Università di Bologna
Facoltà di Scienze

Laurea Triennale in
Ingegneria e Scienze Informatiche

Tesi in materia di
Programmazione ad Oggetti

Tecnologie per la Costruzione di Piattaforme Distribuite basate sul Linguaggio di Programmazione Scala

di
Lorenzo Vernocchi

Relatore:
Professor Mirko Viroli

Anno Accademico 2014/15

Indice

Introduzione

Questo elaborato tratta alcuni dei più noti framework di programmazione avanzata per la costruzione di piattaforme distribuite che utilizzano Scala come fulcro principale per realizzare i propri protocolli. **Scala** è un linguaggio per la programmazione avanzata orientata agli oggetti performante in quanto più potente delle ultime versioni Java, decisamente più versatile degli altri linguaggi nell'implementazione di sistemi di grandi dimensioni.

Per comprendere appieno i contenuti dell'elaborato è necessario prima affrontare un percorso, anche se breve, di approfondimento del linguaggio Scala per comprenderne la logica ed enunciarne le più comuni sintassi di linguaggio.

L'idea di tesi *Tecnologie per la Costruzione di Piattaforme Distribuite basate sul Linguaggio di Programmazione Scala* nasce in seguito a diverse esperienze con la Programmazione ad Oggetti. In particolare, grazie al percorso formativo con il Professor Mirko Viroli, ho avuto l'opportunità di effettuare uno studio approfondito del linguaggio Scala durante un tirocinio presso il suo studio in facoltà.

Inoltre, insieme al Professor Viroli, ho esaminato l'articolo *Eight hot technoloies that were built in Scala* di Laura Masterson, Typesafe Inc. (nella sezione "Sitografia" si potrà trovare un riferimento a tale articolo) che, in occasione del **"Scala Day 2015 – San Francisco"**, fornisce una panoramica delle più famose tecnologie che sfruttano il linguaggio Scala. L'articolo si è dimostrato interessante e stimolante per analizzare a mia volta le tecnologie:

- **Finagle**, sistema RPC per la programmazione di Server altamente concorrenti;
- **Akka**, framework per la programmazione di potenti applicazioni in sistemi distribuiti;
- **Apache Kafka**, sistema di messaggistica per la gestione pulita di grandi quantità di dati;
- **Apache Storm**, sistema distribuito real-time, open source per processare in modo affidabile i flussi di dati di grandi dimensioni.

L’obiettivo dell’elaborato è l’analisi approfondita delle tecnologie sopraelencate per comprendere a fondo le tecniche di programmazione che le rendono uniche nel loro genere. Questo percorso fornisce una chiave di lettura obiettiva e chiara di ciascuna tecnologia, sarà cura del lettore proseguire nello studio individuale della specifica tecnica che ritiene essere più efficace o interessante.

Poiché non è possibile dare un giudizio ed eleggere in questa sede il sistema di programmazione migliore, alla fine della tesi è presente un aperto dibattito in cui le quattro tecnologie vengono messe a confronto e giudicate in base alle loro caratteristiche.

Inoltre vengono ipotizzate realtà in cui si possa trovare collaborazione tra i vari framework ed, infine, è presente una mia personale opinione basata sulla mia esperienza in merito.

La tesi è suddivisa in capitoli dedicati a Scala, Finagle, Akka, Kafka e Storm, ciascun capitolo comprende una sezione di introduzione con lo scopo di offrire un’infarinatura della tecnologia presa in esame, una sezione di documentazione per approfondirne lo studio ed infine una sezione contente una dimostrazione di come tali frame work possono essere messi in pratica.

Il mio personale obiettivo è quello di condividere l’opportunità offertami dal Professor Mirko Viroli. Scala nello specifico permette a coloro che lo studiano di acquisire una visione completa del mondo della programmazione e facilitare di conseguenza lo studio e l’apprendimento di altri linguaggi. Queste quattro tecnologie in quanto accomunate dal linguaggio Scala ma uniche nel loro genere permettono di aumentare le proprie competenze tecniche nel campo della Programmazione ad Oggetti.

Colgo l’occasione per ringraziare il Professor Mirko Viroli per l’opportunità formativa e professionale concessami, per avermi supportato e seguito durante il mio percorso formativo, il mio tirocinio e la stesura della tesi.

Capitolo 1 - Scala

Per comprendere a fondo questa tesi bisogna prima affrontare l'argomento che sta alla base di essa, ovvero **Scala**: un linguaggio per la programmazione avanzata che ridefinisce e perfeziona i metodi della programmazione sia funzionale che ad oggetti e permette la creazione di applicazioni di grandi dimensioni.

L'idea di **Scala**, acronimo per *Scalable Language*, inizia nel 2001 presso *l'Ecole Polytechnique Fédérale di Losanna* da **Martin Odersky**. Dopo il rilascio interno alla fine del 2003, Scala è stato rilasciato pubblicamente all'inizio del 2004 sulla piattaforma Java, e sulla piattaforma .NET nel giugno 2004 (il supporto .NET è stato ufficialmente abbandonato nel 2012). Il 17 gennaio 2011, il "team Scala" ha vinto un assegno di ricerca di cinque anni di oltre 2,3 milioni di € da parte del Consiglio europeo della ricerca. In data 12 maggio 2011, Odersky e collaboratori hanno lanciato **Typesafe Inc.**, una società con lo scopo di fornire supporto commerciale, formazione sul linguaggio di programmazione e creare servizi software in Scala. Typesafe ha ricevuto un investimento di $ 3 milioni in 2011 dal Greylock Partners[1].

Le nuove applicazioni industriali e di rete devono offrire un certo numero di requisiti:

- devono essere implementate velocemente e in maniera affidabile;
- devono offrire un accesso sicuro;
- devono offrire un modello di dati persistenti;
- devono avere un comportamento transazionale;
- devono garantire un'elevata scalabilità, per le quali è necessaria una progettazione che supporti concorrenza e distribuzione;
- le applicazioni sono collegate in rete e forniscono interfacce per essere usate sia da persone sia da altre applicazioni.

[1] **Greylock Partners** è una delle più antiche società di investimenti, fondata nel 1965, con un capitale impegnato di più di 2 miliardi di dollari a titolo di gestione. L'azienda concentra i propri investimenti su società informatiche.

Ad oggi **Scala** è un linguaggio che si rivolge ai bisogni principali dello sviluppatore moderno in grado di soddisfare tutti i requisiti delle applicazioni moderne sopraelencate; di seguito una infarinatura generale di Scala.

Scala è un linguaggio per la Java Virtual Machine a paradigma misto, con una sintassi concisa, elegante e flessibile, un sistema di tipi sofisticato e di idiomi che promuovono la scalabilità dai piccoli programmi fino ad applicazioni sofisticate di grandi dimensioni; molte aziende, tra cui Twitter, LinkedIn e Intel si appoggiano a Scala per implementare gran parte del loro sistema.

Scala supporta appieno la **programmazione orientata agli oggetti**. Concettualmente ogni valore è un oggetto ed ogni operazione è un metodo di chiamata. La tecnica dei **Trait,** per implementare le classi in maniera fluida, migliora il supporto object oriented di Java.

In Scala, ogni cosa è davvero un oggetto, infatti non esistono tipi primitivi come Java, tutti i tipi numerici sono veri oggetti e non sono supportati i membri "statici".

Scala supporta appieno anche la **programmazione funzionale** e, a differenza di molti linguaggi tradizionali, permette una graduale migrazione verso uno stile più funzionale.

Scala opera sulla Java Virtual Machine e **dialoga alla perfezione con Java**:

- Package, classi, metodi e perfino il codice di Java e di Scala possono essere liberamente mischiati. Per quanto riguarda package e classi non importa se essi risiedono in diversi progetti o nello stesso;
- Le classi Scala e le classi Java possono anche riferirsi reciprocamente le une alle altre;
- Il compilatore Scala contiene una parte di un compilatore Java;
- Le biblioteche e gli strumenti di Java sono tutti disponibili all'interno di Scala.

Si può quindi facilmente notare come Scala somigli in tutto e per tutto a Java.

Ma allora perché Scala?

Mentre la sintassi Java può essere prolissa, Scala usa un certo numero di tecniche per minimizzare la sintassi superflua, rendendo il codice tanto conciso quanto il codice scritto nella maggior parte dei linguaggi dinamicamente tipati.

L'inferenza di tipo aiuta a ricavare automaticamente i type nelle dichiarazioni dei metodi e delle funzioni, in modo che l'utente non li debba fornire manualmente, e minimizza il bisogno di esplicite informazioni di tipo in molti altri contesti. Si può dire quindi che Scala estende Java con pattern più flessibili ma più potenti e un altissimo numero di costrutti più avanzati.

In particolare Scala spiazza Java quando si tratta di programmare applicazioni come server che fanno uso di elaborazione simultanea e sincrona, software che utilizzano più core in parallelo oppure protocolli che gestiscano e siano responsabili dell'elaborazione distribuita delle risorse contenute in un Cloud.

Di seguito verrà descritta la sintassi del linguaggio ed i costrutti più comuni ed utili.

Sintassi

La sintassi di Scala non è dissimile da quella del linguaggio Java, per semplificare la comprensione si può pensare a Scala come un "Java abbreviato senza il punto e virgola" (ovviamente Scala non è solo questo).

Innanzitutto si può analizzare come definire una classe Scala. La sintassi è identica a quella di Java ed è *class NomeClasse*.

Scala permette l'utilizzo di due tipologie di variabili: le **val**, sono variabili non modificabili (in Java sono dette final) e le **var**, variabili modificabili. Scala è in grado di riconoscere alla perfezione il tipo di variabile (**tecnica dell'inferenza di tipo**), senza doverlo dichiarare, solamente analizzando il valore assegnato a tale variabile, per esempio:

```
val a = "ciao"
```

Il compilatore Scala non avrà dubbi riguardo alla variabile *a*, è sicuramente di tipo String. Scala è in grado di riconoscere dalle variabili più semplici a quelle più ostiche, ovviamente è possibile dichiarare il tipo di variabile così come segue:

```
val b : String = "ciao"
```

I metodi all'interno di una classe si definiscono con la parola chiave **def** seguita dal nome del metodo e relativi parametri. La dichiarazione del metodo termina con il segno "=" e può essere seguito dalle classiche parentesi graffe (come in Java) se tale metodo richiede più di una istruzione, altrimenti è possibile inserire tale istruzione direttamente dopo il segno "=".

Un altro importante fattore è che **l'inferenza di tipo** di Scala permette al compilatore di capire quale sarà il valore di ritorno di una funzione quindi non occorre né un assegnazione di tipo in fase di definizione del metodo né una clausola di *return*. Ecco vari esempi per definire un metodo:

```
def metodoPrimo = 1
  //metodo senza parametri che ritorna un intero

def metodoSecondo (parametro: Tipo) : valoreDiRitorno = {
  //metodo con varie istruzioni
  /* si noti che definire un valore di ritorno aiuta il
   * compilatore a capire quale sarà il valore all'interno
   * del codice del metodo */
}

def metodoTerzo(par1:Int, par2:Int, par3:Int) =
   par1 + par2 + par3
  //metodo con una unica istruzione
  /* il compilatore non ha dubbi sul valore di
   * ritorno sarà sicuramente un Int */
```

Anche Scala, come Java, mette a disposizione dell'utente le **collezioni**. A differenza di Java, invece, fornisce una panoramica completa di tali oggetti. Infatti non esistono solamente liste, array, mappe e set; Scala mette a disposizione del programmatore anche code e pile. Ciascuna collezione può essere di due tipologie: **mutable** o **immutable**. Non occorre una descrizione dettagliata in quanto è chiaro che le **mutable** possono essere modificate sia in termini di grandezza che di modifica degli elementi all'interno mentre le **immutable** mantengono le dimensioni, l'ordine e i valori settati al momento della creazione.

Per quanto riguarda gli **array**, i **set** e le **mappe**, questi sono identici a quelli messi a disposizione da Java. Le **code** hanno due metodi principali: *enqueue* (che inserisce un valore alla fine della coda) e *dequeue* (che toglie il primo valore in coda). Gli **stack** (o

pile) hanno anch'essi due metodi di base: *push* (che inserisce un valore in cima alla pila) e *pop* (che estrae il primo valore in cima alla pila). Per definire una qualsiasi collezione di Scala si deve seguire la sintassi Java (con qualche vantaggio in più) ovvero:

```
val q = Queue(1,2,3)
/* il compilatore riconosce che q è una Queue[Int]
 * (coda di interi) */

val q = new Queue[Int] //come Java
```

In Scala esistono quattro livelli di classificazione:

- **Il Trait** – corrisponde all'interfaccia di Java con la differenza che un Trait è in grado di implementare tutti i metodi che desidera;
- **L'Abrstact Class** – corrisponde alla classe astratta di Java. Lavorando con queste tipologie di classi occorre fare particolare attenzione a non confonderle con i Trait in quanto c'è pochissima differenza tra i due;
- **Case Class** – tipologia che si interpone tra l'Abstract e la classe standard. Tali classi sono veramente utili per rappresentare oggetti simbolici che non hanno un comportamento specifico o non estendono ulteriormente il comportamento di una classe, non avrebbe quindi senso rappresentarli con una classe standard;
- **Class** – classe standard uguale alla classe Java.

Ovviamente anche Scala fornisce la possibilità di creare **classi generiche** che possono gestire un qualsiasi tipo di valore. La sintassi per implementare una classe generica è la seguente:

```
class Nome [T] { }
```

Con questo si conclude la parte riguardante la sintassi di base del linguaggio Scala, in questo modo si è in grado di comprendere gli eventuali esempi di codice presenti nei capitoli successivi. La sintassi di livello avanzato non verrà trattata. Eventuali tecniche particolari di Scala presenti nei capitoli successivi saranno spiegate in fase di primo riscontro di tali. Il capitolo successivo rappresenta l'inizio effettivo della Tesi sulle *Tecnologie per la costruzione di piattaforme distribuite basate sul linguaggio di programmazione Scala*.

Capitolo 2 - Finagle

di Lorenzo Vernocchi

Tecnologie per la Costruzione di Piattaforme Distribuite
basate sul Linguaggio di Programmazione Scala

2.1 Introduzione

Finagle è un sistema RPC[2] per la Java Virtual Machine (JVM) che viene utilizzato per la high performance computing e per costruire Server ad alta concorrenza. Implementa API uniformi per la programmazione di Client e di Server con prestazioni elevate.

Questo sistema mette in luce, probabilmente, il migliore caso d'uso di Scala: la costruzione di servizi ad elevata scalabilità attraverso l'uso della concorrenza.

Finagle è scritto in Scala e l'intero sistema fa parte dello scheletro del protocollo Client/Server di Twitter.

Questo sistema sfrutta un modello per la programmazione concorrente pulito basato sull'utilizzo dei **Futures**, oggetti che al loro interno incapsulano operazioni concorrenti.

[2] **RPC**: una Remote Procedure Call si riferisce all'attivazione, da parte di un programma, di una procedure o subroutine su un altro computer, diverso da quello sul quale il programma viene eseguito. Quindi l'RPC consente a un programma di eseguire subroutine "a distanza" su computer "remoti", accessibili attraverso una rete.
La chiamata di una procedura remota deve essere eseguita in modo analogo a quello della chiamata di una procedura locale e i dettagli della comunicazione su rete devono essere *trasparenti* all'utente.

I Futures sono la chiave per poter comprendere a fondo Finagle; infatti verranno discussi in modo molto dettagliato così da poter comprendere i Services e di conseguenza i Filters.

I Services sono funzioni utili per implementare sia Client che Server (la Figura n.2 mostra un esempio di un sistema Client/Server). Un Service implementa un metodo che riceve una qualche richiesta di tipo Req e ritorna un Future che rappresenta l'eventuale risultato (o fallimento) di tipo Rep.

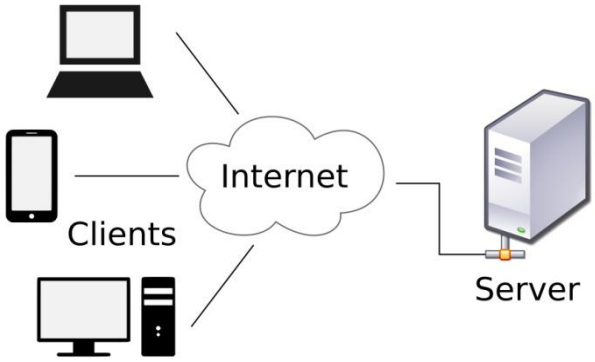


Figura n.1 – Client/Server Network

La Figura n.1 rappresenta una network che vede dispositivi di diversa tipologia (Clients) che tentano di collegarsi allo stesso Server. Un Server deve essere quindi implementato in modo tale da soddisfare qualunque richiesta (pertinente con il suo scopo) indipendentemente dal tipo di dispositivo che la effettua.

Nello specifico un Client utilizza il Service per generare una richiesta e quindi rimane in attesa di una risposta, un Server invece implementa il Service in modo che possa gestire le richieste del Client. A seconda del tipo di richiesta che si vuole gestire, si darà una implementazione appropriata al Service.

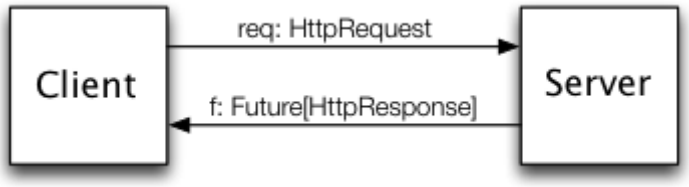


Figura n.2 Client http Request e Server Response

Ai Services Finagle appoggia dei **Filters**, funzioni che permettono di rielaborare i dati passati a e restituiti da una qualsiasi richiesta in modo da renderli compatibili ed elaborabili dai Service. L'immagine sottostante permette di comprendere più a fondo la posizione che Services e Filters ricoprono:

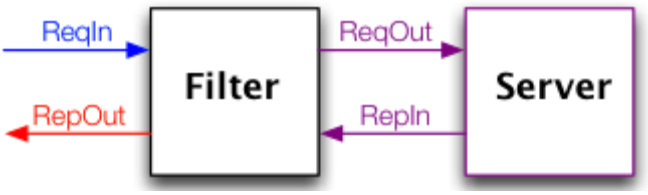


Figura n.3 – Filter prima del Server

Filter e Service possono essere combinati tra loro, inoltre si possono anche combinare filtri con altri filtri. Queste funzioni, combinate correttamente, sono molto utili per creare Client e Server performanti.

I **Server** di Finagle sono molto semplici e sono progettati per rispondere alle richieste rapidamente. Finagle fornisce Server con comportamenti e funzionalità aggiuntive che permettono di debuggare e monitorarne i moduli, inclusi il Monitoring, il Tracing e le statistiche. I **Client** Finagle sono anch'essi progettati per massimizzare il successo e minimizzare la latenza in termini di tempi di attesa.

Ogni richiesta effettuata da un Client Finagle verrà passata attraverso vari *moduli* a suo supporto con lo scopo di raggiungere questo obiettivo.

Nel concreto, ogni componente di un Client e di un Server Finagle è un **ServiceFactory** (vedi par. 2.2.2 – "Service") che permette di creare componenti semplici che, combinate tra di loro, formano un oggetto sofisticato.

Di seguito verranno descritte tutte le componenti di Finagle con lo scopo di approfondirne il funzionamento, verrà poi mostrato un esempio di programmazione di un protocollo Client/Server Finagle.

Si dimostrerà infine che basare l'implementazione dei Client e dei Server su moduli programmati singolarmente sfruttando i Services e i Filters e uniti in seguito mediante uno stack (sfruttando le ServiceFactory) sia decisamente più performante rispetto alla classica (e ormai superata) implementazione mediante Socket.

Finagle infatti sfrutta appieno il linguaggio Scala portando il suo sistema ad un livello avanzato di programmazione.

2.2 Documentazione

2.2.1 I Futures

Finagle sfrutta un modello per la programmazione concorrente pulito, semplice e sicuro basato sull'utilizzo dei **Futures**, oggetti che al loro interno incapsulano ed implementano operazioni concorrenti.

Per capirli meglio possiamo paragonare i Futures ai Thread: infatti agiscono in modo indipendente dagli altri Futures e l'esecuzione di un Future può comportarne la creazione di altri. Inoltre sono poco dispendiosi in termini di memoria, infatti non è un problema gestire milioni di operazioni concorrenziali se queste vengono gestite da futures.

Tra i più comuni esempi di operazioni che utilizzano i Futures troviamo:

- una RPC su host remoto;
- operazioni che richiedono un lungo lasso di tempo computazionale;
- lettura su disco.

Si può notare come queste siano tutte operazioni con possibilità di fallimento: un host remoto può andare in crash, un'operazione può generare eccezioni e la lettura su disco presenta molti casi di fallimento.

Un oggetto di tipo *Future[T]* può presentare, infatti, tre stati:

- **Empty** (in attesa);
- **Succeeded** (ritorna un risultato di tipo *T*);
- **Failed** (ritorna un *Throwable*).

Si può quindi istruire il Future in modo che esegua una determinata istruzione sia in caso di successo:

```
val f: Future[Int]
f.onSuccess { res =>
            /*example*/
            println("Il risultato è " + res)
            //code
            //code
}
```

che in caso di fallimento:

```
f.onFailure { cause: Throwable =>
                    /*example*/
                    println("operazione fallita: " + cause)
                    //code
                    //code
              }
```

Composizione Sequenziale

"The power of Futures lie in how they compose"

Spesso nell'ambito della programmazione è possibile trovarsi di fronte ad operazioni molto costose che possono essere suddivise in una sequenza ordinata di sottoprocessi di dimensioni ridotte e quindi più facili da gestire. L'unione di questi sottoprocessi forma la cosiddetta *operazione composta*.

I Futures permettono di gestire con facilità questa tipologia di operazioni.

Si consideri il semplice esempio di recupero di un'immagine qualsiasi sulla homepage di un sito web. Ciò comporta in genere:

1. Recupero della homepage;
2. Analisi del codice della pagina per trovare un qualsiasi link ad un'immagine;
3. Recupero del link.

Questo è un classico esempio di composizione sequenziale: per poter passare all'i-esimo step occorre prima aver completato lo step precedente. Per gestire questo tipo di operazioni, i Future mettono a disposizione il comando *flatMap*. Il risultato della *flatMap* restituisce il risultato dell'operazione composita. Necessita, ovviamente, di alcuni metodi d'appoggio:

- *fetchUrl* recupera l'URL dato;
- *findImageUrls* analizza una pagina HTML per trovare i collegamenti di immagine.

Possiamo realizzare il nostro **Extractor Images** in questo modo:

```
def fetchUrl(url: String): Future[Array[Byte]]
def findImageUrls(bytes: Array[Byte]): Seq[String]
val url = "http://www.google.com"
val f: Future[Array[Byte]] = fetchUrl(url).flatMap {
        bytes => val images = findImageUrls(bytes)
             if (images.isEmpty)
                Future.exception(new Exception("no image"))
             else
                fetchUrl(images(0))
        }
f.onSuccess { image =>
              println("Found image of size "+image.size)
             }
```

La variabile *f* rappresenta l'operazione composta. È il risultato del primo tentativo di recupero dalla pagina web, nonché il primo link ad un'immagine trovato. In caso una qualsiasi delle sotto operazioni fallisca (*fetchUrl* oppure *findImageUrls*), tutta la composizione sequenziale fallisce.

Composizione Concorrente

È anche possibile comporre Futures in concorrenza. Possiamo estendere il nostro esempio precedente con l'obiettivo di recuperare tutte le immagini della pagina web.

Il metodo *Future.collect* permette di effettuare più operazioni in concorrenza:

```
val collected: Future[Seq[Array[Byte]]] =
fetchUrl(url).flatMap { bytes =>
            val fetches = findImageUrls(bytes).map {
                          url => fetchUrl(url)
                          }
            Future.collect(fetches)
            }
```

In questo esempio ho unito sia la composizione simultanea che la sequenziale: innanzitutto ho recuperato la pagina web, simultaneamente si recuperano tutte le immagini.

Come con la *Composizione Sequenziale*, anche la *Composizione Concorrente* propaga i fallimenti: il *Future Collect* fallirà se uno qualsiasi dei Future sottostanti fallisce. Un codice di questo tipo è molto semplice da scrivere ed intuitivo, inoltre questo

meccanismo è estremamente utile e dà la possibilità di alzare notevolmente il livello di modularità nei sistemi distribuiti.

"Riparare" un fallimento

Un Future composto fallisce quando uno qualsiasi dei Future che lo compone fallisce. A volte può essere utile tentare di riparare un fallimento. Il *rescue combinator* nei *Future* è il duale della *flatMap*: mentre una *flatMap* opera sui valori, *rescue* opera sulle *exceptions*.

L'ideale sarebbe gestire solo un sottoinsieme di possibili eccezioni.

Come si può notare nell'esempio sottostante, per implementare *rescue* occorre innanzitutto passare una *PartialFunction* che mappa un *Throwable* in un *Future*:

```
trait Future[A] {
        def rescue[B>:A](f:PartialFunction[Throwable,
                                           Future[B]]
                        ): Future[B]
}
```

Il codice seguente esegue *rescue* su un'operazione all'infinito, che fallisce tutte le volte a causa di una *TimeoutException*:

```
def fetchUrl(url: String): Future[HttpResponse]
def fetchUrlWithRetry(url: String) =
fetchUrl(url).rescue {
        case exc: TimeoutException => fetchUrlWithRetry(url)
}
```

Nei prossimi paragrafi saranno discussi i *Service* (servizi) e *Filters* (filtri), entità che costituiscono le astrazioni di base con cui i Client e i Server sono costruiti. Sono molto semplici, e molto versatili, la maggior parte dei protocolli di Finagle sono strutturati intorno a servizi e filtri.

2.2.2 Service

Un **Service** è una semplice funzione:

```
trait Service[Req, Rep] extends (Req => Future[Rep])
```

In poche parole, un Service *riceve* una qualche richiesta di tipo Req e *ritorna* un Future che rappresenta l'eventuale risultato (o fallimento) di tipo Rep. I Services sono utili per implementare sia client che server: un'istanza di un Service viene utilizzata da un Client, mentre un Server implementa un Service per processare le richieste e generare risposte.

Ecco un esempio di una richiesta HTTP effettuata da un client:

```
val httpService: Service[HttpRequest, HttpResponse] =
httpService(new DefaultHttpRequest()).onSuccess {
            res => println("received res: "+res)
            }
```

ed un esempio di gestione della richiesta HTTP da parte del server:

```
val httpService = new Service[HttpRequest, HttpResponse] {
        def apply(req: HttpRequest) = /*code*/
}
```

Si può notare come un Client utilizza il Service per generare una richiesta e quindi rimane in attesa di una risposta (DefaultHttpRequest è un Future che in questo caso gestisce solo il successo dell'operazione; vedi par. 2.2.1 – "I Futures"), un Server invece implementa il Service (nello specifico il metodo apply) in modo che possa gestire le richieste del Client. A seconda del tipo di richiesta che si vuole gestire, si darà una implementazione appropriata al metodo apply del Service sul lato Server.

In alcuni moduli, è importante prendere in considerazione il processo di creazione di un Service. Il **ServiceFactory** esiste per questo motivo esatto. Produce dei Services ad hoc per specifici tipi di richieste. La sua definizione:

```
abstract class ServiceFactory[-Req, +Rep] extends
(ClientConnection => Future[Service[Req, Rep]])
```

Nel Client e nei Server di Finagle, i moduli vengono implementati attraverso ServiceFactories e poi composti utilizzando i combinatori di Service e Filters (vedi par. 2.2.3 – "Filters").

2.2.3 Filters

Presentiamo ora un problema:

"Data una richiesta di tipo A e un servizio di tipo Service[A,B],*
restituire un Future di tipo B "*

Abbiamo un Service che non è in grado di gestire richieste di tipo A*, ma solamente richieste di tipo A, inoltre questo Service riesce a mettere a disposizione solo risposte di tipo B, ma a noi serve una risposta di tipo B*, come risolvere questo inconveniente?

Un **Filter** o filtro permette di rielaborare i dati passati a e restituiti da una qualsiasi richiesta in modo da renderli compatibili ed elaborabili dai Service che ricevono la richiesta e/o dai Client che attendono una risposta.

I filtri nello specificano effettuano questo tipo di operazione:

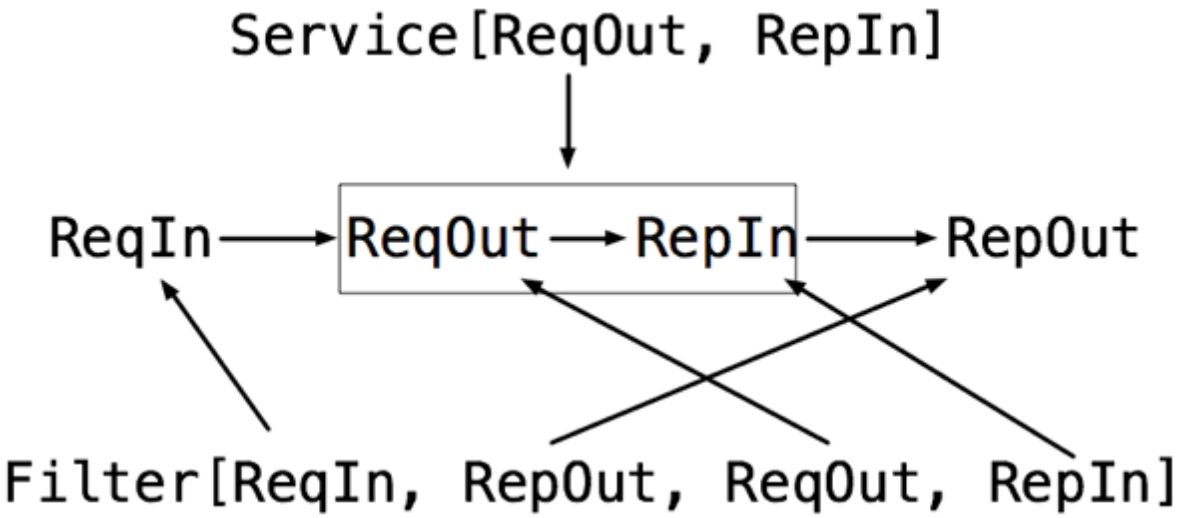


Figura n.4 – Filter e Service

Come i Service, anche i filtri sono semplici funzioni:

```
abstract class Filter[-ReqIn,+RepOut,+ReqOut,-RepIn]
extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
```

Spesso *ReqIn* è uguale a *ReqOut*, e *RepIn* è uguale a *RepOut*. Questo caso è talmente frequente che è stato necessario creare un alias che estenda Filter:

```
trait SimpleFilter[Req,Rep] extends Filter[Req, Rep, Req, Rep]
```

Ecco il **TimeoutFilter** completo:

```
class TimeoutFilter[Req, Rep](timeout: Duration,
                              timer: Timer)
extends SimpleFilter[Req, Rep] {

def apply( request: Req,
           service: Service[Req, Rep]): Future[Rep] = {

                val res = service(request)
                res.within(timer, timeout)

           }
}
```

Al filtro viene passata una richiesta, che crea quindi un Service per gestirla (*val res = service(request)*). *Within* è un metodo che permette di eseguire la richiesta asincronicamente, applicando un tempo di timeout e, se non viene completata la richiesta entro la scadenza, fallisce e lancia un'eccezione.

Filter e Service possono essere combinati tra loro utilizzando il metodo *andThen*. Esempio per fornire ad un Service un comportamento di Timeout:

```
val service: Service[HttpRequest, HttpResponse] = ...

val timeoutFilter = new TimeoutFilter[HttpRequest,
                                      HttpResponse](...)

val serviceWTimeout: Service[HttpRequest, HttpResponse] =
timeoutFilter andThen service
```

L'esempio combina un *Filter* con un *Service*, cosi facendo viene creato automaticamente un nuovo *Service* le cui richieste sono prima filtrate dal *timeoutFilter*.

Si possono anche combinare filtri tra di loro sempre utilizzando *andThen*, per esempio:

```
val timeoutFilter = new TimeoutFilter[..](..)
val retryFilter = new RetryFilter[..](..)
val retryWithTimeoutFilter: Filter[..] =
retryFilter andThen timeoutFilter
```

In questo esempio viene creato un nuovo filtro che invia inizialmente le richieste ad un *retryFilter* e successivamente al *timeoutFilter*. Ora siamo pronti per implementare Client e Server utilizzando le nozioni che Finagle mette a disposizione.

2.2.4 Server

Un Server è un componente (sia hardware che software) di elaborazione che fornisce servizi/risorse ad altre componenti, dette Clients, che ne fanno richiesta.

In altre parole si tratta di un computer o di un programma che fornisce i dati richiesti da altri elaboratori, facendo quindi da host per la trasmissione delle informazioni virtuali.

In Finagle i Server implementano una semplice interfaccia:

```
def serve( addr : SocketAddress,
           factory : ServiceFactory[Req, Rep]):ListeningServer
```

La funzione *serve()*, una volta passati come parametri una ***SocketAddress*** e una ***ServiceFactory***[3], ritorna un oggetto detto ***ListeningServer***, che consente la gestione delle risorse presenti in un Server.

Il metodo *serve()* si richiama con il comando "*Nome_protocollo".serve(...)*, per esempio:

```
val server = Httpx.serve(":8080", myService)
await.ready(server) /* attende finche le risorse presenti
nel Server non sono "rilasciate" */
```

I Server di Finagle sono molto semplici e sono progettati per rispondere alle richieste rapidamente. Finagle fornisce Server con comportamenti e funzionalità aggiuntive che permettono di debuggare e monitorarne i moduli, inclusi il Monitoring, il Tracing e le statistiche.

[3] In alcuni moduli, è importante prendere in considerazione il processo di acquisizione di un Service. Ad esempio, un pool di connessioni dovrebbe svolgere un ruolo significativo nella fase di acquisizione di servizio. Il ServiceFactory esiste per questo motivo esatto. Produce Servizio di oltre il quale le richieste possono essere spediti.

2.2.5 Client

Un Client indica una componente che accede ai servizi o alle risorse del Server. Per accedere a una risorsa, disponibile su un determinato Server, il Client invia una richiesta specificando il tipo di risorsa ed attende che il Server elabori la richiesta e fornisca la risorsa. Il termine Client indica anche il software usato sul computer-client per accedere alle funzionalità offerte da un server.

Come per i Server in Finagle anche i **Client** implementano una semplice interfaccia:

```
def newClient(dest:Name, label:String):ServiceFactory[Req,Rep]
```

Ovvero, data una destinazione logica ed un identificatore, *newClient* restituisce una funzione che produce un service tipizzato attraverso il quale si possono spedire le richieste. Esistono delle varianti per creare dei Client *Stateless* (senza stato) ovvero che al posto di una Factory restituiscono un semplice Service, per esempio:

```
def newService(dest:Name, label:String): Service[Req,Rep]
```

Per creare un nuovo client si usa il comando *"Protocollo".newClient(dest,label)*, per esempio:

```
Http.newClient(…)
```

Di default un Client Finagle è progettato per massimizzare il successo e minimizzare la latenza in termini di tempi di attesa, infatti ogni richiesta effettuata da un Client Finagle verrà passata attraverso vari *moduli* a suo supporto con lo scopo di raggiungere questo obiettivo. Questi moduli sono separati in tre *stacks (pile):*

- **Client Stack** gestisce le richieste di *name resolution* tra end points;
- **Endpoint Stack** fornisce i protocolli per aprire/chiudere una *sessione* e *pool di connessioni*;
- **Connection Stack** gestisce il ciclo di vita di una connessione e implementa il protocollo di tipo *wire*[4].

[4] **Wire**: comunicazione tra sistemi e dispositivi basati su connessioni cablate.

In sostanza un Client di Finagle è un ServiceFactory: produce dei Services che gestiscono le varie richieste. I moduli della figura sottostante sono definiti in termini di ServiceFactory e vengono combinati tra loro (vedi par. 2.2.2 e 2.2.3 – "Service" e "Filters").

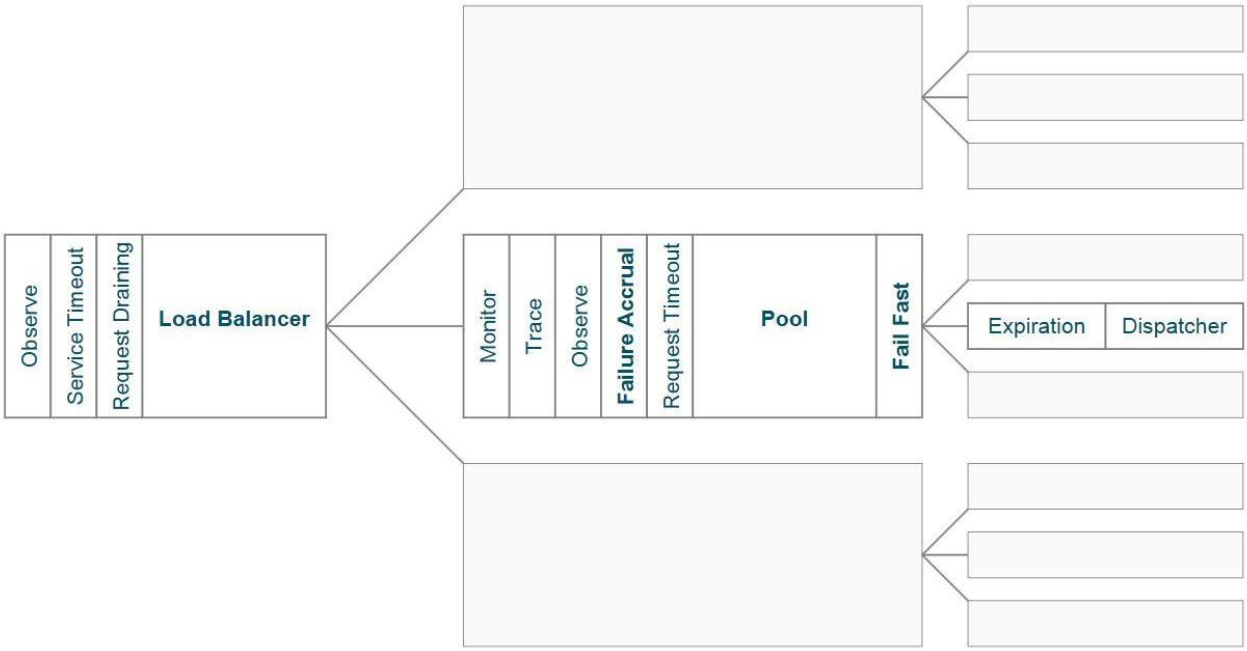


Figura n.5 – Stack dei Moduli di un Client

Una conseguenza è che il comportamento del Client può essere seriamente modificato dai moduli che stanno in fondo alla pila.

Moduli di un Client

In questo sottoparagrafo saranno descritti i vari moduli di un Client come da Figura n.5

Observability

I Moduli **Observe**, **Monitor** e **Trace** forniscono informazioni utili riguardo la struttura interna ed il comportamento di un Client Finagle. Le metriche dei Client vengono esportate mediante uno *StatsReceiver* e il gestore delle eccezioni generiche può essere installato tramite il *MonitorFilter*.

Timeout & Expiration

Finagle fornisce servizi di timeout con **granularità fine**[5]:

- Il modulo **Service Timeout** definisce un timeout per l'acquisizione di un service. Ovvero, definisce il tempo massimo, assegnato a una richiesta, di attesa di un service disponibile. Le richieste che superano il timeout falliscono lanciando *ServiceTimeoutException*. Questo modulo è implementato dal *TimeoutFactory*.
- Il modulo **Request Timeout** è un Filter che impone un limite superiore per la quantità di tempo consentito per una richiesta. Un dettaglio importante nell'implementazione del *TimeoutFilter* è che tenta di annullare la richiesta quando un timeout viene attivato. Con la maggior parte dei protocolli, se la richiesta è già stata spedita, l'unico modo per annullarla è quello di terminare la connessione.
- Il modulo di **Terminazione** (**Expiration**) è situato a livello di connessione e termina un servizio dopo un certo periodo di tempo di inattività. Il modulo è attuato dal *ExpiringService*.

Request Draining

Il modulo di Drain (scarico) garantisce che il Client ritardi la propria chiusura fino al completamento di tutte le richieste in sospeso.

Load Balancer

I Client di Finagle sono dotati di un sistema di bilanciamento del carico (Load Balancer), un componente fondamentale dello stack, il cui compito è di distribuire in modo dinamico il carico attraverso una collezione di endpoint intercambiabili. Questo dà a Finagle l'opportunità di massimizzare il successo e ottimizzare la distribuzione delle richieste, nel tentativo di ridurre al minimo le latenze del client.

[5] **Granularità**: in informatica indica il livello di dettaglio utilizzato per descrivere un'attività o una funzionalità con riferimento alle dimensioni degli elementi che la compongono o che vengono gestiti: si passa dalla *granularità grossolana* (*coarse*) per componenti relativamente grandi alla **granularità fine** (*fine*) per componenti più piccoli.

Il **Load Balancer** è suddiviso in due sottomoduli:

- **Heap + Least Loaded (Carico Minimo)** - Il distributore è un **heap**[6] che è condiviso tra le richieste. Ogni nodo dell'heap mantiene un conteggio di richiesta in sospeso. Il conteggio viene incrementato quando una richiesta viene inviata e decrementato quando si riceve una risposta. L'heap è di tipo min-heap per consentire un accesso efficiente minimizzando il carico.

- **Power of Two Choices (P2C) + Least Loaded** - Il distributore P2C risolve molte delle limitazioni che sono inerenti con il distributore Heap. L'algoritmo sceglie casualmente due nodi della collezione di endpoint e seleziona quello con carico minimo (least loaded). Usando questa strategia, ci si può aspettare un upper bound gestibile del carico massimo di qualsiasi server. La metrica di carico di default per il sistema di bilanciamento P2C è di tipo Least Loaded inoltre, siccome P2C è pienamente concorrente, ci consente di implementare in modo efficiente nodi di diverse metriche di carico con costi minimi per ogni richiesta.

Session Qualification

Il seguente modulo mira a disattivare preventivamente le sessioni le cui richieste presentano un'alta probabilità di fallire. Dal punto di vista del bilanciamento del carico, si comportano come interruttori che, una volta attivati, sospendono temporaneamente l'uso di un particolare endpoint.

Il modulo **Fail Fast** (**fallimento veloce**) tenta di ridurre il numero di richieste spedite che potrebbero fallire. Il modulo opera *marcando* gli host che sono stati abbattuti a

[6] **Heap:** è una struttura dati, più precisamente un vettore o una lista che soddisfa la proprietà *heap* (può essere visto come un albero binario incompleto). È usato principalmente per la raccolta di collezioni di dati, dette dizionari, e per la rappresentazione di code di priorità.
Dato *j*, indice ad un nodo del heap, si definiscono:
- *Padre di j* il nodo in posizione *j/2*;
- *Figlio sx di j* il nodo in posizione *j*2*;
- *Figlio destro di* **j** il nodo in posizione *(j*2)+1*.

Esistono due tipi di heap: **min-heap** e **max-heap**. La scelta di utilizzare un tipo di heap anziché l'altro è data dal tipo di impiego che se ne vuole fare. Dato j indice di posizione della struttura e v lo heap preso in considerazione:
- *min-heap*: se **v[Padre(j)] < v[j]**;
- *max-heap*: se **v[Padre(j)] > v[j]**.

In ogni nodo è presente una coppia *(k,x)* in cui *k* è il valore della chiave associata alla voce *x*. Questi tipi di albero hanno la seguente caratteristica: *qualsiasi nodo padre ha chiave minore di entrambi (se esistono) i suoi figli*. In questo modo si garantisce che compiendo un qualsiasi percorso che parte da un nodo *v* dell'albero e scendendo nella struttura verso le foglie, si attraversano nodi con chiave sempre maggiore della l'ultima foglia visitata.

causa di una connessione fallita e lanciando un processo in background che tenta ripetutamente di ristabilire la connessione. Durante il tempo in cui un host è marcato, la *ServiceFactory* (vedi pag. 12) è contrassegnata come non disponibile e diventerà di nuovo disponibile in caso di successo o quando il processo di background si esaurisce.

Pooling

Finagle fornisce un pool generico detto **Watermark Pool** che mantiene una collezione di istanze di un Service. Al Client, per ogni endpoint al quale si connette, viene messo a disposizione un pool indipendente dagli altri.

Esistono due livelli di assegnazione dei Service:

- *lower bound;*
- *upper bound.*

Il Watermark Pool assegna i Services persistenti (ovvero processi che impiegano un lasso di tempo elevato per terminare) al *lower bound*, mentre assegna all'*upper bound* tutti i nuovi Services entranti nel pool. Ogni qual volta un servizio termina il suo operato viene chiuso e rimosso dal pool. Il programmatore può decidere di spostare un Service da upper bound a lower bound, però se si vuole richiamare un servizio che risiede nel lower bound, il Watermark Pool tenta immediatamente di chiuderlo.

Se l'applicazione richiede frequentemente connessioni di Service di tipo lower bound si rischia la creazione di **collegamenti spazzatura (inesistenti)**, causati dal tentativo del Watermark Pool di chiudere i Service richiamati. Per ridurre il tasso di collegamenti spazzatura, esiste una struttura separata per la cache, con un **TTL (Time to Live)** o "*tempo di vita"*, per tutti i Service che stanno nel lower bound: il **Caching Pool**.

Il **Caching Pool** opera indipendentemente dal numero di Service di tipo lower bound aperti e li mantiene in una cache in modo da non perderne traccia. La cache raggiunge il suo valore massimo quando si raggiunge la vetta della concorrenza e poi lentamente decade, in base alla TTL.

Il Client Finagle, di default, tenta di mantenere al minimo il numero di Services presenti nel lower bound cercando quindi di accodare più richieste possibili nell'upper bound. Ovviamente alcune richieste rischiano di non essere eseguite nell'immediato, occorre quindi dichiararle persistenti e spostarle nel lower bound applicandovi un TTL.

2.2.6 I Nomi

Per introdurre questo paragrafo si riprende brevemente il concetto di percorso di rete. Per comodità degli utenti che navigano la rete, agli indirizzi IP vengono associati dei *nomi simbolici (Domini)* che identificano quindi il percorso di rete a ciascun terminale.

Finagle si avvale dei **Nomi** per identificare i percorsi di rete e quindi associarli ai relativi indirizzi IP. Queste entità vengono utilizzate quando si crea un Client mediante *ClientBuilder.dest* oppure mentre si implementa direttamente il Client.

I Nomi sono rappresentati dalla classe **Name** e si possono definire in due modi:

- `case class Name.Bound(va: Var[Addr])` - Identifica un set di percorsi di rete. **Var[Addr]** rappresenta appunto un insieme di indirizzi intercambiabili;
- `case class Name.Path(path: Path)` - Rappresenta un nome come un percorso gerarchico formato da una sequenza di stringhe.

Resolver.eval[7] converte le stringhe in **Names**. Stringhe di forma:

```
scheme!arg
```

dove ***scheme*** è il tipo di metodo con cui effettuare la conversione in Name, mentre ***arg*** è l'argomento da tradurre; per esempio:

```
inet!twitter.com:80
```

In questo caso arg identifica twitter.com:80 e viene utilizzato *inet* per risolvere l'indirizzo. ***Inet***, nello specifico, utilizza il DNS per effettuare la traduzione.

Inet è anche il metodo utilizzato per default infatti:

```
twitter.com:8080
```

è uguale a:

```
inet!twitter.com:8080
```

[7] **Resolver**: è un object del package *com.twitter.finagle* che presenta tre metodi:
- **eval** (name: **String**):**Name** che traduce l'argomento passato come parametro in un Name (se esiste);
- evalLabeled(addr: **String**): (**Name**,**String**) che traduce l'argomento in una tupla (Name, String)
- get [T <: **Resolver**] che ritorna un Resolver o un suo sottotipo (T)

Name.Bound vuole un Var[Addr] passato come parametro che rappresenta un Address (indirizzo) che cambia dinamicamente; Addr può trovarsi in uno di questi stati:

- **Addr.Pending** - fase di binding (collegamento) ancora in corso, probabilmente in attesa di una risposta da parte del DNS oppure del completamento di un operazione da parte di Zookeeper;
- **Addr.Neg** - binding con esito negativo, significa che la destinazione non esiste;
- **Addr.Failed(cause: Throwable)** - binding fallito con relative causa (cause);
- **Addr.Bound(addrs: Set[SocketAddress])** - binding terminato con successo, *addrs* rappresenta un set di indirizzi tutti validi (ogni indirizzo rappresenta un endpoint).

Quanto spiegato in questo paragrafo è di vitale importanza per permettere ad un Client di raggiungere un Server in modo semplice ed efficace senza dover ricordare tutti gli indirizzi IP e relativi numeri di porta.

Prima mi passare alla sezione relativa all'esempio di codice si aggiunge quanto segue con lo scopo di terminare il percorso con Finagle.

I Client e i Server di Finagle comprendono molti componenti relativamente semplici, disposti insieme in uno *stack* (vedi "Moduli di un Client" in par. 2.2.5 – "Client").

Nel concreto, ogni componente è un ServiceFactory, che a sua volta compone altri Service e ciò permette di creare componenti semplici che, combinate tra di loro, formano un oggetto sofisticato.

Lo **Stack** formalizza il concetto di *componente impilabile* e tratta una sequenza di parti sovrapponibili, ognuna con il proprio comportamento, che possono essere manipolate, possono essere inserite o rimosse dallo stack ed è inoltre possibile associare uno stack ad un altro.

Ora si è a conoscenza del sistema Finagle nello specifico e si è preparati per implementare un proprio protocollo Client-Server, nella sezione successiva sarà mostrato un esempio per realizzarlo.

2.3 Elaborato: Finagle Client e Server

2.3.1 Analisi del Problema

Creare un protocollo Server che riceva un testo da parte di un Client, elaborare il testo inserendo un "a capo" ogni N bit e ritornare il testo formattato. Per facilità il testo è codificato in UTF-8.

Innanzi tutto definisco un protocollo a livello di trasporto; Finagle rappresenta il livello di trasporto OSI come un flusso tipizzato che può leggere e scrivere in modo asincrono. I metodi nel trait sono definiti come tali:

```
trait Transport[In, Out] {
     def read(): Future[Out]
     def write(req: In): Future[Unit]
}
```

Il trait Transport è contenuto all'interno del package *com.twitter.finagle.transport* ed è compito del programmatore creare una classe che estenda da Transport e quindi implementi i metodi read() e write() nel modo ritenuto più opportuno.

Di seguito verranno descritti in modo dettagliato tutti i passaggi per programmare prima il Server con relativa funzione "a capo" poi il Client. Infine si effettuerà la richiesta da parte del Client.

Premessa: Nei prossimi esempi di codice ho utilizzato Netty[8], un framework che mette a disposizione protocolli I/O di tipo client/server sia single che multi - client.

[8] **Netty** è un framework client-server NIO che permette lo sviluppo rapido e semplice di applicazioni di rete come i protocolli server e client . Esso semplifica e snellisce la programmazione di rete come TCP e UDP. Semplice e veloce non significa che un'applicazione risultante soffrono di una manutenzione o di un problema di prestazioni .
Netty realizza molti protocolli come FTP, SMTP, HTTP e vari protocolli binari.

2.3.2 Progettazione

Server

Per questo protocollo Server si è utilizzata un'estensione della **ChannelPipeline**[9] messa a disposizione da Netty. Innanzi tutto si definisce l'oggetto **StringServerPipeline** che effettua l'operazione vera e propria di delimitazione:

```
object StringServerPipeline extends ChannelPipelineFactory {

/* ChannelPipelineFactory è un'interfaccia che sfrutta il
 * pattern Factory per definire una ChannelPipeline
 * semplicemente fornendo un'implementazione del metodo
 * getPipeline */


     def getPipeline = {

          //creo la pipeline
          val pipeline = Channels.pipeline()

          /* definisco il mio distanziatore che manda a capo il
           * testo inviato dal server dopo 30 bit */
          pipeline.addLast( "line",
                           new DelimiterBasedFrameDecoder(
                                30,Delimiters.lineDelimiter:_*
                                )
                           )

          /* definisco uno Decoder e un Encoder di codifica
           * UTF-8 */
          pipeline.addLast("stringDecoder",
                          new StringDecoder(CharsetUtil.UTF_8)
                          )

          pipeline.addLast("stringEncoder",
                          new StringEncoder(CharsetUtil.UTF_8)
                          )

          pipeline //ritorno la pipeline
     }
}
```

[9] Una **ChannelPipeline** è una lista di ChannelHandler. Un **ChannelHandler** è un oggetto che gestisce/intercetta gli eventi su un canale e a sua volta invia un evento al ChannelHandler successivo all'interno della ChannelPipeline.

Ora serve un **Listener**: un oggetto che intercetti eventi generati dalla Socket di rete e che, avvenuto un evento, collochi la pipeline sul canale di trasporto (per esempio dopo un evento di tipo "*send*").

Per facilitare l'implementazione del Server si è scelto di utilizzare come Listener il **Netty3Listener**, oggetto messo a disposizione dal package *com.twitter.finagle*.

```
protected def newListener(): Listener[String, String] =
Netty3Listener(StringServerPipeline, params)
```

Per costruire un Listener occorre estendere il *trait* **Listener** del package *com.twitter.finagle* ed implementare questo metodo[10] :

```
def listen     (ad : SocketAddress)
               (serveTran : Transport[In, Out] => Unit)
               : ListeningServer
   /* Vale a dire, dato un indirizzo ad, viene messo a
    * disposizione un protocollo di trasporto serveTran per
    * ogni nuova connessione stabilita */
```

Il **ServerDispatcher** (package *com.twitter.finagle.dispatch*) è un oggetto messo a disposizione da Finagle che accoda le richieste in entrata e le invia una per volta ad un Transport. Ogni dato letto dal Transport viene incanalato ad un *Service* che lo elabora e restituisce il risultato al Transport stesso.

Inoltre, il ServerDispatcher effettua lo **scarico** (**drain**) delle richieste prima della chiusura del Transport (il *modulo di Drain* di un Server Finagle ha le stesse funzionalità del modulo di Drain di un Client – vedi par. 5.1 *"Moduli di un Client"*).

[10] Si presti particolare attenzione a come è scritto il metodo *listen* e, nello specifico, alla sezione (serveTran:**Transport**[**In**,**Out**]=>Unit). In Scala si definisce la funzione **first-class** ovvero, in questo caso, l'oggetto passato come parametro in realtà è a sua volta un metodo. Possiamo immaginarla come una funzione che è stata istanziata. Scala permette di creare questo tipo di variabili che, ogni volta che vengono richiamate, eseguono il metodo racchiuso al loro interno. È sbagliato paragonare le first-class ad un metodo statico di Java; Scala non prevede metodi statici e l'unico surrogato dello *static* di Java è lo *Scala Pattern Singleton*.

Si può quindi implementare la funzione *serveTran*:

```
/* Service per l'elaborazione dei dati inviati da parte del
Transport (ancora da definire) */
val service = new Service[String, String] {
   def apply(request: String) = Future.value(request)
}

/* Il serveTran, a cui passo un Transport t e il mio Service.
SerialServerDispatcher è un'implementazione standardizzata
della classe ServerDispatcher */
val serveTran = (t: Transport[String, String]) =>
new SerialServerDispatcher(t, service)

/* Il Listener di tipo Netty3Listener a cui passo il processo
cuore del Server e dei parametri di default */
val listener = Netty3Listener[String, String](
               StringServerPipeline,
               StackServer.defaultParams )

/* Un oggetto che resta in ascolto dato un SocketAddress */
val server = listener.listen(ad) { serveTran(_) }
```

Il server non è completo, occorre infatti aggiungere alcune funzionalità per renderlo robusto (per esempio aggiungere un timeout oppure un controllo sulla concorrenza). Finagle mette a disposizione la classe **StdStackServer** che, oltre a combinare un *Listener* e un *Dispatcher*, come visto in precedenza, fornisce un Filter di Timeout, statistiche e controllo sulla concorrenza e permette di rintracciare i messaggi.

La classe StdStackServer fornisce uno stack di strati con le funzionalità appena elencate e può effettuare uno **shutdown** (arresto del server) regolare e sicuro grazie al modulo di Drain messo a disposizione da Finagle.

Utilizzando il Listener ed il Dispatcher come sopra, si può implementare il Server finito. I parametri *In* e *Out* vengono utilizzati solo se il tipo *Listener* differisce dal tipo *Server*. Questo succede spesso siccome alcuni protocolli sono eseguiti all'interno del *Dispatcher*.

Di seguito il codice del **Server** finito:

```
case class Server(
      stack : Stack[ServiceFactory[String, String]]=
      StackServer.newStack,
      params: Stack.Params = StackServer.defaultParams
      ) extends StdStackServer[String, String, Server] {

protected type In = String
protected type Out = String

protected def copy1(
          stack:Stack[ServiceFactory[String,String]]=this.stack,
          params:Stack.Params = this.params): Server =
      copy(stack, params)

protected def newListener(): Listener[String, String] =
      Netty3Listener(StringServerPipeline, params)

protected def newDispatcher(transport: Transport[String,String],
                            service: Service[String,String]) =
      new SerialServerDispatcher(transport,service)
}
```

Infine il **Service**:

```
  val service = new Service[String, String] {
     def apply(request: String) = Future.value(request)
  }

  val server = Echo.serve(":8080", service)
  Await.result(server)
```

Il Server resterà quindi in ascolto di eventuali richieste da parte di un Client, definito nel paragrafo successivo, elaborerà i dati (in questo caso un testo) passati a richiesta e ritornerà il testo formattato, con l'aggiunta di un "a capo" ogni 30 bit (in questo caso).

Ora verrà descritto come creare un Client che si colleghi al Server ed effettui la richiesta.

Client

Per il Client la procedura di implementazione è la medesima del Server ed anche in questo caso ho utilizzato la ChannelPipelineFactory messa a disposizione da Netty.

```
object StringClientPipeline extends ChannelPipelineFactory {

  def getPipeline = {

    val pipeline = Channels.pipeline()
    pipeline.addLast("stringEncode",
                     new StringEncoder(CharsetUtil.UTF_8))

    pipeline.addLast("stringDecode",
                     new StringDecoder(CharsetUtil.UTF_8))

    pipeline.addLast("line", new DelimEncoder('\n'))

    pipeline
   }
}


class DelimEncoder(delim: Char)
extends SimpleChannelHandler {
override def writeRequested(ctx: ChannelHandlerContext,
                            evt: MessageEvent
                            ) = {

        val newMessage = evt.getMessage match {
             case m: String => m + delim
             case m => m
        }

        Channels.write( ctx,
                evt.getFuture,
                newMessage,
                evt.getRemoteAddress )

  }
}
```

Si definisce ora un Transporter ovvero un'entità capace di connettere un Transport ad un peer, stabilendo una sessione. Il Client utilizza il Netty3Transporter.

```
protected def newTransporter(): Transporter[String, String] =
Netty3Transporter(StringClientPipeline, params)
```

Come per il Server occorre creare un **ClientDispatcher**, un oggetto che trasforma un **Transport** (ovvero un flusso di dati) in un Service (ovvero una coppia di tipo *request - response*). Il **Dispatcher** deve gestire tutte le richieste in sospeso ed associare quindi i risultati di ritorno dal Server alle rispettive.

Finagle di default mette a disposizione il **SerialClientDispatcher**, che consente solamente una richiesta in sospeso per volta (le richieste simultanee sono messe in coda e risolte una per volta).

Una volta definiti un **Transporter** ed una strategia di dispatching delle richieste si può implementare un Client, per esempio:

```
val addr = new java.net.InetSocketAddress("localhost", 8080)

val transporter = Netty3Transporter[String,String](
                  StringClientPipeline,
                  StackClient.defaultParams
                  )

val bridge: Future[Service[String, String]] =
    transporter(addr)map {
         transport => new SerialClientDispatcher(transport)
    }

val client = new Service[String, String] {
    def apply(req: String) = bridge flatMap {
         svc => svc(req) ensure svc.close()
    }
}
```

Si può quindi creare la richiesta:

```
val result = client("Un Server è un componente sia hardware che
                     software”)
println(Await.result(result))
```

Assumendo che il Server sia in ascolto, la risposta che ci si aspetta sarà:

```
$ ./sbt run
> Un Server è un componente sia
> hardware che software
```

Come si può notare nella parola “sia” casca il trentesimo bit e quindi il testo è stato formattato per andare a capo.

Nel caso appena descritto il Client è un semplice Service, quindi si possono aggiungere dei nuovi comportamenti, sfruttando la proprietà di Stack discussa precedentemente, in modo da renderlo più robusto:

```
val retry = new RetryingFilter[String, String](
      retryPolicy = RetryPolicy.tries(3),
      timer = DefaultTimer.twitter
      )

val timeout = new TimeoutFilter[String, String](
      timeout = 3.seconds,
      timer = DefaultTimer.twitter
      )

val maskCancel = new MaskCancelFilter[String, String]
```

Ecco il risultato della composizione di questi nuovi componenti con il Client:

```
val newClient = retry andThen
                timeout andThen
                maskCancel andThen
                client

val result = newClient("T e s t o … ")
println(Await.result(result))
```

Per l'implementazione di questo particolare problema si è deciso di seguire le linee guida definite da Finagle pertanto questo Client si è dimostrato funzionale; purtroppo non permette di gestire richieste in concorrenza per un singolo host. Un Client tipico di Finagle deve riuscire a gestire un enorme numero di richieste in concorrenza.

La classe **StdStackClient** di Finagle (package *com.twitter.finagle.client*) combina un *Transporter*, un *Dispatcher* e uno *Stack* per implementare un Client robusto, con gestione del carico in modo bilanciato e che gestisce le richieste concorrenti.

Di seguito in codice del **Client** finito:

```
case class Client(
stack:Stack[ServiceFactory[String,String]]=StackClient.newStack,
params:Stack.Params = StackClient.defaultParams
) extends StdStackClient[String, String, Client] {

  protected type In = String
  protected type Out = String

  protected def copy1(
                stack: Stack[ServiceFactory[String, String]],
                params: Stack.Params
                ): Client = {

                copy(stack, params)
  }

  protected def newTransporter(): Transporter[String, String]={
     Netty3Transporter(StringClientPipeline, params)
  }

  protected def newDispatcher(
                      transport : Transport[String, String]
                      ): Service[String, String] = {
                new SerialClientDispatcher(transport)
  }
}
```

Ed ora si può creare il Client, collegarlo ad una destinazione utilizzando un Name ed avviare la richiesta:

```
  val dest = Resolver.eval("localhost:8080,
                           localhost:8081,
                           localhost:8082"
                           )
  client.newClient(dest)
```

Per concludere, lo scopo di questo capitolo è quello di spiegare nel dettaglio le singole componenti fondamentali di Finagle. La sezione 3 ha lo scopo di mostrare nel concreto un esempio dell'implementazione di queste componenti e il risultato che si può ottenere. Si può facilmente notare come il modello dello stack di funzionalità programmate separatamente e quindi combinate tra loro sia decisamente più pulito e semplice rispetto alla programmazione basata su Socket, per non parlare delle altissime prestazioni che Finagle raggiunge.

Capitolo 3 - Akka

di Lorenzo Vernocchi

Tecnologie per la Costruzione di Piattaforme Distribuite
basate sul Linguaggio di Programmazione Scala

3.1 Introduzione

Akka è un toolkit open-source che è stato sviluppato dalla Typesafe Inc. con lo scopo di semplificare la realizzazione di applicazioni concorrenti e distribuite sulla JVM. Akka supporta più modelli di programmazione per la concorrenza (Futures, Agents …), ma predilige il modello basato sugli **Actors**.

I casi d'uso in cui Akka diventa competitivo e performante sono la creazione di social media, la creazione di giochi multiplayer online e la creazione di sistemi (piattaforme) per le scommesse online.

Esistono binding di linguaggio sia per Java che per Scala. **Akka è scritto in Scala.**

Akka è basato sul linguaggio Erlang che è stato sviluppato per gestire apparecchiature di telecomunicazione parallele e tolleranti ai guasti.

Gli **Actors** (attori) sono oggetti che incapsulano uno stato, un comportamento e una mailbox (casella postale); inoltre un Actor può avere uno o più figli. Questi oggetti comunicano tra loro esclusivamente attraverso lo scambio di messaggi che vengono inviati alla mailbox del destinatario.

In un certo senso, gli Actors sono una forma più severa della programmazione object-oriented.

Gli Actors offrono:

- Un alto livello di astrazione per la concorrenza e il parallelismo;
- Un modello di programmazione asincrono ed altamente performante per la gestione degli eventi;
- Processi event-driven molto leggeri.

Ogni Actor ha un metodo "**receive**" con scopo di gestione dei messaggi ricevuti. Questo metodo può essere paragonato al pattern match[11] di Scala che, a seconda del tipo di messaggio ricevuto, specifica un particolare comportamento da seguire. La receive, implementata in modo indipendente da Actor ad Actor, gestisce tutti i tipi di messaggi che uno specifico Actor è in grado di riconoscere, altrimenti per ogni messaggio sconosciuto si esegue un caso di default.

```
def receive = {

        case m:Message   =>  //comportamento
        case add:Add     =>  //comportamento
        case add:AckAdd  =>  //comportamento
        case rem:Remove  =>  //comportamento
        case p:PrintN    =>  //comportamento
        .
        .
        case _ => //caso di messaggio sconosciuto
}
```

Esistono due tipi di messaggi:

- **Tell –** un processo invia un messaggio ad un Actor senza stopparsi;
- **Ask –** un processo invia un messaggio ad un Actor e si ferma in attesa di una specifica risposta.

Per quanto riguarda l'Ask viene impostato un tempo di **Timeout** in cui il processo mittente attende una risposta dal destinatario; allo scadere del Timeout o in caso di risposta negativa il processo mittente lancia una eccezione.

Akka non permette di creare attori liberamente, bensì mette a disposizione del programmatore un Factory Method che crea degli **ActorRef**, ovvero attori tutti dello stesso tipo che possono inglobare la configurazione di uno specifico Actor. In questo modo l'unica maniera per comunicare con un Actor è inviandogli un messaggio.

[11] **Match**: il match in Scala corrisponde allo switch in Java; il caso di default è individuato con l'underscore "_".

La caratteristica per eccellenza di qusti è che i compiti vengono suddivisi tra Actors che a loro volta ne delegano una parte ad altri Actors fino a che il problema P, suddiviso nei sottoproblemi P1, P2, …, Pn, non diventi abbastanza piccolo da poter essere facilmente risolvibile.

Un Actor che vuole suddividere il suo compito in parti più piccole, in modo da renderlo facilmente gestibile, invia K messaggi (quindi suddivide il compito in K parti) con lo scopo di creare K Actors figli, che a loro volta potrebbero suddividere ulteriormente il loro compito.

Il *requisito necessario* per implementare questa strategia è che ogni Actors che demanda il suo compito a terzi deve essere in grado di tenere traccia dei propri figli (e dei figli dei propri figli) in modo da poter ricostruire la soluzione finale.

Se un Actor figlio non è in grado di gestire il compito assegnatogli dal "padre", invierà a quest'ultimo un messaggio di errore. Il "padre" a sua volta tenterà di gestire il fallimento. Si crea quindi una struttura ad albero che gestisce in modo ricorsivo gli errori generati dai livelli inferiori.

Un'altra importante proprietà degli oggetti Actor è l'assoluta indipendenza tra un Actor e l'altro, infatti un Actor deve "preoccuparsi" esclusivamente di eseguire con efficienza il proprio compito ignorando l'esistenza di altri Actors che operano in parallelo.

Lo scopo di questa sezione è introdurre gli Actors e spiegare come inviare e gestire messaggi. Da questa sezione si può stabilire che tali oggetti sono la più piccola unità prevista all'interno di applicazioni concorrenti.

La sezione successiva esaminerà nello specifico i concetti appena visti, riprenderà il concetto Futures (spiegato dettagliatamente nel capitolo 2 - Finagle) ed introdurrà gli Agents.

Lo studio approfondito di questa documentazione permetterà di realizzare un elaborato altamente concorrente che si basa esclusivamente sugli Actors e sfrutta gli oggetti ed i pattern messi a disposizione da Akka (vedi sez. 3 – "Elaborato: Neighbourhood").

3.2 Documentazione

3.2.1 Gli Attori

Si riprende il concetto enunciato dalla sezione precedente: gli **Actors** o attori sono oggetti capaci di incapsulare uno stato, un comportamento, una mailbox (casella postale) e possono avere figli.

In Akka *non è possibile creare un oggetto Actor* concretamente; quello che Akka permette di fare è la realizzazione di una classe che estenda Actor (contenuta nel package *akka.actor*) al fine di beneficiare del modello ed ereditare tutte le caratteristiche sopra elencate. Pertanto, occorre precisare che non si sta creando un vero e proprio Actor ma si sta implementando un certo tipo di comportamento. Ecco un esempio di classe che estende da Actor:

```
class Inhabitant extends Actor {

   def receive = {
   /* codice con tutti i messaggi
    * gestibili da questo tipo di Actor,
    * ovvero Inhabitant, e relativi
    * comportamenti in risposta a ciascun tipo di
    * messaggio che */
   }
}
```

Il concetto di istanza della classe Actor è molto complesso; occorre infatti tener presente che non è possibile istanziare un Actor (in questo caso prendiamo Inhabitant come esempio) ma si può solamente far riferimento al suo comportamento. Si presti particolare attenzione a quanto segue.

Per creare un oggetto Inhabitant (quindi un attore specifico) Akka non permette l'uso della forma:

```
val myActor : Inhabitant = new Inhabitant ( )
```

Permette invece la creazione di un oggetto che faccia riferimento ad Inhabitant.

Gli Actors sono quindi rappresentati dai cosiddetti **ActorRefs**, che sono oggetti che possono essere passati liberamente come parametri senza restrizioni. Ridefiniamo quindi il concetto di *oggetto* in **oggetto esterno**.

Per semplicità si può paragonare un "**oggetto esterno"** ad un puntatore *P*. *P* punta ad uno specifico *oggetto A* (che mette a disposizione un "ologramma" di se stesso). Questo "puntatore" non permette in alcun modo di interagire con *A* (non è possibile modificarlo o aggiungervi funzionalità), permette di utilizzare il suo "ologramma" sfruttando quindi tutte le sue caratteristiche.

Questo tipo di programmazione è molto sicura e totalmente trasparente. L'utente è convinto di utilizzare un'istanza di *A* senza rendersi conto che sta operando con un oggetto fantasma, l'**ActorRef** nel caso di Akka, che di fatto non è *A*. Esiste solamente un riferimento alle sue funzionalità.

Se si volessero implementare nuove funzionalità di *A* occorrerebbe creare una nuova classe *B* che estenda *A* e realizzi tali funzionalità.

Questa tecnica mantiene l'oggetto (in questo caso *A*) in sicurezza, in quanto non modificabile direttamente, inoltre gli ActorRef sono molto più "leggeri" in termini di memoria rispetto ad un oggetto vero e proprio.

Questa divisione in oggetto specifico ed esterno consente la trasparenza per tutte le operazioni desiderate:

- Il riavvio di un Actor senza la necessità di aggiornare i riferimenti altrove;
- L'oggetto vero e proprio viene mantenuto in sicurezza su host remoti;
- L'invio di messaggi tra Actors può avvenire in applicazioni completamente differenti.

Non è possibile guardare dentro ad un Actor ed entrare in possesso del suo stato dall'esterno, a meno che l'Actor non pubblichi incautamente questa informazione. Questi oggetti comunicano tra loro esclusivamente attraverso lo scambio di messaggi.

Per poter creare un ActorRef occorre prima di tutto definire un **ActorSystem**. Si introduce quindi tale concetto; un ActorSystem è una struttura contenuta all'interno del package *akka.actor* che ha lo scopo di allocare in modo efficace e veloce da 1 a N ActorRef.

Ecco un esempio per la creazione di un oggetto ActorSystem e successivamente un oggetto ActorRef:

```
val system = ActorSystem ("Nome dell'ActorSystem")
val myActor:ActorRef = system.actorOf(Props[ConcreteActor],
                                      name = "nome dell'Actor"
                                      )
```

ConcreteActor è la classe che implementa un comportamento e, come già precisato in precedenza, deve estendere da Actor. Per esempio, in questo caso si sarebbe potuto utilizzare *Inhabitant* come ConcreteActor.

Props è una configuration class che specifica le opzioni per la creazione di ActorRef, per semplificarne la comprensione può essere paragonato ad una "ricetta" immutabile e liberamente utilizzabile per la creazione di Actors.

Avendo esaminato come si implementa un Actor, si definiscono di seguito le sue componenti precedentemente elencate.

Stato

Un oggetto Actor in genere contiene alcune variabili che riflettono possibili stati in cui può trovarsi. Un classico esempio di "stato" potrebbe essere un contatore, un set di richieste, ecc.

Questi dati sono ciò che rende unico uno specifico Actor, e devono quindi essere protetti da corruzione da parte di altri Actors. La buona notizia è che Akka garantisce la sicurezza all'interno di un Actor e lo rende completamente schermato dal resto del sistema.

Ciò significa che si può semplicemente scrivere il codice del proprio Actor, senza preoccuparsi degli altri all'interno del sistema.

Poiché lo stato interno ad un Actor è fondamentale per permettergli di eseguire le operazioni correttamente, uno stato incoerente causa un errore fatale che può propagarsi in tutto il sistema. Così, quando l'Actor risulta possedere uno *stato corrotto* viene riavviato dal suo supervisore (il padre) che riporta lo stato come in occasione della prima creazione. Questo per consentire la capacità di auto-guarigione del sistema.

Comportamento

Ogni volta che un messaggio viene elaborato, ad esso viene associato un **comportamento** che l'Actor deve tenere.

Per **comportamento** si intende una funzione che specifica le azioni da intraprendere in risposta ad un messaggio M ricevuto in un dato istante T. Il comportamento può cambiare nel corso del tempo in seguito ad una variazione di stato oppure in seguito ad un particolare messaggio ricevuto.

Una qualsiasi richiesta che un dato Actor A è in grado di gestire viene definita *forward*, mentre si indica con il termine di *otherwise* una richiesta sconosciuta.

In caso di riavvio dell'Actor, oltre al ripristino dello stato iniziale, verrà reimpostato anche il suo comportamento.

Mailbox

Lo scopo di un Actor è l'elaborazione dei messaggi a lui inviati da altri Actors interni o meno all'applicazione (o dal sistema).

La componente che collega mittente e destinatario è la Mailbox ed ogni Actor è proprietario di esattamente una. Può essere considerata come una sorta di coda che effettua un accodamento ordinato rispetto alla data di invio dei messaggi.

Ci sono diverse implementazioni di una Mailbox: per default è una FIFO (l'ordine dei messaggi elaborati dall'attore corrisponde all'ordine in cui sono stati accodati), ma per alcune applicazioni potrebbe essere necessario assegnare priorità ad alcuni messaggi rispetto ad altri.

In questo caso, una Mailbox con algoritmo di priorità accoderà i messaggi basandosi sulla priorità a loro assegnata.

Actors Figli

Ogni Actor è potenzialmente un *supervisore*: se crea Actors figli per la delega di compiti secondari ne diventa automaticamente supervisore. La lista dei figli viene mantenuta all'interno del contesto dell'Actor e solo lui ha accesso alla propria lista.

Le modifiche alla lista sono effettuate con la **creazione** di un figlio:

```
context.actorOf (...)
```

oppure mediante l'**arresto**:

```
context.stop (mySon)
```

La creazione o l'eliminazione di un figlio **non vengono effettuate nell'immediato**, viene solo creato un riferimento all'azione di creazione/eliminazione in attesa del momento più opportuno per effettuare l'operazione vera e propria. Tale operazione avviene *"dietro le quinte"* in modo **asincrono**.

Più semplicemente, **un Actor A**, che in fase di elaborazione riceve un comando di creazione/eliminazione di un Actor figlio F, **non può interrompere** il proprio compito per eseguire il comando; d'altro canto **F deve essere eliminato/creato**.

Siccome solo A è supervisore di F e, quindi, l'unico a poter delegare sottoprocessi, in caso di creazione si manterrà un riferimento riguardo l'esistenza di F; solo quando A si troverà in stato di attesa (per esempio attesa di I/O) oppure avrà terminato il suo compito F sarà effettivamente creato.

Per quanto riguarda l'eliminazione il principio è il medesimo; F verrà considerato come "eliminato" (anche se di fatto esiste ancora) quindi non potrà più operare in nessun modo e quando A sarà disponibile eseguirà l'eliminazione vera e propria.

Supervisione

Componente finale dell'Actor è la sua strategia per la gestione dei guasti da parte dei suoi figli. La gestione di un guasto viene effettuata in modo trasparente, applicando una delle strategie di cui l'Actor dispone.

Esse vanno implementate all'interno del proprio Actor in modo che possa gestire all'occorrenza gli eventuali errori generati dai suoi figli. Poiché questa strategia è fondamentale, non può essere cambiata una volta che l'Actor è stato creato.

Una volta che un Actor viene terminato o eliminato, libera tutti i messaggi rimanenti dalla sua Mailbox vengono inviati nella "Dead Letters Mailbox" del sistema. Questa tipologia di messaggi viene chiamata Dead Letter.

Actor Reference (Riferimento ad un Actor)

Si riprende ora il concetto di ActorRef, oggetto che Akka mette a disposizione, per descriverlo nel dettaglio. Un **ActorRef** è un oggetto il cui scopo principale è quello di "clonare" il comportamento di un Actor, quindi di effettuare l'invio di messaggi per l'Actor che rappresenta.

Ogni Actor ha accesso al proprio riferimento attraverso il campo di *self*; questo riferimento viene incluso all'interno dei messaggi che l'Actor invia. Un Actor che riceve un messaggio M può entrare a conoscenza del *self* dell'Actor mittente utilizzando il comando *M.sender*.

Ci sono due tipi di ActorRef che sono supportati a seconda della configurazione del sistema:

- **Riferimenti locali** – ActorRefs utilizzati nei sistemi di Actors che non sono configurati per supportare le funzioni di rete;
- **Riferimenti remoti** - rappresentano gli Actors che sono raggiungibili utilizzando comunicazione remota, vale a dire l'invio e la ricezione di messaggi verso/da JVM remote all'interno di altri host.

Ci sono diversi tipi speciali di riferimenti ad attori che si comportano come riferimenti attori locali a tutti gli effetti:

- **PromiseActorRef** – classe che fa riferimento ad una *Promise,* una richiesta che non viene effettuata nell'immediato ma sarà completata in futuro da un Actor. Questa classe *akka.pattern.ask* crea questo tipo di riferimento ad attore;
- **DeadLetterActorRef** – è l'implementazione predefinita del servizio Dead Letters che indirizza tutti i messaggi, presenti nella Mailbox di un Actor dopo la sua terminazione, nella Mailbox Dead Letters;
- **EmptyLocalActorRef** – rappresenta un percorso (ActorPath) inesistente di un Actor disperso o terminato; è equivalente a un *DeadLetterActorRef*, ma mantiene il suo percorso in modo che Akka possa inoltrare tale percorso nella rete e confrontarlo con quello di altri ActorRefs esistenti tentando di recuperare l'Actor.

Actor Path (Percorso di un Actor)

Nella parte precedente si è accennato il concetto di **ActorPath**, questa parte lo descrive nello specifico. Poiché gli Actors seguono una gerarchia ferrea, esiste una sequenza unica di nomi detta ActorPath che permette di identificare un Actor e tutti i suoi "padri" fino all'ActorSystem.

Questa sequenza può essere paragonata alla directory di un file all'interno del file system. Come nei file system, esistono "link simbolici" che permettono di raggiungere un Actor utilizzando più di un percorso.

Un ActorPath è costituito da un'ancora, che identifica l'ActorSystem che lo ha creato, seguita dalla concatenazione degli elementi di percorso; tali elementi sono i nomi degli Actors padri.

Occorre prestare particolare attenzione alla differenza che c'è tra l'ActorRef e l'ActorPath:

- un ActorRef indica un singolo Actor ed il ciclo di vita di un ActorRef corrisponde al ciclo di vita dell'Actor;
- un ActorPath rappresenta un percorso ad un Actor che potrebbe essere ancora in vita oppure potrebbe essere stato eliminato. Il percorso non ha un ciclo di vita e perciò farà sempre riferimento ad oggetto indipendentemente dal fatto che sia stato o meno eliminato.

È possibile creare un ActorPath senza creare un ActorRef, ma non è possibile creare un ActorRef senza un Actor da farvi corrispondere e quindi un ActorPath.

Messaggi

Un messaggio è un qualsiasi oggetto, variabile, classe o case class Scala che l'Actor è in grado di riconoscere e processare. In fasi di implementazione del metodo receive di un Actor occorre definire tutti i messaggi che quest'ultimo conosce.

Esistono due metodi per l'invio di un messaggio e la sintassi è la seguente:

```
myActor ! message
```

Il comando "!" detto anche **Tell** permette di inviare un messaggio a myActor senza stoppare il processo mittente.

```
myActor ? message
```

Il comando "?" detto **Ask** invia un messaggio a myActor e ferma il processo mittente che resta in attesa di una specifica risposta; questo comando ritornerà un oggetto di tipo Future (vedi par. 3.2.3 – "I Futures").

In supporto al comando Ask, viene impostato un tempo di **Timeout** in cui il processo mittente attende una risposta da myActor; allo scadere del Timeout o in caso di risposta negativa (riceverà un eccezione da gestire) il processo mittente riprende l'esecuzione interrotta dall'invio del messaggio.

Nell'esempio sottostante si elencano i passi per la creazione di un Timeout, per l'invio di un Ask e l'attesa della risposta:

```
implicit val timeout = new Timeout(Duration.create(
                                        200,
                                        TimeUnit.SECONDS
                                        )
                                    )
val ask = (y ? new Message())

try {
  var result = Await.result(ask,
                            timeout.duration
                            ).asInstanceOf[String]
} catch {
 case t: Exception => //Nothing
}
```

Timeout è una classe all'interno del package *akka.util* a cui viene passato una durata di tempo. **Duration** è un oggetto di tipo *Singleton* del package *scala.concurrent.duration* che mette a disposizione il metodo *create* per definire un intero è un unità di misura di tempo (in questo caso si utilizza il supporto della **Java enumeration TimeUnit** del package *java.util.concurrent*). Si costruisce la variabile "ask" che rappresenta la richiesta.

Infine si utilizza **Await.result**, racchiuso in un *try-catch*, per attendere una risposta che, come precedentemente accennato, potrebbe avere esito negativo quindi lasciare un eccezione da gestire.

Come precedentemente spiegato, esistono diversi metodi per cerare un messaggio, ecco alcuni esempi per definirne uno:

```
case class Message(val text: String, val mType: MessageType.Value)

case class Add(val elem: ActorRef)

case class Ack()

val DONE = "Done"

class Remove (elem: ActorRef){
    val target = elem
}
```

Non esiste una prassi specifica per definire un messaggio, Akka consiglia di creare messaggi semplici, con nomi logici ed intuitivi, con pochi attributi. Nello specifico si predilige l'uso delle *case class*. Occorre **prestare particolare attenzione** in fase di definizione del metodo *receive,* per esempio:

```
def receive = {

   case m : Message => m.mType match {
                         case MessageType.SEND =>send
                                                 sender ! DONE
                         case MessageType.READ =>read
                                                 sender ! DONE
                         case _ => mailBox.enqueue(m)
                                   sender ! DONE
                       }

   case add : Add   => sender ! DONE

   case rem : Remove=> neighbours -= rem.elem
                       sender ! DONE
   case _ => //Do nothing
}
```

Questo esempio mostra l'implementazione del metodo *receive* di uno specifico Actor e tutti i comportamenti che deve seguire dopo il ricevimento di un messaggio; *read* e *send* sono due metodi privati all'interno dell'Actor e *neighbours* è un set che rappresenta una "*Rubrica di indirizzi di altri Actors*".

Si conclude il paragrafo sottolineando come **la consegna dei messaggi non sia sempre garantita** ed inoltre siccome in un sistema sono presenti molti Actors **l'ordine di invio**

non è sempre uguale a quello di ricezione dei messaggi. Con questo si conclude il paragrafo sugli Actors, si è ora in grado di programmare un sistema concorrente basato su queste entità.

2.2.2 Gli Agents

Gli **Agents (Agenti)** sono oggetti che Akka mette a disposizione per permettere il *cambiamento asincrono dello stato* di una locazione di memoria (in questo caso di un Actor). Essi sono vincolati ad un unico percorso di archiviazione per tutta la vita, e permettono la mutazione (cambio di stato), in conseguenza di un evento, esclusivamente della locazione a cui sono legati.

Le azioni di **update (aggiornamento)** sono funzioni che vengono applicate in modo asincrono allo stato dell'Agent e il cui valore di ritorno diventa nuovo stato dell'Agent stesso.

Mentre gli aggiornamenti degli Agents sono asincroni, lo stato è sempre disponibile per la lettura che può essere effettuata da qualsiasi Thread senza l'invio di alcun messaggio (usando i metodi *get* o *apply*).

Gli Agents sono reattivi, le azioni di **update** degli Agents vengono inserite in un pool di Thread chiamato **ExecutionContext**. In un dato istante, può essere eseguita al massimo una azione di *send* per Agent. Le azioni spedite verso un Agent da parte di un Thread verranno processate nell'ordine d'invio.

Gli Agents vengono creati invocando il comando *Agent(value)*, passando un valore iniziale dell'Agent e fornendo un ExecutionContext implicito da utilizzare.

Ecco un esempio per la creazione di un Agent:

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
val agent = Agent(5)
```

Gli Agents possono essere de-referenziati (ovvero è possibile ottenere il valore di un Agent) invocando questo metodo:

```
val value = agent() /* agent è la val definita nell'esempio
precedente */
```

L’esempio di cui sopra restituisce lo stesso risultato dell’operazione:

```
val value = agent.get
```

Si noti che, mentre gli **aggiornamenti (update)** di un Agent sono asincroni, l’operazione di lettura dello stato di un Agent è sincrona.

Si può effettuare l’operazione di update di un Agent **inviando una funzione** che trasforma il valore corrente oppure **inviando** semplicemente **un nuovo valore**. L'Agent imposterà il nuovo valore o la funzione automaticamente ed in modo asincrono.

Non c'è alcuna garanzia di quando sarà applicato l'aggiornamento, l’unica garanzia che Akka fornisce è che l’aggiornamento sarà effettuato (prima o poi). Nell’esempio sottostante viene spiegato come effettuare l’update di un Agent, sia utilizzando un semplice valore sia inviando una funzione:

```
// Invia un valore
agent send 7

// Invia una funzione
agent send (_ + 1)
agent send (_ * 2)
```

Ricapitolando gli Agents sono oggetti che una volta creati si legano ad una locazione di memoria o ad un Actor fino all’esistenza di tale oggetto, ne permettono quindi il cambiamento di stato. Si ricorda che lo stato di un Actor è fondamentale per poter mutare il comportamento di quest’ultimo in base allo stato che ad esso viene associato.

Esiste una variante dinamica che permette di cambiare stato senza l’utilizzo degli Agents. Si rammenta che un Actor ha un metodo *receive* che gestisce i messaggi in coda e a ciascun messaggio associa un processo specifico. Si può implementare tale metodo suddividendolo in **sottometodi** per ciascuno stato previsto dell’Actor. Per esempio:

*Si suppone si voglia definire un Actor che possa assumere due stati, **activated e stopped**. Se l’Actor è in stato “activated” può operare normalmente, invece se è in stato “stopped” dovrà stampare a video una frase in cui si scusa con l’utente e gli notifica il suo stato.*

Si implementa tale Actor in questo modo:

```
class MyActor extend Actor{

  import context._

  def receive = {
    case StopMessage => become(stopped)
    case ActiveMessage => become(activated)
    case_ => println("Messaggio sconosciuto")
  }

  def activated: Receive = {
    case m:NormalMessage => /* comportamento specifico */
    case StopMessage => println("Stato Modificato
                                 in Stopped")
                        become(stopped)
    case_ => println("Messaggio sconosciuto")
  }

  def stopped: Receive = {
    case m:NormalMessage => println("Scusa!! Ma sono Stopped,
                                     non posso procedere")
    case ActiveMessage => println("Stato Modificato
                                   in Stopped")
                          become(activated)
    case_ => println("Messaggio sconosciuto")
  }
}
```

Il concetto dovrebbe essere abbastanza chiaro, si può comunque spiegare l'esempio visto brevemente:

1. Innanzitutto si importa il *context* (package che contiene il contesto di tutte le funzionalità degli Actors);
2. Si definisce il metodo *receive* che verrà richiamato solamente per il primo messaggio dopo la creazione di MyActor (il messaggio per impostare lo stato iniziale);
3. Si implementano infine N metodi per gli N stati previsti per MyActor (in questo caso due).

È importante assegnare ai metodi *stopped* e *activated* il contesto **Receive**, in questo modo i messaggi verranno incanalati attraverso uno dei due metodi a seconda dello stato di MyActor. Per impostare l'Actor su un determinato stato si utilizza *become*(*stato*); è molto importante che la funzione che gestisce lo *stato* abbia nome uguale.

Questo procedimento per effettuare il cambiamento di stato è decisamente intuitivo nonché pratico e performante.

Si conclude il paragrafo sugli Agents e sul cambiamento di stato di un Actor.

3.2.3 I Futures

In questo paragrafo si riprende il concetto di Future precedentemente spiegato nel capitolo 2 – Finagle e nello specifico nel paragrafo 2.2.1 – "I Futures".

Si ricorda che un **Future** è una struttura di dati utilizzata per recuperare il risultato di una operazione effettuata in parallelo. Questo risultato può essere letto in modo sincrono (bloccante) o asincrono (non bloccante).

Akka utilizza i Futures come oggetti in appoggio agli Actors, questi ultimi infatti possono sfruttare i Futures per inviare messaggi asincroni. Nel paragrafo 3.2.1 – "Gli Actors" e più precisamente nel sottoparagrafo "Messaggi" sono stati analizzati entrambi i metodi per l'invio di un messaggio: **Tell** e **Ask**. In questo paragrafo si concentra la documentazione solo sul protocollo Ask. Riassumendo il comando **Ask** invia un messaggio ad un Actor e ferma il processo mittente che resta in attesa di una specifica risposta; questo comando ritornerà un oggetto di tipo Future. Si riprende in considerazione l'esempio dell'invio tramite Ask tenendo presente che per poter richiamare il comando '?' bisogna importare il package *akka.pattern*:

```
val ask = (y ? new Message())

try {
  var result = Await.result(ask,
                            timeout.duration
                            ).asInstanceOf[String]
} catch {
 case t: Exception => //Nothing
}
```

Si noti inoltre che il Future che viene restituito da un attore è un Future[Some[12]]. Questo spiega il *asInstanceOf* che viene utilizzato nell'esempio.

[12] **Some**: in Scala un oggetto di tipo Some può essere visto come l'oggetto Object in Java, ovvero potrebbe essere un oggetto qualunque.

Il blocco da parte del processo mittente è sconsigliato da Akka in quanto causerebbe problemi di prestazioni. Si preferisce effettuare calcoli contemporaneamente senza la necessità di creare un pool di Actors per la sola ragione di eseguire un calcolo in parallelo.

Esiste un metodo più facile (e più veloce):

```
import scala.concurrent.Await
import scala.concurrent.Future
import scala.concurrent.duration._

val future = Future {
   "Hello" + "World"
}

future foreach println
```

Nel codice di cui sopra il blocco passato al Future sarà eseguito indipendentemente dagli Actor presenti nel sistema. A differenza di un Future che viene restituito da un **Ask** verso un Actor, questo evita sia inutili operazioni di blocco dei processi che eseguono il comando che il sovraccarico di Actors che devono gestire tale richiesta.

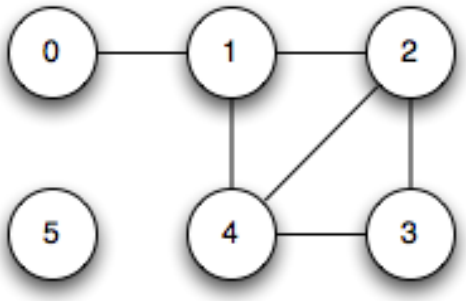
Si conclude la parte di documentazione, ora si hanno gli strumenti necessari per implementare un proprio modello basato sugli Actors.

3.3 Elaborato: Neighborhood

3.3.1 Analisi del problema

Si immagini di trovarsi in un quartiere di una città dove l'unico modo per comunicare tra vicini è mediante lo scambio di messaggi di posta elettronica. La popolazione è quindi composta da N persone (attori), ognuno ha un certo numero, minore di N, di conoscenti (ovvero persone di cui sono a conoscenza dell'indirizzo). Ogni persona (attore) ha la sua mailbox e una rubrica che contiene gli indirizzi dei suoi conoscenti.

Per semplicità si può pensare al quartiere come un ***grafo non orientato G*** *dove i nodi sono gli abitanti mentre gli archi sono rappresentati dagli indirizzi dei contatti di ciascun vicino presente nella rubrica di un qualche abitante. Gli archi sono scelti casualmente allo scopo di simulare il comportamento di un vero quartiere; potrebbero esserci abitanti che non ha nessun contatto dei propri vicini in rubrica.*



A intervalli di tempo, un qualche attore scelto a caso "si sveglia", legge i messaggi ricevuti fino ad allora, li elabora, e risponde ai mittenti ed infine a sua volta invia un messaggio agli altri attori di sua conoscenza.

3.3.2 Progettazione

L'entità principale è l'abitante, ovvero colui che conosce altri abitanti, che invia messaggi e che legge la propria posta; si crea quindi la classe **Inhabitant** un attore che rappresenti tutti i comportamenti *dell'abitante*. Si deduce quindi che questa classe estenderà da Actor e quindi l'implementazione del metodo *receive* dovrà gestire tutti i comportamenti dell'abitante del quartiere e, siccome l'unico modo per comunicare con un Actor è l'invio di messaggi, si dovranno creare messaggi adeguati ad ogni situazione.

Innanzi tutto occorre ragionare sui comportamenti dell'essere umano in un problema del genere, ovvero come si comporterebbe un uomo reale?

Per quanto riguarda le conoscenze, un rapporto di conoscenza tra due esseri umani deve essere reciproco, si potrebbe quindi pensare ad una sorta di "stretta di mano virtuale" tra due entità della classe **Inhabitant.**

Un altro comportamento tipico è quello di "mettere in rubrica" i contatti dei conoscenti (solitamente solo dei più stretti, ma in questo caso si suppone che la conoscenza con un altro abitante implichi la conoscenza del suo "indirizzo di posta").

Si possono ora immaginare i messaggi pertinenti all'applicazione:

```
case class Message( val from: ActorRef,
                    val mType: MessageType.Value)
case class Add(val elem: ActorRef)
case class AckAdd(val elem: ActorRef)
case class Remove(val elem: ActorRef)
case class PrintN()

object  MessageType extends Enumeration{
      val TEXT=Value("Text")
      val RESPONSE=Value("Response")
      val READ = Value("Read")
      val SEND = Value("Send")
}
```

Message: messaggio che permette agli abitanti di comunicare tra loro; esistono 4 sotto tipologie di messaggi:

- **Text** – Messaggio di testo;
- **Response** – Messaggio di risposta ad un messaggio di tipo Text;
- **Read** – Messaggio inviato dal sistema che obbliga l'attore a leggere dalla propria Mailbox;
- **Send** – Messaggio inviato dal sistema che obbliga l'attore ad inviare un messaggio di tipo Text al riferimento presente all'interno del messaggio.

Add (B): comando inviato dal sistema ad un attore A; questo obbliga A e l'ActorRef (B), passato per messaggio, a scambiarsi i propri contatti e "fare conoscenza". L'attore invia a B un "AckAdd" ed attende da B un messaggio di conferma "DONE"; una volta ricevuto, a sua volta aggiunge B nella sua Rubrica ed invia al sistema un messaggio di "DONE";

AckAdd: uguale ad Add inviato da un abitante A ad un abitante B ed obbliga B (destinatario del messaggio) ad aggiungere A nella propria Rubrica, una volta effettuata l'operazione B manda un messaggio di tipo DONE ad A;

Remove (B): messaggio inviato dal sistema che obbliga l'attore a rimuovere B dalla propria rubrica;

PrintN: messaggio inviato dal sistema che comunica l'attore di stampare a video la propria Rubrica;

DONE: Stringa di conferma che viene inviata o al mittente o al sistema in risposta ad un comando di Add, AckAdd o Remove.

Ciascun Inhabitant avrà la propria **Mailbox** (da non confondere con la Mailbox di Akka, in questo elaborato ne verrà creata una dal principio), una *Scala queue* illimitata che conterrà i messaggi ricevuti non ancora letti. Come una qualsiasi casella di posta elettronica la Mailbox verrà letta periodicamente e durante il lasso di tempo tra una visualizzazione e l'altra accumulerà messaggi. La periodicità non è costante per un essere umano (o almeno per la maggior parte) quindi verranno generati step casuali in cui l'abitante si "sveglia" e leggerà la **Mailbox**.

Per semplicità la Mailbox contiene solo i messaggi non ancora letti, quindi appena un messaggio viene letto viene fatto uscire dalla coda e quindi eliminato.

Quando un Inhabitant vuole leggere i propri messaggi eseguirà un metodo di lettura (*read)*. In fase di lettura, la prima operazione che si effettuerà è il comando *mailBox.dequeueAll* che, come un foreach, legge tutti i messaggi presenti in coda, li elabora ed infine svuota la coda.

Si può quindi implementare la classe prima importando i package necessari:

```
import scala.collection.mutable.{HashSet,Queue}
import akka.util.Timeout
import java.util.concurrent.TimeUnit
import scala.concurrent.duration.Duration
import akka.actor._
import akka.pattern._
import scala.concurrent.Await
```

Per poi passare all'implementazione vera e propria:

```
object  MessageType extends Enumeration {
    val TEXT=Value("Text")
    val RESPONSE=Value("Response")
    val READ = Value("Read")
    val SEND = Value("Send")
}

case class Message (val from : ActorRef,
                    val mType : MessageType.Value)

case class Add (val elem : ActorRef)

case class AckAdd (val elem : ActorRef)

case class Remove (val elem : ActorRef)

case class PrintN ( )
```

Si dichiarano i messaggi già descritti dettagliatamente in precedenza e di seguito si dichiara il codice della classe Inhabitant con i metodi *receive, read* e *send:*

```
class Inhabitant extends Actor {


  val DONE = "Done" /* è un Ack che permette il
                     * SystemActor di proseguire */

  val actorName = self.path.name

  val neighbours = new HashSet[ActorRef] //La "Rubrica"

  val mailBox = new Queue[Message] //La Mailbox
```

Si dichiarano alcune variabili come il nome dell'Actor da allegare ai messaggi, un set di riferimenti ad attore che rappresenta i contatti in rubrica da parte dello specifico Actor ed una coda mutabile di messaggi rappresentante la mailbox dell'abitante.

Di seguito il metodo receive dell'Inhabitant che racchiude il comportamento dell'Actor:

```
def receive = {

   case m: Message => m.mType match {
                        case MessageType.SEND => send
                                                 sender ! DONE
                        case MessageType.READ => read
                                                 sender ! DONE
                        case _ => mailBox.enqueue(m)
                                  sender ! DONE
                       }
   case add: Add => if(!neighbours.contains(add.elem)) {
                        println(actorName+" meet "
                                +add.elem.path.name)
                        neighbours += add.elem
                        implicit val timeout = new Timeout(
                        Duration.create(50,TimeUnit.SECONDS)
                        )

                        val future=add.elem ? new AckAdd(self)
                        try {
                          val result = Await.result(
                                       future,
                                       timeout.duration
                                       ).asInstanceOf[String]
                          sender ! result
                        } catch {
                          case t: Exception =>
                          println("I can't wait anymore time")
                        }
                    } else {
                        sender ! DONE
                    }
   case add: AckAdd  => if(!neighbours.contains(add.elem)){
                           println(actorName+ " accept "
                                   +add.elem.path.name)
                           neighbours += add.elem
                        }
                        sender ! DONE
   case rem: Remove  => neighbours -= rem.elem
                        sender ! DONE
   case p: PrintN    => toStringNeighbourhood
                        sender ! DONE
   case _ =>

}
```

La funzione *read* viene richiamata per leggere i messaggi all'interno della coda (Mailbox). Come spiegato in precedenza si suppone che la mailbox contenga solo i messaggi non ancora letti e che, una volta incominciata l'operazione di lettura, non si termini fino al completo svuotamento della coda. Ecco il codice del metodo *read* della classe **Inhabitant**:

```
def read = {
  val writedTo = new HashSet[ActorRef]
  /* traccia delle "persone" alla quale è già stata inviata
   * una risposta */
  println(actorName+" is reading from his mailbox...")
  mailBox.dequeueAll { message =>
                        if(neighbours.contains(message.from)) {
                           message.mType match{
                              case MessageType.TEXT =>
                                   println(actorName+
                                   " find a message from "
                                   +message.from.path.name+
                                   " and reply to him"
                                   )
                                   message.from ! new Message(
                                   self, MessageType.RESPONSE
                                   )

                              case MessageType.RESPONSE =>
                                   println(actorName+
                                   " find a RESPONSE from "
                                   +message.from.path.name
                                   )
                           }

                           writedTo += message.from
                          /* inserisco il destinatario nel set
                           * delle persone a cui ho scritto */

                        } else {
                        /* caso rarissimo dovuto a mal gestione
                         * della concorrenza, quasi impossibile
                         * che si verifichi, ma comunque da
                         * gestire */
                           println(actorName+
                                   " don't knows who is "
                                   +message.from.path.name
                                   )
                        }
                        true
  } //end of dequeue method
```

L'attore in questione può non aver scritto a tutti con la precedente istruzione. Allora al metodo *read* si aggiunge un istruzione finale che filtri in un nuovo set l'insieme neighbours (vicini) all'interno della rubrica che non hanno ancora ricevuto un messaggio e si provvede ad inviarne uno anche a loro. Si utilizza il metodo *filterNot* del set **neighbours** utilizzando come condizione *writedTo.contains* (in questo modo si filtrano solamente i vicini a cui non è stato inviato alcun messaggio). Si completa quindi il metodo read nel seguente modo:

```
    val notWritedYet =
          neighbours.filterNot { x => writedTo.contains(x)}

    notWritedYet.foreach { x => println(actorName+
                                " also send a message to "
                                +x.path.name
                                )

                                x ! new Message(self,
                                                MessageType.TEXT
                                                )
                           }
    println(actorName+" has finished\n")

  }
```

Ora si definisce il metodo send che consiste nell'invio di un messaggio di tipo Text a tutti i vicini presenti nel set *neighbours*. Il metodo toStringNeigborhood effettua la stampa a video di tutti i vicini presenti nel set.

```
  def send = {
      neighbours.foreach { x => x ! new Message(self,
                                                MessageType.TEXT
                                                )
                           }
  }

  def toStringNeighborhood = {
      println(actorName.toUpperCase()+ " NEIGHBOURHOOD")
      neighbours.foreach { N => println(" ° "+N.path.name) }
      println("\n")
  }

}
```

La classe **Main** gestisce gli scambi dei messaggi tra attori, inoltre sfrutta il **Factory Method** per produrre un qualsiasi numero di abitanti all'interno del quartiere.

Inizialmente si crea un **ActorSystem** ovvero, come già spiegato, una struttura capace di allocare in automatico da 1 a N Thread in corrispondenza della creazione di un attore ed associa un Thread a ciascun attore.

```
val system = ActorSystem("System")
```

Successivamente si procede con la creazione di N attori. Per ciascun attore viene richiamato il metodo **actorOf** dell'ActorSystem; questo tipo di procedura è simile al pattern Factory Method di Java.

```
for(i<-1 to N){
  val actorName : String = "ActorN"+i
  val actor = system.actorOf(Props[Inhabitant],
                             name = actorName)
  actors += actor
}
```

Props è una "configuration class" che specifica quale configurazione assegnare all'attore che il sistema sta creando, in questo caso il profilo utilizzato da Props è Inhabitant. Una volta creati gli abitanti, la classe Main provvederà a creare gli archi in modo casuale per formare il grafo (ovvero fa "conoscere" i vari abitanti tra di loro).

```
actors.foreach { x => actors.foreach {
                  y => if(rand.nextInt(RANDOM_LIMIT) == 1 &&
                          x.path.name.compareTo(y.path.name)!=0
                          ) {

                              val future = (y ? new Add(x))
                              try {
                                 var result = Await.result(
                                              future,
                                              timeout.duration
                                          ).asInstanceOf[String]
                              } catch {
                                 case t: Exception => //Nothing
                              }//end try catch
                       }//end if
               }
          }
```

Si può quindi avviare lo scambio di messaggi tra vicini come segue:

```
actors.foreach { x =>if(rand.nextInt(RANDOM_LIMIT)==1){
                                val future = (x ? new Message(x,
                                              MessageType.SEND))
                                try {
                                   var result =
                                       Await.result(future,
                                       timeout.duration
                                       ).asInstanceOf[String]
                                } catch {
                                   case t: Exception => //Nothing
                                }
                            }
    }
```

L'ultima operazione effettuata è inviare a ciascun attore un messaggio **Read**. Gli attori quindi leggeranno i messaggi presenti nella propria Mailbox e risponderanno. Si noti il comando *Thread.sleep* che simula un abitante che si "sveglia" ed inizia la lettura:

```
actors.foreach { x => Thread.sleep(
                        rand.nextInt(SLEEP_LIMIT).toLong
                        )
                        val future = (x ? new Message(x,
                                          MessageType.READ
                                          )
                                     )
                        try {
                          var result = Await.result(future,
                                       timeout.duration
                                       ).asInstanceOf[String]
                        } catch {
                          case t: Exception => //Nothing
                        }
                }
```

Quest'ultima operazione viene ripetuta tre volte per permettere alle varie Mailbox di essere lette più volte ed analizzare come ogni attore reagisce alle differenti tipologie di messaggi ricevuti. Alla fine dell'esecuzione alcune Mailbox potrebbero contenere ancora messaggi, questo perché l'applicazione nel suo totale gestisce sia creazione degli archi che scambio di messaggi in modo casuale; in caso di N (numero di attori) troppo basso, qualche abitante potrebbe non avere nessun contatto in rubrica e non ricevere mai messaggi.

L'applicazione utilizza sia metodi di tipo Tell ('!') per quanto riguarda l'invio di messaggi di tipo *Message.TEXT*, *Message.RESPONSE*, *PrintN* e *DONE*, sia metodi di invio di tipo Ask ('?') per quanto riguarda l'invio di messaggi di tipo *Message.SEND*, *Message.READ*, *Add*, *AckAdd* e *Remove*,.

Per questi ultimi va quindi definito un **Timeout** che ciascun attore, compreso il sistema stesso deve attendere dopo aver inviato un messaggio.

```
implicit val timeout = new Timeout(Duration.create(200,
                                        TimeUnit.SECONDS))
val future = (x ? new Message(x,MessageType.SEND))

try {
   var result = Await.result(future,
                             timeout.duration
                             ).asInstanceOf[String]
} catch {
    case t: Exception => //Nothing
}
```

Ai fini di una comprensione completa del codice si descrivono brevemente alcuni oggetti incontrati:

RANDOM_LIMIT è una costante dichiarata utilizzando la sintassi:

```
val RANDOM_LIMIT : Int = //value
```

Con lo scopo di assegnare un limite per il calcolo della casualità d'invio dei messaggi e della creazione degli archi, aumentando il valore si abbassa la probabilità e viceversa. **SLEEP_LIMIT** è anch'essa una costante con lo scopo di dichiarare un limite massimo in millisecondi di tempo che un Inhabitant può "dormire" prima di effettuare l'operazione di lettura; aumentando il valore si rallenterà l'intera applicazione. La variabile *rand* istanza la classe **Random** del package *scala.util* che permette di generare un numero intero casuale ed, infine, *actors* è un semplice **mutable HashSet** di **ActorRef** per contenere tutti i riferimenti ad attore creati **dall'ActorSystem**.

Si conclude il percorso notando che le dimensioni di un sistema Akka sono direttamente proporzionali al numero di attori coinvolti, inoltre un sistema di adeguate dimensioni è decisamente più performante di un sistema di dimensioni ridotte, questo grazie al modello Actor che permette una suddivisione distinta dei compiti e un'indipendenza tra entità.

Capitolo 4 - Apache Kafka

di Lorenzo Vernocchi

Tecnologie per la Costruzione di Piattaforme Distribuite
basate sul Linguaggio di Programmazione Scala

4.1 Introduzione

Kafka è un sistema distribuito di messaggistica con un design unico, in parte scritto in Scala, costruito ed utilizzato da Linkedin. Kafka permette la gestione di centinaia di megabyte di traffico in lettura e scrittura al secondo da parte migliaia di Client.

Cominciamo con qualche definizione di base:

- Kafka raggruppa i vari feed dei messaggi in categorie dette *topics (argomenti)*;
- Sono detti *Producers (produttori)* i processi che pubblicano messaggi in un topic;
- Sono detti *Consumers (consumatori)* i processi che sottoscrivono i topics ed elaborano i feed dei messaggi presenti nei vari topics;
- Kafka è gestito come un cluster costituito da uno o più server ognuno dei quali viene chiamato *Broker*.

Nel livello più alto i produttori inviano messaggi che passano attraverso il cluster Kafka; a sua volta il cluster li "serve" ai consumatori.

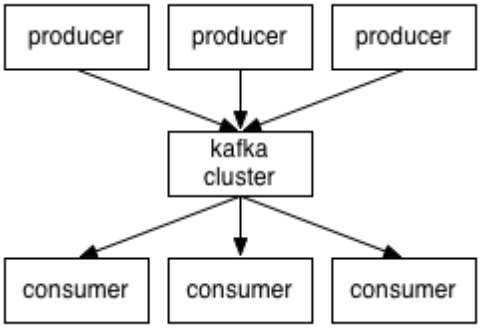


Figura n.6
Sistema Kafka

La comunicazione tra i Client ed i Server avviene tramite il protocollo TCP.

Un *topic* è un insieme di messaggi della stessa categoria o dello stesso feed; per ogni topic il cluster Kafka mantiene un **registro partizionato**. Ogni partizione è una sequenza ordinata ed immutabile di messaggi che vengono aggiunti in continuazione all'interno del registro.

Ad ogni messaggio all'interno della partizione viene assegnato un identificativo numerico, sequenziale e progressivo chiamato **offset** che identifica univocamente ogni messaggio all'interno della partizione.

Il cluster conserva tutti i messaggi pubblicati anche se questi non sono ancora stati "consumati", al registro è assegnato un **tempo configurabile di conservazione** dei messaggi (che chiamiamo *T*). Il messaggio avrà quindi *T* tempo per essere "consumato" altrimenti, allo scadere di *T*, il cluster, semplicemente, scarterà il messaggio con lo scopo di liberare spazio.

Generalmente un cluster riesce a gestire registri di topic contenenti grandi quantità di dati senza problemi, questo perché l'unico dato mantenuto per Consumer è la sua posizione di lettura all'interno del registro; la posizione di lettura è detta "offset" del Consumer. Questo valore è sotto il controllo del Consumer che lo incrementa in modo lineare rispetto **all'offset dei messaggi** che legge (consuma); di fatto però la posizione ("offset" del Consumer) è controllata direttamente dal Consumer che può quindi leggere i messaggi in modo ordinato a partire dalla posizione che vuole. Per esempio un Consumer può liberamente resettare il proprio offset e ricominciare da capo la lettura.

Si può notare come i Consumer siano "a buon mercato" in termini di consumo di memoria, questo permette al cluster di gestire un grande numero di Consumer "contemporaneamente".

Riprendendo il discorso delle **partizioni**, possiamo individuare diversi scopi di utilizzo di queste ultime:

- Permettono ad un topic di adattare le proprie dimensioni per poter essere mantenuto all'interno di un singolo server;
- Un topic più "popolare" può avere più partizioni con lo scopo di gestire grandi quantità di dati rispetto ad un altro topic con tasso di consumo basso;

- Permettono inoltre una sorta di parallelismo (un Consumer può leggere da una partizione di un topic mentre un altro può leggere da un'altra partizione dello stesso).

Le partizioni di un registro vengono spartite e distribuite tra i vari server all'interno del cluster Kafka, ciascun server gestisce i dati all'interno delle partizioni e le *richieste di consumo*. La stessa partizione può essere salvata su più server diversi per mantenere un livello di *fault tolerance*. Il numero di Server su cui viene salvata la stessa partizione si chiama **fattore di replica.**

Ogni partizione ha un server che agisce come un *leader* e zero o più server detti *followers*.
Il leader gestisce tutte le richieste di lettura/scrittura sulla partizione mentre i followers replicano il leader passivamente a scopo appunto di *fault tolerance*. Se il leader **fallisce**, uno dei followers diventerà automaticamente il nuovo leader. Ovviamente un server può gestire più partizioni di topic differenti (o anche dello stesso topic) quindi ciascun server potrà essere leader di alcune partizioni e follower delle restanti.

I **Producers** o produttori pubblicano i messaggi (i dati) all'interno di un topic. Il Producer è responsabile di scegliere in quale partizione del registro del topic inserire un proprio messaggio. Ogni Producer sceglie il proprio algoritmo di assegnamento (per esempio un semplice round robin).

I **Consumers** o consumatori leggono (consumano) i dati presenti all'interno del topic.

La messaggistica prevede due di tipi di modelli:

- **Queuing (Coda)** – un pool di Consumers può leggere dal Server e ciascuno può leggere i dati solamente durante il suo turno;
- **Publish - subscribe** – il messaggio viene trasmesso a tutti i Consumers.

Kafka offre una sola implementazione dell'entità Consumer che generalizza entrambi i modelli, **il *consumer group*.**

Un Consumer *etichetta* se stesso con il nome del gruppo a cui decide di far parte e ciascun messaggio pubblicato in un topic, seguito dal gruppo, viene consegnato a ciascun Consumer presente.

Se tutti i Consumer hanno lo stesso consumer group, il sistema si reduce ad una semplice **coda** con priorità *first in first out*.

Se tutti i consumer hanno un consumer group diverso, il sistema automaticamente diventa di tipo **publish-subscribe**.

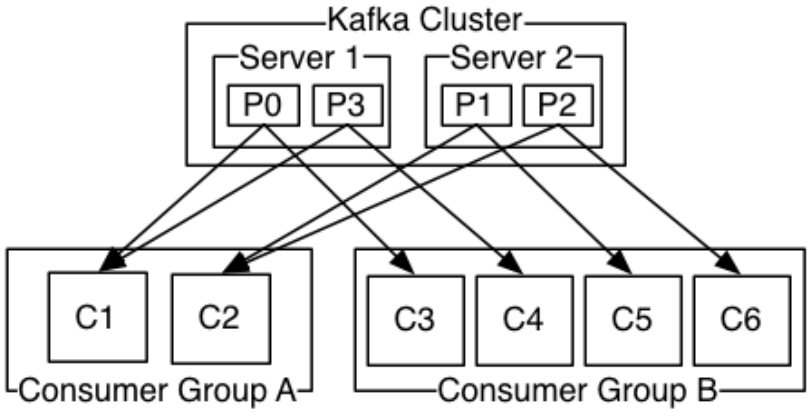


Figura n. 7 - Cluster & Consumers Groups

Nella Figura n.7 il cluster Kafka è composto da due server che mettono a disposizione quattro partizioni (P0-P3) e sono presenti due consumer groups. Il consumer group A ha due consumatori mentre B ne ha quattro.

Una coda tradizionale mantiene i messaggi in ordine sul Server e se più Client effettuano una richiesta di lettura contemporaneamente, il sistema distribuisce i messaggi nell'ordine in cui sono stati salvati. Tuttavia, nonostante il Server distribuisca i messaggi in ordine, i messaggi vengono recapitati in modo asincrono per il Client e quindi c'è il rischio di perderne l'ordine.

Molti sistemi di messaggistica bypassano il problema garantendo mutua esclusione nel consumo dei messaggi (**exclusive consuming**). In questo modo il consumo dei messaggi presenti in coda può essere effettuato da un solo processo per volta, perdendo però il **parallelismo** e la **concorrenzialità**.

"Kafka does it better"

Kafka grazie alle partizioni garantisce parallelismo, ordine e bilanciamento del carico. Ciò si ottiene assegnando le partizioni di un topic al *consumer-group*; in questo modo si garantisce che una partizione venga consumata da un solo Consumer all'interno del

gruppo. Il Consumer è l'unico "lettore" della partizione e può quindi consumare i dati in ordine. Si noti tuttavia che non ci possono essere più istanze di tipo Consumer rispetto le partizioni.

Kafka fornisce solo un ordine globale a livello di topic e non tra partizioni differenti all'interno di uno stesso. Comunque per garantire un ordine totale anche a livello di partizione può essere realizzato un topic formato da una sola partizione, anche se questo significherà solo un processo Consumer.

A livello di topic Kafka dà le seguenti garanzie:

- I messaggi inviati da un Producer ad una particolare partizione di un topic saranno aggiunti in ordine d'invio. Pertanto, se un Producer P invia prima un messaggio M1 poi un messaggio M2, M1 avrà un offset più piccolo rispetto ad M2 in questo modo M1 apparirà prima di M2 all'interno del registro;
- Un'istanza Consumer vede messaggi nell'ordine in cui vengono memorizzati nel registro;
- Un topic con fattore di replica N potrà tollerare fino a N-1 errori del Server senza perdere alcun messaggio contenuto nel registro.

Lo scopo di questa presentazione è di fornire un'infarinatura generale del sistema di Kafka; di seguito, verranno affrontati le varie componenti sopra riportate in maniera più dettagliata.

4.2 Documentazione

4.2.1 I Consumers

Come già precedentemente enunciato, in Kafka i *Consumers o consumatori* sono processi che sottoscrivono i topics ed elaborano i feed dei messaggi presenti all'interno di un determinato topic. In poche parole, leggono i dati contenuti all'interno delle partizioni di un determinato topic.

Il Consumer di Kafka funziona grazie al supporto di **fetch**, richieste passate ai *Brokers* che specificano in quale partizione il Consumer vuole leggere. Il Consumer specifica il suo "offset" nel *registro* (*log*) del topic, dopodiché inizia a consumare la parte di registro a partire dalla posizione richiesta. Per esempio, se un Consumer C volesse leggere i messaggi del topic T a partire dal messaggio M5 (supponendo un ordine crescente M1, M2, ..., Mn) inoltrerebbe una fetch a un Broker specificando 5 come suo "offset". Quindi potrà consumare i messaggi M5, M6, M7 e così via (saltando i precedenti).

Il Consumer ha quindi un controllo significativo sull'offset, inoltre può resettarlo (*rewind*) per rileggere i dati a partire da una posizione a scelta.

Una domanda che ci si deve porre è se sono i Consumers a dover **estrarre** (**pull**) i dati dai Brokers, oppure sono i Brokers a dover **passare** (**push**) i dati ai Consumers; da questo punto di vista Kafka segue un design tradizionale, utilizzato dalla maggior parte dei sistemi di messaggistica, in cui i dati vengono passati (push) al Broker dal Producer e vengono poi estratti (pull) da parte del Consumer (**sistema pull-based**).

Alcuni sistemi, come *Scribe* e *Apache Flume*, seguono un percorso molto diverso in cui ogni componente fa push dei dati al componente successivo (**sistema push-based**).

Ci sono pro e contro per entrambi i **sistemi**. L'obiettivo è generalmente permettere al Consumer di consumare alla velocità massima possibile; purtroppo in un sistema push-based c'è rischio che il Consumer venga soppresso quando il suo tasso di consumo scende al di sotto del tasso di produzione (come in un *attacco DoS*[13]). Un sistema pull-

[13] **Attacco DoS (Denial of Service)**: malfunzionamento di un Server dovuto ad un attacco informatico in cui si esauriscono deliberatamente le risorse del sistema. Lo scopo di tale attacco è quello di saturare e congestionare il Server allo scopo di negare il servizio mandandolo in crash.

based permette al Consumer semplicemente di raggiunge la velocità massima concessa dal sistema di messaggio.

Il problema enunciato in precedenza può essere evitato aggiungendo una sorta di protocollo di backoff con il quale il Consumer può indicare di essere stato soppresso, ma ottenere la velocità di trasferimento massima senza sovra-utilizzarla è comunque molto complicato.

Un altro vantaggio di un sistema pull-based è che la quantità di dati inviati al Consumer dipende dal Consumer stesso, che è in grado di aumentare/diminuire il consumo in caso si trovi o meno in stato di congestione. Un sistema push-based deve scegliere se inviare immediatamente una richiesta oppure se tentare di accumulare più dati possibili e poi inviarli successivamente senza conoscere se il Consumer sarà in grado di elaborare immediatamente. Per questi motivi si preferisce adottare un sistema di tipo pull.

Tracciamento

Tenere traccia di quello che è stato consumato è uno dei punti chiave della performance di un sistema di messaggistica. La maggior parte di questi sistemi mantiene dei metadati relativi a quali messaggi sono stati consumati, ovvero come un messaggio viene consegnato ad un Consumer.

Il *Broker* è responsabile della scrittura dei metadati relativi al consumo e può decidere di crearli immediatamente oppure di aspettare un consenso (*acknowledgement*) da parte del Consumer.

Siccome un singolo Server deve mantenere al suo interno enormi quantità di dati, dopo che un messaggio viene etichettato come consumato da parte del Broker (ovvero viene creato un metadato che specifica che il messaggio ha soddisfatto tutte le richieste di consumo), il Broker può decidere di liberare spazio ed eliminare tale messaggio.

Ottenere un accordo tra Broker e Consumer su ciò che è stato consumato non è un problema banale. Se, per esempio, il Broker registra un messaggio come consumato (**consumed**) dal momento che viene distribuito attraverso la rete ma, per un problema di timeout della richiesta o altro, il Consumer non riesce ad elaborare il messaggio, il messaggio andrà perso.

Per risolvere questo problema, molti sistemi di messaggistica aggiungono una **funzione di riconoscimento**. Significa che inizialmente i messaggi sono contrassegnati solo come *inviato*, ma non consumati (**send not consumed)**, quando vengono trasmessi attraverso la rete, dopo di che il Broker rimane in attesa un riconoscimento (**ack**) specifico da parte del Consumer per registrare il messaggio come *consumato*. Questa strategia consente di risolvere il problema della perdita di messaggi, ma ne crea di nuovi:

- Se il Consumer elabora il messaggio ma fallisce prima di poter inviare un ack, il messaggio sarà consumato due volte;
- Il secondo problema riguarda le prestazioni in quanto il Broker, dovendo registrare più stati su ogni singolo messaggio, dovrà effettuare un numero maggiore di operazioni di lettura/scrittura su ogni messaggio.

Kafka gestisce il problema in modo diverso. Il topic è suddiviso in una serie di partizioni totalmente ordinate (a livello di topic), ciascuna delle quali è consumata da un Consumer in un dato momento. Ciò significa che la posizione del Consumer in ciascuna partizione rappresenta *l'offset* del messaggio successivo da consumare. In questo modo si riesce a capire quali messaggi sono stati consumati. L'operazione di riconoscimento dello stato *consumed* può essere effettuata periodicamente.

C'è un altro vantaggio inaspettato: un Consumer può deliberatamente tornare indietro ad un vecchio offset e ri-consumare dati. Questo viola il comportamento tipico di una coda ma si rivela essere una caratteristica essenziale per molti Consumer. Ad esempio, se il codice del consumo ha un bug e viene scoperto dopo aver già consumato alcuni messaggi, il Consumer può ri-consumare tali messaggi una volta che il bug è stato risolto.

4.2.2 I Producers

Un'altra importante entità in Apache Kafka è il **Producer**: processo responsabile della *pubblicazione dei messaggi*, quindi della *creazione dei dati*, all'interno di un determinato topic. Il Producer deve decidere in quale partizione del registro del topic inserire un proprio messaggio. Per poter pubblicare i messaggi un Producer invia i dati direttamente al Broker leader della partizione interessata.

Ecco come avviene il processo di pubblicazione:

- Il Producer effettua una richiesta di pubblicazione al Cluster;
- Tutti i nodi del Cluster Kafka rispondono alla richiesta informando il Producer su quali Server sono attivi e dove i leader per le partizioni di un topic sono in un dato momento. In questo modo il Producer ottiene le informazioni necessarie per dirigere le sue richieste in modo corretto;
- Il Producer effettua la pubblicazione inviando i dati al Broker della partizione scelta;
- Il Consumer a sua volta identifica in quale partizione sono stati pubblicati messaggi ed inizia il consumo (vedi par. 1 - *"I Consumers"*).

Se per i Consumers l'obiettivo principale era quello di massimizzare il consumo, per i Producers il problema principale che ci si pone è quello di adottare un algoritmo per la pubblicazione efficiente.

Esaminiamo alcuni casi di pubblicazione:

- Come precedentemente spiegato, un Producer per pubblicare deve prima effettuare una richiesta al Cluster e quindi entrare in stato d'attesa. Per questo motivo un Producer non può pubblicare messaggi a raffica ogni volta che ne viene creato un nuovo in quanto comporterebbe uno spreco esagerato di tempo in cui il Producer si trova in stato d'attesa;
- Secondo, ci possono essere spesso messaggi che riguardano lo stesso topic o addirittura la stessa partizione, è evidente quanto sia inefficace effettuare due pubblicazioni distinte per un caso come questo.

Il **batching** è il processo che un Producer Kafka effettua quando tenta di accumulare il maggior numero di dati possibile in memoria ed effettua una pubblicazione di lotti di dati in una singola richiesta.

Questo processo può essere configurato per accumulare non più di un numero fisso di messaggi ed aspettare non più di una certa latenza fissa (per esempio 64k oppure 10 ms). Questo permette un accumulo di più byte da inviare nella stessa richiesta e la possibilità di effettuare alcune operazioni più grandi di I/O sui Server. Per poter mantenere i dati in memoria, i Producers sfruttano un **buffer configurabile**.

L'obiettivo per un Producer è quindi quello di **massimizzare il throughput,** ovvero massimizzare la propria capacità di trasmissione di dati.

4.2.3 Semantica per la consegna dei messaggi

Dopo questa spiegazione del funzionamento dei produttori e dei consumatori, proviamo ad esaminare le garanzie che Kafka fornisce tra questi.

Kafka mette a disposizioni diversi tipi di consegna di un messaggio:

- *At most once* (al massimo una volta) - i messaggi possono essere persi, ma non sono mai riconsegnati/rielaborati;
- *At least once* (almeno una volta) - i messaggi non si perdono mai, ma possono essere riconsegnati/rielaborati.
- *Exactly once* (esattamente una volta) - ogni messaggio viene recapitato/letto una sola volta.

Scomponiamo il problema della garanzia in due sottoproblemi:

- la garanzia di **durabilità** (**durability**) per la pubblicazione di un messaggio;
- la garanzia di consumo di un messaggio.

Molti sistemi di messaggistica, concorrenti di Kafka, sostengono di fornire la semantica di consegna di tipo "*exactly once*"; è possibile che la maggior parte di queste affermazioni sia fuorviante: la semantica di consegna non tiene conto di casi di fallimento da parte dei consumatori o dei produttori, casi in cui vi siano processi multipli di consumo o casi in cui i dati, scritti sul disco, vengano persi.

Kafka d'altro canto utilizza questo tipo di strategia:

*Quando si pubblica un messaggio M, ad esso si assegna lo stato di "**committed**". Una volta che un messaggio pubblicato M assume questo stato, non sarà perso fino a quando esiste un Broker (un nodo), con stato "**alive**"(vivo), che gestisca la partizione in cui M è stato pubblicato.*

La definizione di *alive* verrà trattata in modo più dettagliato nella sezione successiva, per ora si suppone l'esistenza di un Broker perfetto, senza perdite; soffermiamoci quindi

sulle garanzie tra Producers e Consumers. Se un Producer tenta di pubblicare un messaggio e nel frattempo si verifica un *errore di rete*, non c'è garanzia che questo errore sia accaduto prima o dopo che il messaggio è stato dichiarato **committed**. Purtroppo Kafka non ha ancora trovato una soluzione definitiva per questo tipo di problema.

Anche se non si può essere sicuri di ciò che è accaduto nel caso di un *errore di rete*, è possibile consentire al Producer di generare una chiave primaria in modo da permettergli una sorta di ri-pubblicazione.

Questa caratteristica non è banale in quanto deve funzionare anche (soprattutto) in caso di guasto del Server. Con questo tipo di soluzione sarebbe sufficiente per il Producer riprovare a pubblicare finché non riceve un **acknowledgement** (riconoscimento) di un **committed** avvenuto con successo; a quel punto si avrebbe la garanzia che il messaggio è stato pubblicato esattamente una volta.

Non tutti i casi d'uso richiedono tali garanzie così forti. Per i casi sensibili alla latenza di tempo, Kafka permette al Producer di specificare il livello di **durabilità** che desidera: il Producer, una volta specificato di voler rimanere in attesa che il messaggio diventi **committed**, può assumere un ordine di latenza di 10 ms.

Il Producer può anche specificare il desiderio di eseguire l'invio completamente in modo asincrono o aspettare fino a quando il server leader (ma non necessariamente i follower) riceve il messaggio.

Di seguito si descrive la semantica di consegna dei messaggi dal punto di vista del Consumer. Tutte le repliche[14] hanno esattamente lo stesso registro (log) con gli stessi *offset*.

Il Consumer controlla la sua posizione all'interno di una specifica partizione. Se esistesse un Consumer perfetto che non va mai in *crash* potrebbe semplicemente salvare questa posizione nella propria memoria; se il Consumer invece fallisce, il protocollo prevede che questa partizione venga gestita da un altro processo. Il nuovo processo avrà bisogno di scegliere una posizione appropriata dalla quale avviare l'elaborazione.

[14] **Replica**: come precedentemente spiegato, ci possono essere più server che gestiscano lo stesso topic, quindi le stesse partizioni. Per replica si intende una copia di una determinata partizione.

Si suppone che il Consumer legga alcuni messaggi; ha quindi diverse opzioni per elaborarli ed aggiornare la sua posizione:

1. Il Consumer può leggere i messaggi, quindi salvare la sua posizione nel registro ed infine elaborarli. – Questo tipo di approccio non prevede la possibilità che il processo Consumer vada in *crash* dopo il salvataggio della sua posizione ma prima di aver salvato l'output ottenuto dopo l'elaborazione dei messaggi. All'elaborazione successiva un nuovo processo inizierà la lettura a partire dalla posizione salvata in precedenza, ignorando la presenza di messaggi non letti, prima di quella posizione. Questo corrisponde alla semantica di tipo *at-most-once*. Per facilitare la comprensione di quanto detto, immaginiamo che un lettore stia leggendo "Il Signore degli Anelli" ed incominci a leggere da pagina 20 (ovvero la pagina dove aveva posizionato il segnalibro l'ultima volta). Supponiamo che legga 15 pagine ma che presti una scarsissima attenzione nel leggerle, tanto da non ricordarsi cosa abbia letto, e fissi il segnalibro a pagina 35. La prossima volta che leggerà il libro ricomincerà da pagina 35 ma non si ricorderà cosa sia successo nelle precedenti pagine.

2. Il Consumer può leggere i messaggi, elaborarli ed infine salvare la propria posizione. – Questo caso non gestisce la possibilità che il processo Consumer fallisca dopo l'elaborazione dei messaggi, ma prima di aver salvato la sua posizione. In questo caso, quando il nuovo processo Consumer riprende l'elaborazione, i primi messaggi ricevuti saranno già stati elaborati. Questo caso corrisponde alla semantica di tipo *at-least-once*. Per capire a fondo il meccanismo, riprendiamo l'esempio del lettore di libri ed immaginiamo che, dopo aver letto fino a pagina 50, si dimentichi di spostare il segnalibro e lo lasci a pagina 35. Quando riprenderà a leggere, si accorgerà di aver già letto alcune pagine.

3. E per quanto riguarda la semantica *exactly-once* (cioè quella che si desidera veramente)? – Il problema è dovuto dalla necessità di coordinare la posizione del Consumer con l'output che effettivamente è già stato memorizzato. Un modo per raggiungere l'obiettivo sarebbe quello di introdurre un *commit a due fasi*, tra il salvataggio della posizione ed il salvataggio dell'output ottenuto dopo l'elaborazione dei messaggi. La soluzione ideale, definita da Kafka, è quella di

lasciare che il Consumer salvi sia l'offset che l'output nello stesso file; in questo modo se una delle due informazioni manca oppure se il file risulta danneggiato o corrotto o mancante, si prende come *buono* l'ultimo salvataggio effettuato.

Kafka, di default, utilizza la semantica *at-least-once* e consente all'utente di realizzare una consegna di tipo *at-most-once* impedendo al Producer la possibilità di ri-pubblicazione ed obbligando il Consumer a salvare la sua posizione nel registro prima dell'elaborazione dei messaggi.

Ricollegandoci a quanto detto in precedenza, un messaggio M dichiarato committed non sarà perso fino a quando esiste un nodo, con stato "vivo", che gestisca la partizione in cui M è stato pubblicato.

Ricordiamo che Kafka replica le partizioni di ogni topic in un certo numero di Server per garantire fault tolerance. In questo modo i messaggi rimangono disponibili in presenza di guasti.

L'unità di misura per il fattore di replica è la partizione; ogni partizione di un topic ha un solo leader e zero o più followers. Il numero totale dei Server (tra cui il leader) che mantengono la stessa partizione è detto il fattore di replica. Tutte le operazione di lettura e scrittura vengono registrate dal leader della partizione. La copia della partizione sui followers è identica a quella del leader, inoltre tutti hanno gli stessi offset ed i messaggi sono mantenuti nello stesso ordine (anche se, ovviamente, in un dato momento il leader può avere un numero minimo di messaggi non ancora replicati agli altri server).

I followers leggono i messaggi dal leader come se fossero un normale Consumer Kafka.

Per gestire automaticamente i guasti occorre prima di tutto dare una definizione di nodo "vivo" (alive). Per Kafka un nodo è vivo se:

- È in grado di mantenere la sua sessione con **ZooKeeper**[15];
- Se si tratta di un follower, può replicare le operazioni di scrittura che avvengono sul leader rispettando la tempistica degli altri followers.

[15] **Apache ZooKeeper**: un servizio open source ad alte prestazioni per il coordinamento di applicazioni distribuite. Tra i servizi che fornisce troviamo la denominazione, la gestione della configurazione, la sincronizzazione e servizi di gruppo. Esso fornisce un'interfaccia per l'implementazioni di protocolli di tipo Client/Server con scopo di supporto per le applicazioni. Kafka si appoggia a ZooKeeper per l'implementazione dei propri Server.

I nodi che soddisfano queste due condizioni si dicono ***in-sync*** (in quanto *alive* o *failed* risultano essere parole troppo vaghe). Il leader tiene traccia di un set di nodi con stato *in-sync;* se un follower "muore" o non è sincronizzato con gli altri Server, il leader dovrà rimuoverlo dalla lista.

Un messaggio viene considerato "**committed**" quando tutti i nodi (followers) con stato *in-sync* hanno aggiornato la propria replica della partizione. Solo i messaggi committed vengono consegnati al Consumer e quindi consumati.

La garanzia che Kafka offre è che un messaggio committed non verrà mai perso, purché vi sia almeno una replica gestita da un nodo *in-sync*, in ogni momento. Ciò significa che il Consumer non deve preoccuparsi di un eventuale perdita di messaggi da parte del leader, in quanto siamo certi che questi messaggi sono resi disponibili da uno dei followers.

Ovviamente in caso di fallimento del leader, un follower con stato *in-sync* prenderà il suo posto come leader e a sua volta gestirà il set dei nodi "vivi".

Occorre comunque precisare che Kafka è in grado di gestire solamente errori a livello di nodo, **ma non errori di rete**.

4.2.4 Repliche delle Partizioni

In questo paragrafo si analizzeranno gli algoritmi utilizzati da Kafka per la gestione delle repliche delle partizioni di un topic.

Ricollegandosi al paragrafo precedente, un topic contiene un registro partizionato e ciascuna partizione può essere replicata su N server diversi (dove N è detto *fattore di replica*), uno dei quali viene eletto *leader* della partizione ed è responsabile della gestione dei messaggi presenti al suo interno. Ogni volta che un leader va in crash o fallisce **deve essere rimpiazzato** da uno dei restanti N-1 server (detti followers).

Ma alcuni followers potrebbero non essere aggiornati od andare a loro volta in crash; occorre quindi scegliere un follower che abbia una copia corretta della partizione.

La garanzia fondamentale che un **algoritmo di replica** deve fornire è:

Se un client volesse consumare un messaggio dichiarato committed e il leader fallisce, il nuovo leader deve avere quel messaggio.

La domanda che ci si pone è:

***Come** eleggere un leader?*

Ma soprattutto:

***Quale nodo** presente tra i followers ha tutti i requisiti adatti per essere un leader?*

Kafka ha elaborato un **algoritmo di elezione** che mantiene dinamicamente un set di *repliche in-sync* (**ISR**), e solo i membri di questo gruppo sono eleggibili a leader. Un messaggio di una partizione non è considerato committed fino a quando tutti i nodi membri dell'ISR non hanno ricevuto tale messaggio.

Questo tipo di algoritmo garantisce che **qualsiasi nodo membro dell'ISR è idoneo** ad essere eletto come leader. Questo è un fattore importante per il modello di utilizzo di Kafka, dove ci sono moltissime partizioni e garantire la leadership è importante.

Con questo esempio si cerca di capire fino in fondo quali sono le potenzialità dell'algoritmo di elezione con ISR:

Si suppone una partizione avente fattore di replica pari a K, ovvero esistono in totale K nodi (followers e leader) che gestiscono una replica della partizione. Supponiamo che esista un set di ISR che contenga N+1 (N<K) repliche, allora tale partizione può tollerare fino ad N fallimenti senza perdere i messaggi che sono stati dichiarati committed.

Occorre notare che le **garanzie sulla perdita di dati** fornite da Kafka si basano sull'esistenza di almeno una replica su un Server in-sync. Se tutti i nodi che possiedono una replica, sia i membri dell'ISR che i followers che il leader, crollassero, queste garanzie non esisterebbero più. Occorre quindi trovare una soluzione ragionevole per questo tipo di evenienza (rara ma possibile).

Se per sfortuna capita un caso di questo genere, esistono due tipi di soluzioni:

1. Attendere che un nodo qualsiasi nell'ISR si riattivi e sceglierlo come leader (sperando che abbia ancora tutti i dati);
2. Scegliere il primo nodo (non necessariamente membro dell'ISR) che torna in vita come leader.

Un'analisi delle soluzioni:

1. Se si aspetta che un qualsiasi nodo membro dell'ISR ritorni in vita, si rimarrà in attesa per un lasso di tempo indefinito, senza contare la possibilità che la replica contenuta in tale nodo sia andata distrutta o i dati al suo interno siano andati persi;
2. Se, d'altro canto, un nodo *non-in-sync* torna in vita e viene eletto a leader, allora la sua copia della partizione diventa la fonte della verità, anche se potrebbe non essere aggiornata e quindi intrinsecamente errata.

Nella versione corrente, Kafka utilizza la seconda strategia favorendo quindi la possibile **scelta di una replica potenzialmente incoerente** se tutti i nodi membri dell'ISR sono crollati.

Di seguito viene analizzato il modo in cui i Producers interagiscono con le repliche.

Durante la fase di *write* (*scrittura*) di Kafka, i Producers, utilizzando il comando *request.required.acks*, possono decidere di aspettare un **acknowledgement** (**riscontro**) del messaggio inviato da 0,1 o tutti (-1) i nodi che gestiscono una replica della partizione interessata. Si noti che "il riconoscimento da parte di tutti i nodi" non garantisce che tutti i nodi abbiano ricevuto il messaggio. Per impostazione predefinita, quando *request.required.acks = -1*, il riconoscimento avviene non appena tutti gli **attuali** nodi in-sync hanno ricevuto il messaggio.

Ad esempio, si suppone che una partizione P sia stata replicata con fattore 2; esistono quindi due server (S1 ed S2) che ne mantengono una copia. Se S1 fallisce quindi non è più *in-sync*, quando un Producer invia un messaggio destinato a P, specificando *request.required.acks* = -1, il riscontro avrà comunque successo in quanto S2 ricopre la totalità dei Server che si trovano in stato *in-sync*.

Gestione delle Repliche sul Cluster

La discussione, di cui sopra, sulle repliche riguarda solo un singolo registro, ovvero una singola partizione di un topic; ma, come già accennato in precedenza, un cluster Kafka deve gestire centinaia/migliaia di partizioni.

Kafka cerca di bilanciare le partizioni all'interno di un cluster in modalità round robin per evitare di assegnare una quantità esorbitante di partizioni da gestire ad un piccolo numero di nodi. Allo stesso modo cerca di bilanciare la leadership in modo che ogni nodo possa diventare leader di una quota proporzionale delle sue partizioni.

Un grosso problema che può incombere sul sistema è che si verifichi un sovraccarico di ri – elezioni di nuovi leader. È ormai noto che ogni volta che un nodo leader di una partizione fallisce o crolla, tale nodo deve essere rimpiazzato da uno dei suoi followers; questa procedura avviene per ogni partizione e, siccome il cluster ne contiene migliaia, gestire una *failure* di un leader volta per volta grava pesantemente sui tempi computazionali del sistema.

Kafka ottimizza il *processo di elezione* aggiungendo la figura del **Controller**. Questo ruolo viene assegnato ad uno dei Broker all'interno del cluster ed i suo compiti sono:

- rilevare i guasti a livello di Brokers;
- modificare il leader di tutte le partizioni interessate in un broker fallito.

In questo modo il sistema è in grado di raggruppare le richieste di ri-elezione in lotti, che il Controller gestirà ad intervalli regolari di tempo. Questa soluzione rende il processo elettorale di gran lunga più economico e veloce. Se il Controller fallisce o muore, uno dei Broker superstiti diventerà il nuovo controller.

Con questo si conclude la parte di documentazione riguardante Apache Kafka. Sono stati affrontati i concetti di Consumer, Producer e Broker, come queste entità interagiscono tra loro e con quali mezzi; inoltre è stato descritto come il Cluster mantenga i messaggi dei topic e come questi ultimi vengano suddivisi in partizioni per essere gestiti al meglio, sia in termini di consumo efficace che in termini di consistenza dei dati. Ora si è a conoscenza del funzionamento di un sistema di messaggistica ad alte prestazione e degli algoritmi che un sistema di questo tipo necessita. Si può quindi passare alla sezione successiva.

4.3 Elaborato: Conversazione con Cluster Kafka

4.3.1 Analisi del Problema

Kafka è un sistema distribuito di messaggistica gestito come un cluster costituito da uno o più server ognuno dei quali viene chiamato Broker. I messaggi sono suddivisi in categorie dette topics. Sono detti Producers i processi che pubblicano messaggi in un topic, mentre i Consumers i processi che elaborano i feed dei messaggi presenti nei vari topics.

Si crei quindi, utilizzando le API messe a disposizione da Kafka e mediante la riga di comando, un Cluster (a livello locale) in cui risiede uno (o più) topic. Si creino entità Producer che inviano messaggi al Cluster e entità Consumer che leggono i dati presenti. Infine si replichi il topic su un numero N (a scelta) di Brokers e si effettuino test sulla fault-tolerance.

4.3.2 Progettazione

Il codice che segue è stato testato su riga di comando. Si tenga presente che Kafka è stato scritto per ottenere il massimo della performance su sistemi **Unix**. Per questo test si è utilizzato il **terminale Cygwin** che simula la riga di commando Unix.

Il primo step da effettuare per eseguire la conversazione è avviare un protocollo Server. Si tenga presente, come già spiegato in precedenza, che Kafka utilizza **Zookeeper** come supporto al Server. Si aziona quindi un'istanza di **Zookeeper** con il comando:

```
> bin/zookeeper-server-start.sh config/zookeeper.properties

[2013-04-22 15:01:37,495] INFO Reading configuration from:
config/zookeeper.properties
(org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
```

Successivamente si avvia il **Server**:

```
> bin/kafka-server-start.sh config/server.properties

[2013-04-22 15:01:47,028] INFO Verifying properties
(kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes
is overridden to 1048576 (kafka.utils.VerifiableProperties)
...
```

Una volta che il Server è azionato correttamente occorre creare un **topic**, un'entità che contenga i messaggi che si invieranno. Per pura formalità al topic si associa il nome "test". Il comando per la creazione è *--create*, inoltre occorre specificare l'host in cui far risiedere il topic (in questo caso localhost), un fattore di replica (*--replication-factor* N) e il numero delle partizioni (*--partitions* N). Di seguito il procedimento per la creazione di un topic con una partizione ed un fattore di replica pari a 1:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic test
```

Si ignori momentaneamente il numero di partizioni e il fattore di replica in quanto verranno gestiti successivamente. Una volta creato il topic lo si avvia sul Server Zookeeper creato nel passo precedente:

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181

test
```

In caso di successo il Server risponderà stampando a video il nome del topic che è stato avviato. Alternativamente si sarebbe potuto creare un topic manualmente, operazione molto sconveniente in quanto Kafka fornisce pattern di riga di comando per effettuare tale operazione.

Kafka è dotato di un Client a riga di comando in grado di prendere in input dati contenuti in un file (oppure dati forniti mediante lo standard input da tastiera) e inviarli sotto forma di messaggi ad un Cluster Kafka. Il sistema per default considera ogni linea di testo come un messaggio separato.

Per creare un messaggio occorre innanzitutto eseguire un **Producer** e quindi digitare un paio di messaggi nella console. Di seguito il procedimento che esegue il **Producer**, si tenga presente che occorre dichiarare per quale topic sono destinati i messaggi (in questo caso il topic "test"). Non occorre dichiarare l'indirizzo del Server in quanto è un'informazione contetuta all'interno del topic:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092
--topic test

> Ecco un messaggio
> Ecco un secondo messaggio
```

Occorre creare un entità che consumi i messaggi, che li legga e che processi i dati; occorre quindi creare un **Consumer** che stamperà a video i messaggi che leggera sul topic dichiarato (sempre "test"):

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--topic test --from-beginning

Ecco un messaggio
Ecco un secondo messaggio
```

Eseguire i processi Producer e Consumer nel medesimo terminale può risultare estremamente disordinato e difficile da leggere. L'ideale sarebbe aprire due terminali separatamente. Così facendo si dovrebbe essere in grado di scrivere i messaggi nel terminale del Producer e vederli comparire nel terminale del Consumer. Le prove eseguite al momento sono solo a livello locale, si noti che il protocollo da seguire per effettuare una conversazione tra due o più host sparsi nella rete è il medesimo.

Tutti gli strumenti della riga di comando hanno opzioni aggiuntive; l'esecuzione del comando senza argomenti visualizza informazioni d'uso li documenta in modo più dettagliato.

Fin'ora l'esecuzione prevedeva un singolo mediatore, si aumenti quindi il numero di nodi Broker per aumentare il livello di complessità della struttura. Si decide di espandere il gruppo fino a tre nodi (ancora tutti a livello locale).

Per prima cosa si crea un file di configurazione per ciascuno dei Brokers come segue:

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

Successivamente si modifichino questi nuovi file e si impostino le seguenti proprietà:

```
config/server-1.properties:
    broker.id=1
    port=9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    port=9094
    log.dir=/tmp/kafka-logs-2
```

Attraverso la proprietà *broker.id* si assegna un nome univoco e permanente di ciascun ogni nodo del cluster. Occorre assegnare ai Brokers due numeri di porta e due directory differenti in quanto i nodi appena creati sono in esecuzione sulla stessa macchina, lo scopo è quindi quello di evitare che questi si intralcino tra loro.

Siccome è già stato avviato un nodo Zookeeper in precedenza si possono avviare solamente due nuovi nodi digitando i seguenti comandi in cui viene specificato il nome dei file appena creati:

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

Si crea ora un nuovo topic chiamato "*my-replicated-topic"* con fattore di replica pari a tre. Si ricordi che la partizione di un topic può essere salvata su più Server diversi per mantenere un livello di *fault tolerance*.

Il numero di Server su cui viene salvata la stessa partizione si chiama fattore di replica. In questo caso, siccome il topic è di dimensioni ridotte, si prevede un'unica partizione. Per impostare il fattore di replica si utilizza il comando *--replication-factor* seguito da un numero intero:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 3 --partitions 1
--topic my-replicated-topic
```

La domanda che ci si pone al momento è: *come sapere quale broker sta facendo cosa?* Per ottenere queste informazioni occorre digitare il comando *--describe* con relativo nome del topic e dell'indirizzo in cui risiede (localhost):

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic my-replicated-topic

Topic:my-replicated-topic   PartitionCount:1
       ReplicationFactor:3  Configs:
       Topic: my-replicated-topic  Partition: 0  Leader: 1
       Replicas: 1,2,0      Isr: 1,2,0
```

Ecco una spiegazione dell'output appena visto. La prima riga fornisce un riassunto di tutte le partizioni, ogni linea aggiuntiva fornisce informazioni su una specifica partizione.

Dal momento che all'interno del topic è presente una unica partizione l'output sarà composto da un'unica linea.

- **Leader** è il nodo responsabile di tutte le letture e scritture per la partizione data. Ogni nodo sarà il leader di una porzione random della partizione.
- **Replicas** è l'elenco dei nodi che replicano il registro per questa partizione, indipendentemente dal fatto che siano leader o anche se sono attualmente vivi.
- **Isr** è l'insieme di repliche "*in-sync*". Per comprenderlo a fondo può essere considerato come il sottoinsieme della lista **Replicas** contenente i nodi attualmente in vita (incluso il Leader).

Si noti che nell'esempio appena visto il nodo "1" è il leader per l'unica partizione del topic.

Si è naturalmente in grado di eseguire lo stesso comando sul topic "test", creato in precedenza, al fine di visualizzare in che stato si trova:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic test

Topic:test    PartitionCount:1    ReplicationFactor:1  Configs:
       Topic: test   Partition: 0  Leader: 0    Replicas: 0
       Isr: 0
```

Non vi è alcuna sorpresa, il topic originale non ha repliche, l'unico server che gestisce il topic è il cluster creato in precedenza.

Si effettui la pubblicazione di alcuni messaggi sul nuovo topic *my-replicated-topic*:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092
--topic my-replicated-topic
...

messaggio 1
messaggio 2

^C
```

Si proceda quindi con il consumo (si consiglia l'uso di aprire due linee di comando separate):

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--from-beginning --topic my-replicated-topic
...

messaggio 1
messaggio 2

^C
```

Lo scopo successivo è quello di testare la fault-tolerance del sistema, si cerchi quindi di eliminare il Broker "1", ovvero il Leader della partizione:

```
> ps | grep server-1.properties

7564 ttys002    0:15.91
/System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bi
n/java...

> kill -9 7564
```

La leadership è passata a uno degli schiavi e il nodo "1" non si trova più all'interno del set di repliche in-sync (vedi par. 4.2.4 – "Repliche delle Partizioni"). Si esegui nuovamente il comando *--describe* su *my-replicated-topic* e si presti particolare attenzione ai campi *Isr* e *Leader*, confrontando con l'operazione di *--describe* omonima effettuata in precedenza :

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic my-replicated-topic

Topic:my-replicated-topic   PartitionCount:1
       ReplicationFactor:3  Configs:
       Topic: my-replicated-topic  Partition: 0  Leader: 2
       Replicas: 1,2,0      Isr: 2,0
```

Come si può notare il set *Isr* contiene solamente due nodi in quanto "1" è stato ucciso all'iterazione precedente, inoltre il nuovo *Leader* della partizione è il nodo "2". Ma i messaggi sono ugualmente disponibili per il consumo, anche se il leader originario è crollato. Si effettui un ulteriore test di consumo al fine di verificare quanto detto:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--from-beginning --topic my-replicated-topic
...
messaggio 1
messaggio 2

^C
```

Questo è reso possibile dalle **garanzie sulla perdita di dati** fornite da Kafka che si basano sull'esistenza di almeno una replica su un Server in-sync. Ovviamente se tutti i nodi che possiedono una replica crollassero, queste garanzie non esisterebbero più e i messaggi verrebbero perduti.

Si tenti ora di inviare nuovi messaggi al topic:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092
--topic my-replicated-topic
...

messaggio 3
messaggio 4
messaggio 5

^C
```

Si effettui di seguito il consumo utilizzando la clausola *–from-beginning* che permette di consumare i messaggi partendo da offset 0:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--from-beginning --topic my-replicated-topic
...

messaggio 1
messaggio 2
messaggio 3
messaggio 4
messaggio 5

^C
```

Si è dimostrato che Kafka non solo è in grado di processare enormi quantità di dati al secondo ma anche che la politica di replica delle partizioni permette un livello di garanzia molto alto. Il meccanismo della replica mediante ISR permette di istanziare un numero a scelta di nodi che mantengono una copia sincronizzata dei dati. Ogni qualvolta il nodo Leader del ISR crolla un sostituto garantisce la consistenza dei messaggi prendendo il suo posto. Si immagini di aumentare la dimensione dei nodi da tre ad un numero N, il livello di fault-tolerance aumenterebbe proporzionalmente.

Si tenga presente che i test effettuati sono riguardano solo una struttura a livello locale.

Capitolo 5 – Apache Storm

di Lorenzo Vernocchi

Tecnologie per la Costruzione di Piattaforme Distribuite
basate sul Linguaggio di programmazione Scala

5.1 Introduzione

Apache Storm è un sistema distribuito real-time, inoltre è open source e gratuito. Storm permette di processare in modo affidabile i flussi (stream) di dati di grandi dimensioni, è semplice e può essere utilizzato con qualsiasi linguaggio di programmazione. Questa tecnologia presenta molti casi d'uso come l'analisi real-time, calcolo continuo, supporto di RPC distribuite, ecc.

Storm è molto veloce infatti al secondo vengono elaborate circa un milione di tuple per nodo, è scalabile, è altamente resistente ai guasti e garantisce il trattamento sicuro dei dati. Questo sistema è integrato con alcune tecnologie e alcuni database già in uso come **Twitter**, **Spotify**, **Flipboard**, **Groupon** e molti altri.

L'ultimo decennio ha visto una rivoluzione nel trattamento dei dati. MapReduce[16], Hadoop e le relative tecnologie hanno permesso di memorizzare ed elaborare grandi quantità di dati.

[16] Il modello di calcolo **MapReduce** deve il suo nome a due celebri funzioni della programmazione funzionale, map e reduce, delle quali rispecchia in un certo senso il comportamento. In una computazione MapReduce infatti i dati iniziali sono una serie di record che vengono trattati singolarmente da processi chiamati *Mapper* e successivamente aggregati da processi chiamati *Reducer*. Questo modello di calcolo si presta ottimamente alla parallelizzazione e viene utilizzato nelle elaborazioni dei dati generati da enormi applicazioni web (es. Google, Facebook), ma anche per studi di genetica e di molti altri campi.

Purtroppo, queste tecnologie non sono sistemi real-time, né sono destinate a diventare tali in quanto l'elaborazione dei dati in tempo reale ha un insieme diverso di requisiti di elaborazione.

"Storm fills that hole"

Storm espone un insieme di primitive per eseguire real-time computation e scrittura di calcolo parallelo.

I vantaggi principali di Storm sono:

- **Vasta gamma di casi d'uso** - Storm può essere utilizzato per l'elaborazione dei messaggi, per l'aggiornamento dei database (stream processing), per programmare query dinamiche ad alta velocità che operino in parallelo e altro ancora. Un piccolo insieme di primitive Storm è in grado di soddisfare un ampio numero di casi d'uso;
- **Scalabilità** - Storm è in grado di scalare le dimensioni dei propri cluster al fine di poter processare un massiccio numero di messaggi al secondo. Per scalare una *Topology* occorre aumentare le sue impostazioni di parallelismo. Storm sfrutta Zookeeper (presentato precedentemente nel capitolo 4 – Apache Kafka) per il coordinamento del cluster. Zookeeper permette di scalare il cluster fino a raggiungere dimensioni molto grandi;
- **Garanzia di successo** - un sistema real-time deve avere forti garanzie di successo sui dati in corso di elaborazione. Storm garantisce che ogni messaggio venga elaborato correttamente;
- **Robustezza** - a differenza di sistemi come Hadoop[17], che sono noti per essere difficili da gestire, i cluster Storm lavorano in modo semplice. Si tratta di un obiettivo esplicito del progetto Storm per rendere *user-friendly* l'esperienza degli utenti con il sistema;
- **Fault-tolerant** - Storm garantisce che un calcolo possa essere sempre eseguito.
- **Implementazione in più linguaggi di programmazione** - le *Topologies* di Storm e le varie componenti di elaborazione possono essere definite in qualsiasi linguaggio di programmazione, rendendo Storm accessibile a (quasi) chiunque.

[17] **Hadoop**: sistema di calcolo *MapReduce distribuito* per processi di tipo batch estremamente scalabile e in grado di maneggiare terabyte o petabyte di dati senza colpo ferire.

Apache Storm elabora le cosiddette **Topologies,** entità che permettono la computazione in tempo reale. Una **Topology** (o **topologia**) è un grafico di computazione che elabora i *flussi (stream)* di dati; ogni nodo in una **topologia** contiene una logica di elaborazione e i link tra i nodi, i quali indicano come i dati dovrebbero essere passati. Le topologie restano in esecuzione finché non vengono arrestate.

Un **cluster** Storm (vedi Figura n.8) solitamente contiene due tipi di nodi che stanno all'estremità del sistema, i nodi head che eseguono **Nimbus** e i nodi di lavoro che eseguono **Supervisor**; tra i due nodi si interpone un nodo intermedio che esegue **Zookeeper**.

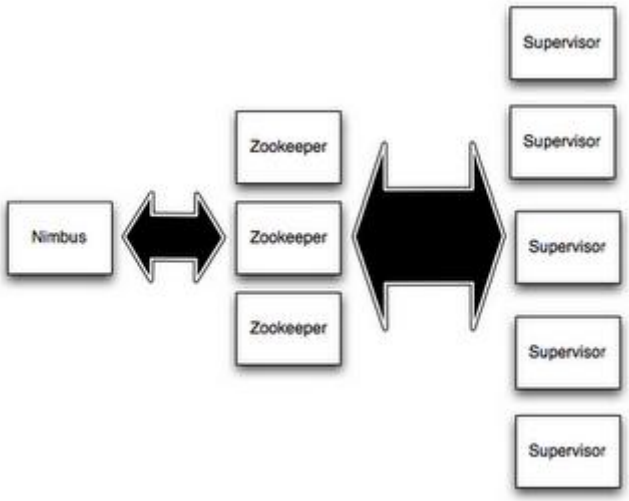


Figura n.8 – Cluster Storm

Il nodo **Nimbus** è responsabile della distribuzione del codice nell'intero cluster, dell'assegnazione delle attività alle macchine virtuali e del monitoraggio degli errori.

Esso assegna attività agli altri nodi del cluster tramite Zookeeper. I nodi **Zookeeper** assicurano la coordinazione del cluster e facilitano la comunicazione tra **Nimbus** e il processo **Supervisor** sui nodi di lavoro. Se un nodo di elaborazione si arresta, il nodo Nimbus riceve una notifica e provvede ad assegnare l'attività e i dati associati a un altro nodo.

La configurazione predefinita di Apache Storm prevede un solo nodo Nimbus. È possibile eseguire anche due nodi Nimbus. In caso di errore del nodo primario, il cluster passerà a quello secondario. Nel frattempo, il nodo primario verrà ripristinato.

Il Nimbus è un servizio offerto da Thrift[18] ed è quindi possibile sviluppare le topologie usando diversi linguaggi di programmazione.

Il nodo **Supervisor (supervisore)** è il coordinatore di ciascun nodo di lavoro, responsabile dell'avvio e dell'arresto dei *processi di lavoro* nel nodo.

Un *processo di lavoro* esegue un sottoinsieme di una **topologia**. Una **topologia** in esecuzione viene distribuita tra più *processi di lavoro* nel cluster.

Si ricorda che la **topologia** elabora i *flussi (stream)* di dati, ovvero raccolte di *tuple* (elenchi di *valori tipizzati dinamicamente*) non associati tra loro (ciascun flusso è indipendente); tali flussi sono prodotti dagli **Spout** e dai **Bolt**.

Uno **Spout** è una fonte di flussi, ovvero legge le tuple provenienti da una sorgente esterna e le pubblica nella topologia. Questi oggetti possono essere sia affidabili che inaffidabili.

Uno *Spout affidabile* è in grado di ritrasmettere una tupla persa o errata, mentre uno *Spout inaffidabile* elimina dalla propria memoria una tupla in seguito alla prima trasmissione (non ne tiene quindi una traccia ri-trasmissibile).

I **Bolt** sono entità che utilizzano i flussi emessi dagli Spouts, eseguono l'elaborazione sulle tuple e generano nuovi flussi in uscita. Essi sono anche responsabili della scrittura dei dati in una risorsa di archiviazione esterna, ad esempio una coda.

Lo scopo di questo paragrafo era fornire una prima panoramica del sistema Storm e di tutte le sue componenti. La sezione Documentazione si addentrerà più in profondità in tutti gli aspetti di Storm. Non verranno ulteriormente approfonditi i concetti **Nimbus**, **Zookeeper** e **Supervisor** in quanto sono servizi dei quali Storm usufruisce e mette a supporto dell'entità di sua proprietà.

[18] **Apache Thrift:** framework software per lo sviluppo di servizi scalabili per più linguaggi. Consente di creare servizi che funzionano con Scala, C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk e altri linguaggi.

5.2 Documentazione

5.2.1 Topologie

La logica di un'applicazione real-time risiede in una **Topology** (**topologia**) di Storm. Una topologia è un grafico di computazione che elabora i *flussi (stream)* di dati; ogni nodo in una topologia contiene una logica di elaborazione ed i link tra i nodi, indicando come i dati dovrebbero essere passati. I nodi presenti in una topologia si suddividono in **Spouts** e **Bolts**. Le topologie restano in esecuzione finché non vengono arrestate.

Ecco un esempio di una topologia che riceve dati da sorgenti esterne, questi dati vengono elaborati dai vari nodi e quindi inviati sottoforma di output stream a dei target che ne fanno richiesta:

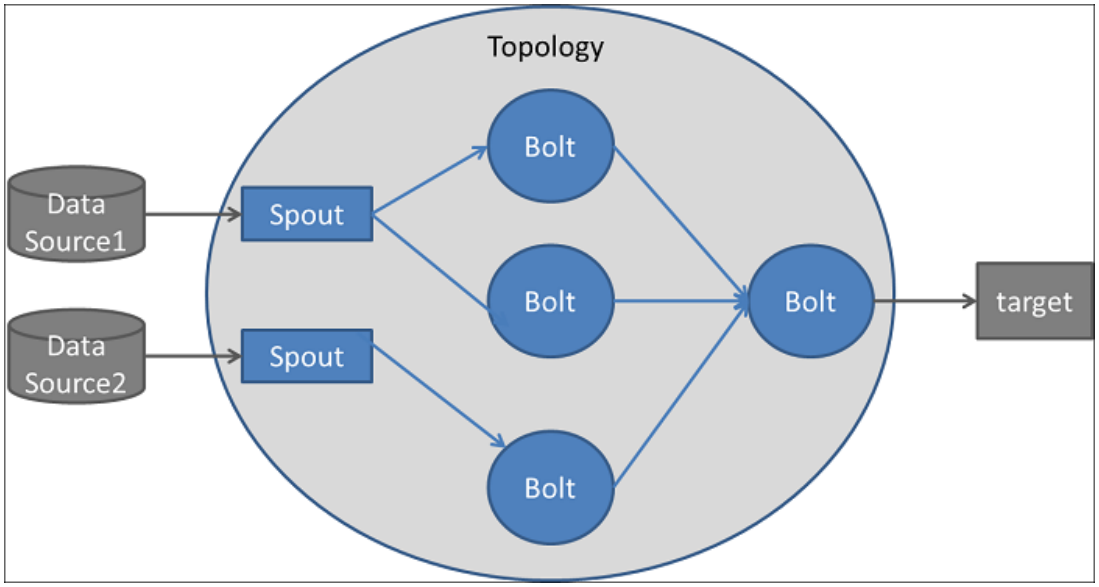


Figura n.9 – Storm Topology

5.2.2 Streams

Lo **Stream** (**flusso**) è l'astrazione di base in Storm; esso è una sequenza illimitata di tuple che vengono elaborate e create in parallelo in modo distribuito. I flussi vengono definiti con uno schema che nomina i campi di tuple al loro interno. Per impostazione predefinita, le tuple possono contenere interi, floats, long, short, byte, stringhe, array e booleani. È anche possibile definire tipologie personalizzate che possono essere utilizzate all'interno delle tuple.

In fase di dichiarazione, al flusso viene assegnato un codice identificativo. In quanto all'interno del sistema possono essere presenti flussi talmente simili che non necessitano

un'identificazione separata, esistono metodi convenienti per dichiarare un unico flusso senza specificare un id. In questo caso, al flusso viene assegnato l'identificatore predefinito "default".

In Storm, la **tupla** è la struttura dati di base che compone un flusso, è una lista denominata di valori, in cui ognuno di essi può essere di un tipo qualsiasi. Le tuple vengono tipizzate dinamicamente, ovvero non è necessario dichiarare la tipologia dei campi; inoltre hanno metodi di supporto come *getInteger* e *getString* per ottenere i valori assegnati ad un campo specifico. Un problema importante per Storm riguarda la serializzazione dei valori di una tupla. Per impostazione predefinita, Storm è in grado di serializzare solamente i tipi primitivi (stringhe e array di byte). Se si desidera utilizzare un altro tipo, è necessario implementare e registrare un serializzatore adatto allo scopo.

5.2.3 Serializzazione

Per completare il percorso riguardante i flussi e le tuple al loro interno si ricorda che le tuple possono essere costituite da oggetti di un qualsiasi tipo. Dal momento che Storm è un sistema distribuito, ha bisogno di sapere come serializzare e deserializzare gli oggetti quando vengono trasferiti tra i vari processi.

Una prima tecnica da considerare è la tipizzazione dinamica che si basa sull'omissione della dichiarazione di tipo per i campi in una tupla. Inserendo gli oggetti nei campi, Storm capisce il tipo di serializzazione da effettuare automaticamente (come **l'inferenza di tipo** in Scala). Per effettuare la serializzazione Storm utilizza **Kryo**[19], una libreria per la serializzazione flessibile e veloce. Di default, Storm è in grado di serializzare tutti i tipi primitivi, mentre se si desidera utilizzare un "tipo più sofisticato", per istanziare le proprie tuple, è necessario implementare un serializzatore personalizzato utilizzando il sistema **Kryo**.

L'aggiunta di serializzatori personalizzati viene fatta attraverso la proprietà *topology.kryo.register* in fase di configurazione della topologia, inoltre occorre definire un nome di una classe di registrazione. In questo caso Storm utilizzerà la classe **FieldsSerializer** di Kryo.

[19] **Kryo**: framework per la serializzazione di grafici di computazione (es. una topologia) veloce ed efficiente per linguaggi object oriented (come Java). Kryo è utile per rendere tali oggetti persistenti all'interno di un file, un database o attraverso la rete.

Storm fornisce degli *helper* per effettuare la registrazione di serializzatori personalizzati. La classe (per esempio **FieldsSerializer**) implementa il metodo *registerSerialization* che accetta una registrazione da aggiungere alla configurazione di una topologia (e quindi dei flussi al suo interno).

5.2.4 Spouts

Gli **Spouts** sono una fonte di flussi in una topologia. Sono oggetti che leggono le tuple provenienti da una sorgente esterna e le pubblicano all'interno della topologia. Esistono due tipologie di Spouts:

- *Spout affidabile*, che è in grado di ritrasmettere una tupla persa o errata;
- *Spout inaffidabile*, che cancella dalla propria memoria una tupla in seguito alla prima trasmissione (non ne tiene quindi una traccia ri-trasmissibile).

Gli Spouts possono emettere più di un flusso. **L'emissione** prevede innanzitutto la **dichiarazione** dei flussi da pubblicare utilizzando il metodo *declareStream* dell'**OutputFieldsDeclarer** (package *backtype.storm.topology*), un'interfaccia che mette a disposizione pattern per dichiarare nuovi flussi ed assegnare loro un identificativo.

Quindi si richiama il metodo *emit* della classe **SpoutOutputCollector (**package *backtype.storm.spout*) specificando il flusso da emettere. Lo **SpoutOutputCollector** è un collettore di **flussi in uscita** (**output streams**) che espone metodi per l'emissione di tuple da uno Spout.

Il metodo principale di uno Spout è *nextTuple* che emette una nuova tupla nella topologia oppure restituisce **null** se non ci sono nuove tuple da emettere. È imperativo che *nextTuple* non blocchi il processo Spout. Altri importanti metodi di uno Spout sono *ack* e *fail*, che vengono richiamati quando Storm rileva che l'emissione di una tupla all'interno di una topologia è avvenuta (*ack*) o meno (*fail*) con successo.
I metodi *ack* e *fail* sono a disposizione solo degli *Spouts affidabili*.

Garantire l'elaborazione dei messaggi

Storm garantisce che ogni messaggio (**tupla**) emesso da una Spout venga completamente elaborato, questo paragrafo descrive come il sistema assicura questa

garanzia e il comportamento che un utente deve seguire per usufruire delle funzionalità di affidabilità di Storm. Una tupla emessa da uno Spout può innescarne la creazione di migliaia. Storm considera una tupla emessa da un Spout come "completamente elaborata" quando l'albero delle tuple è stato esaurito ed ogni messaggio nella struttura è stato elaborato. Nella Figura n. 10 è possibile visualizzare un esempio di albero delle tuple, in questo caso l'albero rappresenta una topologia di flussi di stringhe le cui tuple sono "parole" ed a ciascuna di esse viene assegnato un intero che rappresenta la sua occorrenza nel flusso.

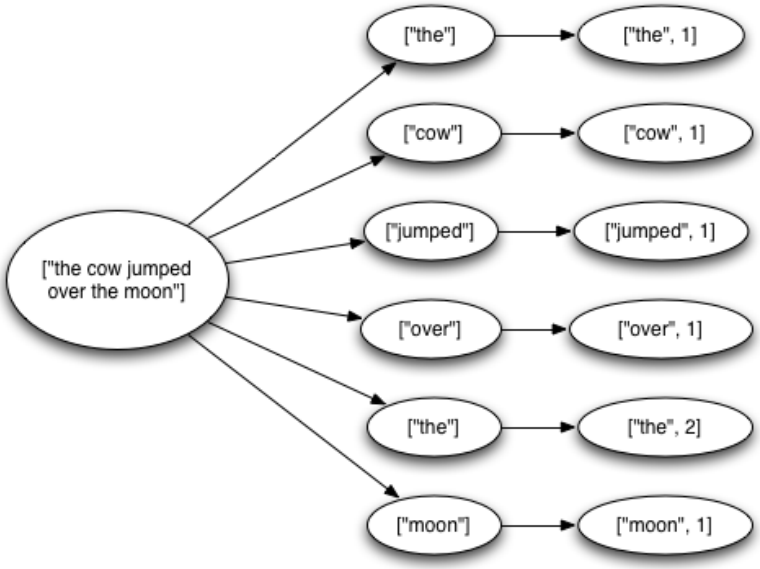


Figura n.10 – Albero delle tuple

Una tupla è considerata *fallita* quando l'albero di cui fa parte non riesce ad essere completamente elaborato all'interno di un timeout specificato. Questo **timeout** può essere configurato per ciascuna topologia utilizzando la configurazione *Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS* e il valore di default è 30 secondi. Di seguito si analizza il trait Spout ed i vari metodi che esso implementa, alcuni di essi illustrati in precedenza:

```
trait Spout {
    def open(conf: Map,
             context: TopologyContext,
             collector: SpoutOutputCollector)
    def close
    def nextTuple
    def ack(msgId : Any)
    def fail(msgId : Any)
}
```

In primo luogo, Storm richiede una tupla dallo Spout richiamando il metodo *nextTuple*. Lo Spout utilizza lo **SpoutOutputCollector** previsto nel metodo *open* per emettere una tupla ad uno dei suoi flussi di uscita. In fase di emissione, lo Spout fornisce un "id messaggio" che verrà utilizzato per identificare la tupla successiva. In un secondo momento, la tupla viene inviata ai **Bolts** che la consumano e Storm si occupa di tracciare l'albero dei messaggi. Se il sistema rileva una tupla completamente elaborata, chiamerà il metodo *ack* sullo Spout con "l'id messaggio" relativo alla tupla. Allo stesso modo, se in fase di elaborazione di una tupla scade il tempo di timeout Storm chiamerà il metodo *fail*.

5.2.5 Bolts

In precedenza si è affermato che una tupla emessa da parte di uno Spout viene inviata ad un entità detta Bolt. I **Bolts** sono oggetti all'interno di una topologia che utilizzano i flussi, eseguono l'elaborazione sulle tuple e sono responsabili della scrittura dei dati in una risorsa di archiviazione esterna. In poche parole, i Bolts ricevono i flussi (streams) di tuple generati ed emessi dagli Spouts (o anche da altri Bolts), successivamente le elaborano ed infine producono nuovi flussi di tuple detti **output streams** che saranno utilizzati in una struttura dati esterna. Tutte le manipolazioni di una topologia vengono effettuate dai Bolts.

In genere si preferisce effettuare semplici trasformazioni su un flusso (stream), le trasformazioni più complesse spesso richiedono un numero maggiore di passaggi e quindi più Bolts. In questo modo all'interno della topologia si formano più livelli di Bolts ed è perciò altamente probabile che un nodo di basso livello si trovi a gestire tuple emesse sia da nodi di tipo **Spout** che da altri di tipo **Bolt**. Ad esempio, la trasformazione di un flusso di tweet in un flusso di immagini di tendenza richiede almeno due fasi:

- Un Bolt dovrà mantenere un conteggio di retweet per ogni immagine;
- Uno o più Bolts dovranno effettuare lo streaming delle top N immagini.

I Bolts, in seguito all'elaborazione delle tuple dei flussi in entrata emessi dagli Spouts, emettono nuovi flussi (streams) di tuple seguendo lo stesso procedimento utilizzato dagli Spouts (utilizzando il metodo *declareStream* seguito dal metodo *emit)*.

Per dichiarare un flusso in entrata in un Bolt (un **input stream**) occorre specificare sia la componente che lo ha emesso, quindi l'identificativo dello specifico Spout (o Bolt) all'interno della topologia, sia il codice identificativo del flusso. È possibile incanalare tutti i flussi verso lo stesso Bolt; per far ciò è necessario richiamare il metodo di input per ciascun componente (Spout o Bolt di livello superiore) nella topologia.

InputDeclarer è una classe generica del package *backtype.storm.topology* che mette a disposizione metodi per sottoscrivere un input stream. Un Bolt che vuole sottoscrivere a se stesso i flussi provenienti dal componente con id = "1" richiamerà il metodo:

```
declarer.shuffleGrouping ("1")
```

Questo metodo è equivalente a:

```
declarer.shuffleGrouping ("1", DEFAULT_STREAM_ID)
```

Il metodo principale di un Bolt è il metodo *execute*, che prende in ingresso una nuova tupla da consumare. I Bolts emettono nuove tuple utilizzando un'istanza della classe **OutputCollector** (package *backtype.storm.task*) e su di essa devono richiamare il metodo *ack* per ogni tupla elaborata, in modo che Storm sia sempre a conoscenza di quali tuple sono state completate. Il sistema può quindi richiamare il metodo *ack* sullo Spout, che ha originariamente emesso tali tuple, per notificare il completamento delle suddette (vedi par 5.2.4 – "Spouts").

L'**OutputCollector** è molto simile allo **SpoutOutputCollector** enunciato nel paragrafo precedente; la differenza principale tra questo collettore di uscita e lo **SpoutOutputCollector** è che il secondo permette agli Spouts di etichettare i flussi con un identificativo in modo che possano essere dichiarati acked o failed in seguito.

Si conclude il paragrafo relativo ai Bolts, si noti che i nodi di tipo Bolt sono l'ultimo step di elaborazione ed emissione di stream all'interno di una topologia. Riassumendo, una topologia è un grafico di nodi, ovvero un grafico orientato di Spouts e Bolts con lo scopo di elaborare i flussi (stream) di dati. Ogni flusso è composto da più tuple.

Gli Spouts sono i primi nodi all'interno di una topologia e il loro scopo è quello di elaborare i dati entranti da una sorgente esterna ed emetterli ai nodi di livello successivo, i Bolts. Questi ultimi ricevono i flussi, elaborati dagli Spouts, processano le tuple ed emettono nuovi flussi. I flussi emessi dai Bolts possono essere sottoscritti a

Bolts di livello inferiore o direttamente inviati a sorgenti esterne che ne fanno richiesta. Per ogni tupla elaborata i Bolts inviano un *ack*, che Storm inoltrerà agli Spouts nella topologia. In questo modo si è sempre a conoscenza di quali tuple sono state completate.
La Figura n.9 presente nel paragrafo 5.2.1 – "Topologie" riassume quanto spiegato.

5.2.6 Stream Grouping

Una parte importante nella definizione di una topologia è specificare, per ogni Bolt, che tipi di flussi (stream) quest'ultimo dovrebbe ricevere come input. Un **raggruppamento di flussi** (o **stream grouping**) definisce come uno stream deve essere partizionato tra le tasks del Bolt. Per il momento si ignori il concetto di task in quanto verrà spiegato successivamente.

Storm mette a disposizione sette tipologie di raggruppamento, inoltre dà la possibilità di creare un raggruppamento personalizzato implementando l'interfaccia **CustomStreamGrouping**:

- **Shuffle grouping** – Le tuple sono distribuite in modo casuale attraverso le tasks del Bolt, in modo tale che la distribuzione sia uniforme;
- **Fields grouping** – Il flusso è partizionato in base ad uno determinato campo specificato nel raggruppamento. Ad esempio, se il flusso è raggruppato per il campo "user-id", tutte le tuple con lo stesso "user-id" saranno sempre assegnate allo stessa task del Bolt, ma le tuple con "user-id" diversi vengono assegnati a tasks differenti;
- **Partial Key grouping** – Il flusso è diviso in base ad un campo specificato nel raggruppamento, come nel Fields grouping, con la differenza che le tuple del campo scelto per il raggruppamento vengono distribuite in modo equilibrato tra due Bolts. In questo modo si fornisce un migliore utilizzo delle risorse;
- **All grouping** – Il flusso viene replicato in tutte le tasks del Bolt. Questo raggruppamento è molto pericoloso in quanto si rischia inconsistenza dei dati, va quindi utilizzato con attenzione;
- **Global grouping** – L'intero flusso viene assegnato ad una singola task del Bolt;
- **None grouping** – Questa "tecnica" di raggruppamento specifica che non si tiene conto di come il flusso è raggruppato;

- **Direct grouping** – Si tratta di un particolare tipo di raggruppamento in cui il produttore del flusso (quindi lo Spout o un Bolt di livello superiore all'interno della topologia) decide a quale task del nodo assegnare le tuple ed anche quali tuple assegnarvi;
- **Local grouping** – Se il Bolt "destinatario" del flusso ha una o più tasks nello stesso processo, le tuple saranno mescolate alle sole tasks "*in-process*". In caso contrario, si ricade in un normale Shuffle grouping.

Tasks & Workers

All'interno del cluster, ogni Spout o Bolt esegue un certo numero di processi (o compiti) chiamati tasks. Ogni **task** corrisponde ad un *Thread* di esecuzione e la tecnica di Stream grouping definisce come inviare tuple da un insieme di tasks ad un altro.

Le topologie vengono eseguite attraverso uno o più processi di lavoro chiamati **workers**. Ogni worker (processo di lavoro) è una JVM fisica che esegue un sottoinsieme di tutte le tasks presenti all'interno della topologia. Ad esempio, se il parallelismo combinato della topologia è di 300 tasks, suddivise tra Spouts e Bolts, e sono allocati 50 workers, allora ciascun worker eseguirà 6 tasks. Storm cerca di distribuire le tasks equamente tra tutti i lavoratori.

Per settare il **parallelismo di una topologia**, ovvero il numero totale delle tasks presenti all'interno di questa, si utilizzano i metodi *setSpout* (per gli Spouts) e *setBolt* (per i Bolts) della classe **TopologyBuilder**. Il parallelismo sarà dato dalla somma delle tasks settate per ciascuno Spout e delle tasks settate per ciascun Bolt. Il **TopologyBuilder**, presente nel package *backtype.storm.topology*, espone i metodi per definire una topologia.

Con questo paragrafo termina il percorso con Storm. Una "carenza" del sistema consiste nel fatto che sono presenti pattern e protocolli per la programmazione user friendly solamente per Java. La sezione successiva mostrerà come implementare tali protocolli anche per linguaggio Scala.

5.3 Elaborato: Storm Scala Spout & Storm Scala Bolt

5.3.1 Analisi del problema

Storm è un sistema distribuito real-time che può essere utilizzato con qualsiasi linguaggio di programmazione, tuttavia mette molte API e pattern a supporto solamente del linguaggio Java. Si implementino in linguaggio Scala due classi che forniscano pattern adeguati per creare Spouts e Bolts dinamicamente.

5.3.2 Progettazione

Uno Spout è un oggetto che legge le tuple provenienti da una sorgente esterna e le pubblica all'interno della topologia. Si prende subito in analisi il metodo principale di uno Spout ovvero l'emissione (pubblicazione) di flussi. Per effettuare un'operazione di emissione si richiama il metodo *emit* oppure il metodo *emitDirect* della classe **SpoutOutputCollector** (passato come parametro del metodo *open* che inizializza lo Spout) specificando il flusso da emettere. Si riscrivono quindi i due metodi in modo da renderli compatibili con il linguaggio Scala.

Nello specifico, il metodo *emit* prende come parametro un numero variabile di argomenti di tipo **Any**[20] che compongono la tupla da emettere. Il metodo *emitDirect* è lo stesso, con la semplice differenza che, oltre agli argomenti della tupla, viene passato un identificativo della task a cui verrà assegnato il compito di emetterla (la task viene generata in automatico dallo **SpoutOutputCollector**).

Nell'analisi dei suddetti metodi, si presti attenzione alla sintassi *_.asInstanceOf[AnyRef]* che verrà riscontrata più volte all'interno dell'elaborato. Il carattere *underscore* viene utilizzato da Scala quando il compilatore non è a conoscenza né del tipo di oggetto che richiamerà tale metodo (in questo caso *instanceOf*) né del numero. Poiché qualsiasi classe che estenda da Any ha a disposizione il metodo *instanceOf* è possibile utilizzare questa sintassi per evitare un *foreach* o una scansione di tutti gli oggetti passati come parametro. Così facendo il compilatore è a conoscenza del fatto che dovrà effettuare l'operazione per tutti i parametri passati a funzione.

[20] In Scala esistono varie tecniche per indicare un oggetto *qualunque;* in questo caso specifico si prendono in considerazione **Any** ed **AnyRef**. La differenza tra le due classi Scala è che Any può fare riferimento sia alla classe java.lang.Object che ad una primitiva qualunque mentre AnyRef può essere considerato il "gemello" del Java Object.

Di seguito il codice dello Spout scritto in Scala:

```
abstract class StormSpout (
     val streamToFields: collection.Map[String, List[String]],
     val isDistributed: Boolean
     ) extends BaseRichSpout
       with SetupFunc
       with ShutdownFunc {

 var _context : TopologyContext = _
 var _collector : SpoutOutputCollector = _
 var _conf: Map[_, _] = _

  def open(conf: Map[_, _],
           context: TopologyContext,
           collector: SpoutOutputCollector) = {

         _context = context
         _collector = collector
         _conf = conf
         _setup()

  }

  override def close() = {
    _cleanup()
  }

  def nextTuple

  def declareOutputFields(declarer: OutputFieldsDeclarer) = {

     streamToFields.foreach { case(stream, fields) =>
               declarer.declareStream (stream,
                                       new Fields(fields:_*)
                                       )
                             }
  }

  def getSpout = this

  def msgId(messageId: Any) =
     new MessageIdEmitter(_collector,
                          messageId.asInstanceOf[AnyRef])

  def toStream(streamId: String) =
           new StreamEmitter(_collector, streamId)
```

```
  def emit(values: Any*) =
          _collector.emit(values.toList.map {
                                  _.asInstanceOf[AnyRef]
                                  }
                          )

  def emitDirect(taskId: Int, values: Any*) =
          _collector.emitDirect(taskId,
                                  values.toList.map {
                                        _.asInstanceOf[AnyRef]
                                        }
                                  )
}
```

Il metodo *nextTuple* permette di passare al processo di emissione della tupla successiva nella topologia oppure restituisce **null** se non ci sono nuove tuple da emettere. Dal momento che Storm non impone vincoli sull'ordine di emissione delle tuple, tale metodo viene dichiarato astratto e potrà essere implementato ad hoc seguendo una politica personalizzata. I metodi *ack* e *fail* vengono ereditati dalla classe Java BaseRichSpout.

Si prendono in analisi i metodi *msgId* e *toStream*. A supporto di questi vengono create le classi **MessageIdEmitter** e **StreamEmitter**: la prima definisce metodi per l'emissione automatica degli identificatori di tupla (o "ID di messaggio"), la seconda permette l'emissione ordinata del flusso convertito in Lista.

```
class MessageIdEmitter (collector: SpoutOutputCollector,
                        msgId: AnyRef) {

  var emitFunc: List[AnyRef] => Seq[java.lang.Integer] =
                                collector.emit(_, msgId).asScala

  var emitDirectFunc: (Int, List[AnyRef]) => Unit =
          collector.emitDirect(_, _, msgId)

  def toStream(streamId: String) = {
    emitFunc = collector.emit(streamId, _, msgId)
    emitDirectFunc = collector.emitDirect(_, streamId, _, msgId)
    this
  }
  def emit(values: AnyRef*) = emitFunc(values.toList)
  def emitDirect(taskId: Int, values: AnyRef*) =
     emitDirectFunc(taskId, values.toList)

}
```

```
class StreamEmitter(collector: SpoutOutputCollector,
                    streamId: String) {

  def emit(values: AnyRef*) =
      collector.emit(streamId, values.toList)

  def emitDirect(taskId: Int, values: AnyRef*) =
      collector.emitDirect(taskId, streamId, values.toList)

}
```

Si passa ora alla progettazione della classe **Bolt**. Lo scopo finale è sempre quello di supportare disegni diversi per i Bolts e facilitarne l'implementazione. Il **Bolt** è una entità all'interno di una topologia che riceve i flussi (streams) di tuple generati ed emessi dagli Spouts (o anche da altri Bolts), successivamente li elabora ed infine produce nuovi flussi di tuple, detti output streams, che saranno utilizzati in una struttura dati esterna.

Il metodo principale di un **Bolt** è il metodo *execute*, che prende in ingresso una nuova tupla da consumare; Storm non definisce vincoli riguardo a politiche di esecuzione di una tupla, occorre quindi implementare il metodo *execute* in modo autonomo. Per maggiori chiarimenti di seguito un esempio di un Bolt concreto che estende la classe Scala **StormBolt** e processa flussi di stringhe:

```
class MyBolt extends StormBolt(outputFields = List("word")) {
  def execute(t: Tuple) = {
    t emit (t.getString(0) + "!!!")
    t ack
  }
}
```

I Bolts emettono nuove tuple utilizzando un'istanza della classe **OutputCollector** che viene passato come parametro del metodo *prepare* (che inizializza il Bolt). I metodi *ack* e *fail* vengono implementati dalla classe StormTuple che verrà definita successivamente.

Di seguito il codice della classe astratta **StormBolt**:

```
abstract class StormBolt(
     val streamToFields: collection.Map[String, List[String]]
     )extends BaseRichBolt
      with SetupFunc
      with ShutdownFunc
      with Bolt {

  var _context: TopologyContext = _
  var _conf: Map[_, _] = _



  def prepare( conf : Map[_, _],
               context : TopologyContext,
               collector : OutputCollector ) = {

      _collector = collector
      _context = context
      _conf = conf
      _setup()

  }

  def declareOutputFields(declarer : OutputFieldsDeclarer) = {
      streamToFields foreach {
           case(stream, fields) =>
                declarer.declareStream(stream,
                                       new Fields(fields:_*))
      }
  }

  override def cleanup() = _cleanup()

  def execute
}
```

Analizzando il codice si può notare l'inclusione del trait **Bolt**, definita esclusivamente per permettere l'emissione di tuple in modo separato dalla classe **StormBolt**. Questa separazione è stata volutamente effettuata poiché, nel caso si preferisca implementare una classe diversa da **StormBolt**, si può sfruttare il trait per evitare l'implementazione dei metodi di emissione di flussi.

Di seguito l'implementazione del trait **Bolt**:

```
trait Bolt {

  var _collector: OutputCollector = _

  def getBolt = this

  implicit def stormTupleConvert(tuple: Tuple) =
    new StormTuple(_collector, tuple)
}
```

Si prendono in analisi la classe **StormTuple**, ovvero l'implementazione di una tupla in linguaggio Scala, e i relativi metodi per l'emissione di tale tupla, *ack* e *fail*. Il metodo di emissione viene definito in una classe astratta chiamata **BaseEmit** mentre i metodi *ack* e *fail* vengono definiti all'interno della classe **StormTuple** che estende da **BaseEmit**. L'implementazione di tali metodi consiste nel richiamare gli omonimi messi a disposizione dall'**OutputCollector,** passato come parametro in fase di definizione della classe. Di seguito il codice delle due classi:

```
abstract class BaseEmit(val collector: OutputCollector) {

  var emitFunc: List[AnyRef] => Seq[java.lang.Integer] =
      collector.emit(_).asScala

  var emitDirectFunc: (Int, List[AnyRef]) => Unit =
      collector.emitDirect(_, _)

  def emit(values: Any*) =
      emitFunc(values.toList.map { _.asInstanceOf[AnyRef] })

  def emitDirect(taskId: Int, values: Any*) =
      emitDirectFunc(taskId,
                     values.toList.map { _.asInstanceOf[AnyRef]})
}//end of class

class StormTuple(collector: OutputCollector,
                 val tuple: Tuple) extends BaseEmit(collector) {

  emitFunc = collector.emit(tuple, _).asScala
  emitDirectFunc = collector.emitDirect(_, tuple, _)

  def toStream(streamId: String) = {
    emitFunc = collector.emit(streamId, tuple, _).asScala
    emitDirectFunc = collector.emitDirect(_, streamId, tuple, _)
    this
  }
```

```
  def ack = collector.ack(tuple)

  def fail = collector.fail(tuple)

}//end of class
```

Si può notare che le classi **StormSpout** e **StormBolt** ereditano da **StetupFunc** e **ShutdownFunc**. Il primo è un trait con lo scopo di definire l'inizializzazione in fase di avvio di ogni istanza di uno Spout/Bolt, il secondo esegue una corretta eliminazione dell'istanza che ne fa uso (dopo aver terminato tutti i suoi compiti):

```
trait SetupFunc {
  //lista di funzioni di setup
  private var setupFuncs: List[() => Unit] = Nil

  // esegue tutte le funzioni di setup
  def _setup() = setupFuncs.foreach(_())

  // aggiunge una funzione di setup in lista
  /* :: è il metodo per aggiungere un oggetto in testa ad una
   * lista */
  def setup(func: => Unit) = { setupFuncs ::= func _ }
}
```

```
trait ShutdownFunc {
  // lista di funzione shutdown
  private var _shutdownFunctions : List[() => Unit] = Nil

  // registra una funzione di shutdown nella lista
  def shutdown(sf: => Unit) = _shutdownFunctions ::= sf _

  // effettua lo shutdown totale
  protected def _cleanup() = _shutdownFunctions.foreach(_())

}
```

Si conclude così il capitolo su Storm. Si è ora in grado di implementare una propria topologia definendo una classe per l'entità Spout e una classe per il Bolt, si sono compresi a fondo i metodi principali per la gestione e l'emissione delle tuple e dei flussi. Con questo capitolo si chiude il percorso con Finagle, Akka, Kafka e Storm; si passa quindi ad un confronto ipotetico delle quattro tecnologie.

Conclusioni

Per concludere *Tecnologie per la Costruzione di Piattaforme Distribuite basate sul Linguaggio di Programmazione Scala* di seguito un breve resoconto in merito a quanto affrontato nel percorso.

Finagle è un sistema RPC utilizzato per la high performance computing e per la costruzione di Server ad alta concorrenza che sfrutta un modello per la programmazione concorrente basato sui **Futures**. A supporto del sistema si trovano i Services, funzioni utili per implementare sia Client che Server capaci di riceve una qualche richiesta di tipo Req e ritornare un Future che rappresenta l'eventuale risultato (o fallimento) di tipo Rep. Ad essi Finagle appoggia dei **Filters**, funzioni che permettono di rielaborare i dati passati a e restituiti da una qualsiasi richiesta, in modo da renderli compatibili ed elaborabili. Queste funzioni, combinate correttamente, sono molto utili per creare Client e Server performanti.

Akka è un toolkit sviluppato dalla Typesafe Inc. con lo scopo di semplificare la realizzazione di applicazioni concorrenti e distribuite sulla JVM, che supporta più modelli di programmazione per la concorrenza (Futures, Agents …), ma predilige il modello basato sugli **Actors**, oggetti che incapsulano uno stato, un comportamento e una mailbox (casella postale). Questi comunicano tra loro esclusivamente attraverso lo scambio di messaggi ed offrono un alto livello di astrazione per la concorrenza e il parallelismo, un modello di programmazione asincrono ed altamente performante per la gestione degli eventi. Ogni Actor ha un metodo "**receive**" con scopo di gestione dei messaggi ricevuti. La loro caratteristica per eccellenza è che i compiti vengono suddivisi tra Actors che a loro volta ne delegano una parte ad altri Actors fino a che il problema non diventi abbastanza piccolo da poter essere facilmente risolvibile. Un'altra importante proprietà degli oggetti Actor è l'assoluta indipendenza tra un Actor e l'altro.

Kafka è un sistema distribuito di messaggistica, in parte scritto in Scala, che permette la gestione dei cosiddetti Big Data, centinaia di megabyte di traffico in lettura e scrittura al secondo da parte migliaia di Client. Kafka raggruppa i messaggi in insiemi della stessa categoria detti topics, per ognuno dei quali un cluster mantiene un registro partizionato.

I Producers pubblicano i messaggi (i dati) all'interno di un topic e sono responsabili di scegliere in quale partizione del registro del topic inserire un proprio messaggio. I Consumers o consumatori leggono (consumano) i dati presenti. Kafka garantisce parallelismo, ordine, bilanciamento del carico e fault tolerance, nonché prestazioni elevate sia a livello di consumo che a livello di produzione.

Storm è un sistema distribuito real-time, open source e gratuito che permette di processare in modo affidabile flussi di dati di grandi dimensioni, è semplice e può essere utilizzato con qualsiasi linguaggio di programmazione. Storm è molto veloce infatti al secondo vengono elaborate circa un milione di tuple per nodo, è scalabile, è altamente resistente ai guasti e garantisce il trattamento sicuro dei dati. Il sistema elabora le cosiddette Topologies, grafici di computazione che elaborano i *flussi (stream)* di dati; ogni nodo in una topologia contiene una logica di elaborazione ed i link tra i nodi, i quali indicano come i dati dovrebbero essere passati. Tali nodi si suddividono in due categorie: Spouts e Bolts. Uno Spout è una fonte di flussi, un Bolt è un'entità che esegue l'elaborazione sulle tuple presenti all'interno dei flussi e genera nuovi flussi in uscita.

Rispetto a quanto detto si mettono di seguito i quattro sistemi a confronto.

Per cominciare si prendano in considerazione Akka, Finagle e Storm; queste tecnologie sono tutte soluzioni di grande raffinatezza per problemi di programmazione concorrente. Un aspetto positivo di Akka è che le dimensioni di un sistema Akka sono direttamente proporzionali al numero di attori coinvolti. Inoltre, un sistema di adeguate dimensioni è decisamente più performante di un sistema di piccole dimensioni, questo grazie al modello Actor che permette una suddivisione distinta dei compiti e un'indipendenza tra entità.

Finagle invece è nettamente più flessibile riguardo alla politica di creazione di un protocollo Client/Server, in quanto la tecnica dello "stack" rivoluziona la programmazione classica delle reti poiché permette di definire un Client (o un Server) come una pila di Service ognuno dei quali opera in favore di quello successivo. Mentre Akka non è in grado di implementare un protocollo Client/Server simile a quello di Finagle, quest'ultimo non regge il confronto con la complessità in termini di dimensioni che un sistema Akka può raggiungere, in quanto consiglia l'implementazione di pochi Service (ad una ventina) per un singolo protocollo Server. Akka, come mostra

l'Elaborato: Neighborhood, può impiegare un numero smisurato di Actors senza appesantire eccessivamente il sistema.

Akka quindi è migliore per gli attori, ma bisogna effettuare operazioni di monitoraggio continuo su di essi e quindi elaborare delle strategie ad hoc. Storm è ideale per effettuare calcoli tra host sparsi all'interno della rete e, perciò, prevede delle strategie molto semplici da impostare per effettuare questa funzione di monitoraggio. Questo permette al sistema di avvantaggiarsi rispetto ad Akka ed anche a Finagle riguardo ai tempi di elaborazione. Un difetto di Storm sta nella sua unilateralità di comunicazione, infatti una topologia presenta un solo ordine di lettura mentre Finagle ed Akka sono in grado di effettuare una comunicazione bilaterale.

Si sposti ora l'attenzione su Kafka e Storm; il primo è in grado di processare enormi quantità di dati rispetto al secondo, inoltre la politica di replica delle partizioni permette un livello di garanzia più alto rispetto allo Stream grouping che Storm mette a disposizione della topologia. Lo Strem grouping è una tecnica per garantire una sorta di ordine all'interno della topologia. A supporto della garanzia di processo dei messaggi Storm utilizza il sistema di acknoledgement ma non tiene conto di errori di trasmissione, ack perduti o fail perduti. Kafka, invece, sfrutta il meccanismo della replica mediante ISR: un set di nodi che mantengono una copia sincronizzata dei dati. Ogni qualvolta il nodo Leader dell'ISR crolla, un sostituto garantisce la consistenza dei messaggi prendendo il suo posto. Si potrebbe pensare che Kafka sia decisamente più efficiente in termini di sicurezza dei dati.

Tuttavia le politiche di Storm riguardo alla terminazione di una topologia sono molto più flessibili riguardo alle politiche di gestione dei messaggi di Kafka. Infatti Storm garantisce che una topologia venga interamente processata ammettendo qualche errore, Kafka invece subisce un rallentamento dovuto ai tentativi di recupero dei nodi *crashed.*

Non è semplice (e forse non è possibile) identificare obiettivamente la tecnologia più performante. Si può sottolineare comunque che il paragone tra i quattro sistemi si suddivide in due principali confronti:

- *Akka contro Finagle*
- *Kafka contro Storm*

Poiché a seguito del confronto tra le tecnologie non è emerso palesemente un "vincitore", si sposti la concentrazione su una visione dedicata alla collaborazione tra queste. Si ipotizzano di seguito alcune realtà con lo scopo di fondere gli aspetti positivi di Finagle, Akka, Kafka e Storm.

Si prendano in considerazione Finagle e Akka. Si potrebbe ipotizzare un protocollo Client/Server che, a livello di rete, sfrutti i pattern per la programmazione messi a disposizione da Finagle, in modo da creare un protocollo Server veloce e performante che gestisce in modo pratico le richieste. A livello locale le richieste verrebbero processate da un modello basato sugli Actors in grado quindi di risolvere ciascuna richiesta in modo indipendente e in completa sicurezza dalle altre. Le richieste più complesse verrebbero suddivise in sotto richieste più facili da risolvere in modo da non rallentarne il processo. Una volta processate, gli Actors di root che hanno ricevuto le richieste direttamente dal Server Finagle risponderebbero con i risultati (o i fallimenti) e quindi il Server potrebbe trasmetterli velocemente al Client. Si potrebbero implementare servizi di grandi dimensioni altamente performanti.

Un altro esempio potrebbe riguardare il rapporto tra Kafka e Storm. Si potrebbe immaginare la fusione di queste tecnologie come una raffineria di petrolio greggio che lo estrae, lo distilla ed infine lo elabora in prodotti utili come il gas.

La "raffineria" di dati in tempo reale *Kafka & Storm* convertirebbe i dati "grezzi" in streaming di dati utili, consentendo nuovi casi d'uso e modelli di business innovativi per l'impresa moderna. Kafka processerebbe i dati provenienti da sorgenti esterne ed effettuerebbe la suddivisioni in topics, dopo di che Storm elaborerebbe un topologia ad hoc per ogni topic ed implementerebbe un numero di Spouts corrispondente al numero di partizioni. In questo modo si sfrutterebbero le garanzie e l'organizzazione equilibrata del sistema Kafka e la potenza di emissione di Storm. Quest'ultimo, quindi, sarebbe il "fornitore diretto" dei dati elaborati per i "consumatori".

Una terza realtà vedrebbe fondersi Akka e Kafka. Si supponga un Actor che incapsuli il comportamento di un Consumer di Kafka. Questa entità permetterebbe asincrona/elaborazione simultanea di un numero limitato di messaggi. Per utilizzare questo tipo di consumatore occorrerà dotarlo di un ActorRef che riceva i messaggi di Kafka e li processi. L'offset del Consumer potrebbe essere rappresentato dallo stato dell'Actor, in questo modo si adegua il comportamento a seconda dei messaggi letti.

Si possono fondere gli esempi in uno unico, semplicemente riprendendo il sistema di elaborazione dati risultante dall'unione di Kafka con Storm e posizionare, alla fine del sistema, degli Actors che rappresentino i Consumers che ricevono i messaggi in modo da rendere il processo scalabile ed ancor più veloce.

Esistono, sicuramente, altre realtà ipotizzabili che non vengono approfondite in questo elaborato, in quanto sono stati affrontati quelli, che secondo l'opinione di chi scrive, risultano essere i più interessanti.

Concludo questa tesi rivolgendomi al lettore in prima persona ed esprimendo il mio parere riguardo a quanto affrontato finora. È stato molto difficile per me decidere quale sia il migliore tra i quattro, all'occorrenza ho notato che Akka si è dimostrato molto utile in diverse situazioni, come incorporare sottosistemi indipendenti (scritti in Scala) tra loro od effettuare chiamate asincrone di servizi che potrebbero fallire. Mi permetto quindi di eleggerlo come il software più vantaggioso, in quanto permette di risolvere i casi d'uso più frequenti.

Comunque, studiando Finagle si è ampliata la mia logica di programmazione di reti, al punto da poter individuare alcune casistiche nelle quali può dimostrarsi utile. Ho trovato molto interessanti e stimolanti anche Kafka e Storm in quanto offrono l'opportunità di mettersi alla prova con sistemi di messaggistica, alcuni dei quali si utilizzano giornalmente (come Linkedin). In questa sede non mi permetto di giudicare negativamente nessuna delle tecnologie, tengo comunque a sottolineare la difficoltà di individuare per Kafka, Storm e Finagle casi di uso diversi dal loro scopo primario. Akka, invece, può essere utilizzato in più ambiti tra cui anche problemi comuni di programmazione.

Bibliografia

Programming in Scala, First Edition by Martin Odersky, Lex Spoon, and Bill Venners

Sitografia

Art. *Eight hot technologies that were built in scala*: typesafe.com/blog/eight-hot-technologies-that-were-built-in-scala

Finagle: twitter.github.io/finagle/

Akka: akka.io

Kafka: kafka.apache.org

Storm: storm.apache.org