Ricordando che:

- The best synchronization technique consists in designing our system so as to avoid the need for synchronization in the first place.

---

1 Architettura generale

- L'attività di scheduling **non** deve essere un'attività relegata ad un solo processo. Una architettura gerarchica del *controller* non è sostenibile poiché:
  - Non è scalabile orizzontalmente
  - Single point of failure
  - Traffico di rete intenso.
- L'architettura del sistema sarà **flat** (***forse*** unstructured P2P).
- Il resource owner è colui che crea e mette a disposizione le coreografie
- Viene introdotto il concetto di responsabilità. Un nodo A si dice **responsabile** delle attività di scheduling per le invocazioni delle composizioni/funzioni serverless associate al resource owner X, se contiene un processo (o thread) p, chiamato **processo scheduler**, che, avendo tutte le strutture dati necessarie e la conoscenza riguardo le composizioni di X, riesce a schedulare e gestire le invocazioni degli utenti delle composizioni di X.
- **Non è necessaria alcuna forma di coordinazione tra i nodi riguardo le attività di scheduling**. Ogni processo scheduler in esecuzione nei vari nodi del sistema lavora indipendentemente (a prescindere da come lo implementiamo).
- In linea di principio deve esistere un **processo scheduler per ogni resource owner.** Più precisamente, se esiste almeno una richiesta di invocazione di una coreografia appartenente ad un resource owner X, deve esistere nel sistema un processo scheduler associato al resource owner X.
  - Eventualmente, se non si registrano invocazioni di composizioni del resource owner X per un "*certo tempo*", è possibile deallocare le risorse associate e il nodo perde la responsabilità della g
- I processi scheduler sono **RISORSE**. Hanno uno **STATO** e una **GUID** (univocamente identificabili a liello GLOBALE)!!!!
- Quando un nodo A riceve una richiesta di esecuzione di una coreografia associata al resoruce owner X, se A è il responsabile delle risorse di X, ovvero contiene il processo scheduler di X, allora accoda la richiesta nello scheduler secondo l'algoritmo implementato.

- Quando un nodo A riceve una richiesta di esecuzione di una coreografia associata al resoruce owner X, se A **NON** è il responsabile delle risorse di X, allora **ESEGUE UN LOOKUP per sapere chi è il responsabile e gli inoltra la richiesta.**

  - Se il lookup da esito negativo, ovvero non esiste ancora un responsabile per X, il nodo che riceve la richiesta diventa automaticamente il responsabile. Quindi allocherà risorse per gestire la richiesta ricevuta.

2 Migrazione degli scheduler

- Per garantire bilanciamento del carico nel sistema, si dovrebbe fare in modo che ogni nodo gestisca un numero pressoché uguale di processi scheduler. Tuttavia, non essendo i processi tutti uguali (esisteranno delle differenze), i nodi dovrebbero supportare un numero equo di richieste.

- I nodi **MENO** sovraccarichi hanno la facoltà di richiedere processi scheduler a nodi vicini. Se si accorge che un nodo vicino è particolarmente carico, richiede una migrazione. Un nodo non può richiedere la migrazione di un processo scheduler se tale processo è già sottoposto a migrazione da parte di un altro nodo. (DA VEDERE... NON E' DETTO CHE DEBBANO ESSERE I NODI MENO CARICHI A RICHIEDERE UNA MIGRAZIONE)

- I processi scheduler hanno la facoltà di migrare da un nodo all'altro secondo due forme di necessità:

  - Bilanciamento carico
  - Posizione geografica. Il processo scheduler associato alla gestione delle composizioni del resource owner X, dovrebbe essere posizionato "vicino" ai richiedenti!!

- Per conoscere lo stato dei nodi vicini, è previsto l'uso di un protocollo di **GOSSIPING** (quello **probabilistico**) attraverso cui ogni nodo dovrebbe riuscire ad ottenere una **visione LOCALE** del sistema.

- Ogni nodo agisce in modo PROATTIVO (con tutti gli svantaggi e i vantaggi del caso) per cui ad intervalli regolari e a bassa frequenza invia ai suoi vicini informazioni sul loro stato (GUID degli scheduler e numero delle richieste pendenti associate a ciasucna di esse che sta gestendo).

  - In realtà, per ottimizzare la propagazione dell'informazione, supponiamo di avere i nodi A e B dove A è vicino al nodo C il quale non è vicino al nodo B.... in sostanza A inoltra anche le informazioni di B verso C.

- Quando avviene una migrazione si da sempre priorità alla migrazione di processi scheduler aventi il MINOR NUMERO di richieste da processare presso il nodo.

- **Un nodo non può migrare un processo scheduler se ne ha uno solo.**
- **Un nodo non può migrare un processo scheduler se la somma delle richieste che sta processando più quella delle richieste del processo scheduler in esecuzione sul nodo sovraccarico è simile a quella del nodo preso in esame anche se hanno meno processi scheduler**
- E' una sorta di Information Sharing Pattern (??)

3 Ulteriori possibilità alla Migrazione degli scheduler

- Ogni nodo tiene traccia, in maniera totalmente indipendente, del numero di richieste associate alle risorse del resource owner X. Se viene a sapere che un nodo vicino possiede lo scheduler di X, richiede la migrazione di X presso di lui IN MODO DA OTTIMIZZARE IL DELAY DI RETE. (il PULL DEI PROCESSI SCHEDULER)
- SI PRESUPPONE CHE OGNI NODO ABBIA UGUALI CARATTERISTICHE IN TERMINI DI POTENZA COMPUTAZIONALE E CHE VENGANO GESTITE DA UNA STESSA AUTORITY
- LA MIGRAZIONE E DI TIPO IBRIDA. Il nodo di destinazione prima alloca tutte le risorse, DIVENTA IL RESPONSABILE, e SOLO DOPO incomincia a ricevere LE NUOVE richieste associate ad al resource owner X. Il vecchio nodo è tenuto a smaltire tutte le richieste prese in carico prima di deallocare tutte le risorse. Quando il vecchio nodo ha smaltito tutte le richieste si considera compiuta la migrazione tuttavia è possibile eseguire una nuova migrazione anche qualora quella precedente associata ad X non è stata completata. (PER ORA!!! NON NE SONO SICURO)
- TUTTE LE informazioni NECESSARIE PER EFFETTUARE LO SCHEDULING DEVONO TROVARSI SUL NODO PRIMA DI DIVENTARE IL NUOVO RESPONSABILE (strutture dati, thread vari...)

3.1 Migrazione dei "Processori" (o Migrazione di capacità) Proattiva

- Ogni provider serverless impone un limite n riguardante il numero di invocazioni. che per noi è arbitrario. E' ragionevole aspettarsi che aumenti.
- Se aumenta "troppo" il design precedente tende a sovraccaricare il nodo responsabile dello scheduler del resource owner di X.
- L'idea quindi è implementare una forma di migrazione della capacità di calcolo a disposizione (o processori).
- Possono esistere più nodi responsabili di X, ma la dimensione (o capacità) delle code che implementano i loro scheduler sono solo una frazione del totale ammesso dai provider.

- Il problema del lookup quando arriva una nuova richiesta viene gestito in questo modo:
  - Se il nodo di arrivo è responsabile di X, il nodo accetta la richiesta e la schedula.
  - Se non lo è, ed il QoS della richiesta è tempo di risposta, allora la inoltra al nodo responsabile più vicino (in termini di rete)
  - Se non lo è ed il Qos della richiesta è di tipo costo, allora la inoltra allo scheduler meno carico.
- La migrazione di capacità avviene in modo **proattivo** tenendo conto del carico. **Idealmente, ogni nodo dovrebbe conservare uno storico delle richieste in tipo ed in quantità per richiedere ai nodi capacità e prepararsi ad un imminente picco di richieste**.

4 Non c'è un singolo scheduler per *resource owner*

- Un *resource owner* potrebbe usare implementazioni di composizioni appartenenti a differenti provider (lambda, IBM etc.). Ogni provider può gestire le funzioni serverless in modo diverso ed avere LIMITI diversi. (**tetto massimo di invocazione di funzioni globale VS tetto massimo di invocazioni per funzione**)
- Per riflettere ciò, ogni nodo gestisce esattamente un numero di processi scheduler, associati TUTTI ad un singolo resource owner, che è pari al numero di piattaforme supportate. (ESEMPIO: supporto IBM e AWS.. ogni nodo gestisce due processi scheduler per resource owner: uno per schedulare funzioni su IBM e l'altro su AWS... LE COMPOSIZIONI POSSONO AVERE TRANQUILLAMENTE IMPLEMENTAZIONI DI ENTRAMBI I PROVIDER)
- Questo è vero se e solo se il resource owner usa effettivamente tutte le piattaforme supportate...PUOI AVERE ANCHE UN SOLO NODO.. PER CUI OGNI NODO GESTISCE UN NUMERO DI SCHEDULER PARI AL NUMERO DEI SERVIZI SERVERLESS EFFETTIVAMENTE UTILIZZATI dal resource owner.

5 OTTIMIZZAZIONI

- Per evitare di allocare strutture dati uguali ogni volta ad ongi invocazione di una corerografia si usano delle cache.
- Ogni nodo a delle proprie cache ed agiscno in maniera indipendente.
- C'è una cache per ogni processo scheduler.
- lo scheduler è implementato con alberi splay (o forse AVL o B-TREE)
- L'accesso alla cache è possibile solo da parte di un solo thread. Non è necessaria alcuna forma di sincronizzazione.

6 Tolleranza ai guasti

- Il design garantisce di per se Maggior tolleranza ai guasti
- La tolleranza ai guasti dei nodi viene raggiunta attraverso uno schema simile a bittorent. In realtà dovremmo abbiamo dei cluster di nodi. Ogni cluster è il responsabile di X. Ogni cluster è organizzato gerarchicamente e, attraverso la elezione di un nodo leader (SUPERPEER), egli gestisce la computazione (POI replicazione attiva)
- CI SONO ALTERNATIVE? (Certo che ci sono...forse i log?)

7 Caratteristiche

- Scalabile orizzontalmente.
- NON E' UN'APPLICAZIONE EDGE-NATIVE MA EDGE-ACCELERATED (per ora...)
- Elegante :)

8 PROBLEMI

- Le informazioni sugli utenti e sulle coreografie sono CENTRALIZZATE su un cluster di zookeeper (CENTRALIZZATO)
- L'OPTIMIZER è anche esso CENTRALIZZATO. ((E VA BENE COSì PER ORA)
- come RISOLVERE IL LOOKUP????????????????????? (CHORD????? **ma non va bene perché l'overlay non tiene conto della posizione geografica**) CONCETTO DI ULTRAPEER ??? (vedi bittorrent)

9 Scheduler

- Scheduler multilivello? Simile a LINUX SCHEDULER???

10 Peer Node

Proposed

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems is to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

Relating to a specified function choreography X belonging to resource owner R, a peer P of our system can be in one of the following states:

Active State When P has been marked as responsible for manage all invocation requests of X forwarded by end users.

Forwarder State Otherwise

function choreographies (FCs) or workflows of functions.

As known, in server-less computing platforms, computation is done in **function instances**. These instances are completely managed by the server-less computing platform provider (SSP) and act as tiny servers where a function is been executed.

11 Server-less Scheduler

In this section, we will describe the very important notion of *server-less scheduler*.

First of all, Let R a resource owner and R_X a set of server-less functions already defined and deployed by R on an *unique* server-less computing platform provider; we assume that $|R_X| \geq 1$.

Lastly, let k the max number of function instances, executable at the same time on the server-less computing platform provider, which are available to execute any function x_j, where $1 \leq j \leq n$, belonging to R_X

Then, it is said that a *server-less scheduler* $S_{(R_X,m)}$ represents a queuing system, implementing any scheduling discipline, equipped with m so-called *virtual function instance*, where $m \leq k$, which aim is to perform scheduling activity managing any function belonging to R_X. The parameter m is also called *scheduler capacity*.

A virtual function instance represents a function instances on the server-less computing platform provider which is *virtually* owned by that scheduler. In other word, m represents the max number of server-less functions, belonging to R_X, which a given scheduler $S_{(R_X,m)}$ can effectively invoke and perform using the server-less computing platform provider.

11.1 Proprieties and constrains regarding *scheduler capacity*

According to proposed solution, a scheduler capable to manage any function belonging to R_X, if exist, is *not* unique, although it is unique inside a peer node.

In order to achieve better performance in terms of network delay experienced by end users, fault tolerance and load balance, any peer nodes can hold a $S_{(R_X,m)}$ scheduler in order to manage incoming request sent by several users spread in different geographic regions.

However, despite there is no upper bound to the number of schedulers of type $S_{(R_X,m)}$ existing at the same time in our system, there is a limitation regarding the scheduler capacity of each existing scheduler.

To be more precise, let's say that, globally, a there are a set of schedulers $S_{1,(R_X,m_1)}, \ldots, S_{p,(R_X,m_p)}$ exist at the same time, where $p \in \mathbb{N}$ with $p \geq 1$. Following constrains must be hold:

$$\sum_{i=1}^{p} m_i \leq k \tag{1}$$

In other words, the sum of all scheduler capacities must be less or equal to the max number of function instances executable at the same time on the server-less computing platform provider for a given R_X set.

Is very important to make clear that different server-less computing platform providers may establish limits, regarding the max number of function instances executable at the same time, in different way; some of them imposed limits per-account, others per-functions. Our design can accommodate both approaches because:

- If a provider imposed limits per-account, then R_X will contains all functions defined and deployed by R, then k will be represent the provider's global limit.
- If a provider imposed limits per-function, then R_X will contains only one function belonging to R, therefore k will be represent the provider's per-function limit.

12 Function choreography

Let $t \in \mathbb{N}$ with $t \geq 1$, a **function choreography** $FC_{(R_{X_1},\ldots,R_{X_t})}$ is a set of server-less functions sets.

of $n \geq 1$ of server-less functions sequences R_{X_j} belonging to R, where any server-less function is connected each other, eventually using data- and control-flow operator in order to construct the work-flow of a generic application.

12.1 Scheduler behaviour

object with only one limitation:

Suppose that globally there are a set of schedulers $S_{1,(R_X,m_1)},\ldots,S_{p,(R_X,m_p)}$, where $p \in \mathbb{N}$ with $p \geq 1$

To be more precise, when a function x_j must to be execute, let s the current number of busy virtual instances, one of the following events may occur:

1. if $s < m$, the scheduler invoke directly the function x_j on the provider.
2. if $s = m$, the scheduler delay the execution of the function x_j on the provider according to implemented scheduling discipline.

A **function choreography**, also called as compound server-less function, are a set of

Let R a resource owner and R_x its function choreography made up of $R_{x_1}, R_{x_2}, \ldots, R_{x_n}$ unique server-less functions; it is said that a peer node P is **responsible** for R_x when it contains a sequence of schedulers $S_{R_1}, S_{R_2}, \ldots, S_{R_k}$ with $k \leq n$, belonging to R, capable to invoke all server-less function belonging to R_x.

It is said that a

Depending on the definition of the function choreography provided by R and the unique characteristics of back-end server-less providers which execute all serverless functions R_{x_n} of

It is said that a scheduler S is capable to invoke a server-less function when , a scheduler S can invoke multiple

When a peer A, placed "*at the edge*" of the network, receives a new request of invocation for X by an end user, it performs following task in that order:

1. If it responsible It check for it is an already an *active peer* to manage

has found the tracker for a file F, the tracker returns a subset of all the nodes currently involved in downloading F.