1 Super-peer Node

An *end-user* represents, instead, a third-party application that wants execute one or more function choreographies.

2 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.

3 Resources

4 System's resources and actors

4.1 Resource owner

A *resource owner*, henceforward denoted with R, represents an entity capable of *creating*, *modifying* and *authorizing* access to several resources of our system.

Given a resource owner R, there are two type of resources which he can manage:

1. Function choreographies.
2. Server-less function implementations (also called *concrete server-less functions*)

4.2 Function choreographies

A *function choreography* is the most important resource of our system and it is used to model both server-less functions and server-less function compositions.

Informally, it represents, using graph notation, all paths that might be traversed through a server-less function composition during its execution. In other words, it describes calling relationships between server-less functions.

Formally, being R the resource owner, a **function choreography**, denoted as FC_R, is a *control-flow graph* $G(V, E)$, where:

- Each node $v \in V$, called *abstract control-flow node*, represents a generic function of a computer program.
- Each edge $(v_i, v_j) \in E$, for any $i, j \in \mathbb{N}$ with $i \neq j$, indicates that the generic function v_i calls that in v_j.

Any function choreography can be uniquely identified by an ordered string pair (a, b), where a is the *resource owner name* while b is the *function choreography name*.

There are two types of abstract control-flow nodes:

Abstract server-less function denoted as $f_{(abstract,i)}$ while the set of all server-less functions is denoted as $F_{abstract}$.

Control-flow function denoted ad c_t while C is the corresponding set of all control-flow functions.

Formally, let $|V| = n$, $|F_{abstract}| = t$, and $|C| = k$, where $t, k \in \mathbb{N}$ with $t + k = n$, we have that:

$$\begin{aligned} V &\stackrel{def}{=} F_{abstract} \cup C \\ &= (f_{(abstract,1)}, \ldots, f_{(abstract,t)}) \cup (c_1, \ldots, c_k) \end{aligned} \tag{1}$$

Clearly, we said that function choreography models a server-less function, when $|V| = 1$ and $|E| = 0$; conversely, it models a server-less function composition, when $|V| > 1$ and $|E| > 0$.

4.2.1 Abstract server-less function

An *abstract server-less function* represents a descriptions of one or more corresponding concrete server-less function implementations. That description includes:

- TODO
- TODO
- TODO

Any abstract server-less function can be uniquely identified by an ordered string pair (a, b), where a is the *resource owner name* while b is the *abstract server-less function name*.

4.2.2 Concrete server-less function

Given an abstract server-less function, a resource owner can provide different implementations which, although they must be semantically equivalent, may eventually expose different performance or cost behaviour.

Therefore, we call *concrete server-less function* any implementation of a given abstract function and it is uniquely identified by an ordered string tuple (a, b, c), where a and b represent, like before, the *resource owner name* and the *abstract function name* respectively, while c represents the so-called *function type*, which is an abstract descriptions of the corresponding function implementation.

4.2.3 Control-flow operator

Like `if`, `for`, etc.

4.3 Server-less function swarms

Informally, a so-called *server-less function swarm* represent a set of concrete server-less function with very specific properties.

Precisely, let $l \in \mathbb{N}$ such that $l \neq 0$, R a resource owner, P a server-less computing platform provider and X_R a set of concrete server-less functions. Moreover, let $\mathbf{X}_R$ the set containing all concrete function implementations defined and deployed by R on any provider.

A set $X_R \subseteq \mathbf{X}_R$ is called a *server-less function swarm*, or simply *swarm*, if:

1. $|X_R| = n \geq 1$, that is X_R must contain at least one concrete function.

2. X_R contain concrete functions that share the same platform provider P where they will be executed.

3. X_R contain concrete functions that share the same limit l in term of max number of server-less function instance runnable at the same time by the corresponding platform provider P. That limit is also called *server-less function swarm's concurrency limit*, or simply, *concurrency limit*.

Is very important to make clear that only at most l concrete functions belonging to X_R can be executed simultaneously by P. The value of l depends by specific policies adopted by P; some of them imposed that limit *per-account*, others *per-functions*. Our model supports both approaches because:

- If P imposed limits *per-function*, then $|X_R| = 1$, that is, X_R will contain only one function defined and deployed by R in P, where l will be represent the provider's per-function limit.

- If P imposed a limit *per-account*, then generally $|X_R| \geq 1$ and it include all concrete server-less function deployed on P while l will be represent the provider's global limit.

4.3.1 Server-less function sub-swarms

A sub-swarm of X_R, which we will denote with Δ_{X_R}, is the term with which we denote any element belong to the power set[1] of X_R, excluding the empty set. Formally, any $\Delta_{X_R} \in \mathcal{P}(X_R) \setminus \oslash$ is a sub-swarm.

Is very important to remember that in our model any sub-swarm Δ_{X_R} of X_R has the *same* concurrency limit of X_R.

[1]The *power set* $\mathcal{P}(S)$ of a set S is the set of all subsets of S, including the empty set and S itself.

5 Function choreography scheduler

5.1 Schedulability condition

Let FC_R a function choreography belonging to a resource owner R and $F_{abstract}$ its server-less abstract functions set. Moreover, be $\mathbf{X}_R$ the set containing all functions deployed by R in any provider.

In order to effectively start the execution of a function choreography, is required that for each abstract function $f_{abstract} \in F$ *at least one* concrete function f, which implements it, exists.

Formally, a function choreography is said *schedulable* when:

$$FC_R \text{ is schedulable } \Leftrightarrow \begin{array}{c} \forall f_{abstract} \in F_{abstract} \\ \exists f \in \mathbf{X}_R \mid f \text{ implements } f_{abstract} \end{array} \tag{2}$$

Although it is correct, the condition expressed by equation 2 is not very precise, because $\mathbf{X}_R$ can contain some functions that doesn't implement any $f_{abstract} \in F_{abstract}$.

Therefore, we define Ω_{FC_R} as the set containing only concrete functions that are needed to execute FC_R, which can both to belong to any provider and to have different concurrency limits.

Since multiple implementations of a same abstract function can exist at the same time, we can exploit the notion of swarm and sub-swarm to formally define the set Ω_{FC_R}.

Let $n \in \mathbb{N}$, such that $n \geq 1$, and X_{R_i} the i-th swarm and $\Delta_{X_{R_i}}$ its sub-swarm which contains only concrete functions implementing one or more $f_{abstract} \in F_{abstract}$, where $1 \leq i \leq n$.

We define Ω_{FC_R} as follows:

$$\begin{array}{c} \Omega_{FC_R} \overset{def}{=} \Delta_{X_{R_1}} \cup \ldots \cup \Delta_{X_{R_n}} = \bigcup_{i=1}^{n} \Delta_{X_{R_i}} \\ \text{where} \\ \Delta_{X_{R_i}} \cap \Delta_{X_{R_j}} = \oslash \quad \text{ for } i, j \in [0, n] \mid i \neq j \\ \forall f \in \Delta_{X_{R_s}} \quad f \text{ implements } f_{abstract} \text{ for } s \in [0, n] \end{array} \tag{3}$$

Since belong to different swarms, please note that any $\Delta_{X_{R_i}}$ and $\Delta_{X_{R_j}}$, for any $i \neq j$, can belong to the same provider but they cannot share the same concurrency limit.

Generally the schedulability condition for FC_R can be written as follows:

$$FC_R \text{ is schedulable } \Leftrightarrow \exists \Omega_{FC_R} \tag{4}$$

5.2 The Δ_{X_R}-Scheduler

Let R a resource owner, P the server-less provider and Δ_{X_R} a sub-swarm of a X_R, where k its concurrency limit.

Let $m \in \mathbb{N}$, a Δ_{X_R}*-Scheduler*, denoted as $S_{(\Delta_{X_R}, m)}$ represents a queuing system, implementing any scheduling discipline, equipped with m so-called *virtual function instance*, where $m \leq k$.

Its aim is to decide when and which function, belonging to Δ_{X_R}, must be performed on P.

The parameter m is also called *scheduler capacity.*

5.2.1 Virtual function instance

A *virtual function instance* represents a real function instances, clearly belonging to the server-less computing platform provider, which is *virtually* owned by $S_{(\Delta_{X_R},m)}$.

Therefore, m represents the max number of server-less function instances usable simultaneously by $S_{(\Delta_{X_R},m)}$.

5.2.2 Proprieties and constrains

According to our model, a Δ_{X_R}-scheduler capable to manage any function belonging to Δ_{X_R}, 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 Δ_{X_R}-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 Δ_{X_R}-schedulers existing at the same time in our system, there is a limitation regarding the scheduler capacity of each existing scheduler.

Let's start summarizing all rules regarding Δ_{X_R}-schedulers:

1. All peer node of our system can hold a Δ_{X_R}-scheduler object.

2. Each node can hold only one instance of type Δ_{X_R}-scheduler.

3. Let $n \in \mathbb{N}$ such that $n \geq 1$, suppose that our system contains n peer nodes holding a Δ_{X_R}-scheduler.

   To be more precise, let's say that a sequence $S_{(1,(\Delta_{X_R},m_1))}, \ldots, S_{(i,(\Delta_{X_R},m_n))}$ exist at the same time in our system, where $S_{(i,(\Delta_{X_R},m_i))}$ represent the Δ_{X_R}-scheduler owned by i-th node having scheduler capacity equal to m_i.

   Following constraint must be hold:

$$\sum_{i=1}^{n} m_i \leq k \tag{5}$$

   where $k \in \mathbb{N}$, with $k > 0$, is the concurrency limit of the swarm X_R.

   Remember that any sub-swarm Δ_{X_R} share the same concurrency limit of X_R. Therefore, equation 5 states that, the sum of all scheduler capacities which manage the functions belonging to Δ_{X_R}, must be less or equal to the max number of function instances executable at the same time on the server-less computing platform provider.

5.3 The FC_R-Scheduler

To support hybrid-scheduling, that is the ability to execute multiple concrete function implementations belonging to different providers or subjected to different concurrency limit, in order to select the most suitable concrete function implementation according to a given QoS, unfortunately only one "*scheduler*" is not enough.

We call *FC_R-Scheduler* a set of *Δ_{X_R}-Schedulers* where $\Delta_{X_R} \in \Omega_{FC_R}$. Is always required that $|FC_R| \geq 1$, that is, at least one scheduler must exist.

6 FC_R-Active Peer Node

According to our model, in order to effectively invoke all server-less concrete function belonging to a function choreography FC_R, is required that a peer node is "active".

We said that a peer node A is FC_R-*active peer node*, or, simply, *active*, when it holds a Δ_{X_R}-scheduler for any sub-swarm in Ω_{FC_R}. Formally:

$$A \text{ is } FC_R\text{-active peer node } \Leftrightarrow \forall \Delta_{X_R} \in \Omega_{FC_R} \quad \exists S_{(\Delta_{X_R},m)} \text{ hold by } A \tag{6}$$

Multiple nodes can be active at the same time. Any node perform its scheduling decision independently.

6.1 FC_R-Request

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.

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.