References

[1] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 129–138, July 2020.

[2] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[3] Xiongfei Weng, Hongliang Yu, Guangyu Shi, Jian Chen, Xu Wang, Jing Sun, and Weimin Zheng. Understanding locality-awareness in peer-to-peer systems. In *2008 International Conference on Parallel Processing - Workshops*, pages 59–66, 2008.

Contents

1 Serverless Computing Paradigm **4**
1.1 Serverless Function 4
1.2 Serverless Application 4

2 Orchestrator Model **4**
2.1 Resource owner 4

3 Serverless Choreography **4**
3.1 Definition 5
3.1.1 Notations 5
3.2 Abstract Serverless Function 6
3.2.1 Special Abstract Serverless Functions 7
3.3 Serverless Choreography Configuration 8
3.3.1 Serverless Functions Configuration 8
3.3.2 Serverless Choreography Configuration 8
3.4 Metrics 9
3.5 Serverless Sub-choreography 9
3.6 Structures 9
3.6.1 Parallel 10
3.6.2 Branch 10

4 Optimization **11**
4.1 Performance Modeling 11

5 Super-peer Node **11**

6 Peer Node **12**

7 Resources **12**

8 System's resources and actors **12**
8.1 Server-less function swarms 12
8.1.1 Server-less function sub-swarms 13

9 Function choreography scheduler **13**
9.1 Schedulability condition 13
9.2 The Δ_{X_R}-Scheduler 14
9.2.1 Virtual function instance 14
9.2.2 Proprieties and constrains 14
9.3 The FC_R-Scheduler 15

10 FC_R-Active Peer Node **15**

11 Architecture overview **15**
11.1 Overlay network 16
11.1.1 Locality-awareness property 16
11.2 Fault Tolerance 16
11.2.1 Availability 16
11.2.2 Replication 17
11.3 FC_R-Request 17

12 Queuing system of a Δ_{X_R}-scheduler **19**
12.1 Server-less function preemption 19
12.2 Queuing system design 19
12.3 The virtual function instances allocation problem 20
12.3.1 Reactive scaling policy 20
12.3.2 Server-less function scheduling for QoS "Minimum Response Time" 21

fdsfsfsdfsdfsd f sd fsd f sdfsd fsdfsdtertertertereterterterterterterterter

$$
\begin{aligned}
\min \quad & \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} g(x)^{f_k^i} Y_x^{f_k^i} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} T(x)^{f_k^i} Y_x^{f_k^i} \quad \leq \quad D_{\mathbf{F}} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} Y_x^{f_k^i} \qquad\quad = \quad 1 \\
& Y_x^{f_k^i} \in \{0,1\}
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
\min \quad & \sum_{\pi\in\Pi}\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} g(x)^{f_k^i} Y_{(\pi,x)}^{f_k^i} \\
& \sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} T(x)^{f_k^i} Y_x^{f_k^i} \qquad\quad \leq \quad D_{\mathbf{F}} \\
& \sum_{\pi\in\Pi}\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{x\in C} Y_{(\pi,x)}^{f_k^i} \qquad\quad = \quad 1 \\
& Y_{(\pi,x)}^{f_k^i} \in \{0,1\}
\end{aligned}
\tag{2}
$$

1 Serverless Computing Paradigm

In serverless computing platforms, computation is done by so-called *function instances* which are completely managed by the serverless computing platform provider and act as tiny servers that can be invoked based on events forwarded by end-users [2].

Serverless computing platforms handle almost every aspect of the system administration tasks needed to deploy a workload on the cloud, providing a new simplified programming model according to which developers can focus on the business aspects of their applications only [1].

Moreover, the paradigm lowers the cost of deploying applications too, adopting a so-called "*pay as you go*" pricing model, by charging for execution time rather than for allocated resources [1].

1.1 Serverless Function

A *serverless function* represents a stateless, event-driven, self-contained unit of computation implementing a business functionality.

Although a serverless function generally represents a unit of executable code, submitted to FaaS platforms by developers using one or a combination of the programming languages supported by FaaS providers, a serverless function can be any cloud services eventually necessary to business logic, like cloud storage, message queue services, pub/sub messaging service etc.

When it represents executable code, developers can specify several configuration parameters, like timeout, memory size or CPU power [1].

A serverless function can be triggered through events or HTTP requests following which the FaaS provider executes it in a containerized environment, like container, virtual machine or even processes, using the specified configuration.

1.2 Serverless Application

A *serverless application* represents a stateless and event-driven software system made up of a set of serverless functions hosted on one or more FaaS platforms and combined together by an *orchestrator*, which handles events among functions and triggers them in the correct order according to the business logic defined by developers.

2 Orchestrator Model

2.1 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.

3 Serverless Choreography

According to our framework, a *serverless choreography* is a *resource* used to model and implement both serverless functions and serverless applications.

Informally, a serverless choreography is derived by the control-flow graph which describes, using graph notation, all paths that might be traversed through a serverless application during its execution. Similarly, a serverless choreography describes calling relationships between serverless functions which can be combined using several types of control-flow connectors.

3.1 Definition

Formally, let $n \in \mathbb{N} \setminus \{0\}$, being R the resource owner, a *serverless choreography*, or simply *choreography*, $\mathcal{S}_R$ owned by R is a weighted directed graph defined as follows:

$$\mathcal{S}_R \stackrel{def}{=} (\Phi, E) \tag{3}$$

where:

- Φ represents a finite set of vertices such that $|\Phi| = n$.

  Each vertex $\phi \in \Phi$ is called *abstract serverless function* which represents, generally, a computational unit; we will describe more in detail what we mean by abstract serverless function shortly;

- $E \subseteq \Phi \times \Phi$ represents a finite set of directed edges.

  For all $i, j \in [1, n]$, each directed edge from $\phi_i \in \Phi$ to $\phi_j \in \Phi$, denoted as (ϕ_i, ϕ_j), indicates that the abstract serverless function ϕ_j *can* be called by ϕ_i, representing the interaction between ϕ_i and ϕ_j defined by the business logic;

- For all $i, j \in [1, n]$, the number $p_{ij} \in [0, 1]$ is the weight assigned to the edge (ϕ_i, ϕ_j) and represents the *transition probability* from ϕ_i to ϕ_j, that is the probability of invoking ϕ_j after finishing the execution of ϕ_i.

  $P : \Phi \times \Phi \to [0, 1]$ is defined as the *transition probability function* according to which $P(\phi_i, \phi_j) = p_{ij}$. Clearly, when $P(\phi_i, \phi_j) = 0$, the directed edge (ϕ_i, ϕ_j) does not exist according to business logic, therefore ϕ_i cannot invoke ϕ_j;

A choreography $\mathcal{S}_R$ can be uniquely identified by an ordered pair (a, b), where a is the name of the resource owner R, while b is the function choreography name.

Clearly, we say that choreography models a serverless function when $|\Phi| = 1$ and $|E| = 0$; conversely, it models a serverless function application when $|\Phi| > 1$ and $|E| > 0$.

From now, a choreography $\mathcal{S}_R$ will be briefly denoted by $\mathcal{S}$ when no confusion can arise about the resource owner R.

3.1.1 Notations

Let $\mathcal{S} = (\Phi, E)$ a choreography. Let's now introduce following notations and functions:

- A *path* π is defined as a finite sequence of distinct vertices and edges as follows:

$$\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n \tag{4}$$

where:

- $\phi_i \in \Phi$, for all $i \in [1, n]$
- $e_i = (\phi_i, \phi_{i+1}) \in E$, for all $i \in [1, n-1]$

- Given a path π, we define the *transition probability* of the path π the following quantity:

$$TPP(\pi) = \prod_{i=1}^{n-1} P\left(\phi_i, \phi_{i+1}\right) \tag{5}$$

- Let $\phi_i \in \Phi$ to $\phi_j \in \Phi$ for all $i, j \in [1, n]$, the set $\Pi(\phi_i, \phi_j)$ identifies all possible paths starting from vertex ϕ_i and ending at vertex ϕ_j.

- The set $out(\phi_u)$ ($in(\phi_u)$) denotes all edges starting (ending) from (to) vertex ϕ_u, while the set $succ(\phi_u)$ ($pred(\phi_u)$) includes all direct successor (predecessors) vertices of ϕ_u. Formally:

$$out(\phi_u) \stackrel{def}{=} \{(\phi_u, \phi) \in E, \quad \forall \phi \in \Phi\} \tag{6}$$

$$in(\phi_u) \stackrel{def}{=} \{(\phi, \phi_u) \in E, \quad \forall \phi \in \Phi\} \tag{7}$$

$$succ(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi_u, \phi) \in out(\phi_u)\} \tag{8}$$

$$pred(\phi_u) \stackrel{def}{=} \{\phi \in \Phi \mid (\phi, \phi_u) \in in(\phi_u)\} \tag{9}$$

- $NI : \Phi \to [0, \infty)$ is a function representing the average number of invocations of an abstract serverless function $\phi \in \Phi$ during the execution of a serverless choreography $\mathcal{S}$.

- $D : \Phi \times \Phi \to [0, \infty)$ represents a delay function according to which $D\left(\phi_i, \phi_j\right)$ identifies the delay from ϕ_i to ϕ_j due to network delay or orchestration task;

3.2 Abstract Serverless Function

Supposing that a choreography $\mathcal{S} = (\Phi, E)$ is given, an *abstract serverless function* $\phi \in \Phi$, or simply *abstract function*, is a *resource* that represents a computational unit needed by business logic.

According to our model, there are two types of abstract functions implementations:

- ϕ is defined as *serverless executable functions*, or simply *executable function*, when ϕ contains executable code which implementing business logic.

  $\mathcal{F}_{\mathcal{E}}$ is defined as the set containing all executable function of $\mathcal{S}$ and it is formally defined as follows:

$$\mathcal{F}_{\mathcal{E}} \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless executable function } \} \tag{10}$$

However, multiple different implementations of a given executable function can be provided by developers which, although they must be semantically and logically equivalent, may eventually expose different performance or cost behavior.

Therefore, for each $\phi \in \mathcal{F}_{\mathcal{E}}$, the set $\mathbf{F}_{\phi}$, containing all implementations of ϕ, generally denoted by f_{ϕ}, such that $|\mathbf{F}| \geq 1$, must exist.

We will explain later that the aim of our framework is to pick exactly one $f_{\phi} \in \mathbf{F}_{\phi}$ whose properties allow us to meet user-specified QoS objective.

We assume that every $f_{\phi} \in \mathbf{F}_{\phi}$ for all $\phi \in \mathcal{F}_{\mathcal{E}}$ is already deployed on one or more FaaS platform by developers.

- ϕ is called *serverless orchestration functions*, or *orchestration functions*, when ϕ contain orchestration code needed to decide which and how the next abstract functions must be called; in other words, orchestration code is control-flow code.

  $\mathcal{F}_{\mathcal{O}}$ is defined as the set containing all executable function of $\mathcal{S}$ and it is formally defined as follows:

$$\mathcal{F}_{\mathcal{O}} \stackrel{def}{=} \{\phi \in \Phi \mid \phi \text{ is a serverless orchestration function } \} \tag{11}$$

Clearly, based on above definition, we can say:

$$\mathcal{F}_{\mathcal{E}} \cap \mathcal{F}_{\mathcal{O}} = \emptyset \tag{12}$$
$$\mathcal{F}_{\mathcal{E}} \cup \mathcal{F}_{\mathcal{O}} = \Phi \tag{13}$$
$$|\mathcal{F}_{\mathcal{E}}| + |\mathcal{F}_{\mathcal{O}}| = |\Phi| \tag{14}$$

Any abstract function ϕ is uniquely identified by an ordered pair (a, b), where:

- a represents the identifier of the choreography $\mathcal{S}$;
- b is the name of the abstract serverless function ϕ;

3.2.1 Special Abstract Serverless Functions

According to the serverless paradigm, the execution of a serverless application starts with a particular function, which we will denote as the *entry point*, while any other serverless functions, belonging to the application, will be invoked subsequently according to specified business logic.

Clearly, the execution of a serverless application ends when ends the execution of the last function according to the business logic. That ending function will be denoted as *end point*. According to our point of view, the end point is not unique.

In order to appropriately model a serverless application, any choreography $\mathcal{S}$ has two special abstract functions:

- One acting as the entry point which is denoted by ϕ_{entry};
- Another acting as the end point and denoted by ϕ_{end};

Formally:

$$\phi_{entry} \in \Phi \text{ is the entry point of } \mathcal{S} \quad \Leftrightarrow \quad in(\phi_{entry}) = \emptyset \tag{15}$$

$$\phi_{end} \in \Phi \text{ is the end point of } \mathcal{S} \quad \Leftrightarrow \quad out(\phi_{end}) = \emptyset \tag{16}$$

3.3 Serverless Choreography Configuration

Given a choreography $\mathcal{S} = (\Phi, E)$, the basic goal of our framework is to determine the so-called *serverless choreography configuration*, also called *choreography configuration* or simply *configuration*, that allows us to meet user-specified QoS objectives.

Informally, a configuration specifies which implementation $f_\phi \in \mathbf{F}_\phi$, for each $\phi \in \mathcal{F}_\mathcal{E}$, will be effectively executed, including its parameters such as memory size or CPU power.

3.3.1 Serverless Functions Configuration

Let $\phi \in \mathcal{F}_\mathcal{E}$ an executable function and $\mathbf{F}_\phi$ the corresponding set of serverless functions which implement it.

We say that a *configuration* for ϕ is a vector $x_\phi \in \mathbf{F}_\phi \times \mathbb{N}$. Let $m \in \mathbb{N}$ and supposing that $x_\phi = (f_\phi, m)$, f_ϕ represents a particular serverless function implementing ϕ while m represents the allocated memory size of f_ϕ.

At this point, we can define some useful functions:

- $C_{\mathbf{F}_\phi} : \mathbf{F}_\phi \times \mathbb{N} \to [0, \infty)$ is the cost function for the serverless functions implementing ϕ.

  To be more precise, for all $f_\phi \in \mathbf{F}_\phi$ and $m \in \mathbb{N}$, $C_{\mathbf{F}_\phi}(f_\phi, m)$ returns the average cost paid by developers to execute f_ϕ using an allocated memory size equal to m.

- $RT_{\mathbf{F}_\phi} : \mathbf{F}_\phi \times \mathbb{N} \to [0, \infty)$ is a delay function for the serverless functions implementing ϕ.

  For all $f_\phi \in \mathbf{F}_\phi$ and $m \in \mathbb{N}$, $RT_{\mathbf{F}_\phi}(f_\phi, m)$ returns the average response time when f_ϕ is invoked with memory size equal to m;

3.3.2 Serverless Choreography Configuration

Let $n, k \in \mathbb{N}$ and a choreography $\mathcal{S} = (\Phi, E)$ such that $|\Phi| = n$ and $|\mathcal{F}_\mathcal{E}| = k$ where $k \leq n$.

Formally, a configuration for $\mathcal{S}$ is a vector $\mathbf{X}$ such that:

$$\mathbf{X} \in \{\mathbf{F}_{\phi_1} \times \mathbb{N}\} \times \ldots \times \{\mathbf{F}_{\phi_k} \times \mathbb{N}\} = \bigtimes_{i=1}^{k} \{\mathbf{F}_{\phi_i} \times \mathbb{N}\} \tag{17}$$

For all $i \in [1, k]$, supposing that $\mathbf{X} = \{x_{\phi_1}, \ldots, x_{\phi_k}\}$, x_{ϕ_i} is a configuration for ϕ_i; we will use $X(i)$ notation as a reference to the configuration x_{ϕ_i} while we will use $X(i, 1)$ and $X(i, 2)$ to denote respectively the implementation $f \in \mathbf{F}_{\phi_i}$ and the allocated memory size $m \in N$ for the function f.

3.4 Metrics

Let $\mathcal{S} = (\Phi, E)$ a choreography such that:

- $\mathcal{S}$ is a DAG without any cycles or loops;
- Is true that the transition probabilities of all paths between the entry point and the end point of $\mathcal{S}$ can sum up to 1. Formally:

$$\sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) = 1 \tag{18}$$

Suppose to have a path $\pi = \phi_1 e_1 \phi_2 e_2 \dots e_{n-2} \phi_{n-1} e_{n-1} \phi_n$ and a choreography configuration $\mathbf{X}$ for $\mathcal{S}$.

We can define the *response time* of π given $\mathbf{X}$ as follows:

$$RT_P(\pi, \mathbf{X}) = \sum_{i=1}^{n} (N(\phi_i) \cdot RT_{\phi_i}(X(i)) + \sum_{i=1}^{n-1} N(\phi_i) \cdot D(\phi_i, \phi_{i+1}) \tag{19}$$

Similarly, we define the cost of π given $\mathbf{X}$ as follows:

$$C_P(\pi, \mathbf{X}) = \sum_{\substack{1 \leq i \leq n \\ \phi_i \in \overline{\mathcal{F}}_{\mathcal{E}}}} N(\phi_i) \cdot C_{\mathbf{F}_{\phi_i}}(X(i)) \tag{20}$$

Then the response time and the cost of the choreography $\mathcal{S}$ a follows:

$$RT_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot RT_P(\pi, \mathbf{X}) \tag{21}$$

$$C_C(\mathcal{S}, \mathbf{X}) = \sum_{\pi \in \Pi(\phi_{entry}, \phi_{end})} TPP(\pi) \cdot C_P(\pi, \mathbf{X}) \tag{22}$$

3.5 Serverless Sub-choreography

For any given serverless choreography $\mathcal{S} = (\Phi, E)$, is possible to identify a *serverless sub-choreography*, or simply *sub-choreography*, denoted by S^*, which is a weighted directed sub-graph of $\mathcal{S}$ formally as follows:

$$S^* \stackrel{def}{=} (\Phi^*, E^*) \qquad \text{where } \Phi^* \subseteq \Phi \wedge E^* \subseteq E \tag{23}$$

Through $\phi^*_{start} \in \Phi^*$ and $\phi^*_{end} \in \Phi^*$ notation we denote, respectively, the *entry point* and the *end point* of the sub-choreography S^*.

3.6 Structures

Supposing to have a serverless choreography $\mathcal{S} = (\Phi, E)$, we call *structure* any sub-choreography $\mathcal{S}^* = (\Phi^*, E^*)$ whose entry point and the end point are, respectively, an opening and a closing orchestration functions of the same type. Formally:

$$S^* \text{ is a structure } \Leftrightarrow \begin{cases} \phi^*_{entry} & \in \mathcal{F}^*_{\mathcal{O}} \quad \wedge \quad \mathcal{T}(\phi^*_{entry}) = \tau_\alpha \\ \phi^*_{end} & \in \mathcal{F}^*_{\mathcal{O}} \quad \wedge \quad \mathcal{T}(\phi^*_{end}) = \tau_\omega \end{cases} \tag{24}$$

The most important aspect is that every structure can be viewed as a set of sub-choreographies of $\mathcal{S}^*$. Formally, suppose that $\Lambda = (A, B)$ is a graph such that:

$$\begin{array}{rcl} A & \stackrel{def}{=} & \Phi^* \setminus \left\{\phi^*_{entry}, \phi^*_{end}\right\} \\ B & \stackrel{def}{=} & E^* \setminus \left(out\left(\phi^*_{entry}\right) \cup in\left(\phi^*_{end}\right)\right) \end{array} \tag{25}$$

Let $c \in \mathbb{N}$ and suppose that c is the number of connected components of the graph Λ.

We denote by $\Theta_{\mathcal{S}^*}$ as the set containing all connected components of Λ every of which is considered as a sub-choreography of $\mathcal{S}^*$. Formally:

$$\begin{gathered} \Theta_{\mathcal{S}^*} \stackrel{def}{=} \{\theta_i, \ldots, \theta_c\} \\ \text{where} \\ \theta_i \stackrel{def}{=} (\Phi^{**}_i, E^{**}_i) \text{ is a sub-choreography of } \mathcal{S}^* \\ \Phi^{**}_i \subset \Phi^* \wedge E^{**}_i \subset E^* \qquad \forall i \in [1, c] \end{gathered} \tag{26}$$

3.6.1 Parallel

A structure $\mathcal{P} = (\Phi, E, \Theta)$ is called *parallel structure* when:

$$TPP(\pi) = 1 \qquad \forall \pi \in \Pi(\phi_{start}, \phi_{end}) \tag{27}$$

Let $n \in \mathbb{N}$, supposing that $|\Theta| = n$, the response time of a parallel structure is the longest response time of all sub-choreographies $\theta_i \in \Theta$ for all $i \in [1, n]$, while its cost is equal to the sum their costs of execution. Formally, being $\mathbf{X}$ a configuration of $\mathcal{P}$:

$$RT_{parallel}(\mathcal{P}, \mathbf{X}) = max\left\{RT_C(\theta, \mathbf{X}) \mid \theta \in \Theta\right\} \tag{28}$$

$$C_{parallel}(\mathcal{P}, \mathbf{X}) = \sum_{\theta \in \Theta} C_C(\theta, \mathbf{X}) \tag{29}$$

3.6.2 Branch

We say that A sub-choreography $\mathcal{B}$, such that $\phi'_{start} = \phi_i$ and $\phi'_{end} = \phi_j$, is called *branch structure* when:

$$\begin{gathered} TPP(\pi) \neq 1 \qquad \forall \pi \in \Pi(\phi^*_{start}, \phi^*_{end}) \\ \wedge \\ \sum_{\pi \in \Pi(\phi^*_{start}, \phi^*_{end})} TPP(\pi) = 1 \end{gathered} \tag{30}$$

4 Optimization

4.1 Performance Modeling

We define the utility function as follows:

$$F(\mathbf{x}) \stackrel{def}{=} w_{RT} \cdot \frac{RT_{max} - RT(\mathbf{x})}{RT_{max} - RT_{min}} + w_C \cdot \frac{C_{max} - C(\mathbf{x})}{C_{max} - C_{min}} \tag{31}$$

where $w_{RT}, w_C \geq 0$, $w_{RT} + w_C = 1$, are weights for the different QoS attributes, while RT_{max} (RT_{min}) and C_{max} (C_{min}) denote, respectively, the maximum (minimum) value for the overall expected response time and cost.

Before to proceed, when the serverless workflow configuration x is used,

- $D(\mathbf{x})$ denotes the end-to-end response time of current serverless workflow G_s when the serverless workflow configuration x is used. At the same time, $C(\mathbf{x})$ represents the
- $C(\mathbf{x})$ re

gdfgdfgdfgd

- $\mathbf{x}(\mathcal{F})$ is used to denote the configuration (f, m) of the serverless abstract function $\mathcal{F}$.
- $C_{func}(\mathbf{x}(\mathcal{F}))$ represents the cost to pay to execute the concrete serverless function f with memory size m.

$$C_{walk}(w, \mathbf{x}) = \sum_{\mathcal{F} \in w} C_{func}(\mathbf{x}(\mathcal{F})) \tag{32}$$

$$C_{conf}(\mathbf{x}) = \sum_{w \in SP_{G_s}(\mathcal{F}_{start}, \mathcal{F}_{end})} TPP(w) C(w, \mathbf{x}) \tag{33}$$

$$\begin{aligned} \arg\min_{\pi} \quad & \sum_{i=1}^{n} C_{\pi}(x_{ij}) x_{ij} \\ & \sum_{n=1}^{N} \sum_{k=1}^{K} \sum_{x \in C} T(x)^{f_k^i} Y_x^{f_k^i} \quad \leq \quad D_{\mathbf{F}} \\ & \sum_{j=1}^{n_i} x_{ij} \quad = \quad 1 \quad \forall i \in \{0, n\} \\ & x_{ij} \in \{0, 1\} \end{aligned} \tag{34}$$

5 Super-peer Node

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

6 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.

7 Resources

8 System's resources and actors

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*)

8.1 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.

8.1.1 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.

9 Function choreography scheduler

9.1 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:

$$FC_R \text{ is schedulable } \Leftrightarrow \begin{array}{c} \forall f_{abstract} \in F_{abstract} \\ \exists f \in \mathbf{X}_R \mid f \text{ implements } f_{abstract} \end{array} \tag{35}$$

Although it is correct, the condition expressed by equation 35 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 $1 \leq i \leq n$.

[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.

We define Ω_{FC_R} as follows:

$$\begin{gathered}\Omega_{FC_R} \stackrel{def}{=} \Delta_{X_{R_1}} \cup \ldots \cup \Delta_{X_{R_n}} = \bigcup_{i=1}^{n} \Delta_{X_{R_i}} \\ \text{where} \\ \Delta_{X_{R_i}} \cap \Delta_{X_{R_j}} = \oslash \quad \text{for } i,j \in [0,n] \mid i \neq j \\ \forall f \in \Delta_{X_{R_s}} \quad f \text{ implements } f_{abstract} \text{ for } s \in [0,n]\end{gathered} \tag{36}$$

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:

$$FC_R \text{ is schedulable } \Leftrightarrow \exists \Omega_{FC_R} \tag{37}$$

9.2 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*.

9.2.1 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 $S_{(\Delta_{X_R},m)}$.

9.2.2 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:

$$\sum_{i=1}^{n} m_i \leq k \tag{38}$$

   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 42 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.

9.3 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.

10 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:

$$A \text{ is } FC_R\text{-active peer node} \Leftrightarrow \forall \Delta_{X_R} \in \Omega_{FC_R} \quad \exists S_{(\Delta_{X_R},m)} \text{ hold by } A \tag{39}$$

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

11 Architecture overview

Our system design is based on a network of nodes, or *peers*, every of which has the *same functionality*; in fact, any of them is able to handle request submission, request scheduling and, potentially, request execution.

This is the reason according to which we can mark our proposal as a *P2P system*.

11.1 Overlay network

Our system's nodes are connected by an *overlay network*.

Definition 11.1 (Overlay network)**.** An overlay network is a virtual network built on top of a physical network according to which each nodes can communicate with other if and only if they are connected by virtual links belonging to the virtual network.

Remark. A node may not be able to communicate directly with an arbitrary other node although they can communicate through physical network.

To be more precise, we have adopt a fully *centralized unstructured overlay network* because the *peer-resource index*, sometimes called *directory*, is centralized.

Please note that hybrid unstructured or fully decentralized solutions are technically possible, but guarantees about quality of service are very difficult.

11.1.1 Locality-awareness property

Locality-awareness is one of the essential characteristics of our system. In fact, if each peer is able to select his neighbours exploiting a suitable locality aware algorithm, is possible to decrease user experienced delays.

We have decided to adopt a multi-level based locality-aware neighbour selection called *intra-AS lowest delay clustering algorithm* (ASLDC).

When ASLDC algorithm is used, each peer chooses nearby peers only from those within the same AS; then it ranks its neighbours in terms of transmission latency, preferring to o establish the connection with the node with the shortest latency to itself.

TODO [3]

11.2 Fault Tolerance

Like in many other P2P implementation, there are two ways of detecting failures in our proposed solution:

1. If a node tries to communicate with a neighbour and fails.
2. Since all nodes send to all his neighbour nodes so-called "heartbeat" messages, that is messages sent at fixed time intervals to indicate that the sender is alive, is possible to detect a failure by not receiving aforementioned periodic update messages after a long time.

11.2.1 Availability

Let $n \in \mathbb{N}$, R a resource owner and FC_R his function choreography, since multiple FC_R-Scheduler object can exist in n different nodes, our proposed solution can guarantee an high degree of availability.

In fact, when a node node holding an FC_R-Scheduler fails, all end users requests related to FC_R can be routed to any other node holding a FC_R-Scheduler.

11.2.2 Replication

Although multiple FC_R-Scheduler object can exist, it's not mean that FC_R-Scheduler object are replicated, because every of them manage different virtual function instances.

Except the peer-resource index, any form of replication is performed by our system.

Algorithm 1 An algorithm with caption

$y \leftarrow 1$

and Δ_{X_R}-Scheduler

The locality awareness of the overlay network is used in scheduling jobs, described in Section 3.2

physical network but with added properties such as fault tolerance and flexibility.

The first problem is scheduling. Since there is no central scheduler it is difficult

Hybrid unstructured overlay

11.3 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.

Replication and Fault Tolerance. There are two ways of detecting failures in CAN, the first if a node tries to communicate with a neighbor and fails, it

takes over that neighbor's zone. The second way of detecting a failure is by not receiving the periodic update message after a long time. In the second case, the failure would probably be detected by all the neighbors, and all of them would try to take over the zone of the failed node, to resolve this, all nodes send to all other neighbors the size of their zone, and the node with the smallest zone takes over.

$$E[T] = \sum_{i=0}^{n} E[S_i] + E[T_{Q_i}] \tag{40}$$

12 Queuing system of a Δ_{X_R}-scheduler

Let $k \in \mathbb{N}$ such that $k \geq 0$. Moreover, suppose that R represents a resource owner, P a server-less provider and Δ_{X_R} a sub-swarm of the swarm X_R where k its concurrency limit. Finally, let $S_{(\Delta_{X_R},m)}$ a Δ_{X_R}-scheduler.

Obliviously, since there are only m available virtual function instances, if there are more than m server-less functions waiting to be execute, a choice has to be made about which server-less function has to run next to ensuring QoS guarantees for latency critical applications.

Because we expect to execute server-less functions with different response-time requirements, which may have different scheduling needs, i have designed the Δ_{X_R}-scheduler as a queuing system implementing a *multilevel queue scheduling algorithm* which partitions the ready queue into several separate queues.

According to our solution, each queue has its own scheduling algorithm and any server-less function is permanently assigned to one queue according to his class.

In addition, we have adopt *round-robin* (RR) scheduling algorithm to perform scheduling activity among the queues.

12.1 Server-less function preemption

A very important consideration regards server-less function preemption.

In our context, due to FaaS paradigm, which hides the complexity of servers where our functions will be executed, the ability to preempt functions is not naturally available.

For that reason, only non-preemptive scheduling algorithms, according to which once a server-less function starts running it cannot be preempted, even if a higher priority server-less function comes along, can be adopted.

Since many scheduling algorithms require job preemption to run optimally, this situation can lead to a suboptimal resource management.

12.2 Queuing system design

A Δ_{X_R}-scheduler is made up of three queues:

- One queue implements a *Non-Preemptive Least-Slack-Time-First* (LST) scheduling algorithm. We will refer to that queue using Q_{LST} notation.

  That algorithm implements a *dynamic priority scheduling approach* where priorities are assigned to server-less functions based on their *slacks*.

  At any time t, the *slack* of a job with deadline at d is equal to $d - t - s$, where s is the time required to complete the job.

  Any job having strict latency requirements must to be assign to this queue.

- Another queue implements, instead, a *Non-Preemptive Shortest-Job-First* (*SJF*) scheduling algorithm. That queue is denoted with Q_{SJF}.

  That policy assigns priorities to jobs based on their size: the smaller the size, the higher the priority.

  This queue is reserved for any function choreography that tolerates high latency. However, according to queueing theory, since SJF performs very bad when service time distribution is heavy-tailed; in fact

$$E[T_Q(x)]^{SJF} = \frac{\rho E[S^2]}{2E[S]} \cdot \frac{1}{(1-\rho x)^2} \tag{41}$$

  This queue is not suitable for very, very large server-less function. The variance in the job size distribution must be low.

- Finally, the third queue exploits the the simplest scheduling algorithm, that is the *First-Come-First-Served* (*FCFS*) policy, and it is denoted using Q_{FCFS} notation.

  This queue is used for any heavy-tailed server-less function which can tolerate high latency.

12.3 The virtual function instances allocation problem

Let $n \in \mathbb{N}$ such that $n \geq 1$ and suppose that, at a certain time, a sequence including n unique Δ_{X_R}-schedulers exist in our system.

Formally, that sequence is denoted with $S_{(1,(\Delta_{X_R},m_1))}, \ldots, S_{(i,(\Delta_{X_R},m_n))}$, where $S_{(i,(\Delta_{X_R},m_i))}$ represents the Δ_{X_R}-scheduler owned by i-th node having scheduler capacity equal to m_i.

We have already established that the following constraint must be hold:

$$\sum_{i=1}^{n} m_i \leq k \tag{42}$$

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

12.3.1 Reactive scaling policy

Every node of the system monitors all queues belong to its Δ_{X_R}-Scheduler and adjusts the number of reserved virtual function instances to preserve quality-of-service guarantees. This activity is called *reactive scaling policy*.

Although a $S_{(\Delta_{X_R},m)}$ owns m virtual function instances, they are not equally shared between aforementioned run queue.

First of all, let be v_{LST}, v_{SJF} and v_{FCFS} the numbers of the virtual function instances bounded to the Q_{LST}, Q_{SJF} and Q_{FCFS} run-queue.

At any time, following constrains must be hold:

$$v_{LST} \stackrel{def}{=} \frac{N \cdot E[S]}{\sum slack} \tag{43}$$

$$v_{SJF} \stackrel{def}{=} \frac{len(Q_{SJF})}{\alpha} \tag{44}$$

$$v_{FCFS} \stackrel{def}{=} \frac{len(Q_{FCFS})}{\alpha} \tag{45}$$

$$m = v_{LST} + v_{SJF} + v_{FCFS} \tag{46}$$

1. If the current number of virtual function instances is not enough to avoid deadline misses by latency constrained server-less function inside Q_{LST}, will be add to $VCPU_{LST}$ a number of virtual processor equal to

$$v_{LST} < \frac{N \cdot E[S]}{\sum slack} \tag{47}$$

This form of aging prevents starvation.

Finally, suppose that a node peer A holds a $S_{(\Delta_{X_R}, m_A)}$, which represents the Δ_{X_R}-Scheduler object.

1. Once a request is received over HTTP
2. Ask to the coordinator for

$$m = \frac{\sum slack}{E[S] \cdot m_{current}} + \frac{N}{\alpha} \tag{48}$$

$$\lambda_{FC_R} \geq \lambda_{threshold} \tag{49}$$

$$RTT_{(A,B)} > RTT_{(A,C)} + \tag{50}$$

$$f_{abstract} = next \tag{51}$$

$$f_{abstract} = min(E[f_{\Delta_{X_{R_i}}}] + E[T_{Q_{LST}}]F \tag{52}$$

12.3.2 Server-less function scheduling for QoS "Minimum Response Time"

Let $n \in \mathbb{N}$, such that $n \geq 1$, supposing to have an FC_R-*active peer node* and let Ω_{FC_R} the set of all sub-swarms containing at least one function implementing any $f_{abstract} \in \mathbf{F}_{abstract}$, where $|\Omega_{FC_R}| = n$.

Finally, suppose that $f_{abstract}$ is the next abstract function that have to be executed according to the control-flow logic described by FC_R.

1. For $1 \leq i \leq n$, let $\Delta_{(X_{R_i}, f_{abstract})} \subseteq \Delta_{X_{R_i}}$ containing all concrete functions implementing $f_{abstract}$.
2. For $1 \leq i \leq n$, select $\mathbf{f}_i \in \Delta_{(X_{R_i}, f_{abstract})}$ functions having the minimum response time $E[T_{\mathbf{f}_i}]$.

3. Found $i \in [0, m]$ such that:

$$E[T_{\mathbf{f}_i}] + E[T_{Q_i}]^{\mathbf{LST}} < E[T_{\mathbf{f}_j}] + E[T_{Q_j}]^{\mathbf{LST}} \quad \text{for } j \in \mathbb{N}, 1 \leq j \leq n, i \neq j \tag{53}$$

4. Select the i-th Δ_{X_R}-*Schedulers* belonging to FC_R-*Scheduler* of the node and add $\mathbf{f}_i$ into the Q_{LST} run-queue.