Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Greedy Nominator Heuristic (GNH): Virtual Function Placement on Fog Resources

Osama Almurshed[1] | Omer Rana[1] | Kyle Chard[2]

[1]School of Computer Science & Informatics, Cardiff University, Cardiff, UK
[2], Argonne National Lab. & University of Chicago, Illinois, USA

Correspondence
Osama Almurshed, School of Computer Science & Informatics, Cardiff University.
Email: AlmurshedO@cardiff.ac.uk

Summary

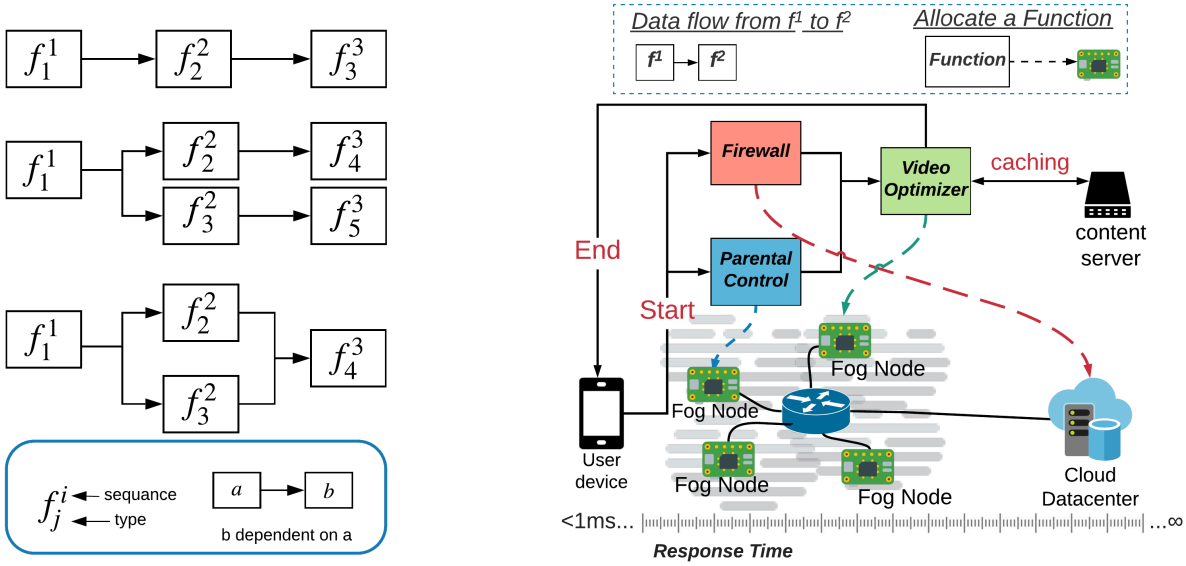
Fog computing is an intermediate infrastructure between edge devices (e.g., Internet of Things) and cloud systems that is used to reduce latency in real-time applications. An application can be composed of a collection of virtual functions, between which dependency constraints can be captured in a Service Function Chain (SFC). Virtual functions within an SFC can be executed at different geo-distributed locations. However, virtual functions are prone to failure and often do not complete within a deadline. This results in function reallocation to other nodes within the infrastructure; causing delays, potential data loss during function migration, and increased costs. We proposed Greedy Nominator Heuristic (GNH) to address these issues. GNH is based on redundant deployment and failure tracking of virtual functions. GNH places replicas of each function at multiple locations—taking account of expected completion time, failure risk, and cost. We make use of a MapReduce-based mechanism, where *Mappers* find suitable locations in parallel, and a *Reducer* then ranks these locations. Our results show that GNH reduces latency by up to 68%, and is more cost effective than other approaches which rely on state-of-the-art optimization algorithms to allocate replicas.

KEYWORDS:
edge computing, resource management, virtual function chaining, workflows

1 | INTRODUCTION

Computational offloading is often used to overcome the resource limitations of IoT devices by migrating the application towards the cloud. Cloud computing provides a salable, cost-effective, and easy-to-use platform for storing and processing data. While IoT devices often rely on cloud platforms, cloud platforms do not satisfy one important requirement of many IoT applications: low latency actions[1]. In other words, by the time data are transferred from an IoT device to the cloud, they may be obsolete and the opportunity to act on those data may be gone[2].

Fog computing extends the cloud computing model to consider the needs of processing time-sensitive data at the edge of the network, near the location in which it was generated[2]. Fog infrastructure can therefore enable applications to act on IoT data within milliseconds. Moreover, a hybrid IoT-Fog-Cloud model can send selected data to the cloud and thus provide IoT applications with more computing power for executing non-time-sennsitive parts of applications, such as offline data processing[1]. Therefore, IoT-Fog-Cloud infrastructure provides numerous advantages for instance low latency, high reliability and enhanced Quality of Service (QoS).



(a) Examples of input-output data dependencies

(b) SFC allocation processes in the Fog and Cloud

FIGURE 1 Abstract service function chaining (SFC) and a scenario of SFC deployment

An application in the IoT-Fog-Cloud ecosystem is composed of sub-applications called virtual functions which can be distributed for execution across available infrastructure. However, determining where virtual functions should be executed is challenging and must consider not only application requirements, but also utilization of infrastructure and the need to avoid bottlenecks that may delay the entire IoT applicaiton.

Virtual functions can be network services or application-specific services. Common examples include firewall (FW), parental control (PC), and video optimizer (VO) applications. These functions are typically combined, in some order, to form an application, known as a Service Function Chain (SFC)[3]. SFC have been defined by the Internet Engineering Task Force (IETF) as RFC7665. Figure 1b shows a video streaming application combining the FW, PC, and VO functions. An SFC can take several forms based on data dependencies (e.g., Figure 1a). Edge device requests are dynamically deployed in IoT-Fog-Cloud infrastructure, where each function can be located on a fog node or Virtual Machine/container hosted on the cloud. Service function chaining enables the creation of composite (network) services that comprise an ordered set of functions that must be applied to packets and/or frames. These functions are triggered based on packet classification as they pass through the function chain. Each service function is referenced using an identifier that is unique within an SFC domain. There are a number of overlaps between an SFC and a workflow, both involve: (i) aggregating functions and services; (ii) ensuring data dependencies are maintained; and (iii) deploying services across one or more physical nodes. A key difference is the level at which deployment of such functions is supported.

The unique nature of SFCs and IoT-Fog-Cloud infrastructure leads to various complications that do not arise in traditional systems, many of which negatively affect the Quality of Service (QoS) of a SFC in IoT-Fog-Cloud infrastructure. We specifically consider three such challenges: (i) nodes may be unreliable and therefore functions may fail or be unable to meet their deadlines when deployed to these nodes; (ii) the IoT-Fog-Cloud infrastructure can dynamically change when locations are added or removed; and (iii) the search space of possible deployment locations can be large and costly to explore.

In this paper, we consider an SFC architecture in an IoT-fog-cloud ecosystem, where fog devices request to host service functions via a controller fog node. Such a node determines the placement of virtual functions and the number of function replicas needed given the failure rate of the nodes. We propose a scalable decision-making algorithm, called Greedy Nominator Heuristic (GNH), to identify the "best" deployment locations. The main contributions of this paper are as follows.

- We formulate the application deployment problem in the IoT-Fog-Cloud infrastructure as an Integer Linear Programming (ILP) problem.
- We explore tradeoffs between cost and performance when deploying replica functions.

- We propose Greedy Nominator Heuristic (GNH) to search for locations that optimize delay, risk, and cost.
- We implement GNH in the MapReduce framework[4], using *Parsl*[5] to place single/multiple replicas of each function at multiple locations.
- We validate GNH via simulation where location characteristics are generated randomly (e.g., availability and latency).

Our results show that GNH is scalable in terms of location search, and also mitigates the effects of a single point of failure in managing virtual functions placement. The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 describes the system model and defines the problem we aim to solve. Section 4 proposes the distributed GNH algorithm for placing IoT applications in the IoT-Fog-Cloud infrastructure. Section 6 validates the performance of the GNH algorithm experimentally and Section 7 concludes the paper.

2 | CONTEXT & RELATED WORK

This section provides the context for this work, and outlines related efforts on developing edge-based computational clusters. The increasing availability of low-cost devices (e.g. Raspberry Pi) has made it cost-effective to build and deploy distributed environments.

2.1 | Single-Board Clusters

A number of efforts focus on implementing computational clusters composed of single-board computers (SBCs):(i) using Raspberry Pi and the Python programming language[6,7,8,9]; (ii) Message Passing Interface (MPI) and SSH connections to securely communicate between SBC nodes. In addition to Raspberry Pi and Python, researchers have used Arduinos and C++ to implement an application that collects data from sensors embedded in IoT devices[9].

Mollava et al.[6] study fault tolerance mechanisms for a cluster of SBCs, which are composed of four Raspberry Pi nodes. Their approach proposes a mechanism to prevent data loss by replicating SBCs. In another effort, six Raspberry Pi workers are controlled by a coordinating Virtual Machine (VM)[7], to support process cross-correlation similarity analysis in parallel. Misbahuddin et al.[9] proposed an IoT e-health system to support Electroencephalography (EEG), Electrocardiogram (ECG) analysis, and temperature sensors which monitor the well being of patients in a hospital. Raspberry Pi and Arduino are used in the system to support data analysis. Similarly, researchers have studied the power consumption, hardware cost, and performance of an RPiCluster composed of 32 Raspberry Pi nodes[8]. The RPiCluster is used to solve 900M Monte Carlo simulation iterations. Results show that increasing the number of nodes reduces the time to completion. Considering a greater number of nodes in a cluster, researchers have designed an affordable cloud infrastructure with 300 SBCs to offer cloud services[10]. The system consisted of a monitoring panel, supporting automatic service provisioning and online access to acquired resources using a centralized manager. A 22-node SBC[11] was implemented to study the possibility of developing an edge-based system to run MapReduce. This system demonstrates a low-cost cluster with each node supporting 1GB of RAM that runs Apache Spark. However, it was found that tasks must consume less than 62.4% of a node's memory, otherwise they would fail to complete successfully.

Morabito[12] compared SBC variants for edge computing applications, comprised of Raspberry Pi and Odroid hardware. The paper shows that Raspberry Pi devices are better in terms of power consumption. However, Odroid devices outperform Raspberry Pi when using a container-based (Docker) OS-level virtualization.

MapReduce has been used by a number of researchers for processing large volumes of data in SBCs clusters[13,14,15,11]. Srinivasan et al.[14] used Hadoop to run the MapReduce programming model on a Raspberry Pi3 cluster with 10 nodes. Nodes run "speeded up robust features" (SURF) to extract features from an image. The authors compared performance between the cluster and a single desktop computer (with Intel I5-4440 CPU and 8GB RAM) and in the case of a large dataset, the SBC cluster has 20% better performance than the desktop computer. However, if the dataset size is reduced to 12.5% of the original, the desktop computer provides better performance. Using a 20 Raspberry Pi cluster for robotics applications, Qureshi et al.[13] increased the operating speed of an SBC (using over-clocked CPUs from 700MHz to 1GHz) to analyse images. At 700MHz, Hadoop execution takes between 100ms to one second, for a data size range between 3MB to 300MB. The overclocked (1GHz) model finishes the tasks within 100ms on any data size between 3MB to 300MB. Other similar approaches include the system by Kaewkasi et al.[11] focusing on the development of a lower-power Hadoop cluster for processing Wikipedia articles. Scolati et al.[15] proposed

containerized clustered SBCs architecture that runs Apache Spark and processes data generated by IoT devices. The system uses 8 Raspberry Pi B2 nodes, three that collect data, four that process this data, and a single node that controls the data processing. Unlike related work[14], this architecture takes advantage of in-memory data storage (provided by Apache Spark) to reduce I/O delay causes by moving data from disk to memory.

2.2 | System Dependability and Service Availability

A system that can be categorized as trustworthy and dependable must cope with any failures that occur during operation, and reduce the impact of the failure on hosted services and applications[16]. These failures can be attributed to abnormal behavior, such as a denial-of-service (DoS) attack, or may be caused by increases in traffic or workload during normal operation. Various proposed solutions[17,18,19,20,21] focus on detecting failure and recovering processing nodes (e.g., a fog node). However, simply recovering a node does not ensure the completion of a service or application hosted on the node, this is especially relevant in the context of a service function chain (SFC). On the other hand, there are a number of research projects that monitor application completion time, and propose mechanisms that either prevent application failure or additional delay[22,23,24,25].

Having highly available services on any SFC architecture is an essential requirement. Recovery from service failure in an SFC deployment should be quick, to meet low-latency requirements. Redundancy mechanisms such as duplicating service functions would fulfill availability requirement[26]. For example, researchers[27,28] have studied SFC failure caused by processing node or network link failures. Service functions generally have a backup that can be instantiated in different locations. This backup provides a redundant deployment, supporting switching to the backup deployment as a recovery strategy. Dinh et al.[30] proposed an algorithm based on the priority level of service functions, and deployment redundancy is modelled as a cost minimization problem. In a cloud infrastructure, Scholler et al.[31] proposed a resilience mechanism based on duplicating the service function deployment to guarantee availability; OpenStack is used to develop the prototype.

The redundancy allocation problem in SFC infrastructure has also been investigated[29], where a particle swarm optimization (PSO) algorithm was used for redundant deployment of SFC in cellular Long-Term Evolution (LTE) networks. Several research efforts focus on minimizing makespan using a PSO approach in Cloud environments. For example, PSO-based dynamic scheduling[32] focuses on maximizing utilization and minimizing makespan in the cloud. DNCPSO is a PSO-based approach that reduces makespan during graph deployment in a cloud-edge environment. Shahid et al.[33] proposed a decentralized autonomous PSO algorithm that can be used to support load balancing in geo-distributed systems. It focuses on minimizing execution cost along with balancing the workload between cloud and edge resources. The global criterion method[34] utilizes scalarization, i.e., a mathematical function that maps two vectors to a single value, to capture the similarity between a solution and the ideal solution. A multi-objective optimizer can utilize the global criterion to rank solutions. For example, Euclidean distance[35,36,37] measures the distance between two points (which represents a solution and the ideal solution) in the Euclidean space. Whereas, Cosine Similarity[38] measures the cosine of the angle between the two vectors solutions. Some researchers[39,40] have proposed their own similarity measures to rank the best solution from a set of solutions. Our approaches makes use of *Parsl* to split IoT applications dynamically across fog and cloud resources. The multiple executors and pilot-job oriented mechanism used in Parsl can support dynamic deployment of functions across a number of different endpoints, two key aspects that can be used to differentiate this work from other efforts. We also propose a novel approach for dynamic (and redundant) SFC allocation management, where the number of replicas is determined based on the overall impact of service functions on the execution of an application.

3 | SFC CONTROL ARCHITECTURE

In our proposed SFC Control Architecture, *locations* are processing nodes that host service functions. They are either Virtual Machines (VMs) in the cloud or Fog Nodes (FNs). Locations receive requests to execute service functions from a *controller* node which is responsible for managing SFC placement. Controllers manage communications between locations, and monitor their capabilities, availability, and SFC operations. However, updating a controller with the infrastructure's status and searching optimal locations to deploy functions can be a time consuming, error prone process. Consequently, the decision-making overhead is divided between workers, that is, the FNs that support a controller to acquire the current status of the infrastructure and find optimal locations, as illustrated in Figure 2.

Controllers make use of *Parsl*[5]—a Python library for parallel programming. Parsl has a DataFlow Kernel (DFK) that orchestrates the execution of individual Python functions on geo-distributed locations. Further, the DFK manages data dependencies

between an SFC's service functions. This is achieved by defining functions that are annotated with a decorator (`@python_app`) to indicate that they can be executed on asynchronously and on remote computing resources (i.e., locations). Annotating Python functions in this way declares these functions as *Parsl apps*, and they return objects, referred to as AppFutures, in lieu of immediate results. Figure 3 shows a simplified SFC (without redundancy) that is executed using Parsl—where Parsl apps are the service functions. The locations are declared in the `'@python_app'` decorators as Parsl *executors*. In the figure, two locations, 'Fog Node 1' and 'Virtual Machine', are specified and indicate that these apps will execute on FW and PC, respectively. Within every Parsl app, Parsl imports a Python module that has the program logic associated with the service function, for example, FW and PC in this instance. Next, it passes 'input' to the 'run' method of the imported service function module, and assigns the result to the 'output' variable. The returned 'output' in both FW and PC are passed to VO as illustrated in Figure 1b.

3.1 | Usage Scenario

This section illustrates a scenario of application placement in the IoT-Fog-Cloud, as shown in Figure 1b. The application is composed of three functions: FW, PC, and VO. FW and PC are independent of on another, but VO depends on the outputs of FW and PC. FW blocks any traffic intended to harm the end-user, where PC filters any multimedia content deemed inappropriate for the intended user based on a number of pre-defined filtering rules. VO enhances the video stream by caching fragments of video content 'near' to the end user.

A user device (e.g., a smart phone) may request application functions from controller Fog Node (FN) in an SFC, then the controller places these functions across locations in the infrastructure. Since the VO depends on FW and PC, the application can experience delays if either FW or PC fail. Therefore, FW and PC must have higher redundancy than OV, as shown in Figure 2, to save the time of redeployment in case of failure. This is based on the observation that functions that occur earlier in an SFC are more significant, as their failure will cause an effect downstream in the SFC pipeline. Having greater redundancy at the early stages of the pipeline is likely to provide greater benefits to avoid delays at subsequent stages in the SFC.

3.2 | System Model

The controller receives an application (i.e., SFC) as a graph $A = (F, D)$, where F is a set of functions, $F = \{f_1^1, f_2^2, \ldots, f_j^i\}$, and D is a set of pairs representing dependencies between functions, $D = \{(f_1^1, f_2^1), \ldots, (f_{j-1}^{i-1}, f_j^i)\}$. The sequence in A is i, where j is a function type identifier (ID). The sequence number i and set D indicate the dependency between SFC functions, in which the function with index i is dependent on the outputs of functions $i-1$ if $(f^{i-1}, f^i) \in D$. For example, in Figure 1b, PC, FW ,and VO represent functions that have an ordering in their execution: PC and FW need to execute before VO. The SFC therefore can be specified as:$(\{f_1^1, f_2^1, f_3^2\}, \{(f_1^1, f_3^2), (f_2^1, f_3^2)\})$, and f_3^2 is dependent on the output of f_1^1 and f_2^1. Table 1 summarizes the associated resource and application properties.

Every function has execution requirements, $q_{j,p}$, $q_{j,m}$, and $q_{j,w}$, which represent process, memory, and storage, respectively, needed to execute function f_j^i. Set L represents all locations that are registered with the controller, and $x_{j,k}^i$ is an auxiliary variable that indicates execution of f_j^i on l_k, where $l_k \in L$ (i.e., f_j^i placed in l_k). p_k, m_k, w_k, and b_k are the available resources at location l_k (i.e., the process handling capacity, memory, storage, and bandwidth respectively). Processor usage and delay (i.e., between the controller and l_k), are represented as u_k and d_k, respectively. Table 2 summarizes decision outcomes and mapping result variables.

3.3 | Estimating Completion Time

The system minimizes the end-to-end latency by attempting to reduce the delay between controller fog node and deployment locations. Moreover, it estimates the time to process every function at a locations as follows:

Network delay between l_k and controller, that is, d_k, is the round-trip time to l_k. The average package size generated by Parsl during the SSH connection between controller and l_k is considered in d_k calculation.

Processing time of f_j^i in l_k depends on f_j^i processing requirement ($q_{j,p}$) and l_k's processing speed (p_k). The total time to place f_j^i in l_k, ($T_{j,k}$), including transmission and processing as defined:

$$T_{j,k} = d_k + \frac{q_{j,p}}{p_k} \tag{1}$$

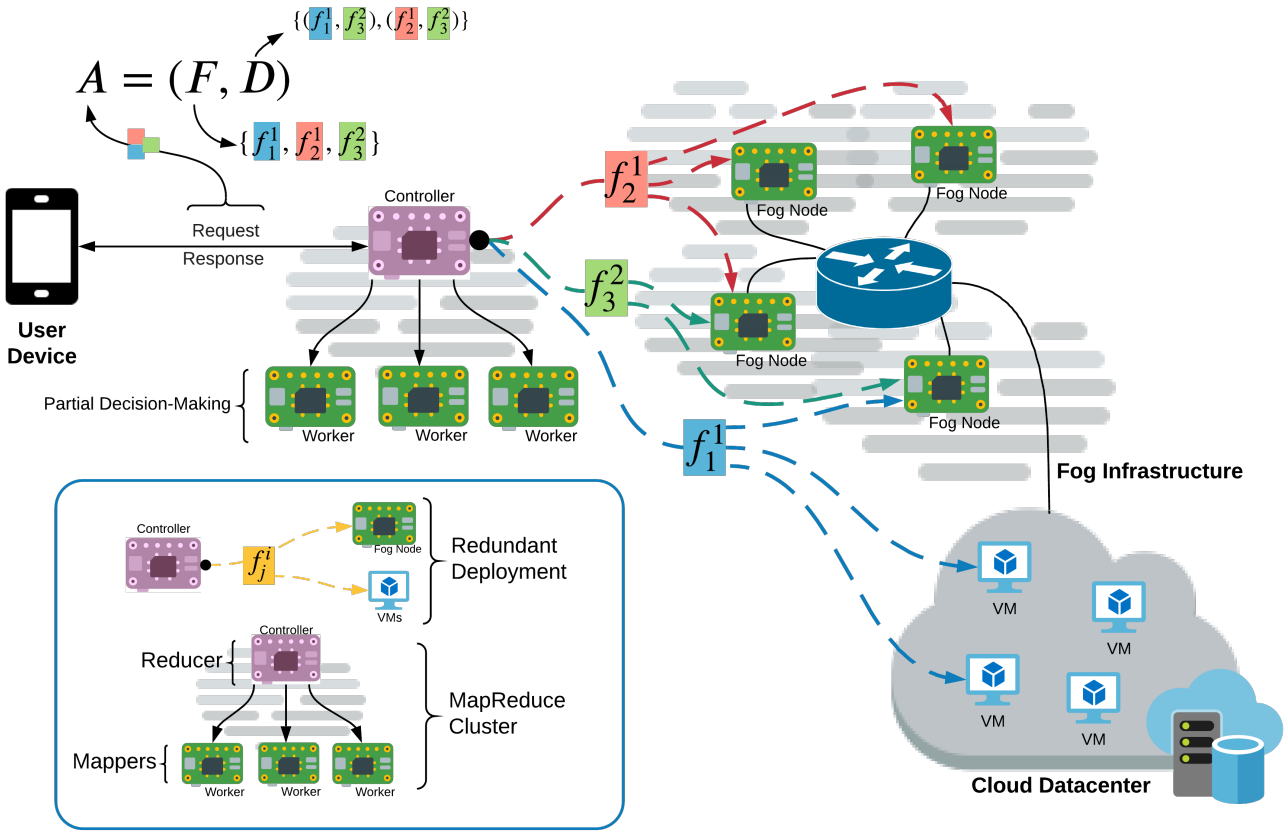


FIGURE 2 Redundant Deployment of application A in the Fog and Cloud

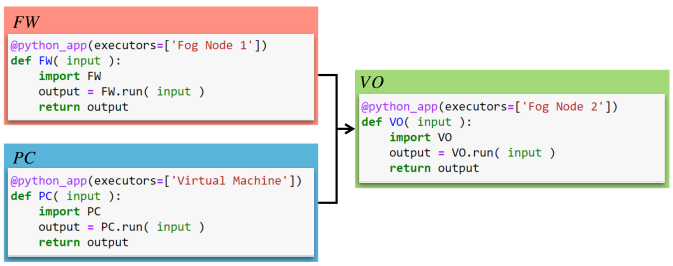


FIGURE 3 SFC from the scenario shown in Figure 1b implemented with Parsl. The *'executors'* argument in the function decorator specify the location that runs service function.

A path in an SFC placement is a sequential execution of service functions, $e = \{T_{1,1}, T_{2,1}, \ldots T_{j,k}\}$ defined in a graph. A **path time** in an SFC placement is the sum of all execution time in the path, $\sum T_{j,k}$ where $T_{j,k} \in e$. Also, E is the set of all paths in an SFC.

The longest path (M) is measured based on the longest time from the placement of the first service function to the end of the last service function within all SFC paths. The longest path is equivalent to the makespan of the SFC and should satisfy constraints in formula 8.

RESOURCE PROPERTIES

Symbol	Description
L	Locations set all locations that are controlled by controller
l_k	Locations k, $l_k \in L$
p_k	l_k's CPU processing power, instruction per second
m_k	l_k's available memory, in bytes
w_k	l_k's available storage, in bytes
d_k	Delay to l_k, Millisecond
b_k	l_k's available bandwidth in percentage
u_k	l_k's processor usage in percentage

APPLICATION PROPERTIES

Symbol	Description
A	Application, consist of SFC which is (F, D)
F	functions in A which is $\{f_1^1, f_2^2, ..., f_j^i\}$
D	Dependencies between A's functions, which is $\{(f_1^1, f_1^1) \dots (f_{j-1}^{i-1}, f_j^i)\}$
n	Number of sequence in A
f_j^i	Service function of i-th in execution and has type j
$q_{j,p}$	The number of instruction needed for f_j, integer value
$q_{j,m}$	The memory needed for f_j, in bytes
$q_{j,w}$	The storage needed for f_j, in bytes

TABLE 1

DECISION OUTCOMES

Symbol	Description
$x_{j,k}$	Auxiliary variable indicate that $f^i j$ is executed in l_k value is 0 or 1
$y_{j,k}$	Auxiliary variable indicate that $T_{j,k}$ is part of the longest path value is 0 or 1
o_k	Auxiliary variable indicate that l_k is obtained by A value is 0 or 1

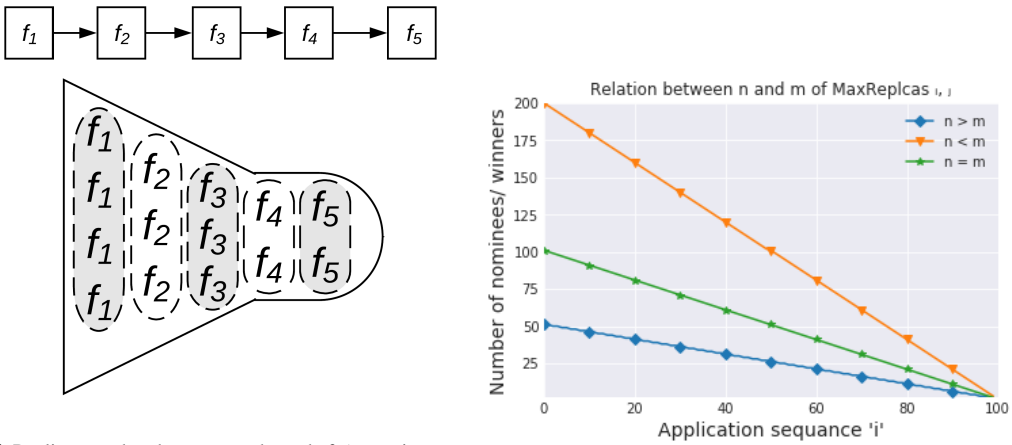
DECISION SUPPORT VARIABLE AND FUNCTIONS

Symbol	Description
$T_{j,k}$	Time to send and process f_j in l_k
E	The set of all placed paths of A
$Risk_{j,k}$	Risk of executing f_j in location l_k
$MaxReplicas_{i,j}$	The maximum possible replicas for f_j^i is integer
m	Constant adjust maximum possible replica is m + 1 is intege
r_k	Loss probability, the number of failures per allocations
M	The path with longest time elapses in A, $M = \{T_{1,1}, T_{2,1}, \dots T_{i,j}\}, M \in E$

TABLE 2

3.4 | Application redundancy and cost

The controller avoids allocating a function to a location that has a high risk of failure. The risk of placing a function at a specific location is specified as: **Risk of allocating** f_j^i in l_k, i.e., $Risk_{j,k}$, is failure/loss probability times the impact of failure ($T_{j,k}$). Loss impact is $T_{j,k}$, as it is the time of the first f_j^i's allocation that failed to complete on time $T_{j,k}$, results in reallocation. Loss probability, i.e., r_k, is the number of failures per allocations, and is derived from historical l_k failure data, hence $Risk_{j,k} = r_k \times T_{j,k}$. Even after calculating the risk, there are chances of reallocating a failed function. Thus, the system deploys replicas of the function to avoid losing time in case of failure. **The redundancy** of application uses a "funnel-shape" of replicated functions, as illustrated in Figure 4a. The initially executed functions (i.e., at an early stage in the SFC) have the maximum replicas, $MaxReplicas_{i,j}$. This value decreases as we progress through an application composed of n functions, as illustrated



(a) Replicas number decreases to the end of A, creating funnel-shaped

(b) Relation between n and m of $MaxReplicas_{i,j}$

FIGURE 4 $MaxReplicas_{i,j}$ Controlling the number of replicas in an SFC

conceptually in Figure 4b. The constant m adjusts $MaxReplicas_{i,j}$, and $MaxReplicas_{i,j}$ does not exceed $m+1$. Formula 2 is used to calculate $MaxReplicas_{i,j}$. The replication strategy is based on the observation that functions that occur at an early stage of the SFC should have higher priority (in terms of resilience), as inability to complete these successfully will cause failure downstream in the SFC. Consequently the number of replicas follow the funnel shape illustrated in Figure 4b, where the actual number of replicas are based on the observed failure rate.

$$MaxReplicas_{j,i} = 1 + \lceil (1 - \frac{i}{n+1})m \rceil \tag{2}$$

The **Cost** of deploying application A is controlled by tracking the locations obtained by A in variable o_k, Table 2 summarizes the decision support variables.

3.5 | Problem Formulation

The main goal of this paper is to provide SFC placement aiming to minimize overall application time, deployment cost, and risk of application failure (to maximize availability). We formulate the optimization problem as follows:

$$MIN \quad C \quad and \quad MIN \quad R \quad and \quad MIN \quad O \tag{3}$$

$$C = \sum_{j \in F} \sum_{k \in L} T_{j,k} \cdot y_{j,k} \cdot x_{j,k} \tag{4}$$

$$R = \sum_{j \in F} \sum_{k \in L} Risk_{j,k} \cdot x_{j,k} \tag{5}$$

$$O = \sum_{k \in L} o_k \tag{6}$$

Where objective function C (formula 4) is the total completion time, and objective function R (formula 5) is the total risk of application A completing successfully. The number of locations used to execute A, including redundancy, is represented by objective function O (formula 6)

Subject to

$$MaxReplica_{i,j} \geq \sum_{k \in L} x^i_{j,k}, j \in F, 1 \leq i \leq n \tag{7}$$

$$\sum_{T_{j,k} \in M} T_{j,k} \cdot y_{j,k} \cdot x_{j,k} \geq \sum_{T_{j,k} \in P} T_{j,k} \cdot x_{j,k}, M \in E, P \in E \tag{8}$$

Algorithm 1 Mapper receives L and f_j^i and Return $MapperResult$ of size $MaxReplicas_{i,j}$

```
1:  class MAPPER
2:      method MAP (L; f_j^i)
3:          count ← 0
4:          for all l_k ∈ L do
5:              result ← ED(l_k, f_j^i)
6:              if count < MaxReplicas_{i,j} then
7:                  MaxHeap.PUSH(⟨result; l_k⟩)
8:                  count ← count + 1
9:              else
10:                 if MaxHeap.PEAK() > ⟨result; l_k⟩ then
11:                     MaxHeap.PUSH-POP(⟨result; l_k⟩)
12:                 MapperResult ← MaxHeap
13:         return MapperResult
```

$$w_k - q_{j,r} \geq 0 \tag{9}$$

$$m_k - q_{j,m} \geq 0 \tag{10}$$

$$u_k \cdot x_{j,k}^i < 1.0 \tag{11}$$

$$b_k \cdot x_{j,k}^i < 1.0 \tag{12}$$

$$x_{j,k}^i - o_k \geq 0, \forall j \in F \tag{13}$$

$$n \geq 1 \tag{14}$$

$$m < |L| \tag{15}$$

4 | GREEDY NOMINATOR HURISTIC (GNH)

The key idea of our proposed optimization algorithm (GNH) is to search for optimal solutions for every function in an application which results in finding the optimal deployment for the application. The GNH uses MapReduce to identify potential locations, where the search space is divided between workers (i.e., Mappers) whose result is sent to the Reducer (i.e., control fog node) to decide the overall optimal locations for redundant deployment. The components used to realise GNH are as follows:

A **similarity function** is used to compare the general solution to an ideal solution[34]. Usually, it is a norm function, for example, Euclidean distance is 2-norm. For all functions, l_{ideal} is the location that has zero execution time and no risk, and no additional locations obtained by A, i.e., O_{ideal}, and is also represented as point $(0, 0, O_{ideal})$. GNH uses Euclidean Distance ($ED_{j,k}$), shown in formula 16 in the Mapper to compare l_k with l_{ideal}[†]. A number of other measures may also be used to perform similarity comparison, such as applying fuzzy metrics that capture a degree of membership. Our implementation can be generalised and extended to use other measures also, as the distance measure can be application- and context-dependent.

$$ED_{j,k} = \sqrt{(0 - T_{j,k})^2 + (0 - Risk_{j,k})^2 + (O_{ideal} - O_k)^2} \tag{16}$$

Max-heap is a complete binary tree that is used to store the Mapper(s) and Reducer results. The root of the tree has the maximum value in the tree, and the value decreases as we move to lower levels in the tree. Max-heap is of $MaxReplica_{i,j}$ size, where each node has key-value pair $\langle key; value \rangle$ (e.g., $\langle result; l_k \rangle$ in algorithm 1). The key is the $ED_{j,k}$ result, whereas the location is the value.

Mappers apply the $ED_{j,k}$ function to all locations, and then store them in Max-heap of $MaxReplica_{i,j}$ size. Max-heap order is based on the key (i.e., $ED_{j,k}$ result), not the values (i.e., locations). Every Mapper keeps a local record of the locations they monitor, and the records are results of monitoring the computing resources and the network connections linking these locations.

[†] O_{ideal} is simply the number of locations that executed the previous functions in A. Therefore, O_k in the next function is equal to O_{ideal} or O_{ideal} incremented by one. The incremented value is multiplied by a weight to have a higher impact on $ED_{j,k}$

Algorithm 2 Reducer receives $MapperResult$ and Return MAXHEAP of size $MaxReplicas_{i,j}$

```
1:  class REDUCER
2:      method REDUCE (MapperResult)
3:          count ← 0
4:          for all ⟨key; value⟩ ∈ MapperResult do
5:              if count < MaxReplicas_{i,j} then
6:                  MaxHeap.PUSH(⟨key; value⟩)
7:                  count ← count + 1
8:              else
9:                  if MaxHeap.PEAK() > ⟨key; value⟩ then
10:                     MaxHeap.PUSH-POP(⟨key; value⟩)
11:         return MaxHeap
```

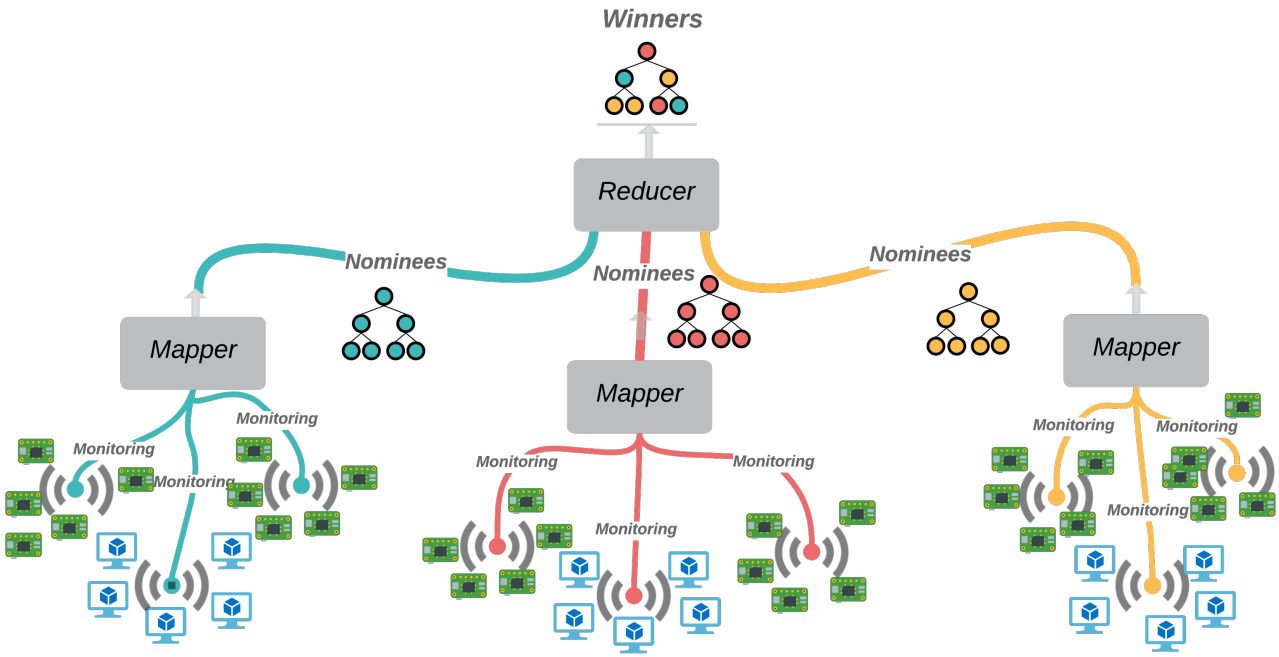


FIGURE 5 Each Mapper has a group of locations to monitor, each group has its color (green, red and yellow). The final Max-heap has a variety of node's color, due to them coming from different Mappers

The **Reducer** receives results from every Mapper, concatenates them, and applies a Max-heap push-pop function to each location in the Mappers' results. Finally, the Mappers' results are reduced in a single Max-heap that has the locations (in the value of $\langle key; value \rangle$ in algorithm 2) to deploy the current function, i.e., f_j^i in the requested application A.

The system has a controller that plays the role of a reducer, whereas workers in Figure 2 are Mappers. In Figure 5, Mappers monitor network performance and available computing resources at specific locations. Moreover, the system nominates locations to execute service functions. Whereas, the Reducer chooses from the nominated locations to place redundant function instances. This is achieved by running MapReduce using Parsl.

Both Mapper and Reducer algorithms 1 and 2 use similar search mechanisms: they loop through the search space and update Max-heap. However, the difference is that Mappers apply the $ED_{j,k}$ function (line 5 of algorithm 1), then compare the result with the worst location within the Max-heap, that is, the peak or root of the tree. Both algorithms initialize Max-heap tree of $MaxReplicas_{i,j}$ size (lines 2-8 in algorithm 1 and lines 2-7 in algorithm 2). Inside the for loop (line 4 in algorithm 1 and 2)

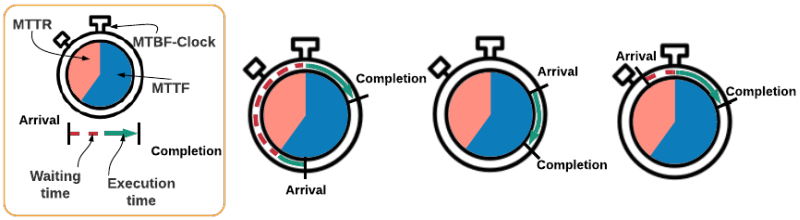


FIGURE 6 The same virtual function's execution at a location showing completion time for three different arrival times.

Names	Definitions
Failure	A period of time where a location freezes and does not respond
Recovery	Restoring after freezing, i.e, failure
MTTF	The time after recovery to location's failure
MTTR	The duration of time were a locaiton restoring after failure
MTBF	Period of time between two failures
MTBF-clock	A location's clock that runs from Failure to Failure and is divided into 2 Periods, MTTF and MTTR
Execution time	The time to run function in an location
Waiting time	The time where functions waits the location to recover

TABLE 3 Failure mode definitions

the new results (*result* in mapper, and *key* in reducer) are compared with the peak of the Max-heap, if the result is less then the peak, then the new result replaces the Max-heap.

5 | FAILURE MODEL

The section describes the failure model we used in this work, specifying when failure and recovery events happen at each processing node (or location). We model and use location failure to validate GNH performance; however, we do not consider software errors, which may be caused by a faulty service function implementation, in this model.

Failure of a processing node is the period during which a location does not respond to user commands. This lack of response adds additional delay, until the node is restored—specified as the *recovery* time. The time to discover the next failure is the Mean Time To Failure (*MTTF*) and time period until the node is restored is the Mean Time To Recovery (*MTTR*). The sum of *MTTF* and *MTTR* is the mean time between two failures *MTBF*.

Every location has a repeating timer, referred to as *MTBF-clock*, that indicates when requests arrive at a location *MTTF* or *MTTR*. This method (algorithm 3) defines whether the allocation location has failed or succeeded. It uses a predefined clock (line 3 in algorithm 3). This mechanism helps to track *MTBF*, where the *MTBF-clock* resets to zero and starts measuring a new *MTBF* time interval. Further, *MTBF-clock* starts with *MTTF* periods followed by *MTTR*. Hence, MTBF-clock of a chosen location tracks the current local *MTBF* time. Table 3 summaries the failure model used in this work. Figure 6 illustrates service functions that have been submitted over three different periods of *MTBF-clock*. If a request arrives within *MTTF* period and is not interrupted (i.e., does not overlap/fall during the MTTR period), then the location will process the function within the expected execution time, otherwise the execution must wait until MTTR period passes to execute the service function.

Algorithm 3 (line 2 and 3) calculate the *MTBF* and the remaining period to the next cycle, MTBFREMAIN. ARRTIME is the timestamp at which a request arrives. If a request arrived within *MTTF*, then the method checks *MTTF* has sufficient time to execute the requested function, EXECTIME, otherwise it is considered a failure (if statement in line 4). The method determines the COMPLETIONTIME of the function (line 6-12).

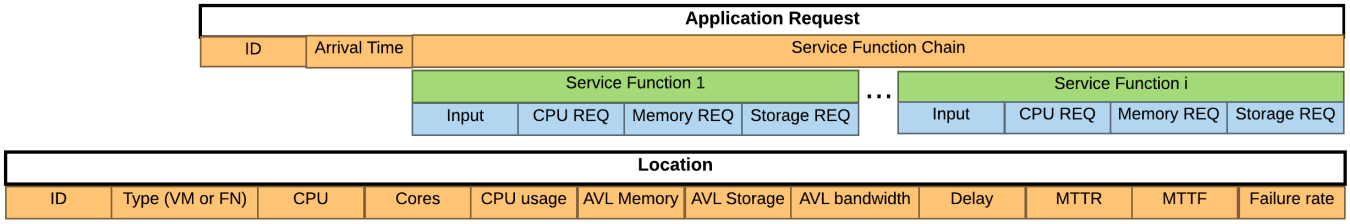


FIGURE 7 The structure of the random generated records

Algorithm 3

```
1:  method FinishingTime ( MTTF, MTTR, execTime, arrTime )
2:      MTBF = MTTF + MTTR
3:      MTBFremain = arrTime % MTBF
4:      if MTBFremain < MTTF and |MTTF-MTBFremain| ≥ execTime then
5:          allocStat = True
6:      if allcStat then
7:          CompletionTime = execTime + ArrTime
8:      else
9:          if MTTF ≥ execTime then
10:             CompletionTime = ArrTime + execTime + |MTTF-MTBFremain| )
11:         else
12:             CompletionTime = ∞
13:     return allcStat, CompletionTime
```

6 | EVALUATION

We evaluate GHN via a based on a Raspberry Pi deployment platform. The simulation dynamically generates requests and varies the number of functions in an SFC (from 1 to 20). It also varies the availability and failure profile of resources using a clock mechanism that is divided into two periods: (i) a failure event (MTTR), and (ii) normal resource operation (MTTF). We create several simulation scenarios and compare GNH with random allocation and PSO-based approaches.

The simulation generates records that contain application request arrival time and computing requirements. Records are passed to GNH, and the allocation decision is evaluated by applying the FinishingTime method as shown in algorithm 3. **Location** records include computational attributes, such as, capability, capacity, delay and available resources. The location generator chooses specifications randomly from Table 6 and Table 7. Figure 7 shows the structure of the records that hold this randomly generated data. All CPU usage and the available resources (e.g., available memory) are frequently and randomly updated to simulate system behaviour. Failure rate is updated every time the FinshingTime (in algorithm 3) returns the allocation state, AllcStat. MTTR and MTTF are constants, and do not change after initialization.**Application** requests are generated with unique IDs and chained service functions then map them to Arrival Time. Arrival times, capturing generated application requests, are uniformly distributed over a single day. This simulates data streams coming to a single controller in the fog infrastructure. Applications are SFCs and can vary in length from 1 to 20 chained virtual functions.

Output of the FinishingTime method determines allocation state, whether a failure has occurred and the actual finishing time of service functions. Service function failure in GNH occurs if all its redundant deployed instances (i.e. replicas for a single function) fail to complete within the estimated time.

6.1 | Experiment Setup

We generate 10 million application requests distributed uniformly over a 24 hour period. Requests are sent in order of their arrival time to a controlling fog node, which is responsible for 100 locations in the IoT-Fog-Cloud ecosystem. 20% of the computing

Variable	Number/Rangs
Application requests	10,000
SFC length	(1-20)
Location	100
FNs	80
VMs	20
FN's Latency	(21 - 50 ms)
VM's Latency	(50 - 300 ms)
FN MTTF	(10 - 30 ms)
FN MTTR	(5 - 15 ms)
VM MTTF	(30 - 300 ms)
VM MTTR	(2 - 10 ms)

TABLE 4 The simulation parameters are chosen randomly from these ranges.

Service requirements	Max Value
CPU	2,000,000
Memory	6MB
Storage	5MB

TABLE 5 Maximum computational resource requirements of the generated functions

resources are virtual machine (VMs) in a cloud data centre, and the remainder are Fog Nodes (FNs) in the fog infrastructure. We do not consider SFCs with more than 20 functions in this experiment.

During the GNH evaluation, we assume that application placement is undertaken over unreliable infrastructure, where failure occurs frequently. Moreover, the *MTTF* of locations can range between *10ms and 300ms* and *MTTR* between *2ms and 15ms*. Table 4 and Table 5 summarizes the parameters used in the experiments. These parameter ranges are based on measured values in a distributed environment consisting of Raspberry Pi nodes connected to a gateway machine over a Wireless (Wifi) network. The experiments are used to evaluate the behaviour and the accuracy of the placement decisions under different network latencies. Moreover, in case of additional delay, the controller will not reallocate the application or functions (in this experiment). This is done to measure the additional delay associated with the placement of the GNH.

We compare *GNH* to two other approaches based on random allocation and a Particle Swarm Optimization (*PSO*).

We consider random allocation with and without replicas. The first algorithm *Rand*, places each service function in an SFC randomly by the controlling fog node. The second algorithm is the same as *Rand* but with replicas *RP*, which alters the number of replicas based on Algorithm 2.

We choose *PSO* due to the similarity between GNH and PSO, where a population of candidate solutions are generated and then a decision is made about the best solution out of those candidate solutions (by announcing a winner in GNH or the *global best* in PSO). One PSO-based algorithm deploys applications with two replicas (RPSO) and the other (PSO) deploys services functions without replicas.

6.2 | Results

The heat map in Figure 8 shows application completion times. We see that 61.18% of *GNH* allocations take less than 100 milliseconds (ms). Whereas 29.78% and 5.45% of applications finished within 101ms-200ms and 201ms-300ms, respectively. The 29.78% applications that have longer completion times are due to the application having longer chained service functions. Around 1.37% of the applications fail to complete with low delay. Compared to GNH, Particle Swarm Optimization with Replicas (RPSO) completed most of its applications within 300-600 ms. This is because RPSO takes more time to decide where to deploy applications; the PSO-based decision-making process takes 200-300 ms before deploying the application (Table 8) with 5 particles and 50 iterations. Particle Swarm Optimization without replica (PSO) did not perform well compared to either GNH or RPSO. Around half of the applications failed to complete on time, and 25% of the deployed applications completed between

Version	CPU	Core(s)	Memory			Storage			Network Interface Speed	
RPi 3 Model A+	1.4 GHz	4	256 MB	512 MB		8 GB	16 GB	32 GB	300 Mbps	
RPi 1 Model B	700 MHz	1	256 MB	512 MB		8 GB	16 GB	32 GB	100 Mbps	
RPi 1 Model B+	700 MHz	1	256 MB	512 MB		8 GB	16 GB	32 GB	100 Mbps	
RPi 2 Model B	900 MHz	4	1 GB			8 GB	16 GB	32 GB	100 Mbps	
RPi 3 Model B	1.2 GHz	4	1 GB			8 GB	16 GB	32 GB	100 Mbps	300 Mbps
RPi 3 Model B+	1.4 GHz	4	1 GB			8 GB	16 GB	32 GB	300 Mbps	1000 Mbps
RPi 4 Model B	1.5 GHz	4	1 GB	2 GB	4 GB	8 GB	16 GB	32 GB	300 Mbps	1000 Mbps
RPi Zero W	1 GHz	1	512 MB			8 GB	16 GB	32 GB	300 Mbps	

TABLE 6 variety of raspberry pi (RPi) models choose from

Version	CPU		Core(s)	Memory	Storage	Network Interface Speed	Max NICs
VM 1	2.35 GHz	3.35 GHz	2	8 GB	50 GB	1000 Mbps	2
VM 2	2.35 GHz	3.35 GHz	4	16 GB	100 GB	2000 Mbps	2
VM 3	2.35 GHz	3.35 GHz	8	32 GB	200 GB	2000 Mbps	4
VM 4	2.35 GHz	3.35 GHz	16	64 GB	400 GB	2000 Mbps	8
VM 5	2.35 GHz	3.35 GHz	32	128 GB	800 GB	16000 Mbps	8

TABLE 7 Possible Virtual Machines (VMs) that are chosen from

	No Replicas (*seconds*)				**With 2 Replicas** (*seconds*)			
#iteration / #particles	**50**	**100**	**200**	**400**	**50**	**100**	**200**	**400**
5	0.238	0.486	1.002	1.985	0.343	0.611	1.255	2.442
10	0.849	1.698	3.397	6.802	0.972	1.953	3.884	7.665
20	3.110	6.166	12.332	24.717	3.349	6.670	13.323	26.709
40	11.910	23.855	47.010	93.905	12.548	24.900	49.874	99.550

TABLE 8 Comparing PSO performance - SFC length is 10

300-600 ms. Increasing the number of particles also increases the chances of converging to the global optimum. However, more particles and iterations will increase execution time, as shown in Table 8. For example, in a Raspberry Pi 3B+, 10 particles with 250 iterations can increase the failure rate from 38.83% to 33%. However, it will take between 3.5 to 4.5 seconds to complete the 250 situations.

Using *RP*, 74.56% applications completed in less than 200ms. Approximately 16.4% of applications can be allocated to faster locations, if the *RP* was aware of location completion times. Therefore, even though the replica-based strategy mitigates failure, on its own it will not guarantee an optimal completion time. Finally, since *Rand* does not use replicas, it is more prone to failure when compared to *GNH* and *RP*. More than 70% of *Rand* allocations finished in the order of seconds not milliseconds. However, 23% of the Rand applications finished faster than RPSO and PSO because it also takes time to decided on the allocations.

Figure 9 shows the average completion time of applications and their execution time. *GNH* on average completes the application request in less than 200ms. Whereas *RP* has an average completion time of around 600ms.

Table 9 shows the failure percentage for each algorithm. When using *GNH* there is less than 4% failure rate, whereas when using *RP* 18% of the allocated applications fail to complete within the expected time. With *Rand* we see approximately 24% of the allocated applications succeed in the expected time. Comparing to Rand, PSO enhanced the success rate by 11.86%, whereas RPSO decreases the failure rate by 49.18%.

Finally, we show the **Cost** of application deployment of *RP*, *GNH*, and *RPSO* in hex-bins and bar-charts in Figures 10 and 11, respectively. *GNH* has a maximum of 5 locations used per application. Despite the number of service functions, it has only one extra location than the $MaxReplicas_{i,j}$ of the first function in the SFCs. Even though *Rand* uses $MaxReplicas_{i,j}$ to determine

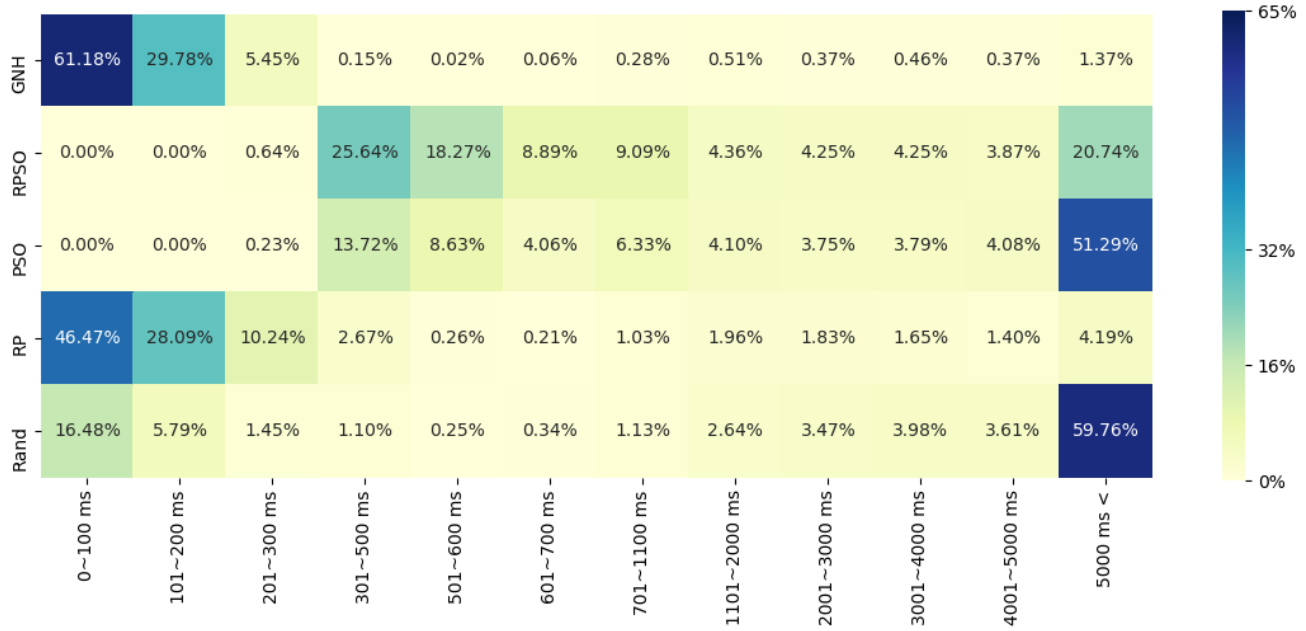


FIGURE 8 heat-map shows applications completion time

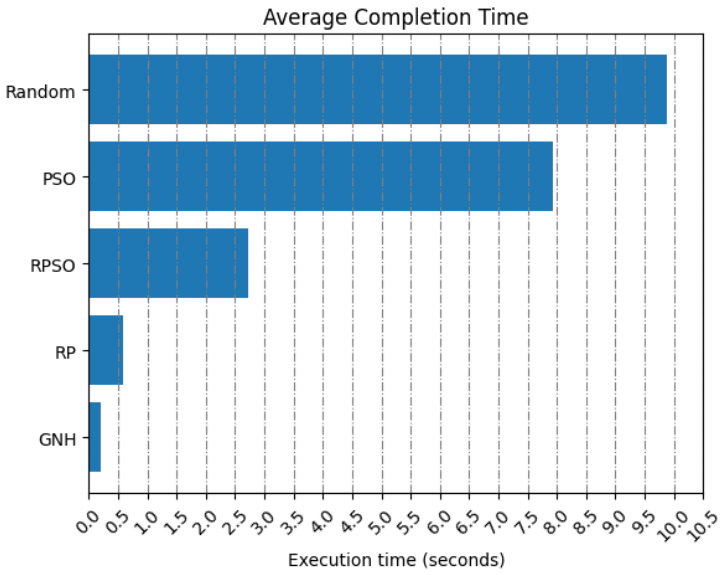


FIGURE 9 Algorithm comparison: completion time in seconds

the number of replicas, 50 locations can be used for a single application deployment, and around 40% exceeded 20 locations for an application. This occurs because *Rand* does not save locations that have been used for previously executed functions within the same SFC. *RPSO* on average uses around 9 locations per application and mostly uses 1.5 times the SFC length of the application.

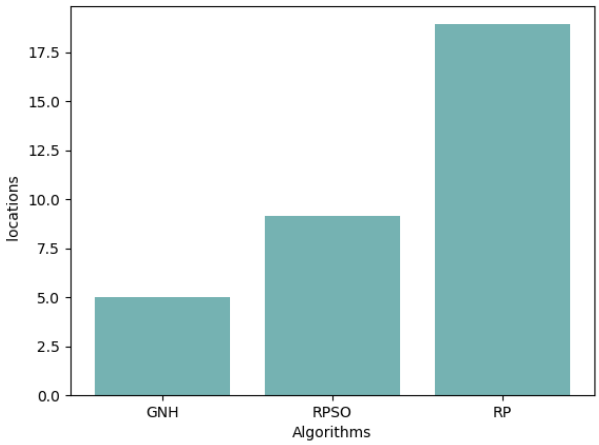


FIGURE 10 Average cost – based on the number of locations used

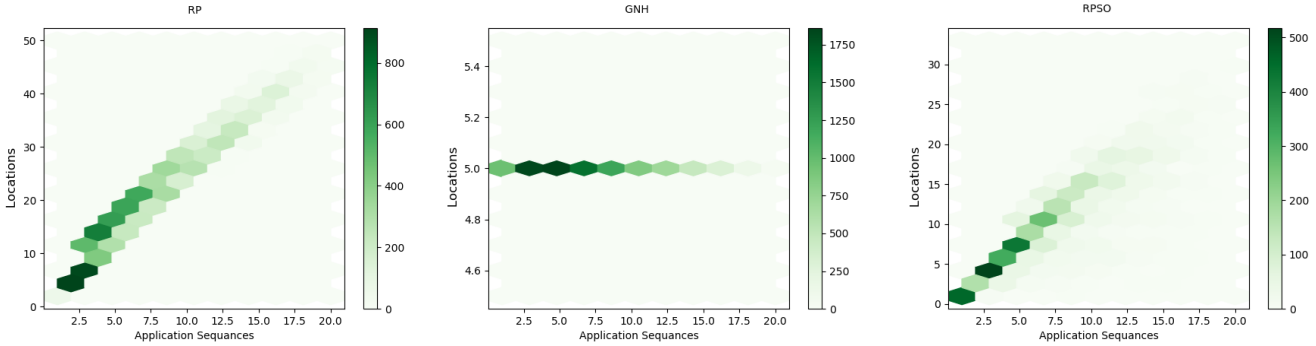


FIGURE 11 Hex-bins shows the relations between SFC length with the number of locations

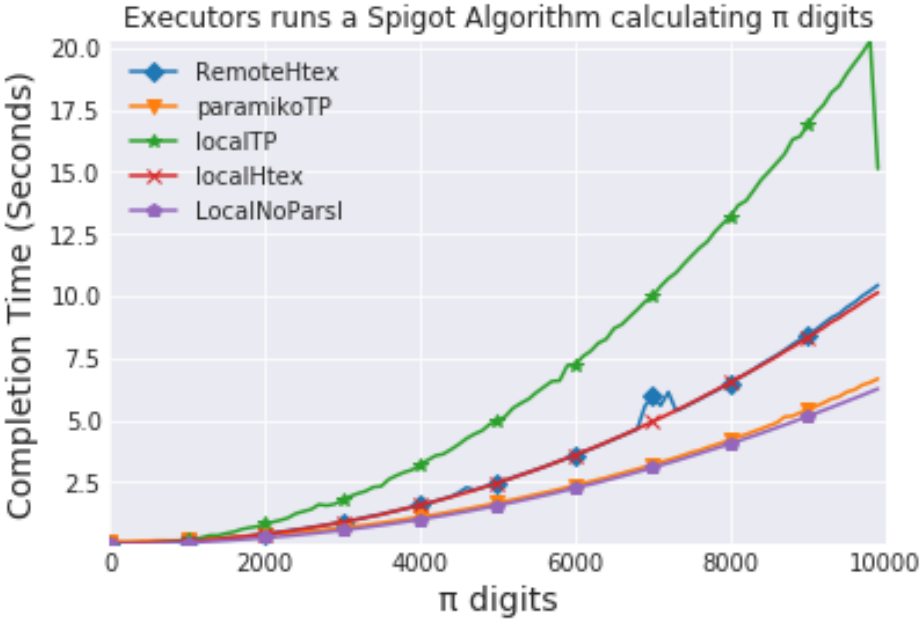


FIGURE 12 Benchmarking performance of Parsl using different execution models both on local and remote nodes to compute Pi to increasing decimal places.

Algorithm	Failure rate
GNH	3.5%
RP	18.00%
RPSO	38.83%
PSO	67.35%
Rand	76.41%

TABLE 9 Failure Percentage (on average) in each algorithm

6.3 | *Parsl* Performance on Fog Node

In this section we benchmark Parsl performance using fog nodes deployed over a Raspberry Pi3 model B+ device (with 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU, and 1GB RAM), connected over a WiFi network. The average response time for the controller is 0.47ms (minimum latency 0.4ms, maximum latency 0.65ms). The controller is also a Raspberry Pi3 model B+ device. We consider both the ThreadPoolExecutor and HighThroughputExecutor Parsl executors (where executors are the method via which service functions are executed).

The *HighThroughputExecutor* allows service function execution, concurrently within local (*localHtex*) or remote (*remoteHtex*) machines using Python processes. ThreadPoolExecutor uses multiple threads on a single local node (*localTP*). In this experiment, we created a Paramiko script to SSH to a Raspberry Pi and run service functions, and the script was run using the ThreadPoolExecutor (*paramikoTP*) to connect it to Parsl. We also executed the targeted service function without Parsl (*LocalNOParsl*), to compare its execution time on the Raspberry Pi. Moreover, we observed that multiple controllers accessing a single Raspberry Pi using *remoteHtex* negatively affected performance, unlike *paramikoTP*. Therefore, in this experiment we used a single controller, running the spigot algorithm remotely using *remoteHtex* and *paramikoTP*. We used the *spigot* algorithm[41] as a service function that evaluates performance using the different execution models. This service function aims to calculate the value of π, where the user specifies the number of decimal places required (varied from 10 to 10,000 decimal place accuracy in this experiment).

Figure 12 shows the time to complete the spigot algorithm using the different execution methods. The x-axis shows the input size and the y-axis shows the completion time. *LocalNOParsl* and *paramikoTP* are almost identical and take around 6.26 seconds for the former and 6.66 seconds for the latter. *LocalNOParsl* exhibits marginally better performance as it does not use a network to send the output. The next set of experiments using *localHtex* and *remoteHtex* show similar results. *remoteHtex* has a delay of 0.3 seconds when it reaches 10,000 digits. Even though the *localTP* does not use the network it has the worst completion time, which exceeds 15 seconds to calculate π to 10,000 decimal places. Although *remoteHtex* runs on a Raspberry Pi, there is an opportunity to achieve faster completion time, similar to *paramikoTP*. A possible solution is to design a *lightWeight parsl* executor that is similar to *paramikoTP*, which allows multiple controllers to access the same Raspberry Pi, and has a performance comparable to execute service functions through a normal SSH connection.

6.4 | GNH Performance

In this section we evaluate the speed of GNH in decision making. Both the controller and workers (i.e., reducer and mappers) are Raspberry Pi 3 models B+. We tested the algorithm in two search space one with 1,000 locations, and another with 100,000 locations. Since Parsl allows decisions to be processed in parallel, we evaluate speed with a single service function. We also considered locality of data, whether they are on disk (in a file) or in memory.

As can be seen in Figure 13, the time to make a decision is not affected by the number of replicas. This is due to the efficiency of the Max-heap operations which reduce the time to compare locations.

With less than 1000 location, adding a new mapper to the GNH can increase the performance from 2% to 15%. Moreover, dividing the search space into 10 disjoint spaces, in which each mapper has 100 locations, can speed up decisions to about 48% when data are stored in a file on disk. However, if the data are already in memory it is faster by about 35%.

When 100,000 locations are considered the decision-making overhead has a more significant impact. Every added mapper can boost the performance from 10% to 45 %. Moreover, 10 mappers in the GNH is faster than single mapper by 88%.

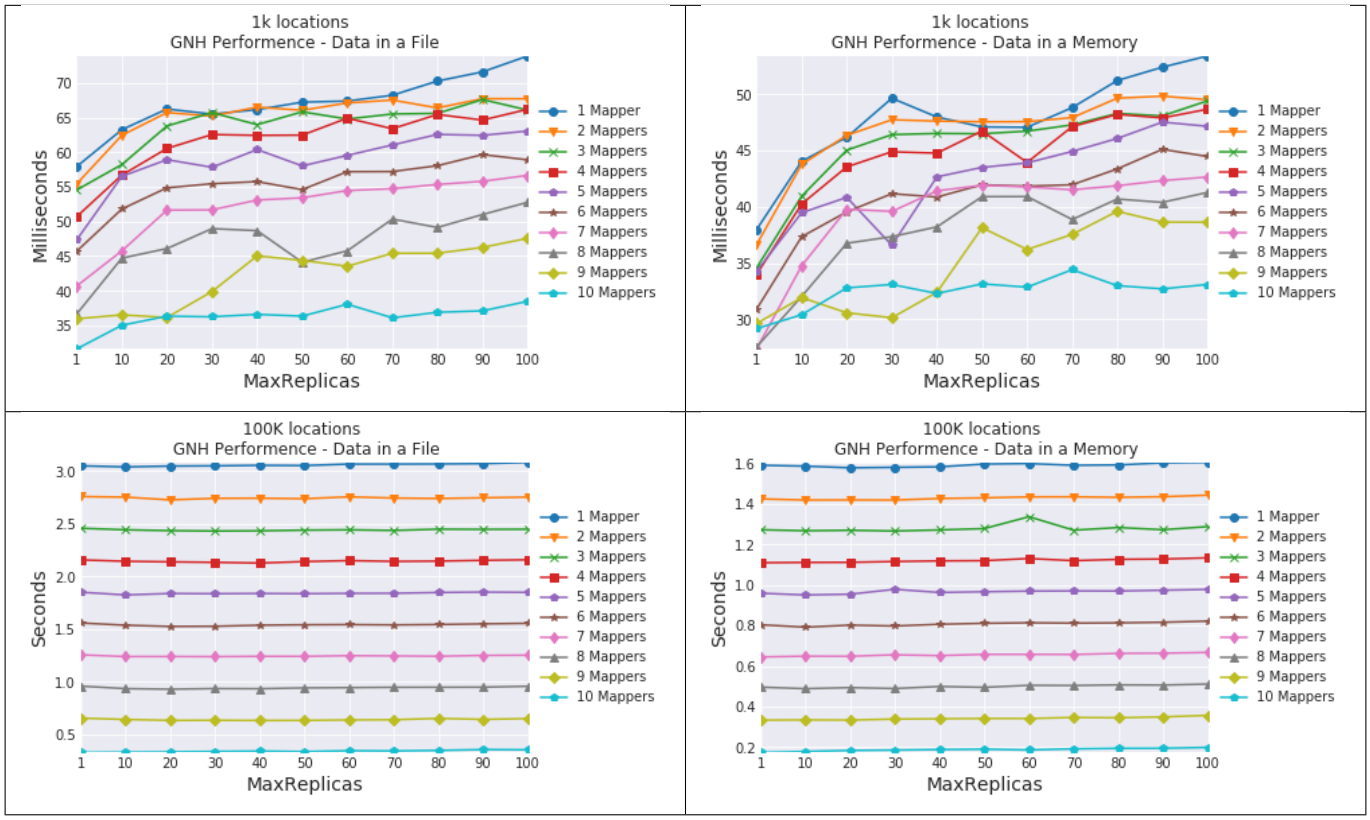


FIGURE 13 GNH Performence

Every time we add a mapper the performance improves by 10-14%. Until we add the 5th mapper the performance exceeds 15%. Addining additional mappers continues to improve performance. For example, 6 mappers outperform 5 mappers by 20%, 7 mappers outperform 6 mappers by around 25%, and 10 mappers outperform 9 mappers by 47%.

Loading a file with 1000 locations to memory takes between 8ms and 23ms. Therefore, keeping the data in memory can boost the decision-making speed by up to 40%. However, loading data from files to mappers will not take much time if there are fewer than 100 locations. On the other hand, with 100,000 locations a single mapper can take 1.25 seconds to load the locations to memory, dividing them between 10 Mappers reduces it to 100ms.

Files with less than 100 locations are easily loaded in memory with minimal delay. Therefore, it would be better to divide the locations recorded into files that can be loaded to the memory concurrently on a separate thread while the mappers do partial decision.

Another solution can be to make partial decisions within the mappers during the periodical update for all service functions and then ranking them in max-heaps; each max-heap ranks locations by single service function. The partial decision will be passed to the reducer which will decide from the pre-processed partial decisions. This solution saves time, especially with large search space, since the reducer will not wait for mappers to produce results. However, this solution will need an adjustment to the MapReduce implementation, for example, the Mapper will only calculate a similarity function for the time and risk, but the number of locations per application is done by Reducer.

6.5 | Comparing GNH with other approaches

Hmaity et al.[27] proposed an approach that chooses two nodes for deployment—primary and back up—where functions are deployed to primary nodes, and redeployed to back up nodes in case of failure. However, this approach has two issues, real-location increases delay, as well as the obvious limitation if both of the chosen nodes fail to complete. Under a high failure rate situation, increasing the number of replicas will reduce the failure rate and outperform a state-of-the-art approach with two replicas running simultaneously.

Beck et al.[28] used breadth-first to search for an SFC placement, and in case of deadlock, they use backtracking to avoid deadlocks. Nevertheless, as there is no redundant allocations, there is an overhead for reallocation, also, the highly influential functions in the chain, i.e., the first in the sequence, are randomly placed.

Chantre et al.[29] used classic particle swarm optimization (PSO) to redundantly deploy SFCs, where particles move around in the search-space (simulating social behaviour of swarms) to search for optimal placement. PSO is a stochastic approach, which is suitable for a large search space. However, due to non-determinism, it cannot guarantee optimality. For example, 38.83% of RPSO's applications failed to complete in time; while only 3.5% failed when using GNH (Table 9). Additionally, the algorithm needs stopping criteria, such as, maximum number of iterations, which can result in long execution times which increase the end-to-end latency.

GNH varies the number of replicas based on the impact of specific service functions on the SFC completion time, and replicas are all deployed at the same time. It finds the replicas by dividing process nodes, and searching each in parallel to speed up the process. Moreover, the optimal allocation (i.e., local optimum) is guaranteed since all nodes are covered and ranked. GNH is deterministic, which means with the same environmental condition and the same input it will generate the same output.

7 | CONCLUSION

The Service Function Chain (SFC) model is increasingly being used to deploy functions across the Fog-Cloud environment. Fog nodes can be in close proximity to a user, and may have lower capabilities than cloud nodes. We consider a variety of different types of fog and cloud resources, and explore how these resources can be used as "locations" to host service functions. We make use of Parsl (a Python-based parallel programming library) to manage dataflow within a SFC, propose the Greedy Nominator Heuristic (GNH), using the Map-Reduce paradigm, to reduce end-to-end latency across an SFC. GNH applies two key strategies: (i) avoiding placement of functions on unstable computing resources, that is, resources that historically have demonstrated a high failure rate; and (ii) deploy functions across multiple locations, using a replication strategy that takes into account the location of the function in the SFC. Functions that occur at an early stage of the SFC have a greater replication factor, as executing these functions successfully has an impact on completion of dependent functions further down the SFC.

We conducted a simulation-based evaluation of this work using parameters based on a Raspberry Pi deployment platform. The simulation is used to: (i) dynamically generate requests and vary the number of functions in an SFC (from 1 to 20); (ii) vary the availability and failure profile of resources using a clock mechanism that aligns resource unavailability with request arrival rate (using Mean-time-to-Failure and Mean-time-to-Recovery metrics). We create a number of possible simulation scenarios to compare GNH with two random placement algorithms, one with replicated placement of functions. On unreliable infrastructure, our results show that with the two strategies (i.e., redundancy and failure tracking), the system is able to reduce function execution latency by up to 68.38% compared to a redundancy only strategy. Moreover, the GNH redundancy is also shown to be cost-effective compared to a random redundant deployment.

However, our current solution does not support the decoupling of physical-virtual layers in the fog; the virtual system layer provides flexibility in managing the infrastructure. Moreover, the user needs a way to define applications easily. Our future work includes (i) investigating the use of parallelism to improve optimization, for example, particle swarm and genetic algorithms for a single-board computer; and (ii) considering user-defined requirements to service functions placement, such as, taking account of security tags or hardware requirements (e.g., use of a GPU).

References

1. Bittencourt L, Immich R, Sakellariou R, et al. The Internet of Things, Fog and Cloud continuum: Integration and challenges. *Internet of Things*. 2018;3-4:134-155. doi:10.1016/J.IOT.2018.09.005

2. Cisco White paper. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*; 2015. https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf. Accessed May 28, 2020.

3. Quinn Paul and Beliveau A. Service Function Chaining (SFC) Architecture. *Draft (work progress)*. 2014

4. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107-113. doi:10.1145/1327452.1327492

5. Babuji Y, Woodard A, Li Z, et al. Parsl: Pervasive parallel programming in Python. In: *HPDC 2019- Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. New York, New York, USA: Association for Computing Machinery, Inc; 2019:25-36. doi:10.1145/3307681.3325400

6. Mollova S, Georgieva P, Kostadinov A. Fault-tolerance of a laboratory computer cluster. In: *2018 20th International Symposium on Electrical Apparatus and Technologies, SIELA 2018 - Proceedings*. Institute of Electrical and Electronics Engineers Inc.; 2018. doi:10.1109/SIELA.2018.8447083

7. Vujicic D, Markovic D, Veskovic M. Practical Aspects of Using Virtualization with Raspberry Pi Clusters Full-wave rectifier for low-level signals View project. In: *International Scientific Conference UNITECH 16-17 2018 Gabrovo*: UNITECH 2018; 2018:113-116.

8. Kiepert J. Creating a raspberry pi-based beowulf cluster. *Boise State University* 2013

9. Misbahuddin S, Ibrahim MM, Alnajar AM, Alolabi BQ, Ammar AF. Automatic Patients' Vital Sign Monitoring by Single Board Computer (SBC) Based MPI Cluster. In: *2nd International Conference on Computer Applications and Information Security, ICCAIS 2019*. Institute of Electrical and Electronics Engineers Inc.; 2019. doi:10.1109/CAIS.2019.8769551

10. Abrahamsson P, Helmer S, Phaphoom N, et al. Affordable and energy-efficient cloud computing clusters: The Bolzano Raspberry Pi cloud cluster experiment. In: *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*. Vol 2. IEEE Computer Society; 2013:170-175. doi:10.1109/CloudCom.2013.121

11. Kaewkasi C, Srisuruk W. A study of big data processing constraints on a low-power hadoop cluster. In: *2014 International Computer Science and Engineering Conference, ICSEC 2014*. Institute of Electrical and Electronics Engineers Inc.; 2014:267-272. doi:10.1109/ICSEC.2014.6978206

12. Morabito, Roberto Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access* 2017;

13. Qureshi B, Javed Y, Koubâa A, Sriti MF, Alajlan M. Performance of a Low Cost Hadoop Cluster for Image Analysis in Cloud Robotics Environment. In: *Procedia Computer Science*. Vol 82. Elsevier B.V.; 2016:90-98. doi:10.1016/j.procs.2016.04.013

14. Srinivasan K, Chang CY, Huang CH, Chang MH, Sharma A, Ankur A. An efficient implementation of mobile Raspberry Pi Hadoop clusters for Robust and Augmented computing performance. *J Inf Process Syst*. 2018;14(4):989-1009. doi:10.3745/JIPS.01.0031

15. Scolati R, Fronza I, El Ioini N, Samir A, Pahl C. A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In: *CLOSER 2019 - Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress; 2019:68-80. doi:10.5220/0007695000680080

16. Sterbenz JPG, Hutchison D, Çetinkaya EK, et al. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Comput Networks*. 2010;54(8):1245-1265. doi:10.1016/j.comnet.2010.03.005

17. Kreutz D, Ramos FMV, Verissimo P. Towards secure and dependable software-defined networks. In: *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, New York, USA: ACM Press; 2013:55-60. doi:10.1145/2491185.2491199

18. Manuel Sanchez Vilchez J, Grida I, Yahia B, Crespi N. Self-Healing Mechanisms for Software Defined Networks. In: *Jose Manuel Sanchez Vilchez, Imen Grida Ben Yahia, Noël Crespi. Self-healing Mechanisms for Software Defined Networks. 8th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2014), Brrno, Czech Republic.* ; 2014. https://hal.archives-ouvertes.fr/hal-01068045. Accessed April 8, 2020.

19. Da Rocha Fonseca PC, Mota ES. A Survey on Fault Management in Software-Defined Networks. *IEEE Commun Surv Tutorials*. 2017;19(4):2284-2321. doi:10.1109/COMST.2017.2719862

20. Guha A, Reitblatt M, Foster N. Machine-verified network controllers. In: *ACM SIGPLAN Notices*. Vol 48. ; 2013:483-494. doi:10.1145/2499370.2462178

21. Lee J, Ko H, Suh D, Jang S, Pack S. Overload and failure management in service function chaining. In: *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: En Route to 5G, NetSoft 2017*. Institute of Electrical and Electronics Engineers Inc.; 2017. doi:10.1109/NETSOFT.2017.8004241

22. Turchetti RC, Duarte EP. Implementation of Failure Detector Based on Network Function Virtualization. In: *Proceedings - 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2015*. Institute of Electrical and Electronics Engineers Inc.; 2015:19-25. doi:10.1109/DSN-W.2015.30

23. Lee SI, Shin MK. A self-recovery scheme for service function chaining. In: *International Conference on ICT Convergence 2015: Innovations Toward the IoT, 5G, and Smart Media Era, ICTC 2015*. Institute of Electrical and Electronics Engineers Inc.; 2015:108-112. doi:10.1109/ICTC.2015.7354505

24. Chiti F, Fantacci R, Paganelli F, Picano B. Virtual Functions Placement with Time Constraints in Fog Computing: a Matching Theory Perspective. *IEEE Trans Netw Serv Manag*. 2019. doi:10.1109/TNSM.2019.2918637

25. Tajiki MM, Shojafar M, Akbari B, Salsano S, Conti M. Software defined service function chaining with failure consideration for fog computing. *Concurr Comput Pract Exp*. 2019;31(8):e4953. doi:10.1002/cpe.4953

26. Halpern J, Pignataro C RFC 7665: Service function chaining (sfc) architecture. *RFC*. 2015:1-31

27. Hmaity A, Savi M, Musumeci F, Tornatore M, Pattavina A. Virtual Network Function placement for resilient Service Chain provisioning. In: *Proceedings of 2016 8th International Workshop on Resilient Networks Design and Modeling, RNDM 2016*. Institute of Electrical and Electronics Engineers Inc.; 2016:245-252. doi:10.1109/RNDM.2016.7608294

28. Beck MT, Botero JF, Samelin K. Resilient allocation of service Function chains. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2016*. Institute of Electrical and Electronics Engineers Inc.; 2017:128-133. doi:10.1109/NFV-SDN.2016.7919487

29. Chantre HD, Da Fonseca NLS. Redundant placement of virtualized network functions for LTE evolved Multimedia Broadcast Multicast Services. In: *IEEE International Conference on Communications*. Institute of Electrical and Electronics Engineers Inc.; 2017. doi:10.1109/ICC.2017.7996870

30. Dinh NT, Kim Y. An Efficient Reliability Guaranteed Deployment Scheme for Service Function Chains. *IEEE Access*. 2019;7:46491-46505. doi:10.1109/ACCESS.2019.2908185

31. Schöller M, Stiemerling M, Ripke A, Bless R. Resilient deployment of virtual network functions. In: *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*. IEEE Computer Society; 2013:208-214. doi:10.1109/ICUMT.2013.6798428

32. Khalili A. and Babamir S.M. Makespan improvement of PSO-based dynamic scheduling in cloud environment. In *23rd Iranian Conference on Electrical Engineering*. IEEE; 2015

33. Shahid H.F., Pahl Enhanced Particle Swarm Optimisation and Multi Objective Optimization for the Orchestration of Edge Cloud Clusters. In: *IJCCI*; 2019: 155-162

34. Zeleny, M. Compromise programming. In: *Multiple criteria decision making.* 1973: 263-301

35. Cardellini, V., Grassi, V., Lo Presti, F. Nardelli, M., Distributed QoS-aware scheduling in Storm. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*; 2015: 344-347

36. Pietzuch, P., Ledlie, J., Shneidman, J. Roussopoulos, M., Welsh, M., Seltzer, M., Network-aware operator placement for stream-processing systems. In: *22nd International Conference on Data Engineering (ICDE'06) IEEE.*; 2006: 49-49

37. Jiang, F., Ferriter, K., Castillo, C. A cloud-agnostic framework to enable cost-aware scheduling of applications in a multi-cloud environment. In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium* ; 2020: 1-9

38. Radouche, S., Leghris, C. Network selection based on cosine similarity and combination of subjective and objective weighting. In: *2020 International Conference on Intelligent Systems and Computer Vision (ISCV) IEEE*; 2020: 1-7

39. Baranwal, G., Yadav, R., Vidyarthi, D. P. QoE aware IoT application placement in fog computing using modified-topsis. In: *Mobile Networks and Applications.*; 2020;25(5): 1816-1832

40. Samriya, J. K., Kumar, N. An optimal SLA based task scheduling aid of hybrid fuzzy TOPSIS-PSO algorithm in cloud environment. In: *Materials Today: Proceedings.*; 2020

41. Rabinowitz S, Wagon S. "A spigot algorithm for the digits of π." *The American mathematical monthly* 1995;102(3): 195-203. doi:10.1080/00029890.1995.11990560