Fault Tolerant Placement of Stateful VNFs and Dynamic Fault Recovery in Cloud Networks

Guochang Yuan[1], Zichuan Xu[1], Binxu Yang[2], Weifa Liang[3], Wei Koong Chai[4], Daphné Tuncer[5], Alex Galis[2], George Pavlou[2], Guowei Wu[1]

Abstract

Traditional network functions such as firewalls and Intrusion Detection Systems (IDS) are implemented in costly dedicated hardware, making the networks expensive to manage and inflexible to changes. Network function virtualization enables flexible and inexpensive operation of network functions, by implementing virtual network functions (VNFs) as software in virtual machines (VMs) that run in commodity servers. However, VNFs are vulnerable to various faults such as software and hardware failures. Without efficient and effective fault tolerant mechanisms, the benefits of deploying VNFs in networks can be traded-off. In this paper, we investigate the problem of fault tolerant VNF placement in cloud networks, by proactively deploying VNFs in stand-by VM instances when necessary. It is challenging because VNFs are usually stateful. This means that stand-by instances require continuous state updates from active instances during their operation, and the fault tolerant methods need to carefully handle such states. Specifically, the placement of active/stand-by VNF instances, the request routing paths to active instances, and state transfer paths to stand-by instances need to be jointly considered. To tackle this challenge, we devise an efficient heuristic algorithm for the fault tolerant VNF placement. We also propose two bicriteria approximation algorithms with provable approximation ratios for the problem without compute or bandwidth constraints. We then consider the dynamic fault recovery problem given that some placed active instances of VNFs may go faulty, for which we propose an approximation algorithm that dynamically switches traffic processing from faulty VNFs to stand-by instances. Simulations with realistic settings show that our algorithms can significantly improve the request admission rate compared to conventional approaches. We finally evaluate the performance of the proposed algorithm for the dynamic fault recovery problem in a real test-bed consisting of both physical and virtual switches, and results demonstrate that our algorithms have potentials of being applied in real scenarios.

Keywords: Fault-tolerance; network function virtualization; cost minimization; bicriteria approximation algorithms; algorithm analysis.

1. Introduction

The operation of cloud networks, such as data center networks, geo-distributed networks, needs various network functions, such as network address translation (NAT), firewall and deep packet inspection (DPI), to improve the network performance and security. These network functions are typically implemented in dedicated hardware that are costly and difficult to reconfigure. The advent of Network Function Virtualization (NFV) provides a flexible and inexpensive support of network functions [1], by decoupling network functions from physical devices to software in virtual machines (VMs). Therefore, such virtualized network functions (VNFs) can be instantiated on any data center (DC) with enough compute resources. This flexibility further enables advanced VNF placement schemes [2], through which the cost and flexibility of network functions can be largely improved [3].

Despite the achieved flexibility, moving network functions from hardware to software poses grand concerns especially in terms of fault tolerance and reliability. For instance, VNFs are software running in VMs, which are vulnerable to various problems such as software misconfiguration, faulty VMs and software malfunctions [4]. In order to enhance VNF fault tolerance and reliability, backup VNF instances are required [5]. In case of failures, requests of stateless VNFs can be immediately redirected to one of their stand-by instances. In contrast, stateful VNFs generate states during traffic processing [6] that need to be transferred to stand-by instances in order to guarantee seamless request redirection. For instance, a stateful NAT VNF needs to maintain existing user connections to support its correct operation. If a NAT fails, the transient states created by the traffic itself have to be transferred to the backup NAT to avoid NAT disconnection. Given that such state transfers need to be continuously performed while active instances are in operation [7, 8], it could consume considerable network bandwidth resources. As such, decisions regarding 1) the placement of active instances, 2) the placement of stand-by instances, 3) traffic routing, and 4) the state transfer paths need to be jointly considered so that the number of admitted user requests can be maximized, subject to computing and bandwidth resource constraints. In this paper, we study the *fault-tolerant stateful VNF placement problem* and *the fault recovery problem* when VNFs fail, whereby the aforementioned four decisions are jointly determined under compute and bandwidth resource constraints of data centers in a cloud network.

Providing efficient solutions to the fault-tolerant VNF placement and fault recovery problems poses several fundamental challenges. First, as stated earlier, a naive solution that separately determines the

*Corresponding author: Zichuan Xu. Tel.: +8641162274514. Email address: z.xu@dlut.edu.cn

Email addresses: ygc@mail.dlut.edu.cn (Guochang Yuan), z.xu@dlut.edu.cn (Zichuan Xu), binxu.yang.13@ucl.ac.uk (Binxu Yang), wliang@cs.anu.edu.au (Weifa Liang), wchai@bournemouth.ac.uk (Wei Koong Chai), d.tuncer@imperial.ac.uk (Daphné Tuncer), a.galis@ucl.ac.uk (Alex Galis), g.pavlou@ucl.ac.uk (George Pavlou), wgwdut@dlut.edu.cn (Guowei Wu)

[1]Dalian University of Technology, P. R. China

[2]University College London, UK

[3]the Australian National University, Australia

[4]Bournemouth University, UK

[5]Imperial College London, UK

instance locations and routings may result in network congestion and admission failures. It may also lead to significant network communication costs if the active/stand-by instances are placed with long network distance to the source and destination nodes of requests. Second, the placement of stand-by instances directly influence the state update cost for VNFs. Also, the number of stand-by instances affects the fault tolerance of the networks. Clearly, a higher number of stand-by instances indicates a higher degree of fault tolerance; however, a higher number of stand-by VNF instances mean higher cost and overhead. Third, there exist various players in the NFV market. Although all of them aim to maximize their revenue in offering NFV services, they usually have different resource settings and thus will need different objectives in optimizing the performance of VNF provisioning in their networks. For example, some start-up service providers may have limited compute and bandwidth resources leased from infrastructure providers, and they want to admit as many requests as possible while guaranteeing the fault tolerance of the NFV services. Furthermore, service providers that aim to compute-intensive services, e.g., big data processing services usually want to minimize the maximum resource utilization at different locations. How to provide a series of efficient and effective solutions for different network service providers with different objectives is challenging.

There are a few studies on the fault-tolerant VNF placement. Most of them focused on either backup instances or stateless VNFs [5, 9, 10, 11, 12]. For example, Kanizo *et. al.* [10] investigated the planning-stage VNF backup instances (i.e., do not consider active instances) deployment problem while taking into account the failure probabilities of network nodes. Chantre *et. al.* [11] studied the placement problem of redundant stateless VNFs in LTE networks with a focus on deriving the optimal number of VNFs to guarantee reliability. Carpio *et. al.* [5] investigated the joint active and backup stateless VNF placement problem, but did not consider request routing and VNF state transfers. Almost all of them only focus on a specific optimization objective in their problems. To the best of our knowledge, this work is the first study that jointly considers stateful active/stand-by VNF placement, request routing and state transfers. We are also the first to consider different scenarios for network service providers with different optimization objectives to optimize the provisioning of their NFV services.

The main contributions of this paper include:

- We define a comprehensive series of optimization problems on fault-tolerant VNF placement in cloud networks for various service providers with different resource configurations to offer compute-intensive or bandwidth-hungry network services

- For start-up service providers with both limited compute and bandwidth resources,we formulate the fault-tolerant VNF placement problem to maximize the number of requests that can be admitted while minimizing the implementation cost of the admitted requests. We propose an efficient heuristic based on the joint availability of compute resources of a data center and the accumulative bandwidth resources of its inbound links. The proposed heuristic jointly computes the placement of both active

and stand-by stateful VNF instances

- For service providers offering compute-intensive network services, we consider the fault-tolerant VNF placement problem without bandwidth constraint that aims to minimize the maximum resource utilization in different locations. We propose a $(2, 4+\epsilon)$ bicriteria approximation algorithm with provable approximation ratios on the achieved cost and maximum utilization of data center in Sections 5, where ϵ is a constant that represents the accuracy parameter in the algorithm [21] for the unsplittable flow problem. The proposed algorithm exploits an approach based on auxiliary graph that allows active/stand-by instances, request routings and state update paths to be jointly considered

- Similarly, the bandwidth resource may be the bottleneck of the networks of some service providers. We thus consider the VNF placement problem that aims to minimize the maximum congestion of the links in the network while minimizing the implementation cost, by proposing a $(2\Delta, 8+2\epsilon)$ bicriteria approximation algorithm

- For the problem of dynamic fault recovery, we also propose an efficient approximation algorithm with an approximation ratio

- We then investigate the performance of the proposed algorithms by simulations and results show that the performance of the proposed algorithms is promising

- We finally build a prototype to evaluate the algorithm for the dynamic fault recovery problem in a real test-bed with both hardware and virtual switches, and results show that the proposed algorithm can be applied in real environments.

The remainder of this paper is organized as follows. We give a survey of the state-of-the-art on fault tolerance in NFV-enabled networks in Section 2. We introduce the considered scenario, the related definitions, and a series of optimization problems related to fault-tolerant placement of stateful VNFs in NFV-enabled networks in Section 3. We then propose a heuristic algorithm for the fault-tolerant VNF placement problem in Section 4. For the fault-tolerant VNF placement without compute or network bandwidth resource constraints, we propose two bicriteria approximation algorithms with approximation ratios in Sections 5 and 6, respectively. We also investigate the dynamic fault recovery problem with VNFs being placed in the network in Section 7. We finally study the performance of the proposed algorithms in Section 8, and conclude in Section 9.

2. Related Work

Previous work have focused on different aspects of the deployment of VNFs, *e.g.,* [13, 14, 9, 15]. Fayazbakhsh *et al.* [16] proposed FlowTags for flow scheduling in a network in the presence of dynamic modifications performed by middleboxes. Martins *et al.* [13] developed a virtualization system that aims to

improve network performance, by deploying modular, virtual middleboxes on lightweight VMs. Qu *et al.* [17] studied the problem of delay-aware scheduling and resource optimization for VNFs. Wang *et al.* [15] studied the problem of dynamic network function composition, and proposed a distributed algorithm, using Markov approximation method for the problem. Huang *et al.* [9] studied the problem of jointly routing and placing network functions to some servers in a data center, with the aim to maximize network throughput while meeting end-to-end delay requirements of user requests. These studies assumed failure-free scenarios and as such, are not adapted to support fault tolerant routing in case of VNF malfunctions.

VNF failures can however occur frequently [18], due to a variety of reasons, such as connectivity errors (e.g., link flaps, device unreachability, port errors), hardware faults (memory errors, defective chassis), misconfiguration (wrong rule insertion, configuration conflicts), software faults (reboot, OS errors) or excessive resource utilization. Not handling these failures seamlessly and correctly can cause significant degradation in terms of service performance and reliability. It is therefore crucial to enable robust routing by placing active and stand-by instances of VNFs, such that traffic is processed by the active instance under normal operating conditions and by one of the stand-by instances when a failure occurs. Most studies on providing fault tolerance support for NFV-enabled networks have been focusing on either designing and implementing systems with fault tolerance support [8, 19, 20] or plan-stage VNF placements based on statistical methods [10]. For example, Pico [8] was designed and implemented to provide fault tolerance support at the flow level. Kanizo *et. al.* [10] investigated the problem of NFV backup instances deployment, given the distribution of failure probabilities for different network functions. None of these studies, however, takes into account how to jointly route user requests and place active and stand-by instances of their service chains while optimizing network performance metrics.

In contrast to previous work, we investigate a comprehensive set of optimization problems related to enhancing the fault tolerance of NFV-enabled network services. Specifically, we study the problem of joint routing and placement of active and stand-by instances, such that either network throughput, congestion, or implementation cost of admitted requests is optimized, while meeting compute and network bandwidth resource constraints.

3. Preliminary

In this section, we first introduce the system model. We then describe the stateful VNFs and cost model, and we finally define a series of optimization problems.

3.1. System model

We consider a cloud network $G = (V \cup \mathcal{DC}, E)$ operated by a cloud service provider with a set V of switches, a set $\mathcal{DC}$ of data centers that are attached to some switches in V, and a set E of network links

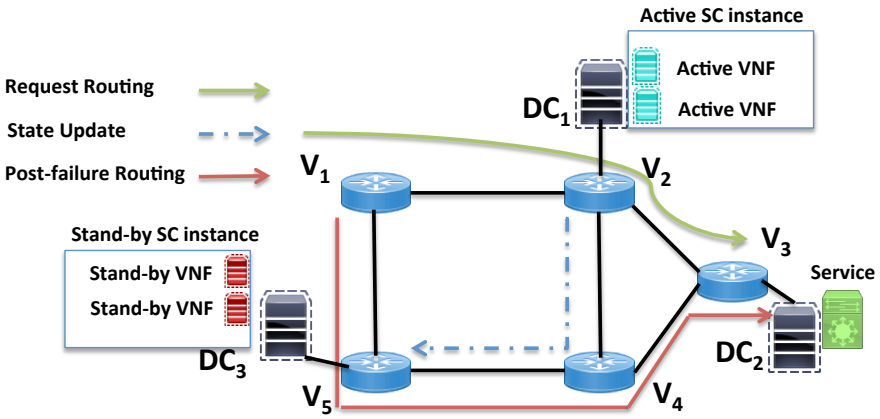


Figure 1: An example of fault-tolerant placement problem in G with a set $\mathcal{DC} = \{DC_1, DC_2, DC_3\}$ connected by a set $V = \{v_2, v_3, v_5\}$ of switches.

that interconnect the switches in V (see Fig 1). We follow the convention to assume that the number of data centers is far less than the number of switches. Each data center $DC_i \in \mathcal{DC}$ has a capacity of compute resources $C(DC_i)$ that can instantiate a limited number of VNFs. Also, each link $e \in E$ has a capacity $B(e)$ of bandwidth resource that can be allocated to transfer data traffic of user requests. Furthermore, the transmission delay on each link $e \in E$ is denoted as d_e, representing the delay for transmitting a unit packet rate over edge e. In this work, we focus on inter-data center resource allocations and hence assume that each data center and the switch node attached to it (usually the core switch in a leaf-spine data center network) are connected by high-speed optical cables with abundant network bandwidth (see Fig 1), so that the delay and communication cost on these links can be considered as negligible.

3.2. User requests with service chain requirements

We denote as $r_j = (s_j, t_j, SC_j, \rho_j, D_j)$ a user request. Each user request r_j requires to route its traffic from a source node s_j to a destination node t_j at a given packet rate ρ_j within D_j time, such that its traffic passes through one instance of its required service chain SC_j. *An instance of a service chain* is defined as an implementation of its specified VNFs in a VM. We denote as $\mathcal{R}$ the set of all r_j.

User requests usually require different types of service chains, with each type of service chains having a different sequence of VNFs. We assume that the compute resources requested by an instance of service chain SC_j for processing the traffic of r_j is proportional to its packet rate, i.e., $\rho_j \cdot c^{unit}$, where c^{unit} is a given constant representing the amount of compute resources that is needed to process each packet rate unit. The total amount of compute resource allocated to all instances of service chains in data center DC_i must not exceed its computing capacity $C(DC_i)$. Similarly, let b^{unit} be the amount of bandwidth resource that is assigned to transfer each data unit via each link $e \in E$.

The *end-to-end delay requirement* D_j of each request r_j is defined as the maximum tolerable delay experienced by its traffic from its source node s_j to destination node t_j. It consists of the processing delay of

service chain SC_j in a data center and the transfer delay due to traffic routing. Let $d(SC_j, DC_i)$ be the delay incurred by an instance of SC_j at DC_i for processing a packet rate unit. We denote by y_{ji} a binary variable indicating whether an instance of service chain SC_j of r_j is placed in DC_i. We also use a binary variable w_{je} to indicate whether edge e is used to transfer the data of request r_j. The end-to-end delay requirement of r_j thus can be defined by

$$\rho_j \Big(\sum_{e \in E} w_{je} \cdot d_e + \sum_{DC_i \in \mathcal{DC}} y_{ji} \cdot d(SC_j, DC_i) \Big) \leq D_j. \tag{1}$$

3.3. Stateful active and stand-by VNF instances

Faults can occur anywhere and at anytime in a network due, for example, to natural disasters at the locations of data centers, software malfunctions in VNFs, and hardware failures. To avoid service interruption due to such failures, active-standby failover mechanisms are usually adopted in many systems. We consider that an *active instance* of a service chain is placed into a data center, and a few *stand-by instances* of the service chain are placed into other data centers.To reduce communications overhead among different VNFs, the instances are considered at the service chain level instead of VNF level. It must be mentioned that the number of data centers that are selected for the stand-by instances play a vital role in guaranteeing the fault-tolerance level of the system. Too many stand-by instances may lead to a high overhead of maintenance and cost, while too few stand-by instances may not be able to respond to some failures.

We consider *stateful* VNFs (i.e., stateful service chains), whereby the states from the active instance need to be continuously transferred to stand-by instances while the active instance is still in operation. Such state transfer plays a vital role in enabling the seamless and correct request redirection from an active instance to a stand-by instance. Specifically, once the active instance fails (e.g., one of the VNFs in a service chain fails), its traffic can be seamlessly redirected to one of the stand-by instances for processing, as long as the states are updated to the stand-by instances. The re-directed flows may not be processed correctly otherwise. For example, maintaining statistic information about the observed traffic flows is essential to perform intrusion detection. The absence of relevant statistics in the stand-by instances of an Intrusion Detection VNF can compromise the detection.

We assume that the state update rate of each request from its active instance to stand-by instances is proportional to its packet rate, i.e., $\beta \cdot \rho_j$, where β (> 0) is a given constant. We further assume that the demand for compute resources is allocated on a stand-by instance only when this is activated. In other words, compute resources are not pre-allocated to a specific stand-by service chain instance. We assume instead that there is a *stand-by resource pool* in each data center with reserved compute resources that can be used by any stand-by instance when it becomes activated.

3.4. Cost model

Minimizing the implementation cost for user requests is usually considered as an effective objective to reduce the operational cost of network service providers. Here, *the implementation cost* of request $r_j = (s_j, t_j, SC_j, \rho_j, D_j)$ consists of (i) the processing cost incurred by the processing of user traffic by an active instance of service chain SC_j in data center DC_i, (ii) the communication cost of transferring its traffic from s_j to DC_i for processing, (iii) the communication cost of transferring the processed data from DC_i to its destination t_j, and (iv) the communication cost of updating status from DC_i to a set of data centers with the stand-by instances of SC_j. Let $c(SC_j, DC_i)$ be the cost of implementing an instance of SC_j in DC_i for the processing each packet rate unit, and $c(e)$ be the cost of transferring each packet rate unit of request r_j through link $e \in E$. Without loss of generality, we assume that the edge cost $c(e)$ is within the range of $(0, 1]$. Let y'_{ji} be the binary variable indicating whether data center DC_i has a stand-by instance of SC_j or not. The implementation cost $c(r_j)$ of r_j in an active data center and a set of data centers with stand-by service chain instances is:

$$c(r_j) = \rho_j \cdot \Big(\sum_{DC_i \in \mathcal{DC}} y_{ji} c(SC_j, DC_i) + \sum_{e \in E} w_{je} \cdot c(e) + \sum_{DC_i \in \mathcal{DC}} y_{ji} \sum_{DC_{i'} \in \mathcal{DC}} y_{ji'} \sum_{e \in p_{DC_i, DC_{i'}}} c(e) \Big), \quad (2)$$

where $p_{y,z}$ is the shortest path in G from node y to node z (in terms of path cost). As explained in Section 3.3, the stand-by instances of a service chain are only activated in case of failures but their resource demands are reserved in advance. As such, the placement of service chain instances does not take into account the cost incurred by the consumption of compute resources by stand-by instances of the service chain.

3.5. Problem definitions

Different network performance indicators can be taken into account when optimizing the service delivery process. In particular, the choice of the objective depends on the characteristics of the provider, *e.g.,* types of available resources, geographical scope of the infrastructure, nature of the offered services *etc.* We here define different versions of the VNF deployment optimization problem that cover the requirements of a wide range of network service providers.

Problem 1: A key objective for service providers with limited compute and bandwidth resources (*e.g., start-ups service providers*) is to maximize the number of admitted requests while both fully utilizing their limited resources and incurring the least operational cost. In addition, it is also essential for their services to be fault tolerant and delay-aware in order to maximize the quality of experience offered to their customers. The optimization objective thus is defined as the maximization of the admitted number of requests. More specifically, the goal of *the fault-tolerant VNF placement problem* is to determine, for all user requests r_j in $\mathcal{R}$, *i)* the placement of the active instance of service chain SC_j on data center DC_i, *ii)* the number and placement of stand-by instances to a set of other data centers, as well as *iii)* the routing path for requests

from s_j to t_j via DC_i and the state update path from DC_i to the data centers with stand-by instances of SC_j, so that the number of admitted requests is maximized while the total cost associated with their implementation is minimized, subject to compute resource capacity $C(DC_i)$, network bandwidth capacity $B(e)$ for $e \in E$, and end-to-end delay constraints.

Let x_j be a binary variable that indicates whether user request r_j is admitted. We denote by q_{jv} a binary indicator variable that shows whether switch $v \in V$ is used to forward the traffic of r_j. Let $\delta(v)$ denote the incident edges of switch node $v \in V$. Denote by χ_{je} a binary variable that shows whether edge e is used to transfer the state update data of request r_j. Assuming that $\mathbb{B}$ represents the budget of implementing requests, we can formulate the objective of **Problem 1** as an Integer Linear Program (ILP) with quadratic constraints as follows,

$$\textbf{ILP}: \qquad \max \sum\nolimits_{r_j \in \mathcal{R}} x_j, \tag{3}$$

subject to the following constraints,

$$\sum\nolimits_{DC_i \in \mathcal{DC}} y_{ji} = x_j, \qquad \forall r_j \in \mathcal{R} \tag{4}$$

$$\sum\nolimits_{DC_i \in \mathcal{DC}} y'_{ji} \geq x_j, \qquad \forall r_j \in \mathcal{R} \tag{5}$$

$$\sum_{r_j \in \mathcal{R}} y_{ji} \cdot \rho_j \cdot c^{unit} \leq C(DC_i), \quad \forall DC_i \in \mathcal{DC} \tag{6}$$

$$\sum_{r_j \in \mathcal{R}} \big(w_{je} + \chi_{je} \cdot \beta\big) \cdot \rho_j \cdot b^{unit} \leq B(e), \quad \forall e \in E \tag{7}$$

$$\sum_{e \in \delta(v)} w_{je} \leq 2 \cdot q_{jv}, \quad \forall r_j \in \mathcal{R} \tag{8}$$

$$\sum_{e \in \delta(s_j)} w_{je} \leq 1, \quad \forall r_j \in \mathcal{R} \tag{9}$$

$$\sum_{e \in \delta(t_j)} w_{je} \leq 1, \quad \forall r_j \in \mathcal{R} \tag{10}$$

$$\sum_{r_j \in \mathcal{R}} c(r_j) \leq \mathbb{B}, \tag{11}$$

$$\rho_j \Big(\sum_{e \in E} w_{je} \cdot d_e + \sum_{DC_i \in \mathcal{DC}} y_{ji} \cdot d(SC_j, DC_i) \Big) \leq D_j, \quad \forall r_j \in \mathcal{R}, \tag{12}$$

$$x_i,\ y_{ji},\ y'_{ji},\ w_{je} \in \{0, 1\}, \tag{13}$$

Constraints (4) guarantee that each admitted user request has its service chain SC_j active instance in a data center. Constraints (5) make sure there is at least one data center that implements a stand-by instance of SC_j of request r_j if it is admitted. Constraints (6) and Constraints (7) guarantee that the computing capacity of each data center DC_i and the bandwidth resource capacity of each link $e \in E$ are satisfied, respectively. Constraints (8) makes sure that if a switch $v \in V$ is selected to forward the traffic of r_j, either at most two of its incident edges are used to forward traffic (one for incoming traffic and one for outgoing

traffic) or one incident edge is used to forward both incoming and outgoing traffic. Constraints (9) and (10) ensure that no traffic goes to source node s_j and no traffic leaves destination node t_j of request r_j, respectively. Constraints (11) imposes a budget on the cost of implementing all admitted requests. The budget can be used to control and possibly minimize that cost. It should be highlighted that quadratic nature of constraints (11) affects the hardness of the ILP's resolution and optimal solutions can only be obtained for small problem sizes. Constraints (12) guarantee that the end-to-end delay requirement of each admitted request r_j is met.

Problem 2: An important objective for service providers with distributed data centers offering compute-intensive services is to balance resource usages between available locations (*e.g.,* geographical load balancing), such that users in different locations can be provided with maximum resource availability and guaranteed quality of experience. In this setting, we assume that links in G have abundant resources to implement all requests in $\mathcal{R}$. The objective of *the fault-tolerant VNF placement problem without bandwidth capacity constraint* is to minimize the maximum data center utilization for all data centers, i.e.,

$$\min \max_{DC_i \in \mathcal{DC}} \sum_{r_j \in \mathcal{R}} \frac{y_{ji} \rho_j \cdot c^{unit}}{C(DC_i)}, \tag{14}$$

subject to constraints (6), (8), (9), (10), (11), (12), (13), and:

$$\sum\nolimits_{DC_i \in \mathcal{DC}} y_{ji} \geq 1, \qquad \forall r_j \in \mathcal{R} \tag{15}$$

$$\sum\nolimits_{DC_i \in \mathcal{DC}} y'_{ji} \geq 1, \qquad \forall r_j \in \mathcal{R}. \tag{16}$$

In this problem, all resources in the network are sufficient to implement all requests. Constraints (15) and (16) are used to guarantee that each request in $\mathcal{R}$ is admitted.

Problem 3: We focus in this problem on service providers offering intensive data-transfer services, *e.g.,* big data processing network functions. Given that these services are bandwidth hungry, network link bandwidth can act as a bottleneck. *The fault-tolerant NFV placement problem without computing capacity constraint* consists in determining the placement of the service chain active instance of each request $r_j \in \mathcal{R}$, as well as the number and placement of stand-by instances, such that the congestion on links in G and the cost of implementing all requests are minimized, *i.e.,*

$$\min \max_{e \in E} \sum_{r_j \in \mathcal{R}} \frac{w_{je} \cdot \rho_j}{B(e)}, \tag{17}$$

subject to constraints (15), (16), (7), (8), (9), (10), (11), (12).

Problem 4: Problems 1, 2 and 3 compute for each request a set of stand-by instances. In Problem 4, we focus on determining which of these stand-by instances to select when the active instance of a VNF fails. We

refer to this problem as *the dynamic fault recovery problem*. We assume a model where time is divided into equal slots. The active instance of a request can fail at any future time slot once the request is admitted. Let $\mathcal{R}_f(t)$ be the set of requests with active service chain instances failed at time slot t. As explained in Section 3.3, compute resource are reserved at each data center hosting stand-by instances. Let $C_f(DC_i)$ be the computing resource capacity of the stand-by resource pool at data center DC_i. The total amount of resource allocated to activated stand-by instances of service chains should not exceed $C_f(DC_i)$. Similarly, denote by $B_f(e)$ the bandwidth resource capacity that is reserved for transmitting the data traffic from/to activated stand-by instances. Note that without such computing and bandwidth resource reservation, the stand-by instances of VNFs may not be able to be activated when necessary.

The objective of *the dynamic fault recovery problem* is to select, for each request r_j ($\in \mathcal{R}_f(t)$), a data center from the set $\mathcal{DC}_j^s$ and determine the routing of the traffic of r_j to the selected data center, such that the cost of the recovery procedure in terms of compute and network bandwidth resource consumption is minimized, subject to compute resource capacity $C_f(DC_i)$ and bandwidth resource capacity $B_f(e)$ constraints.

All these problems are clearly NP-hard given that special versions without considering fault-tolerant requirements and/or bandwidth resource constraints are NP-hard by simple reduction from another NP-hard problem, the unsplittable single-source flow problem [21]. In this paper, we build upon the algorithm proposed in [21] to solve this problem.

More specifically, in *the unsplittable single-source flow problem* [21], we are given a network $G = (V, E, u)$, a source node s, and a set of commodities with a set of sink nodes. Each commodity m has a demand σ_m for transferring σ_m number of flows from the source node to its sink node t_m along a single $s - t_m$ path. The total number of flows routed across any edge $e \in E$ should not violate its capacity u_e.

4. An Efficient Heuristic for the Fault-Tolerant VNF Placement Problem

Due to the NP-hardness of the fault tolerant VNF placement problem, we here propose an efficient heuristic to solve it.

4.1. Algorithm

To avoid poor performance in terms of request admission rate and cost, the placement of active/stand-by instances, request routings and update paths need to be jointly computed. Conventional approaches such as naive *greedy algorithm* select data centers for the active and stand-by instances separately. It first finds the data center with the largest amount of available compute resources to host the active instance for SC_j of r_j, and then selects a random number of data centers with lowest transfer costs to host stand-by VNF instances for r_j. As a result, the separate placement and routing decision may result in situations where no update paths are available from the active instance to one of its stand-by instances due to link congestions.

Table 1: Symbols

Symbols	Meaning
$G = (V \cup \mathcal{DC}, E)$	a cloud network with a set V of switches, a set DC of data centers, and a set E of edges
DC_i	a data center DC_i in DC
v_{DC_i}	the switch that attaches DC_i
$C(DC_i)$	the capacity of compute resources of data center DC_i
e and d_e	link $e \in E$ and the delay of implementing a unit packet along e
$B(e)$	the network bandwidth capacity for $e \in E$
r_j, s_j, t_j	a request and its source and destination nodes
D_j	the end-to-end delay requirement of request r_j
ρ_j and SC_j	the packet rate and service chain of request r_j
$\mathcal{R}$	a set of requests
c^{unit}, b^{unit}	the amounts of compute and bandwidth resources that is needed to process each unit packet rate.
$d(SC_j, DC_i)$	the processing delay by an instance of SC_j at DC_i for the processing of a unit amount of packet rate
y_{ji}	the binary variable that shows whether an instance of service chain SC_j of r_j is placed to DC_i.
w_{je}, χ_{je}	the binary indicator variables that show whether edge e is used to transfer the data and the state update of r_j, respectively.
β	the ratio between the state update rate and the packet rate of user requests
$c(SC_j, DC_i)$	the cost of implementing an instance of SC_j in DC_i for the processing a unit amount of packet rate
$c(e)$	the cost of transferring a unit packet rate for request r_j through link $e \in E$.
y'_{ji}	the binary variable showing whether data center DC_i has a stand-by instance of SC_j
$c(r_j)$	the implementation cost of r_j
$p_{y,z}$	the shortest path in G from node y to node z in terms of path cost
q_{jv}	the binary indicator variable that shows whether switch $v \in V$ is used to forward the traffic of r_j
$\delta(v)$	the incident edges of switch node $v \in V$
$\mathbb{B}$	the budget of implementing requests
$\mathcal{R}_f(t)$	the set of requests that have their assigned active service chain instances failed at time slot t
$\mathcal{DC}^s_j$	the data centers where the stand-by instances of request $r_j \in \mathcal{R}_f(t)$ are hosted
$C_f(DC_i)$	the amount of computing resource in the stand-by resource pool with resources reserved for activated stand-by instances
$B_f(e)$	the bandwidth resource capacity that is reserved for transmitting the data traffic from/to activated stand-by instances
s and t_m	the single source and the sink of commodity m in the unsplittable single-source flow problem
σ_m	the demand of each commodity m in the unsplittable single-source flow problem
u_e	the capacity for each link e in the unsplittable single-source flow problem
$NR(DC_i, j)$	the ranking of DC_i after considering the $(j-1)$th request in the sorted list
$A(DC_i, j)$, $A(e, j)$	the available compute and bandwidth resources of DC_i and link e after considering the $(j-1)$th request
E^i_{adj}	the set of inbound links of DC_i
DC_{hr}	based on the obtained ranking, the algorithm selects the data center with the highest rank
K	a threshold for the number of data centers that can be used for stand-by instances($1 \leq K \leq \lvert\mathcal{DC}\rvert$)
L_{hr}	the list of data centers
ϵ	a constraint with $\epsilon > 0$ that represents the accuracy parameter in the algorithm [**?**] for the unsplittable flow problem
$G' = (V', E')$	the auxiliary graph for the problem without bandwidth constraint
DC'_i, s_0	the virtual data center node and the common source in auxiliary graph G'
$\mathcal{DC}^s_j$	the set of data centers for stand-by instances of SC_j
$T_2(m, n)$	the time to solve a fractional minimum-cost flow problem with m edges and n nodes in the flow graph
Δ	the diameter of the given network
$G'' = (V'', E'')$	the auxiliary graph for the fault-tolerant VNF placement problem without computing capacity constraints
$r_{j,1}$ and $r_{j,2}$	the two virtual requests of user request r_j
$f_{j,1}$	the unsplittable flow for virtual request $r_{j,1}$ from s_0 to virtual request node in G''
$f_{j,2}$	the unsplittable flow for virtual request $r_{j,2}$ from s_0 to its virtual request node in G''
f'	the single-source unsplittable flow in the auxiliary graph G''
$f_{j,1}$ and $f_{j,2}$	the flows for virtual requests $r_{j,1}$ and $r_{j,2}$ in auxiliary graph G''
$G''' = (V''', E''')$	the auxiliary graph for the fault recovery problem
$B_{f,max}$, $C_{f,max}$	the maximum edge and node capacities
f'''	the unsplittable flow from s_0 to request node r_j in G'''

In contrast, our heuristic jointly selects a data center for the active instance and a number of data centers for its stand-by instances. Specifically, the heuristic first sorts all requests in $\mathcal{R}$ in increasing order of their packet rates, and then sequentially considers the requests in the sorted list. Next, for the jth request r_j in the sorted list, the algorithm ranks data centers based on the increasing order of the product of the available compute resources and the accumulative available network bandwidth resources of data centers' inbound links. Let $NR(DC_i, j)$ be the ranking of DC_i after considering the $(j-1)$th request in the sorted list. Also, denote by $A(DC_i, j)$ and $A(e, j)$ the available compute and bandwidth resources of DC_i and link e after considering the $(j-1)$th request. Then,

$$NR(DC_i, j) = A(DC_i, j) \cdot \sum_{e \in E^i_{adj}} A(e, j), \tag{18}$$

where E^i_{adj} is the set of inbound links of DC_i. The idea of such ranking is to find a set of data centers with enough compute and network bandwidth resources for both active and stand-by instances.

Based on the obtained ranking, the algorithm selects the data center with the highest rank, denoted DC_{hr}. Then, the algorithm checks if (1) DC_{hr} has enough compute resources to host an active instance of SC_j for r_j; and (2) if the shortest path from s_j to t_j via DC_{hr} has enough bandwidth resources to transfer r_j at rate ρ_j. The algorithm also checks whether (3) DC_{hr} conforms to r_j's delay requirement. If the above three requirements are all satisfied, DC_{hr} is selected as the data center for the active service chain instance of SC_j. The algorithm then searches data centers for the stand-by service chain instances of r_j. Let $\mathcal{DC}^s_j$ be the set of data centers for stand-by service chain instances of r_j. To this end, the rest of data centers except DC_{hr} are sorted in the increasing order of state update costs to DC_{hr}. Each data center in the sorted DC list is further added to $\mathcal{DC}^s_j$ until there is a data center that cannot meet the bandwidth resource requirement for updating states from DC_{hr}. To avoid all the other data centers to be selected to host stand-by instances, we set a threshold K $(1 \leq K \leq |\mathcal{DC}|)$ for the number of data centers that can be used for stand-by instances. This prevents a large number of data centers to be selected to place stand-by instances and as such avoids the creation of unnecessary burden for state updates. If no stand-by data center exists after considering the rest of the data centers, request r_j is rejected.

In case the aforementioned constraints cannot be satisfied, DC_{hr} is added to $\mathcal{DC}^s_j$ as the accumulative bandwidth resources to nearby data centers might make DC_{hr} a promising candidate for stand-by instances. Data centers other than DC_{hr} are sorted in a list based on the increasing accumulative communication cost to DC_{hr}. Denote by L_{hr} such a list of data centers. The algorithms then iterates through data centers in L_{hr} until a data center, say DC_i, that can serve the active service chain instance is found, i.e., a data center that meets constraints (1), (2) and (3). Once DC_i is found, it is used to host the active instance of r_j. Then, in L_{hr}, only the data centers that have enough bandwidth resources for state updates rate $\beta \cdot \rho_j$ from DC_i

(the data center with the active instance) are added to $\mathcal{DC}_j^s$ (with $|\mathcal{DC}_j^s| \leq K$). If neither such data center can be found for its active instance nor a set of data centers can be determined for its stand-by instances, r_j is rejected.

The above procedure continues until all requests in $\mathcal{R}$ are considered. The details of the proposed heuristic are shown in Algorithm 1.

Algorithm 1 `Heuristic`

Input: Network $G(V \cup \mathcal{DC}, E)$; Set of requests $r_j \in R$ where $r_j = (s_j, t_j, SC_j, \rho_j, D_j)$, K.
Output: Assignments of each request in $r_j \in \mathcal{R}$ to a data center for the active instance of its service chain SC_j, and to a set $\mathcal{DC}_j^s$ of data centers for stand-by instances of r_j.

for $r_j \in \mathcal{R}$ **do**
  $Sorted_{list} \leftarrow$ SortIncreaseOrder($\mathcal{DC}$) based on Eq. (18)
  $DC_{hr} \leftarrow Sorted_{list}.getFirst()$;
  $\mathcal{DC}_j^s \leftarrow \emptyset$;
  Let DC_j^a be the data center for the active instance of SC_j;
  $A(p_{(s_j,DC_{hr})}) \leftarrow G.shortestPathAvailBandwidth(s_j, DC_{hr})$;
  $A(p_{(DC_{hr},t_j)}) \leftarrow G.shortestPathAvailBandwidth(DC_{hr}, t_j)$;
  if $\rho_j \leq A(p_{(s_j,DC_{hr})})$ && $\rho_j \leq A(p_{(DC_{hr},t_j)})$ && $D_{hr} \leq D_j$ **then**
    $DC_j^a \leftarrow DC_{hr}$;
    $Update_{list} \leftarrow$ SortIncreaseOrder($\mathcal{DC} \setminus DC_{hr}$) based on state update costs to DC_{hr};
    for each $DC_i \in L_{hr}$ **do**
      $\mathcal{DC}_j^s \leftarrow \mathcal{DC}_j^s \cup \{DC_i\}$
      if $K = |\mathcal{DC}_j^s|$ or $A(p_{(DC_i,DC_{hr})}) \leq \beta \cdot \rho_j$ **then**
        Break;
  else
    $\mathcal{DC}_j^s \leftarrow \mathcal{DC}_j^s \cup \{DC_{hr}\}$
    $L_{hr} \leftarrow$ SortIncreaseOrder($\mathcal{DC} \setminus DC_{hr}$) following state update costs to DC_{hr};
    for each $DC_i \in L_{hr}$ **do**
      if $DC_j^a \neq NIL$ && $A(p_{(DC_j^a,DC_i)}) \geq \beta \cdot \rho_j$ && $|\mathcal{DC}_j^s| \leq K$ **then**
        $\mathcal{DC}_j^s \leftarrow \mathcal{DC}_j^s \cup \{DC_i\}$;
      else
        if $\rho_j \leq A(p_{(s_j,DC_i)})$ && $\rho_j \leq A(p_{(DC_i,t_j)})$ && $D_i \leq D_j$ **then**
          $DC_j^a \leftarrow DC_i$;
        else
          if $|\mathcal{DC}_j^s| \leq K$ **then**
            $\mathcal{DC}_j^s \leftarrow \mathcal{DC}_j^s \cup \{DC_i\}$;
  Update the available resources of all data centers and network link resources
return The assigned data center to place the service chain of each request for the processing of its traffic, and a set of data centers to replicate its service chain.

The performance of the proposed heuristic is given by the following theorem.

Theorem 1. *Given a cloud network $G = (V \cup \mathcal{DC}, E)$, a set of requests with each represented by $r_j = (s_j, t_j, SC_j, \rho_j, D_j)$, there is an algorithm, i..e,* **Algorithm** *1, which delivers a feasible solution to the fault-tolerant VNF placement problem in $O(|\mathcal{R}|(|\mathcal{DC}| \log |\mathcal{DC}|) + (|V| + |\mathcal{DC}|)^3)$ time.*

Please see the proof in the appendix.

5. A $(2, 4+\epsilon)$ Bicriteria Approximation Algorithm for the Problem without Bandwidth Constraint

We now consider the fault-tolerant VNF placement problem without the bandwidth capacity constraint of links in the cloud network G. We assume that all requests in $\mathcal{R}$ can be admitted, and the objective thus is to minimize the maximum data center utilization for all data centers. We here propose a bicriteria approximation algorithm with an approximation ratio of $(2, 4+\epsilon)$. Such a ratio indicates that (1) the implementation cost of all requests is twice the optimal cost, and (2) the minimum maximum utilization of compute resources in a data center is $(4+\epsilon)$ times the optimal one, where ϵ is a constant with $\epsilon > 0$.

5.1. Overview

Solving the fault-tolerant VNF placement problem without bandwidth capacity constraint is to balance the workloads among data centers by not only minimizing their maximum resource utilization but also minimizing the total implementation costs of the requests. One challenge is with respect to the tradeoff between the balance of data center resource utilizations and the implementation costs of requests. For instance, the active instance of some requests may have to be placed into data centers with high communication costs in order to achieve a balanced workload among data centers. In order to achieve a near optimal solution, we jointly consider the active/stand-by instance placements, request routings and state update paths.

The idea behind the proposed approach is to reduce the fault-tolerant NFV placement problem without the bandwidth capacity constraint in G into a single-source unsplittable flow problem [21] in an auxiliary graph $G' = (V', E')$. Then, a feasible unsplittable flow in G' that minimizes both the implementation cost of requests and the maximum congestion of links in G' is a feasible solution to the original problem in G. Note that the aim of the single-source unsplittable flow problem is, given a network $G = (V, E, u)$, a source vertex s, and a set of M commodities with sinks t_1, ..., t_M and associated real-valued demands σ_1, ..., σ_M, to route the demand σ_m of each commodity m along a single $s - t_m$ flow path so that the congestion, i.e., $max_{e \in E}\{\frac{f_e}{u_e}, 1\}$, and the cost of flow f are minimized, while the edge capacities constraints of G are met.

5.2. Bicriteria approximation algorithm

We now describe the bicriteria approximation algorithm. We first construct the auxiliary graph $G' = (V', E')$. Recall that the traffic of each request r_j is processed by an active instance of its SC_j in a data center, and by one of its stand-by instances in other data centers if the active instance fails. Thus, each data center DC_i corresponds to a *data center node* (see Fig. 2), and is added into the auxiliary graph G', i.e., $V' \leftarrow \{DC_i \mid 1 \leq i \leq |\mathcal{DC}|\}$. For each data center node DC_i, we further add a *virtual data center node* DC'_i (see Fig. 2) into V', i.e., $V' \leftarrow V' \cup \{DC'_i\}$ so that the computing capacity constraint of each data center is converted into an edge constraint in the auxiliary graph. Next, for each data center node, we add a few *stand-by set nodes* to G', whereby each stand-by set node represents a set of candidate data centers

for stand-by instances (see Fig. 2). Specifically, the stand-by set nodes of DC_i are different combinations of data centers from $\mathcal{DC} \setminus \{DC_i\}$ whereby each stand-by set node has no more than K data centers. For example, in Fig. 2, DC_1 has three stand-by set nodes, i.e., $\{DC_2\}$, $\{DC_3\}$, and $\{DC_2, DC_3\}$. This means that the stand-by instance for an active instance in DC_1 may be placed to DC_2, DC_3 or both of them. Note that a stand-by set node will not be added twice (e.g., there is only one DC_1 in stand-by set nodes). Last, we add a *request node* into V' for each request r_j, and add a common source s_0 for all requests into V'.

An edge from the common source s_0 to each of stand-by set node is added into E'. Its capacity and cost are set to infinity and zero, respectively (i.e., no bandwidth constraint). Also, there is an edge from each stand-by set node to a data center node DC_i if DC_i is not in the set of data centers represented by the stand-by set node (e.g., DC_1 has edges to DC_2, DC_3 and DC_2 & DC_3 in Fig. 2). The capacity of the edge is set to infinity, and its cost is the accumulative cost of state updates from DC_i to the data centers within the set of data centers represented by the stand-by set node. Further, an edge from DC_i to DC_i' is added. Its capacity is the processing capacity of DC_i, and its cost is set to 0. We add an edge from each DC_i' to a request r_j if DC_i provides a total delay (e.g., sum of processing and communication delay) for request r_j smaller than the request delay requirement. The capacity of this edge is set to infinity. Its cost is the total cost of processing costs of DC_i for request r_j plus the communication costs from s_j to DC_i and from DC_i to t_j at packet rate ρ_j. Fig. 2 shows an example of the constructed auxiliary graph G'. Given the constructed

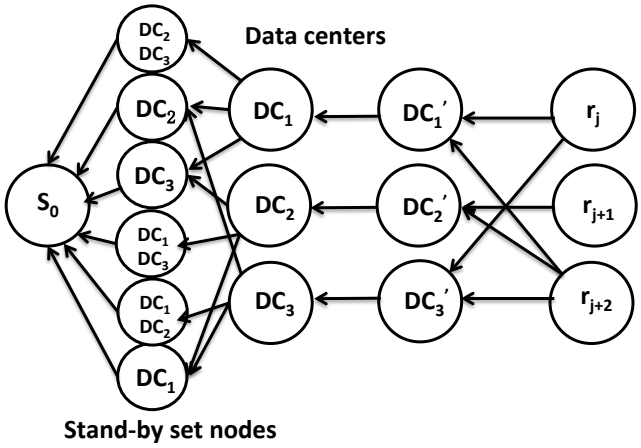


Figure 2: An example of the auxiliary graph $G' = (V', E')$ constructed from network G with a set $\mathcal{DC} = \{DC_1, DC_2, DC_3\}$ of DCs that are connected by a set $V = \{v_2, v_3, v_5\}$ of switches. $\mathcal{R} = \{r_j, r_{j+1}, r_{j+2}\}$.

auxiliary graph $G'(V', E')$, the original problem is transferred to the problem of single source unsplittable flow problem in G'. To find a feasible flow f in G', the algorithm presented in [21] is invoked. The main steps of the approximation algorithm are shown in Algorithm 2.

Algorithm 2 A $(2, 4+\epsilon)$ bicriteria approximation algorithm for the fault-tolerant VNF placement problem without network bandwidth constraint

Input: A network $G(V \cup \mathcal{DC}, E)$, a set requests $r_j \in R$ where $r_j = (s_j, t_j; SC_j, \rho_j, D_j)$.

Output: Assignments of each requests in $r_j \in \mathcal{R}$ to a data center for active instances of service chain SC_j and to a set $\mathcal{DC}_j^s$ of data centers for stand-by instances.

1: Construct an auxiliary graph $G' = (V', E')$ from network $G(V \cup \mathcal{DC}, E)$ as exemplified by Fig. 2;
2: Find a single-source unsplittable flow$f$ in the auxiliary graph G' by applying the algorithm presented in [21];
3: The requests that are assigned into DC_i in the flow f will be processed by an instance of a service chain in DC_i, and request will be assigned a set of data centers that are represented by the stand-by set node in f.
4: **return** The assigned data center to place the service chain of each request for the processing of its traffic, a set of data centers to replicate its service chain, the request routings and update paths.

5.3. Algorithm analysis

We now analyze the correctness and performance of the proposed algorithm.

Theorem 2. *Given a network $G = (V \cup \mathcal{DC}, E)$, let $\mathcal{R}$ be a set of requests with each represented by $r_j = (s_j, t_j, SC_j, \rho_j, D_j))$. Algorithm 2 delivers a bicriteria approximate solution with an approximation ratio of $(2, 4+\epsilon)$ with (1) the implementation cost of all requests twice the optimal cost, and (2) the minimum maximum utilization of compute resource in a data center $(4+\epsilon)$ times the optimal one, for the fault-tolerant VNF placement problem without bandwidth capacity constraint, in $O(T_2(|\mathcal{R}| + |V| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}, |\mathcal{R}| \cdot |\mathcal{DC}| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}))$ time, where $T_2(m, n)$ is the time to solve a fractional minimum-cost flow problem with m edges and n nodes in the flow graph, and ϵ is a constant with $\epsilon > 0$.*

Please see the proof in the appendix.

6. A $(2\Delta, 8+2\epsilon)$ Bicriteria Approximation Algorithm for the Fault-Tolerant VNF Placement Problem without Computing Capacity Constraints

In this section we deal with the fault-tolerant NFV placement problem that aims to minimize the maximum congestion of links in network G and the cost of implementing all requests in $\mathcal{R}$. According to realistic traffic patterns in various networks, we assume that all requests are mice flows with packet rates far smaller than the minimum edge capacity in G [22]. Following ubiquitous heavy-tailed distributions in the Internet – most (80% of the traffic is actually carried by only a small number of connections (elephants), while the remaining large amount of connections are very small in size or lifetime (mice)), we further assume that the total network bandwidth demand of all mice flows is far less than the total available network resources of G. With the mentioned assumptions, we first devise a $(2\Delta, 8+2\epsilon)$ bicriteria approximation algorithm by a non-trivial reduction to the single-source unsplittable flow problem, where Δ is the diameter of the given network, i.e., the longest shortest path between any two nodes in the network. We then analyze the approximation ratio of the proposed algorithm.

6.1. Basic idea

To reduce the problem to the single-source unsplittable flow problem in an auxiliary graph G'', we treat each request in $\mathcal{R}$ as two single *virtual requests* without service chain requirements before finding routes for

them in the auxiliary graph G''. The rationale behind is that the path that routes the data traffic of r_j can be split into two segments: the segment from its source node s_j to a data center DC_i and the other segment from DC_i to its destination node t_j. We thus can consider these two segments as two paths that are found to route the traffic of two independent virtual request that require to transfer data from source node s_j to a data center and from destination node t_j to a data center. To this end, we carefully set the sources and destinations in auxiliary graph G'' of such virtual requests, which will be introduced later once the construction of G'' is elaborated on in the rest of this section. However, if the two virtual requests of each request are dealt with independently, the solution obtained may not correspond to a feasible solution to the original problem, since the two virtual requests of each request may be assigned to different data centers violating the request's "unsplittable" constraint for its traffic. We thus modify the solution to the single-source unsplittable flow problem in G'' to a feasible solution to the original problem without two much deterioration of the obtained approximation guarantee.

6.2. An approximation algorithm

We now describe the algorithm by first introducing the construction of the auxiliary graph $G'' = (V'', E'')$, and then elaborating on the algorithm.

To construct the auxiliary graph G'', we add all switch nodes in the original network G into auxiliary graph G''. We also create a *DC node* for each data center DC_i, and add it into the auxiliary graph G'', i.e., $V'' \leftarrow V'' \cup \{DC_i\}$. Similar to the auxiliary graph construction in Section 2, we add a *stand-by set node* for a candidate set of data centers where a number of stand-by instances of the service chain of a user request may be placed. In addition, for each virtual request, we add a *virtual request node* into V''. A common source s_0 for all virtual requests are added into V''.

For the edges in G'', we first add the links in the original network G into G''. The costs and capacities of these edges are the same as those in G. There is an edge from each virtual request node to each stand-by set node. The capacity and cost of each of these edges are set to infinity and zero, respectively. An edge from each stand-by set node to its corresponding data center node DC_i is added, where its capacity is infinity and cost is the cost of updating processing states from DC_i to the data centers within the set of data centers represented by the stand-by set node. In addition, an edge from each data center node DC_i to the switch node in V where DC_i is attached is added, with its capacity and cost being set to infinity and zero, respectively. To make sure the demands of virtual requests $r_{j,1}$ and $r_{j,2}$ are met, an edge from s_j to $r_{j,1}$ and an edge from t_j to $r_{j,2}$ is added into E''. The capacity and cost of each of these two edges is set to infinity and zero, respectively. Fig. 3 shows an example of the constructed auxiliary graph G''.

Let $r_{j,1}$ and $r_{j,2}$ be the two virtual requests of user request r_j. Given the construction of G'', virtual request $r_{j,1}$ has source s_0 and destination s_j, and $r_{j,2}$ has source s_0 and destination t_j. Clearly, if $r_{j,1}$ is implemented by G'', along its implementation path from s_0 to s_j, there will be a data center. This represents

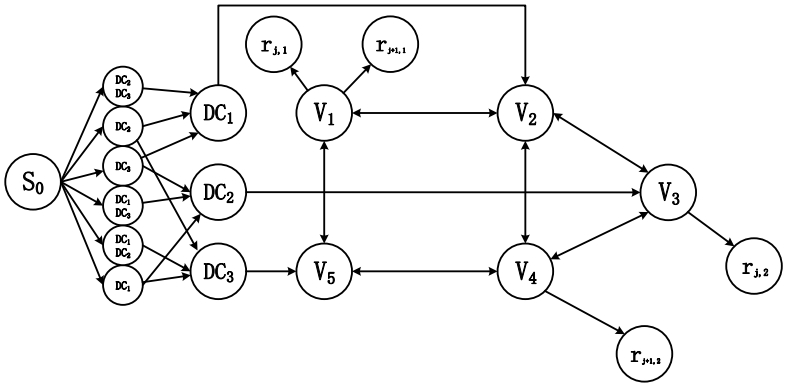


Figure 3: An example of the auxiliary graph $G'' = (V'', E'')$ constructed from network G with a set $\mathcal{DC} = \{DC_1, DC_2, DC_3\}$ of data centers that are connected by a set $V = \{v_2, v_3, v_5\}$ of switches. $\mathcal{R} = \{r_j, r_{j+1}\}$, where r_j has source v_1 and destination v_3 and r_{j+1} has source v_1 and destination v_4.

that the traffic of r_j will be routed from s_j to the selected DC for processing. Similarly, the implementation of $r_{j,2}$ means that the traffic of r_j will be forwarded to its destination t_j, after being processed by a data center on the path.

Having constructed the auxiliary graph G'' and the set of virtual requests, we proceed by transferring the original problem into the single-source unsplittable flow problem. To this end, we consider each virtual request as a commodity that has a demand of ρ_j from the common source s_0 to its corresponding virtual request node in G''. Let $f_{j,1}$ be the unsplittable flow for virtual request $r_{j,1}$ from s_0 to virtual request node in G'', denote by $f_{j,2}$ the unsplittable flow for virtual request $r_{j,2}$ from s_0 to its virtual request node in G''. Clearly, according to the construction of the auxiliary graph, virtual requests $r_{j,1}$ and $r_{j,2}$ of each request r_j may be assigned to different data centers, which is not feasible for the original problem.

To make the obtained solution feasible for the fault-tolerant VNF placement problem, we then adjust it by routing the traffic of one virtual request from its assigned data center to the other data center that is assigned to another virtual request. Specifically, suppose the traffic of virtual request $r_{j,1}$ of r_j is assigned to data center DC_m by flow $f_{j,1}$ in G'' and virtual request $r_{j,2}$ is allocated to DC_n by flow $f_{j,2}$. The data center with a smaller processing cost, say DC_m, will be selected to process the traffic of request r_j. The traffic of r_j then will not be processed to the other data center DC_n. Instead, DC_n will be serving as a switch node, and the traffic of r_j will be forwarded from DC_n to DC_m for processing. This can be done by a simple application of the algorithm for the single-source unsplittable flow problem in the network G'', where its capacities on links and edges are the residual capacities after implementing the virtual requests. Namely, the traffic of r_j will be considered as a community that requires to be routed from DC_n to DC_m. Let $p_{n,m}$ be the path found by the single-source unsplittable. The overall path that route the traffic of r_j will be composed of three segments: (1) the segment from s_j to data center DC_n, which is derived by the

Algorithm 3 A $(2\Delta, 8 + 2\epsilon)$ bicriteria approximation algorithm for the NFV placement and replication problem

Input: A network $G(V \cup \mathcal{DC}, E)$, a set $\mathcal{R}$ of requests with each request r_j demanding to transfer its traffic from source s_j to destination t_j at packet rate ρ_j, service chain SC_j requested by each request r_j, and its end-to-end delay requirement D_j.

Output: An assignment of requests in $\mathcal{R}$ to a DC to place an active instance of its service chain and a set of nearby data centers to place a number of stand-by instances of its service chain in case failures happen in the active instance.

1: Split each user request r_j into two virtual requests without service chain requirements, i.e., $r_{j,1}$ and $r_{j,2}$, where $r_{j,1}$ has source s_0 and destination s_j, $r_{j,2}$ has source s_0 and destination t_j;
2: Construct an auxiliary graph G'' as illustrated in Fig. 3;
3: Consider each virtual request as a commodity that has a demand of ρ_j from the common source s_0 to its corresponding virtual request node in G''.
4: Find a single-source unsplittable flowf' in the auxiliary graph G'' by applying the algorithm in [21], given the constructed auxiliary graph G'' and set of commodities;
5: If the virtual requests of each request are not assigned to a single data center in f', assign both virtual requests to one of the data centers, by routing the traffic of one virtual request from its assigned data center to the other data center with lower processing cost;
6: **return** The assigned data center to place the service chain of each request for the processing of its traffic, and a set of data centers to replicate its service chain.

flow $f_{j,1}$ for virtual request $r_{j,1}$ in auxiliary graph G'', (2) the path $p_{n,m}$ from DC_n to DC_m, derived by a single-source unsplittable flow in the residual network G after implementing the virtual requests of each request, and (3) the segment from DC_m to t_j, which can be derived from the flow $f_{j,2}$ for virtual request $r_{j,2}$ in G''.

The details of the proposed algorithm is described in algorithm 3.

We here analyze the performance of the proposed approximation algorithm in Theorem 3.

Theorem 3. *Given a network $G = (V \cup \mathcal{DC}, E)$, let $\mathcal{R}$ be a set of requests with each represented by $r_j = (s_j, t_j; SC_j, \rho_j, D_j))$. Algorithm 3 delivers a bicriteria approximate solution with an approximation ratio of $(2\Delta, 8 + 2\epsilon)$ in $O(T_2(|\mathcal{R}| + |V| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}, |E| + |\mathcal{R}| \cdot |\mathcal{DC}| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}))$ time, where $T_2(m, n)$ is the time to solve a fractional minimum-cost flow problem with m edges and n nodes in the flow graph, ϵ is a constant with $\epsilon > 0$, and Δ is the diameter of network G that is given constant.*

Please see the proof in the appendix.

7. An Approximation Algorithm for the Fault Recovery Problem

Once the locations for the active and stand-by service chain instances for each request are determined, one of the stand-by instances will be activated once some errors happen in the active instance. In this section, we study the problem of fault recovery problem by proposing an approximation algorithm with an approximation ratio.

7.1. Algorithm

Recall that once request r_j is admitted by the network G, a data center and a set $\mathcal{DC}_j^s$ of data centers will be selected for the active and stand-by instances of its service chain, respectively. Once faults happen in active instances of such requests, the traffic of the requests need to be routed from the switch that attaches

the faulty data centers to one of the data centers that host their stand-by instances. As the compute and bandwidth resources in data centers and links for handing the fault recovery process when faults happen are reserved in advance, the resource consumed by back-up instances and traffic re-routing should not exceed the reserved amounts. The basic idea of the approximation algorithm is to jointly consider these node and edge capacities in an auxiliary graph $G''' = (V''', E''')$ in a unified way, and then transfer the fault recovery problem into a single-source unsplittable flow problem [21] in the auxiliary graph G'''. A solution to the latter will return a feasible solution to the former.

To jointly consider the edge and node capacities, we start by normalizing not only edge and node capacities but also compute and network bandwidth resource demands into ranges of $(0, 1]$. Specifically, we divide the edge capacity $B_f(e)$ and node capacity $C_f(DC_i)$ by the maximum edge capacity $B_{f,max}$ and node capacity $C_{f,max}$, respectively. Accordingly, we then transfer the packet rate and compute resource demand of each request r_j into the ranges of $(0, 1]$.

We proceed by constructing the auxiliary graph $G'''(V''', E''')$ that enables the algorithm due to [21] for the single-source unsplittable flow problem to jointly consider the edge and node capacities. Specifically, we split each data center node DC_i into two virtual nodes, i.e., DC_i' and DC_i'', are added into V''', and a directed edge $\langle DC_i', DC_i'' \rangle$ is added into E'''. For the switch v_{DC_i} that attaches DC_i, there is an edge from v_{DC_i} to DC_i' and an edge from DC_i' to v_{DC_i}, i.e, $\langle v_{DC_i}, DC_i' \rangle$ and $\langle DC_i', v_{DC_i} \rangle$. For simplicity, we refer to such edges as *data center derived edges*. All the other switch nodes in V are added into V''', and all other edges in E are added into E''' with each edge being represented as an bidirectional edge. We further add a common source node s_0 into V''', and a request node r_j for each request in $\mathcal{R}_f(t)$. For each request r_j, there is an edge from s_0 to a data center with its faulty active instance of service chain, and an edge from the second virtual data center DC_i'' of each $DC_i \in \mathcal{DC}_j^s$. Fig. 4 shows an example of the constructed auxiliary graph.

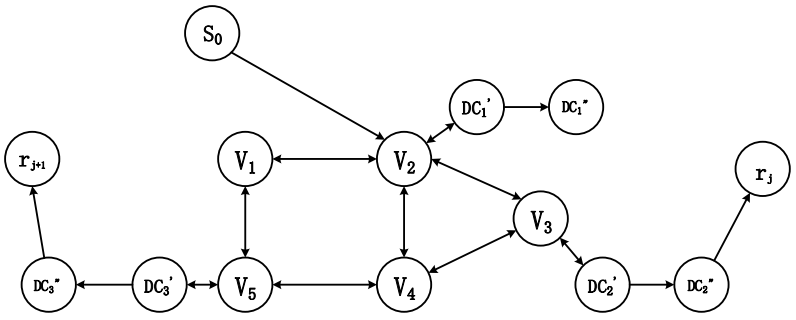


Figure 4: An example of the auxiliary graph $G''' = (V''', E''')$ constructed from network G with a set $\mathcal{DC} = \{DC_1, DC_2, DC_3\}$ of DCs that are connected by a set $V = \{v_2, v_3, v_5\}$ of switches. $\mathcal{R}_f(t) = \{r_j, r_{j+1}\}$, where r_j and r_{j+1} have both of their active instances located in DC_1 and stand by instances in DC_2 and DC_3, i.e., $\mathcal{DC}_j^s = \mathcal{DC}_{j+1}^s = \{DC_2, DC_3\}$.

Algorithm 4 An approximation algorithm for the fault recovery problem

Input: A network $G(V \cup \mathcal{DC}, E)$, a set $\mathcal{R}_f$ of requests with each request r_j having the data center DC_j for its active instance of its service chain at faulty, and the set $\mathcal{DC}_j^s$ of data centers for its stand-by instances.

Output: An assignment of each request in $\mathcal{R}_f$ to one of the stand-by instances in data centers $\mathcal{DC}_j^s$ to be activated to processing its traffic.

1: Construct an auxiliary graph $G''' = (V''', E''')$ from network $G(V \cup \mathcal{DC}, E)$, as illustrated in Fig. 4;
2: Consider the route of each request r_j's traffic from the data center with its active service chain instance to a data center in $\mathcal{DC}_j^s$ with stand-by instances as a commodity with source node s_0 and destination node r_j in auxiliary graph G''';
3: Find a single-source unsplittable flow f''' in the auxiliary graph G''' by applying the algorithm in [21];
4: The requests that are assigned into DC_i in the flow f''' will have their stand-by instances activated in DC_i, and their traffic will be forwarded from the data centers with their active instances to DC_i;
5: **return** The assigned data center to place the service chain of each request for the activation of its stand-by instance of its service chain.

Constructed the auxiliary graph, we consider the route of each request r_j's traffic from the faulty active instance of service chain to a data center in $\mathcal{DC}_j^s$ with stand-by instances as a commodity with source node s_0 and destination node r_j in auxiliary graph G'''. The demand of this commodity is ρ_j. All these commodities will be routed in from their common source to destinations in auxiliary graph G'''. Algorithm 4 lists the detailed steps of the proposed approximation algorithm.

7.2. Algorithm analysis

We now show the correctness and performance of the proposed algorithm in the following lemma and theorem.

Lemma 1. *Let f''' be a unsplittable flow from s_0 to request node r_j in G'''. There is only one data center derived edge in f'''.*

Please see the proof in the appendix.

Theorem 4. *Given a network $G = (V \cup \mathcal{DC}, E)$ with an amount $B_f(e)$ of bandwidth resource and an amount $C_f(DC_i)$ of compute resource to be reserved to enable fault tolerance, let $\mathcal{R}_f(t)$ be the requests that have their active instances being faulty. There is an approximation algorithm for the fault recovery problem that delivers a feasible solution with an approximation ratio of $1+\delta$ in $O(T_2(|\mathcal{R}|+|V|+|\mathcal{DC}|, |E|)+|E| \log((|\mathcal{R}|+|V|+|\mathcal{DC}|)/\epsilon))$ time, where $T_2(m, n)$ is the time to solve a fractional minimum-cost flow problem with m edges and n nodes in the flow graph, δ and ϵ are constants with $\delta > 0$ and ϵ.*

Please see the proof in the appendix.

8. Experiments

In this section, we evaluate the performance of the proposed algorithms through not only simulations in both synthetic networks and real networks but also a real test-bed with both hardware and virtual switches.

8.1. Experiment settings

We consider synthetic networks generated by GT-ITM [23]. The network size ranges from 50 to 250 nodes with a node connectivity of 0.2 (i.e., the probability of having an edge between two nodes is 0.2) [24]. In these networks, the switch to data center ratio is set to 0.1, and each data center has a CPU capacity in the range 4,000 to 8,000 Mhz. The transmission delay of a network link varies between 2 milliseconds (ms) and 5 ms [25]. The costs of transmitting and processing 1 GB (approximately 16,384 packets with each having size of 64 KB) of data are set within [\$0.05, \$0.12] and [\$0.15, \$0.22], respectively, following typical charges in Amazon EC2 with small variations [26]. We consider five categories of network functions: Firewall, Proxy, NAT, DPI, and Load Balancer, their computing demands (e.g., CPU) are adopted from [13]. Further, the consumed compute resources of a service chain is the sum of the computing demands of its contained VNFs (the number of contained VNFs is randomly selected between 1 and 50). The processing delay of a packet for each NF is randomly drawn from 0.045 ms to 0.3 ms [13], and the processing delay of a service chain is the total processing delay of its NFs. Each request r_j is generated by randomly selecting its source s_j and destination t_j from G with packet rate ρ_j randomly selected between 400 and $4,000$ packets/second [27]. Each request has a delay requirement ranging from 10 ms to 100 ms [28, 29]. The running time is obtained based on a machine with a 3.40GHz Intel i7 Quad-core CPU and 16 GB RAM. All results are averaged based on 15 runs in each network topology.

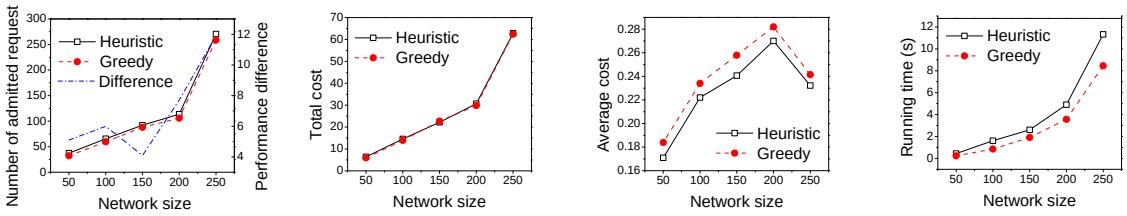
There has not been any existing work considering the fault-tolerant stateful VNF placement. One possible solution is to derive each decision variable in a separate step (similar to an existing approach for stateless VNF placement problem [5]). In this sense, we compare our algorithms against a greedy algorithm (described in 4.1) that separately selects the placement of active/stand-by service chain instance, request routings and state transfer paths. The greedy aims to maximize the throughput by admitting requests with small packet rates first. For simplicity, we refer to this greedy algorithm as algorithm `Greedy`, and the greedy without bandwidth and computing capacity constraints as `Greedy_noBW` and `Greedy_noCP`, respectively. The proposed heuristic and approximation algorithms (Algorithms 1, 2, 3, and 4) are referred to as `Heuristic`, `Appro_noBW`, `Appro_noCP`, and `Appro_Recovery`, respectively.

8.2. Performance evaluation of the heuristic algorithm

We first compare the performance of algorithm `Heuristic` against that of algorithm `Greedy` for networks with various sizes, in terms of the number of admitted requests, the total cost of implementing the admitted requests, the average cost of implementing a request, and the running time.

Fig. 5 shows the result in terms of the number of admitted requests, the average cost of admitting a request, and the running time. We see from Fig. 5 (a) that the proposed algorithm `Heuristic` consistently admits more requests than `Greedy`. This is due to the fact that algorithm `Heuristic` jointly selects the active and stand-by instances. As such, both network resources and the compute resources of data centers are

efficiently utilized, which avoids the request rejections that happened with `Greedy` due to separate selection process. The surge of admitted requests observed with both algorithms for networks with size equal to 250 can be explained by the fact that in this case, the bandwidth resources between any two network nodes are on average increased (i.e., more network links exist between two nodes when network size becomes larger), which results in relaxed constraints in terms of bandwidth. From Fig. 5 (b), we observe that the two algorithms achieve almost the same total cost. However, since the overall admitted request number obtained with `Heuristic` is higher than that of `Greedy`, we see from Fig. 5 (c) that the `Heuristic` achieves a lower per request cost than `Greedy`. Furthermore, `Heuristic` achieves a lower cost in terms of the average cost per admitted request than that of `Greedy`. Meanwhile, we see from Fig. 5 (d) that `Heuristic` slightly results in a longer running time than that of algorithm `Greedy`.



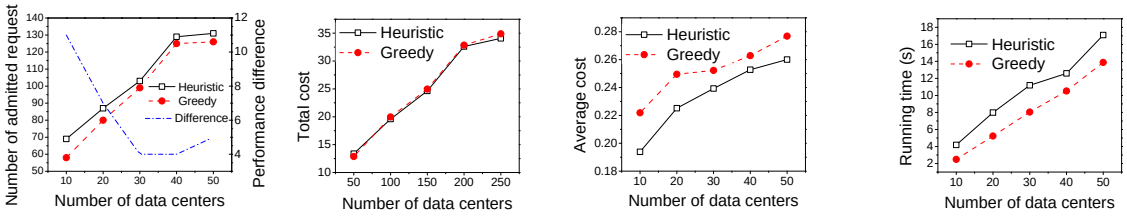
(a) Number of admitted requests (b) Total cost (c) Average cost per request (d) Running time

Figure 5: Performance of algorithms `Heuristic` and `Greedy`.

We then study the performance of algorithm `Heuristic` against that of algorithm `Greedy` in real network AS1755, in terms of the number of admitted requests, the total cost of implementing the admitted requests, the average cost of implementing a request, and the running time, by varying the number of data centers in the network from 10 to 50.

Results are shown in Fig. 6. From Fig. 6 (a), we can see that algorithm `Heuristic` admits many more request than algorithm `Greedy`. For example, when there are 20 data centers in the network, algorithm `Heuristic` has an over 10% higher throughput than algorithm `Greedy`. Also, the number of admitted requests is consistently increasing with the growth of data centers. The rationale behind is that more data centers mean more resources to implement a higher number of requests. In addition, algorithm `Heuristic` implements a request in a much lower cost, as shown in Figures 6 (b), and 6 (c).

It must be mentioned that the results obtained from synthetic and real networks can differ significantly. For example, in Fig 5, the number of admitted requests by algorithm `Heuristic` for network size 250 is slightly over 250. Notice that the ratio of number of switches to number of data centers is 0.1. We thus compare this result with the number of data centers being set to 25 in real network AS1755, as shown in Fig 6 (a). We can see that the number of admitted requests is roughly 95. The reason that the results differ is due to the network topology. Specifically, for the synthetic networks, we connect each pair of nodes



(a) Number of admitted requests (b) Total cost (c) Average cost per request (d) Running time

Figure 6: Performance of algorithms `Heuristic` and `Greedy` in real network AS1755.

with probability 0.2. This means that any pair of nodes has the same probability of being connected. This however is not the case in real network AS1755. Consequently, the real network AS1755 may have more bottleneck links with the synthetic one, leading to more requests being rejected due to the lack of bandwidth resource in the bottleneck links.

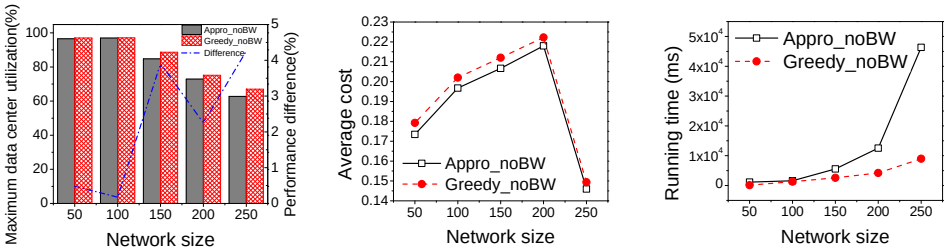
8.3. Performance evaluation of the approximation algorithm without bandwidth capacity constraints

We then compare the performance of algorithm `Appro_noBW` with that of algorithm `Greedy_noBW` in terms of the maximum resource utilization of data center resource, the average cost of implementing a request, and the running time, by varying the network size from 50 to 250.

It can be seen from Fig. 7 (a) that the proposed approximation algorithm `Appro_noBW` delivers solutions with lower maximum data center resource utilization than that by algorithm `Greedy_noBW`. For example, when the network size is 250, the minimum maximum resource utilization of data centers of `Appro` is 5% lower than that of algorithm `Greedy_noBW`. The rationale behind is that algorithm `Appro_noBW` explores a fine-grained trade-off between the resource utilizations and the cost of implementing requests, by moving both node and edge costs in the original network to the edge costs in the constructed auxiliary graph. Fig. 7 (a) also shows that the maximum resource utilization of data centers is decreasing with the network size. This is because larger networks mean on average more compute resources in data centers, which incurs lower resource utilization. In addition, as shown in Fig. 7 (c), algorithm `Appro_noBW` also delivers a lower implementation cost. Regarding the running time, it should be noted that our algorithms are intended to be executed offline and to compute solutions that will be implemented in networks at the network configuration stage. The running time of `Appro_noBW` is therefore considered as tolerable. Finally, we observe that both the maximum data center utilization in Fig. 7 (a) and the total cost in Fig. 7 (b) are not increasing as the network size grows, which further justifies the performance guarantee of the proposed algorithm `Appro_noBW`.

We also investigate the performance of algorithms `Appro_noBW` and `Greedy_noBW` in real network AS1755 in terms of the maximum resource utilization of data center resource, the average cost of implementing a request, and the running time, by varying the number of data centers in the network from 10 to 50.

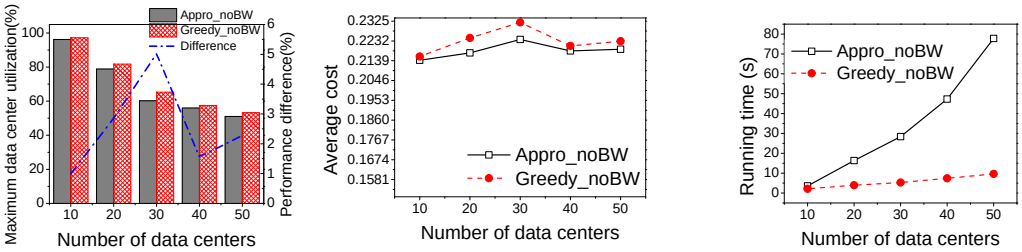
It can be seen from Fig. 8 (a) that algorithm `Appro_noBW` has a lower maximum data center resource utilization. Also, the data center utilization is decreasing with the grow of the number of data centers, because



(a) Maximum data center utilization (b) Average cost per request (c) Running time

Figure 7: Performance of algorithms `Appro_noBW` and `Greedy_noBW`.

there are more compute resource in a higher number of data centers. In terms of the cost of implementing requests, we can see from Figures 8 (c) that `Appro_noBW` has a lower average cost of implementing a request.



(a) Maximum data center utilization (b) Average cost per request (c) Running time

Figure 8: Performance of algorithms `Appro_noBW` and `Greedy_noBW` in real network AS1755.

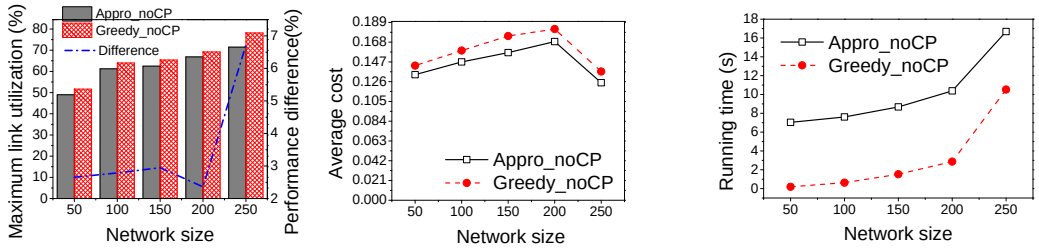
8.4. Performance evaluation of the approximation algorithm without computing capacity constraints

We now show the performance of algorithm `Appro_noCP` with that of algorithm `Greedy_noCP` in terms of the maximum link utilization, the average cost of implementing a request, and the running time, by varying the network size from 50 to 250.

The results are shown in Fig. 9. We can see from Fig. 9 (a) that the link utilization obtained from the solution by algorithm `Appro_noCP` is lower than that of algorithm `Greedy_noCP`. As shown in Fig. 9 (b), the average cost by algorithm `Appro_noCP` is much lower than that of algorithm `Greedy_noCP`. For example, when the network size is 150, algorithm has at least 15% percentage lower average cost of implementing a request, as shown in Fig. 9 (b). The running times are shown in Fig. 9 (c).

We then evaluate the performance of algorithms `Appro_noCP` and `Greedy_noCP` in terms of the maximum link utilization, the average cost of implementing a request, and the running time in real network AS1755, by varying the number of data centers in the network from 10 to 50.

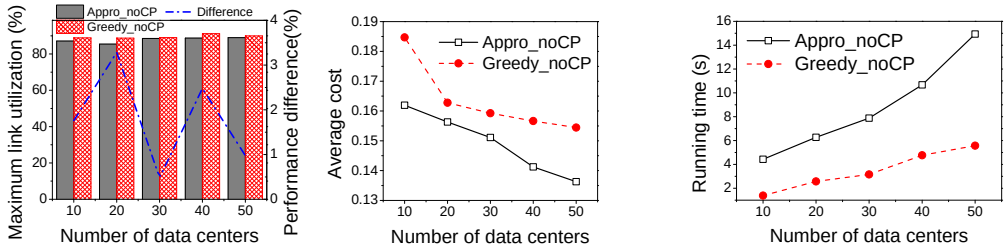
From Fig. 10 (a), we can see that there is no obvious trend of the maximum link utilization with the growth of data center numbers. As shown in Fig. 10 (b), the average cost of implementing a request decreases with the growth of data centers. This is because the compute resource capacity of data centers is not a



(a) Maximum link utilization (b) Average cost (c) Running time

Figure 9: Performance of algorithms `Appro_noCP` and `Greedy_noCP`.

constraint in this problem, and more data centers usually mean more requests will be served in the data centers that are closer to their sources or destinations. This will reduce the transmission cost significantly. The running times of algorithms `Appro_noCP` and `Greedy_noCP` are shown in Fig. 10 (c), from which we can see that algorithm `Appro_noCP` has a tolerable higher running time than algorithm `Greedy_noCP`.



(a) Maximum link utilization (b) Average cost (c) Running time

Figure 10: Performance of algorithms `Appro_noCP` and `Greedy_noCP` in real network AS1755.

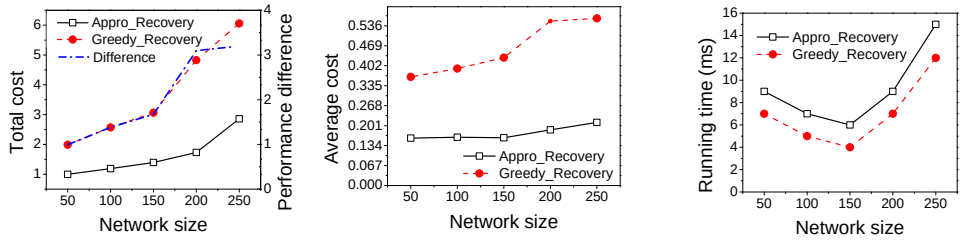
8.5. Performance evaluation of the approximation algorithm for fault recovery

We continue by investigating the performance of algorithm `Appro_Recovery` against algorithm `Greedy_Recovery` in terms of the total cost of implementing admitted requests, the average cost of each admitted request, and the running time, by varying the network size from 50 to 250.

From Fig. 11 (a), it can be seen that the total cost of implementing all requests is increasing with the growth of the network size. The reason is that although the requests thus have more choices of being implemented in data centers with lower costs, when the network size keeps growing, each request has a higher probability of being implemented in a data center that is far from its source and destination, thereby increasing the transmission cost.

We then evaluate the performance of algorithms `Appro_Recovery` and `Greedy_Recovery` in real network AS1755 in terms of the total cost of implementing admitted requests, the average cost of each admitted request, and the running time, by varying the number of data centers in the network from 10 to 50.

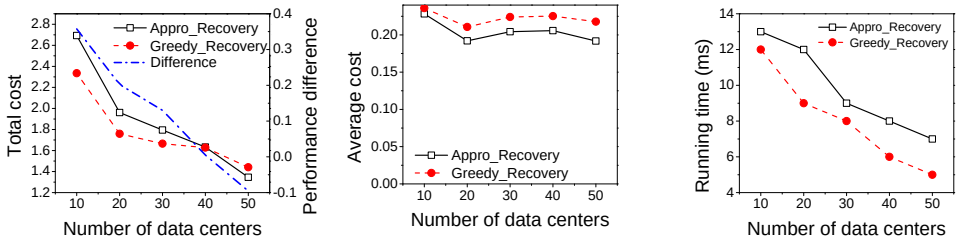
From Fig. 12 (a), we can see that the total cost of implementing all admitted requests is decreasing because less requests are admitted, as shown in Fig. 12 (b). However, the average cost does not experience



(a) Total cost (b) Average cost per request (c) Running time

Figure 11: Performance of algorithms `Appro_Recovery` and `Greedy_Recovery`.

some obvious changing patterns, and algorithm `Appro_Recovery` has a lower average cost than algorithm `Greedy_Recovery`.



(a) Total cost (b) Average cost per request (c) Running time

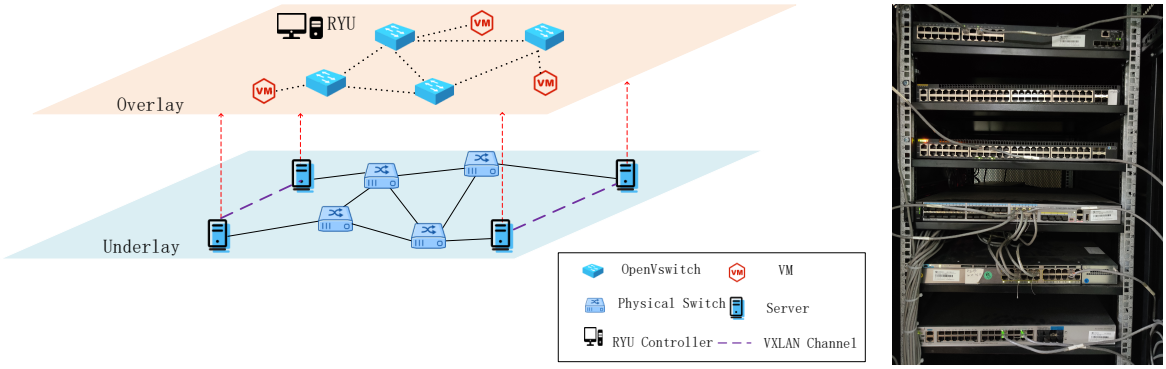
Figure 12: Performance of algorithms `Appro_Recovery` and `Greedy_Recovery` in real network AS1755.

8.6. Performance of algorithms `Appro_Recovery` *and* `Greedy_Recovery` *in a real test-bed*

To testify the applicability of the proposed algorithm `Appro_Recovery` in real settings, we finally evaluate the performance of algorithms `Appro_Recovery` and `Greedy_Recovery` in a test-bed with both hardware-switches and virtual switches. Notice that the reason that why only algorithm `Appro_Recovery` is implemented in the test-bed is that it deals with the online fault recovery problem while the rest algorithms focus on offlline fault-tolerant placement of VNFs. In the following, we first describe the building of the test-bed, settings, and the obtained results.

Test-bed: To evaluate both the applicability in real environments and scalability of algorithm `Appro_Recovery`, the built test-bed consists of both a underlay with hardware appliances and an overlay with virtual resources, as shown in Fig. 13. Specifically, in the underlay, we connect five hardware heterogeneous switches and four servers, and their specifications are listed in Table 2. By using the VXLAN technique, we build a virtual network with a number of Open vSwitch (OVS) [30] nodes and virtual machines. Notice that not all switches in Table 2 support VXLAN. We thus attach a server node in each of the switches that do not support VXLAN, and instantiate a virtual node with built in support for VXLAN, by replacing the network stack with OVS in Linux kernel. Also, we use Netconf and SNMP protocols to manage the switches and the links that interconnect them. The overlay network is built following the real topology AS1755. Its OVS nodes and

virtual machines are controlled by a Ryu [31] controller. Notice that we do not consider the fault recovery within a data center, we consider a simplified scenario with each virtual machine being used to simulate a data center. Algorithms `Appro_Recovery` and `Greedy_Recovery` are implemented as Ryu applications, and control the fault recovery process of the overlay network following OpenFlow protocol [32]. All the rest settings are the same as the aforementioned simulations.



(a) The underlay and overlay of the test-bed

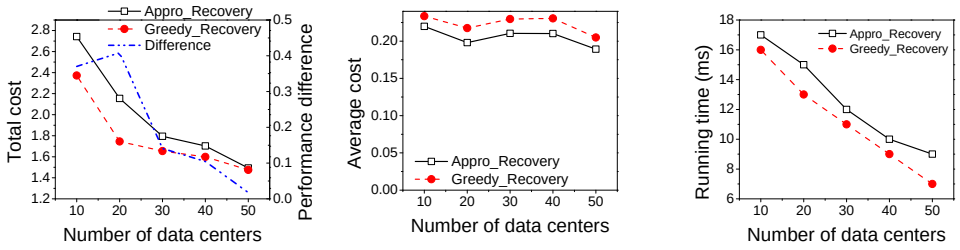
(b) The physical deployment of the hardware switches

Figure 13: A test-bed with both hardware switches and virtual resources.

Table 2: Physical switches in the test-bed

Manufacturer	Model	Support VXLAN?	Network management protocol
Huawei	S5720-32C-HI-24S-AC	YES	Netconf
H3C	S5560-30S-EI	NO	Netconf
Ruijie	RG-5750C-28Gt4XS-H	NO	Netconf
CISCO	CISCO3750X-24T	NO	SNMP
Centec	aSW1100-48T4X	NO	SNMP

Results: We evaluate the performance of algorithms `Appro_Recovery` and `Greedy_Recovery` in terms of the total cost of implementing all fault recovery requests, the average cost of implementing the requests, and the running time. Fig. 14 shows the results, from which we can see that total cost obtained by `Appro_Recovery` is 20% lower than that of algorithm `Greedy_Recovery`, and the average cost is 15% lower than that of algorithm `Greedy_Recovery`, when there are 20 data centers in the network. However, the running times between `Appro_Recovery` and `Greedy_Recovery` are higher than that of simulation results in Fig.12. The reason is that the algorithms have to deal with more types of packets, such as PacketIn, PacketOut, and probe packets.



(a) Total cost (b) Average cost per request (c) Running time

Figure 14: Performance of algorithms `Appro_Recovery` and `Greedy_Recovery` in a real test-bed.

9. Conclusion

In this paper, we proposed a novel efficient heuristic approach for the fault-tolerant VNF placement problem that jointly computes the placement active and stand-by instances of stateful VNFs, the routing paths and update paths of user requests. For the problem without network bandwidth or compute resource constraints, we proposed two bicriteria approximation algorithms with performance guarantees, respectively. Given the placed active and stand-by instances of stateful VNFs, we then investigated the dynamic fault recovery problem, by assuming that the fault recovery requests arrive into the system without the knowledge of their future arrivals. We proposed another approximation algorithm for the problem. We finally evaluated the performance of the proposed algorithms based on simulations using both synthetic and real networks. Our evaluation results show that the performance obtained with each algorithm outperforms existing solutions that separately determine placements, routings and state update paths.

Acknowledgment

The work of Zichuan Xu is supported by the National Natural Science Foundation of China (Grant No. 61802048, 61802047, 61772113, 61872053), the fundamental research funds for the central universities in China (Grant No. DUT17RC(3)061, DUT17RC(3)070), and the Xinghai Scholar Program in Dalian University of Technology, China. The work by Alex Galis is supported by EU NECOS projects (777067).

References

[1] ETSI, *NFV white paper 1*, https://portal.etsi.org/NFV/.

[2] Z. Xu, W. Liang, A. Galis, and Y. Ma, "Throughput maximization and resource optimization in nfv-enabled networks," in *IEEE ICC*, 2017.

[3] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *IEEE NOMS*, 2014.

[4] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: a field study of middlebox failures in datacenters," in *ACM IMC*, 2013.

[5] F. Carpio, S. Dhahri, and A. Jukan, "Vnf placement with replication for loac balancing in nfv networks," in *IEEE ICC*, 2017.

[6] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed vnf state management," in *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[7] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for nfv: Simplifying middlebox modifications using statealyzr." in *UNSENIX Proc. of NSDI*, 2016.

[8] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *ACM Proc. of SoCC*, 2013.

[9] M. Huang, W. Liang, Z. Xu, M. Jia, and S. Guo, "Throughput maximization in software-defined networks with consolidated middleboxes," in *IEEE LCN*, 2016.

[10] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz, "Optimizing virtual backup allocation for middleboxes," *IEEE ICNP*, 2016.

[11] H. D. Chantre and N. L. da Fonseca, "Redundant placement of virtualized network functions for lte evolved multimedia broadcast multicast services," in *IEEE ICC*, 2017.

[12] H. Huang, S. Guo, J. Wu, and J. Li, "Service chaining for hybrid network function," *IEEE Trans. on Cloud Computing, DOI:10.1109/TCC.2017.2721401*, 2017.

[13] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX Proc. of NSDI*, 2014.

[14] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 27–38, 2013.

[15] P. Wang, J. Lan, X. Zhang, Y. Hu, and S. Chen, "Dynamic function composition for network service chain: Model and optimization," *Elsevier Computer Networks*, vol. 92, pp. 408–418, 2015.

[16] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags." in *USENIX Proc. of NSDI*, 2014.

[17] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *IEEE Trans. on Communications*, vol. 64, no. 9, pp. 3746–3758, 2016.

[18] D. Cotroneo, L. De Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network function virtualization: Challenges and directions for reliability assurance," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2014, pp. 37–42.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. of NSDI, USENIX*, 2008.

[20] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "Colo: Coarse-grained lock-stepping virtual machines for non-stop service," in *ACM Proc. of SoCC*, 2013.

[21] S. G. Kolliopoulos and C. Stein, "Approximation algorithms for single-source unsplittable flow," *SIAM J. on Computing*, vol. 31, no. 3, pp. 919–946, 2001.

[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *ACM Proc. of IMC*, 2009.

[23] GT-ITM, http://www.cc.gatech.edu/projects/gtitm/.

[24] B. Yang, W. K. Chai, Z. Xu, K. V. Katsaros, and G. Pavlou, "Cost-efficient nfv-enabled mobile edge-cloud for low latency mobile applications," *IEEE Trans. on Network and Service Management, DOI:10.1109/TNSM.2018.2790081*, 2018.

[25] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[26] I. Amazon Web Services, *Amazon ec2 instance configuration*, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html.

[27] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *IEEE INFOCOM*,

2016.

[28] Microsoft, *Plan network requirements for Skype for business*, https://technet.microsoft.com/en-us/library/gg425841.aspx.

[29] B. Yang, W. K. Chai, G. Pavlou, and K. V. Katsaros, "Seamless support of low latency mobile applications with nfv-enabled mobile edge-cloud," in *IEEE Cloudnet*, 2016.

[30] *Open vSwitch*, https://www.openvswitch.org/.

[31] *Ryu SDN Controller*, https://osrg.github.io/ryu/.

[32] *OpenFlow*, https://www.opennetworking.org/.

Appendix

Proof of Theorem 1

Proof. To show the feasibility of the algorithm, we need to show that the resource demands of each admitted request and its end-to-end delay requirement are met. Clearly, this is true due to steps 8 and 22.

For the running time of the proposed heuristic, we can see that the most time-consuming phases of Algorithm 1 are (1) finding all pair shortest paths in G, (2) ranking all data centers, and (2) iteratively selecting a number of data centers for each request. Clearly, phase (1) takes $O((|V| + |\mathcal{DC}|)^3)$ time, phase (2) takes $O(|\mathcal{DC}| \log |\mathcal{DC}|)$ time, and phase (3) takes $O(|\mathcal{DC}|)$ time. Since the ranking of DCs is performed every time when a request is admitted, the overall running time of algorithm 1 is $O(|\mathcal{R}|(|\mathcal{DC}| \log |\mathcal{DC}|)+(|V|+|\mathcal{DC}|)^3)$. □

Proof of Theorem 2

Proof. We first show the feasibility of the proposed algorithm. Given an unsplittable flow f, it starts at a request node r_j and ends at the common source s_0 in G' according to the construction of auxiliary graph G'. Clearly, a data center node DC_i for active instance and a stand-by set node exists in the route. The traffic of request r_j is processed by the placed active instance in DC_i, and it is routed on the paths from r_j's source s_j to DC_i and from DC_i to destination t_j (e.g., represented by edge $\langle DC'_i, r_j \rangle$ in auxiliary graph). Also, since an auxiliary edge between a stand-by node to DC_i denotes the state update path from DC_i to one of the stand-by set nodes, the processing states are then updated to one of the stand-by set nodes following the traversed edge by f. In addition, the delay requirement of r_j is met, as f only exists when there is an edge between r_j and DC'_i (i.e., delay is met).

We then show the approximation ratio of the devised approximation algorithm. It is clear that the solution to the single-source unsplittable flow problem in auxiliary graph G' corresponds to the solution to the VNF placement problem without bandwidth constraint in network G. The approximation ratio obtained for the former problem thus is the approximation ratio for the latter, i.e., $(2, 4 + \epsilon)$.

We then show the time complexity of the approximation algorithm, which can be divided into two stages: (1) the construction of the auxiliary graph $G'(V', E')$; and (2) finding an unsplittable flow in the constructed auxiliary graph using the algorithm proposed by Kolliopoulos and Stein [21]. Clearly, the construction of G'

takes $(|V'|+|E'|)$ time, where $|V'| = O(|\mathcal{R}|+|V|+|\mathcal{DC}|+\sum_{k=1}^{|\mathcal{DC}|-1}\binom{|\mathcal{DC}|-1}{k}) = O(|\mathcal{R}|+|V|+|\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k})$, and $E' = O(|\mathcal{R}|\cdot|\mathcal{DC}|+|\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k})$, where $\sum_{k=1}^{|\mathcal{DC}|-1}\binom{|\mathcal{DC}|-1}{k}$ is the maximum number of stand-by set nodes for all DCs. According to [21], finding a unsplittable flow in G' takes $O(T_2(|V'|,|E'|)+|E'|\log(|V'|/\epsilon)) = O(T_2(|\mathcal{R}|+|V|+|\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k}, |\mathcal{R}|\cdot|\mathcal{DC}|+|\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k})) = O(T_2(|\mathcal{R}|+|V|+|\mathcal{DC}|2^{|\mathcal{DC}|-1}, |\mathcal{R}|\cdot|\mathcal{DC}|+|\mathcal{DC}|2^{|\mathcal{DC}|-1}))$ time. □

Proof of Theorem 3

Proof. We first show the feasibility of the solution. Recall that each request is split into two virtual requests. It is clear that each virtual request will have its traffic going through a data center node and a stand-by set node. This is because any path that starts with the common source s_0 in G'' and ends with a request node r_j has a data center node and a stand-by set node in the path, according to the construction of G''. However, different virtual requests may be assigned to different data centers. In the step (5) of algorithm 3, the data traffic of r_j is routed from its source s_j to the data center assigned to one of its virtual request, then from the virtual request to another data center that is assigned to its the other virtual request, and finally its forwarded to its destination t_j. Also, the service chain of r_j is assigned to the DC and stand-by set with smaller cost. This clearly makes the solution feasible.

We then show the approximation ratio of the devised approximation algorithm. It is clear that the solution to the single-source unsplittable flow problem with a set of virtual requests without service chain requirements in auxiliary graph G'' has an approximation ratio of $(2, 4+\epsilon)$ 1 for cost and congestion. Since virtual requests of each request may be assigned to different data centers, the approximation algorithm performs further adjustments to make it feasible. To show the approximation ratio, we only need to show how much of the approximation ratio is deviated from the one for the solution to the single-source unsplittable flow problem with a set of virtual requests. To this end, we first show the derivation of the approximation ratio on cost. The additional cost of such adjustment is mainly due to the path cost of the shortest path between the two data centers that are assigned to the virtual requests of each request. Let Δ be the diameter of the network G. Clearly, the cost will be no more than Δ times the cost obtained for the single-source unsplittable flow problem with a set of virtual requests without service chain requirements. This is because any shortest path in G will be no greater than Δ, considering the cost of each edge is in $(0, 1]$. Thus, the approximation ratio for cost is 2Δ. We then show the derivation of the approximation ratio on congestion. In the worst case, each request may has its two virtual request distributed into two different data centers. This is to consider an addition set of requests with the same resource demands as the ones in $\mathcal{R}$ but with different sources and destinations. Recall that we assume that the total network bandwidth demand of all mice flows is far less than the total available network resources of G. It thus is clear the total demand of the additional set of requests is still smaller than the total residual capacity of G after step 4. Clearly, by reapplying the single-source unsplittable flow algorithm with the additional set of requests with equal total

amount of packet rates as the ones in $\mathcal{R}$, the congestion will be enlarged by at most twice. We thus have an approximation ratio of $8 + 2\epsilon$.

We then show the time complexity of the approximation algorithm, which can be divided two steps: (1) construction of the auxiliary graph $G''(V'', E'')$; and (2) finding a unsplittable flow in the constructed auxiliary graph by the algorithm due to Kolliopoulos and Stein [21]. Clearly, the construction of G'' takes $|V''| + |E''|$ time. First, for the node set, we know that there are three types of nodes in G'': (1) the nodes in the original network G, (2) the virtual request and source nodes, and (3) the stand-by set nodes. For types (1) and (2), it takes $O(|\mathcal{R}| + |V| + |\mathcal{DC}|)$ time. For stand-by nodes, since we do not fix the number of data centers in each stand-by set, we thus have $O(\sum_{k=1}^{|\mathcal{DC}|-1} \binom{|\mathcal{DC}|-1}{k})$ choices, which is also the time spent in adding the nodes into V''. Therefore, we have $|V''| = O(|\mathcal{R}| + |V| + |\mathcal{DC}| + \sum_{k=1}^{|\mathcal{DC}|-1} \binom{|\mathcal{DC}|-1}{k}) = O(|\mathcal{R}| + |V| + |\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k})$, and $E'' = O(|E| + |\mathcal{R}| \cdot |\mathcal{DC}| + |\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k})$, where $\sum_{k=1}^{|\mathcal{DC}|-1} \binom{|\mathcal{DC}|-1}{k}$ is the maximum number of stand-by set nodes for all data centers. For step (2), according to Theorem 2, finding an unsplittable flow in G' takes $O(T_2(|V''|, |E''|) + |E''| \log(|V''|/\epsilon)) = O(T_2(|\mathcal{R}| + |V| + |\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k}, |E| + |\mathcal{R}| \cdot |\mathcal{DC}| + |\mathcal{DC}|\binom{|\mathcal{DC}|-1}{k}))) = O(T_2(|\mathcal{R}| + |V| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}, |E| + |\mathcal{R}| \cdot |\mathcal{DC}| + |\mathcal{DC}|2^{|\mathcal{DC}|-1}))$ time. □

Proof of Lemma 1

Proof. To show the feasibility of the proposed algorithm, we only need to show that any path from s_0 to a request node r_j in auxiliary graph G''' has only one data center node. We prove by contradiction. Let DC_i be the data center to active the stand-by instance of request r_j, clearly, edge $\langle DC'_i, DC''_i \rangle$ is the edge that flow f''' traverses to reach its destination node r_j. Assume there is another edge $\langle DC'_{i+1}, DC''_{i+1} \rangle$ that is traversed by the flow before edge $\langle DC'_i, DC''_i \rangle$. According to the construction of the auxiliary graph G''', there is no edge from DC''_{i+1} to DC'_{i+1}. If DC''_{i+1} is connected to r_j (i.e., $DC_{i+1} \in \mathcal{DC}^s_j$), the flow will reach r_j via edge $\langle DC''_{i+1}, r_j \rangle$, meaning that DC_{i+1} is the DC to active the stand-by instance of request r_j rather than DC_i, which contradicts the assumption. Otherwise, edge $\langle DC'_i, DC''_i \rangle$ will not be traversed either. □

Proof of Theorem 4

Proof. According to Lemma 1, the solution obtained by algorithm 4 is feasible. We here analyze the approximation ratio and running time of the algorithm.

We first show that a unsplittable flow f''' in auxiliary graph G''' from source node s_0 to destination node r_j corresponds to the routing of the traffic of request r_j from its faulty data center to a data center with an available stand-by service chain instance. Recall that in algorithm 4, the construction of auxiliary graph G''' divides each data center to two virtual data centers and add a directed edge from the first virtual data center to the second one. Also, there is an edge from the second virtual data center to the request node. This guarantees that the compute resource reserved in each data center is considered as edge capacities, and thereby makes the capacity of reserved compute resource is met. That is to say, any feasible flow f''' starting

from s_0 will have a data center derived edge in it, and the corresponding data center will active a stand-by service chain instance to further process the traffic. We thus say a unsplittable flow corresponds the routing of user request r_j from its faulty service chain instance to a stand-by instance.

In the construction of the auxiliary graph, the cost of using reserved compute resource is transferred to edge costs in the auxiliary graph. The cost due to fault recovery thus is the cost of a unsplittable flow f''' in G'''. Also, considering the solution obtained from finding a single-source unsplittable flow in G''' can be directly transferred to a solution to the fault recovery problem, the approximation ratio obtained of the algorithm for the single-source unsplittable flow problem is the approximation ratio for the fault recovery problem. That is, $1+\delta$ due to [21], where δ is a given positive constant.

For the running time, algorithm 4 has two parts: (1) the construction of auxiliary graph G''', and (2) the invoking of algorithm due to [21]. For part (1), it can be seen the construction of auxiliary graph G''' takes $|V'''|+|E'''| = O(|\mathcal{R}|+|V|+|\mathcal{DC}|+|E|)$ time, where $|V'''| = |\mathcal{R}|+|V|+|\mathcal{DC}|$ and $|E'''| = O(|E|)$. For part (2), according to Theorem 1, finding a unsplittable flow in G''' takes $O(T_2(|V'''|,|E'''|)+|E'''|\log(|V'''|/\epsilon)) = O(T_2(|\mathcal{R}|+|V|+|\mathcal{DC}|,|E|)+|E|\log((|\mathcal{R}|+|V|+|\mathcal{DC}|)/\epsilon))$. Overall, algorithm 4 takes $O(T_2(|\mathcal{R}|+|V|+|\mathcal{DC}|,|E|)+|E|\log((|\mathcal{R}|+|V|+|\mathcal{DC}|)/\epsilon))$ time. □