计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210300304

故障场景下的边缘计算 DAG 任务重调度方法

蔡凌峰[1]　魏祥麟[2]　邢长友[1]　邹　霞[1]　张国敏[1]
1 陆军工程大学指挥控制工程学院　南京 210007
2 国防科技大学第六十三研究所　南京 210007
(sawakita1122@foxmail.com)

摘　要　边缘计算将计算和存储资源部署在靠近数据源的网络边缘,并高效调度用户卸载的任务,从而极大地提升了用户的服务体验(Quality of Experience,QoE)。但是,边缘计算缺乏可靠的基础设施保护,服务器节点或通信链路的突发故障可能会导致服务失败。为此,建立了边缘计算中的计算节点和通信链路故障模型,并针对依赖型用户任务的调度,提出了资源故障场景下的任务重调度算法 DaGTR(Dependency-aware Greedy Task Rescheduling)。DaGTR 包括两种子算法,即 DaGTR-N 和 DaGTR-L,分别用于处理节点和链路故障事件。DaGTR 能够感知任务的数据依赖关系,并基于贪心方法对所有受故障影响的用户任务进行重调度,以保证每个任务的成功执行。仿真结果显示,所提算法能够有效避免节点或链路故障导致的任务失败,提高了资源故障情况下任务的成功率。

关键词:边缘计算;有向无环图;任务调度;资源故障
中图法分类号　TP398.08

Failure-resilient DAG Task Rescheduling in Edge Computing

CAI Ling-feng[1],WEI Xiang-lin[2],XING Chang-you[1],ZOU Xia[1] and ZHANG Guo-min[1]
1 The Command & Control Engineering College,Army Engineering University,Nanjing 210007,China
2 The 63rd Research Institute,National University of Defense Technology,Nanjing 210007,China

Abstract　By deploying computation and storage resources at the network edge that is close to the data source,and scheduling tasks offloaded by users efficiently,edge computing can greatly improve the quality of experience (QoE) of users. However,due to the lack of the reliable infrastructure support,the failure of edge servers or communication links could easily fail the edge computing service. To handle this problem,we establish the failure models of the computing nodes and communication links in edge computing,and then propose the rescheduling algorithm DaGTR (Dependency-aware Greedy Task Rescheduling) for the scheduling of dependent user tasks in resource failure scenarios. DaGTR includes two sub-algorithms,DaGTR-N and DaGTR-L,which are responsible for handling the node and link failure events respectively. DaGTR can sense the data dependency of tasks,and reschedule the tasks affected by failure events based on greedy method to ensure the successful execution of each task. Simulation results show that the algorithm can effectively avoid the task failure caused by failure events and improve the success rate of tasks in the case of resource failure.

Keywords　Edge computing,Directed acyclic graph,Task scheduling,Resource failure

1　引言

随着 5G 和物联网技术的不断发展,越来越多的数据开始在网络边缘产生和被处理。中国互联网络信息中心(CNNIC)发布的《第 45 次中国互联网络发展状况统计报告》[1] 显示,2019 年全年移动互联网接入流量达 1 220.0×10[8] GB,是 2016 年全年的 13 倍。然而,受到成本和当前集成电路工艺的限制,移动终端设备的处理能力难以满足大数据时代的需求,在运行计算密集型任务时,智能手机或物联网设备往往需要经历较长的响应时延,影响用户体验,且耗能较高从而减短寿命。移动设备可以通过计算卸载技术[2-5],利用算力强大的服务器来执行计算密集的任务,但如果将任务卸载到云端执行,不仅要经历较长的通信延迟,而且存在数据隐私泄露的风险,大量上传到云端的数据也会大大加重骨干网的负担。由此,边缘计算(Edge Computing,EC)应运而生[6-7]。部署在靠近数据源位置的边缘服务器能够及时处理用户卸载的任务,避免将数据上传到远端的云中心,大大提高了数据处理的实时性和私密性[8]。但不像云计算中心有稳定的基础设施保护,许多边缘计算节点(如路旁单元(Roadside Unit,RSU)、基站)暴露于自然环境下,其硬件容易受到外部物理损伤[9];并且由于节点的地

到稿日期:2021-03-31　返修日期:2021-04-28
通信作者:邢长友(changyouxing@126.com)

理位置比较分散，边缘计算集群难以提供与云计算中心相当的维护和管理水平，节点可能因配置错误或遭受攻击而发生软件层面的崩溃。当发生硬件损伤或是软件崩溃时，边缘系统提供的服务可能会中断，影响用户的服务体验。

卸载到边缘的用户任务通常分为独立任务和依赖型任务，独立任务只对用户提供的原始数据进行处理并返回结果；而依赖型任务则包括若干子任务，每个子任务的执行依赖于前驱子任务的执行结果，通常将这种具有依赖关系的任务表示为一个有向无环图(Directed Acyclic Graph，DAG)，如图 1 所示。大数据时代，这种依赖型的应用任务十分常见，如一个增强车辆现实系统就包括了目标追踪、模型映射、目标识别、透视变化以及合并处理等多种类型的计算任务。

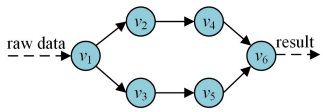


图 1 DAG 任务示例

Fig. 1 Example of DAG task

为了最大化边缘系统的效能，部署在一定范围内的多个服务器节点可以通过组网的方式来协作提供服务。任务可以根据用户的服务质量(Quality of Service，QoS)需求以及边缘系统中的资源状况选择合适的计算节点来执行。对于一个 DAG 任务，其每个子任务也可以进行调度，分别卸载到不同节点执行，以实现更高的并发性和资源利用效率，DAG 任务的调度是一个 NP-hard 问题[10]。在该领域，已经有许多研究者[11-15]提出了不同的方法，能够实现 DAG 任务的高效调度。但是，这些调度方法均没有考虑节点或链路发生故障时，如何调整任务的调度以避免任务失败的发生。本文将研究资源发生故障的场景下，DAG 任务的重调度问题。

2 相关工作

针对边缘计算中的 DAG 任务调度问题，学术界已经提出了一些解决方案。Liu 等提出了 GenDoc 算法来处理边缘计算中的 DAG 调度问题[11]。GenDoc 通过灵活配置网络中服务的部署来减少配置和通信开销，然后基于贪心思想，将 DAG 应用的每个子任务调度到边缘服务器上执行，调度的目标是最小化 DAG 应用的完成时间。He 等提出了一种基于任务复制的 DAG 调度算法[12]。其中，一个 DAG 任务的调度需要经历 4 个阶段，包括任务聚类、任务复制、处理器合并以及任务插入，经过这些阶段后，DAG 任务可以以较低的延迟完成调度。Qi 等将深度强化学习应用到车联网 DAG 任务调度问题中，将任务的响应延迟作为收益函数，提出了一种知识驱动的调度方法[13]，该方法利用计算节点(如 RSU 或基站)训练神经网络模型以进行调度决策，而后将训练好的神经网络分发给行驶中的车辆。车辆在运行服务时执行异步在线学习，并将模型更新到边缘节点，这样模型便能够有效地适应环境的变化，并在线做出最优调度决策。

上述研究实现了边缘系统中 DAG 的高效调度，但均未考虑系统中的节点或链路故障问题，难以从调度策略层面来保证服务的可靠性和持续性，也就是服务的韧性[16]。实际上，当前还没有针对资源故障场景中 DAG 任务重调度的相关研究。

本文针对此问题进行了研究，并提出了解决方法，主要贡献如下：

(1)建立了边缘系统中计算节点及通信链路的故障模型，并分析了故障事件对 DAG 任务执行的影响。

(2)提出了面向故障事件的 DAG 任务调度方法 DaGTR (Dependency-aware Greedy Task Rescheduling)，该算法包括 DaGTR-N 和 DaGTR-L 两种子算法，分别对节点和链路故障事件中受影响的 DAG 任务进行重调度，基于贪心思想为这些受影响的子任务或数据边选择其他可行的执行节点或传输路径，以有效避免故障事件造成的任务失败。

(3)使用 Python 语言开发了一个边缘计算仿真器 EdgeSimDAG，并进行了大量仿真实验。仿真结果证明，DaGTR 算法能够有效避免 DAG 任务因受到故障事件影响而导致的执行失败。

3 系统模型及问题表述

3.1 边缘-云系统模型

边缘网络由边缘服务器、交换设备和通信链路组成，表示为无向图$G_{topo}=(U_F \cup U_S, E)$。其中，$U_F$是边缘服务器的集合，假设系统中包括了 M 台服务器，那么$U_F=[u_1, u_2, \cdots, u_M]$，其中第 i 台表示为u_i；U_S是交换节点的集合，用于边缘服务器组网；E 是通信链路的集合，链路连接服务器或交换节点，节点 u_i到节点 u_j的直连通信链路表示为$e_{i,j}$。

系统中每台边缘服务器与接入点(Access Point，AP)共同部署，用户任务从 AP 接入边缘系统。边缘服务器通过虚拟化技术[17]实现的服务功能(Service Function，SF)实例来提供计算服务，将系统中所有 SF 的集合表示为 F，假设系统中一共包括 N 种不同的 SF，表示为 $F=[sf_1, sf_2, \cdots, sf_N]$。每台边缘服务器部署 F 的一个子集，其中第 k 台服务器部署的 SF 集合为F_k，并且所有边缘服务器部署的 SF 的全集等于 F，因此有：

$$F_1 \cup F_2 \cup \cdots \cup F_M = F \tag{1}$$

考虑到边缘系统的异构性，每台服务器部署的 SF 集合、CPU 资源、处理能力都不同，用 CPU 内核数量c_k来表示服务器u_k的可用资源，用f_k表示u_k每个内核的处理能力。假设 DAG 的每个子任务都是单线程的，即在同一时刻只占用一个内核来运行。

边缘系统可以通过光纤来连接设备，因此在一个较小的服务范围内，数据在通信链路中的传播时延可忽略不计。对于每条通信链路$e_{i,j}$，其最大可用带宽表示为$b_{i,j}$。

云计算中心部署在较远的位置，每个 AP 都可通过互联网连接到远程云。假设远程云可以处理所有类型的任务，并且可用的 CPU 资源是无限的，即有$F_{cloud}=F$，$c_{cloud}=\infty$，其处理能力表示为f_{cloud}。但是将任务数据发送到云端需要经历一个较长的传播时延，表示为d_{cloud}，传输带宽则为定值b_{cloud}。

图 2 给出了一个边缘-云协同的网络架构。

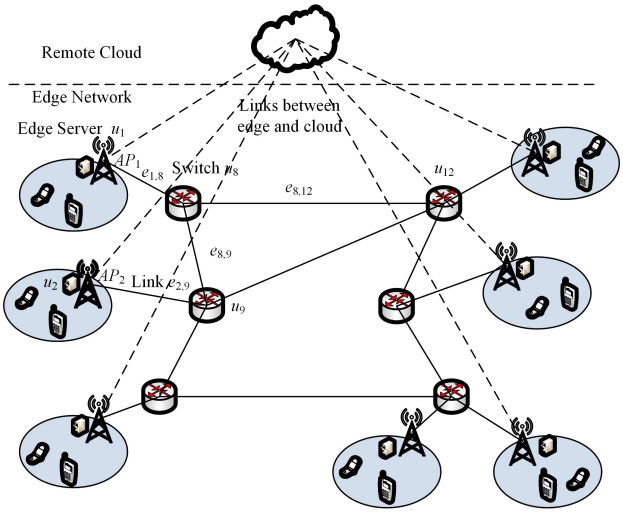


图 2 典型边缘-云网络

Fig. 2 Typical edge-cloud network

3.2 DAG 任务模型

用户卸载的 DAG 任务可能从任意一个 AP 接入边缘网络，DAG 任务包含若干子任务和数据依赖边，表示为有向无环图 $G=(V,\varepsilon)$。其中，V 是子任务的集合，用 n_f 表示 V 中的子任务个数，则 $V=[v_1,v_2,\cdots,v_{n_f}]$；$\varepsilon$ 则是数据边的集合，存在一条数据边 $\varepsilon_{i,j}\in\varepsilon$ 表示子任务 v_j 的执行依赖于 v_i 执行的结果。将第一个子任务 v_1 作为 G 的入口子任务处理原始数据，子任务 v_{n_f} 为出口子任务，用于进行最后的处理并返回结果。如图 1 所示，v_1 和 v_6 分别为入口和出口子任务，v_4 和 v_5 是 v_6 的直接前驱子任务，因此存在数据边 $\varepsilon_{4,6},\varepsilon_{5,6}\in\varepsilon$，表示 v_6 的执行依赖于 v_4 和 v_5 的执行结果。子任务 v 的执行节点为 $A(v)$，每个子任务属于某种特定的 SF 类型 $SF(v)$，有 $SF(v)\in F$，子任务的卸载调度需满足约束：

$$SF(v)\in F_{A(v)} \tag{2}$$

即只有部署了子任务对应服务功能的边缘服务器可以作为该子任务的卸载目的地。

根据 DAG 中子任务间的数据依赖关系，可以得出每个子任务开始和完成时间的计算公式，某个子任务 v_i 的开始时间表示为：

$$SF(v_i)=\max_{v_j\in pre(v_i)}(FT(v_j)+t_{j,i}) \tag{3}$$

其中，$pre(v_i)$ 表示 v_i 所有直接前驱子任务的集合，v_j 则表示 v_i 某一个直接前驱子任务；$FT(v_j)$ 为 v_j 的完成时间，$t_{j,i}$ 表示 v_j 到 v_i 的数据边 $\varepsilon_{j,i}$ 的传输耗时，计算式为：

$$t_{j,i}=w_{j,i}/B_{j,i} \tag{4}$$

其中，$w_{j,i}$ 是 $\varepsilon_{j,i}$ 的数据负载量，$B_{j,i}$ 是分配给 $\varepsilon_{j,i}$ 的传输带宽，$B_{j,i}$ 的大小不能超过 $\varepsilon_{j,i}$ 经历的任一条链路的最大可用带宽。用 $Path_{A(v_j),A(v_i)}$ 表示数据边 $\varepsilon_{j,i}$ 传输的路径，e 表示路径所经过的通信链路，b_e 表示该通信链路的最大可用带宽，那么分配给 $\varepsilon_{j,i}$ 的传输带宽应当满足约束：

$$\forall e\in Path_{A(v_j),A(v_i)},B_{i,j}\leqslant b_e \tag{5}$$

由此，式(3)表示子任务 v_i 必须等到其所有直接前驱子任务执行完毕，并将执行结果传输到 v_i 所在服务器后才能开始执行。

v_i 的最早完成时间为：

$$FT(v_i)=ST(v_i)+t_i^{A(v_i)}+WT_i \tag{6}$$

其中，$t_i^{A(v_i)}$ 表示 v_i 在其卸载位置的执行耗时，计算式为：

$$t_i^{A(v_i)}=\frac{w_i}{f_{A(v_i)}} \tag{7}$$

其中，w_i 表示 v_i 的计算负载，即所需的 CPU 周期数。式(6)中的 WT_i 则表示系统忙碌时，v_i 执行前可能需要经历的等待时延。DAG 的出口子任务执行完毕后，其执行结果需要返回给用户，任务的执行结果往往只有几个比特，即其传输时延可以忽略不计[18]，因此一个 DAG 任务 G 的响应延迟 $Latency_G$ 可以表示为：

$$Latency_G=FT(v_{exit}^G)-T_{arrival}^G \tag{8}$$

其中，$FT(v_{exit}^G)$ 表示 G 出口任务的完成时间，$T_{arrival}^G$ 表示 G 到达边缘网络的时刻。

3.3 资源故障模型

为了能够细粒度地分配边缘系统中的计算和带宽资源，并且能够精确地对节点和链路的故障事件进行建模，本文利用软件定义网络(Software Defined Network，SDN)[19]提出了资源的集中式矩阵表示形式。将时间线划分为等长的若干时隙，作为矩阵的横轴，将系统中的计算和带宽资源作为矩阵的纵轴，以时隙为资源的最小分配单位，一个典型的时隙-资源矩阵如图 3 所示。图 3 中，横轴为等长的若干时隙，纵轴为网络中的服务器和通信链路可用资源，图中圆圈表示来自某个 DAG 的子任务，矩形表示子任务间的数据依赖关系。一个圆圈或矩形占据了矩阵中某个位置，表示其在某一时隙占用了某段资源的一部分。

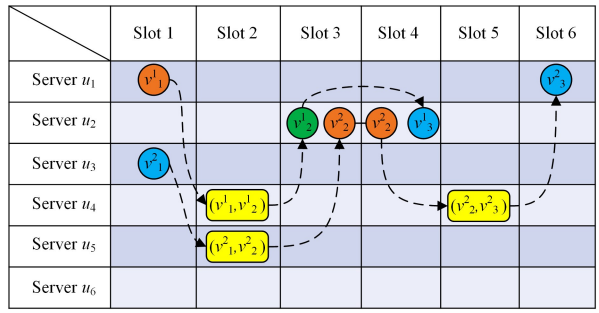


图 3 时隙-资源矩阵示意图

Fig. 3 Schematic diagram of time slot and resource matrix

对边缘网络中的服务器节点来说，承载 SF 示例的虚拟机(Virtual Machine，VM)可能由于配置错误或遭受黑客攻击而故障宕机[20]，但在一个高效的虚拟化架构(如通过 OpenStack 管理的虚拟化平台)中，VM 和服务实例的重启恢复时间可以缩短到秒级[21]，在服务重启的时间内，服务器的可用资源为 0，可以通过时隙-资源矩阵体现，假设服务器节点 u_k 在时刻 T_k^f 发生故障，服务重启需要 n 个时隙，每个时隙大小为 τ，那么有：

$$c_k^t=0,t\in[T_k^f,T_k^f+n\tau] \tag{9}$$

其中，c_k^t 表示 u_k 在时隙 t 的可用内核数量。除了服务器 VM 故障，服务器硬件和网络中的通信链路可能因为遭受外部破坏而发生故障，需要数小时乃至数日才能恢复，在相当长的一段时间内故障部位处于不可用状态，服务恢复时长可认为是

无穷大，式(10)、式(11)分别表示节点u_k和通信链路$e_{k,l}$发生长时故障时的可用资源状态。

$$c_k^t = 0, t \in [T_k^f, \infty] \tag{10}$$

$$b_{k,l}^t = 0, t \in [T_{k,l}^f, \infty] \tag{11}$$

其中，$b_{k,l}^t$表示链路$e_{k,l}$在时隙t的可用带宽。

故障事件的发生会影响部分正在执行或将要执行的子任务以及相关的依赖数据边，因此必须对受故障影响的任务进行重新调度以避免任务失败。

3.4 问题表述

假设在时刻T_f，边缘系统中某个节点或某条链路发生故障，受故障事件影响的 DAG 任务集合表示为D_f，调度算法的目标是避免D_f的任务因为故障事件而失败。对于D_f中的每个 DAG 任务G，用指示变量x_G^f表示G是否成功执行，0 表示失败，1 表示成功，那么优化目标可以表示为：

$$\max \sum_{G \in D_f} x_G^f \tag{12}$$

表 1 列出了本文中的变量符号及其含义。

表 1 符号表示

Table 1 Symbolic description

Symbol	Meaning
G_{topo}	Topology of the edge network
U_F	Set of edge servers
u_k	The kth edge server
AP_k	Access point deploying with u_k
f_k	Processing capacity of CPU core in u_k
U_s	Set of switching nodes
E	Set of communication links
$e_{i,j}$	Communication link connecting u_i and u_j
$b_{i,j}$	Available bandwidth of $e_{i,j}$
F	Universal set of SFs in the system
sf_i	The ith SF in F
F_k	A subset of F deployed on u_k
$Path_{m,n}$	Shortest path between nodes u_m and u_n
f_{cloud}	Processing capacity of the remote cloud
b_{cloud}	Bandwidth between edge and cloud
d_{cloud}	Propagation delay between edge and cloud
G	Arriving DAG Task
$T_{arrival}^G$	Arrival time of G
$Latency_G$	Response latency of G
v_i	The ith sub-task of G
$A(v_i)$	Execution server of v_i
$SF(v_i)$	SF to which v_i corresponding
w_i	Workload of v_i
$pre(v_i)$	Predecessor sub-tasks of v_i
$succ(v_i)$	Successor sub-tasks of v_i
$\varepsilon_{i,j}$	Data edge from v_i to v_j
$w_{i,j}$	Workload of $\varepsilon_{i,j}$
$B_{i,j}$	Bandwidth allocated to $\varepsilon_{i,j}$
t_i^k	Execution latency of v_i on u_k
$D(v_i)$	The DAG task to which v_i belongs
$ST(v_i)$	Start time of v_i
$FT(v_i)$	Finish time of v_i
WT_i	Waiting latency of v_i execution
$t_{i,j}$	Transmission latency of $\varepsilon_{i,j}$
τ	Size of time slot
c_t^k	Available CPU cores of u_k at time slot t
D_f	Set of tasks affected by the failure event
S_f	Set of sub-tasks affected by the failure event
x_G^f	Indicator of whether task G in D_f is successful

4 依赖关系感知的任务重调度方法

为了避免故障事件造成的任务失败，接下来针对 DAG 中子任务间的依赖关系提出了故障场景下的任务重调度方法。

为了仿真故障场景的影响，首先采用一种随机调度算法来为每个到达的 DAG 任务进行调度并分配资源，算法的伪代码如算法 1 所示。值得注意的是，在算法中将远程云中心视作一个特殊的边缘服务器，其可以处理所有类型的子任务，并且为每个子任务处理分配的计算能力f_{cloud}以及为数据边分配的传输带宽b_{cloud}都是固定值，但是数据在边缘和云之间进行上传下载都需要经历额外的传播时延d_{cloud}。

算法 1 DAG 随机调度算法

输入：任务 G 的拓扑 G=(V,ε)

每台边缘服务器部署的 SF 集合

输出：任务 G 的调度结果

更新后的时隙资源矩阵

1. 调用算法 2 对 G 中的所有子任务进行逻辑排序，得到子任务调度序列 V′
2. foreach sub-task v_i in V′ do
3. 找出能够执行v_i的服务器集合 S
4. 从 S 中随机选择一个节点 u
5. 将v_i分配到 u 执行，即 A(v_i)=u
6. foreach precursor sub-task of v_i, v_j do
7. 使用 OSPF 算法得到 A(v_j)到 A(v_i)的最短路径$Path_{A(v_j),A(v_i)}$，将其作为数据边$\varepsilon_{j,i}$的传输路径
8. 查看资源矩阵中$Path_{A(v_j),A(v_i)}$所经过链路的资源状况，根据式(5)为数据边$\varepsilon_{j,i}$分配带宽及传输时段
9. end foreach
10. 根据式(3)得到v_i的开始时间，查看资源矩阵中 u 的内核资源状况，据此来为v_i分配计算资源及执行时段
11. end foreach
12. 根据资源分配结果更新时隙资源矩阵
13. 返回 G 的调度结果及更新后的资源矩阵

算法 1 以一种“预约”的形式来调度每个任务。在任务G到达某个 AP 后，算法根据时隙资源矩阵显示的未来一段时间内系统的可用资源，为G所有的子任务随机选择可行的执行位置，并为数据边分配传输路径和带宽。决策完毕后，G将严格按照算法 1 的调度结果在未来的时间线上执行，分配给G的资源将从资源矩阵中扣除，从而进行矩阵更新。

算法 1 的 Step 1 根据子任务间的数据依赖关系来决定所有子任务的调度执行顺序，一条数据边的起点任务必须在终点任务之前被调度执行，利用算法 2 对任务图中的所有子任务进行拓扑排序，得到的排序结果能够满足数据依赖关系。在边缘系统正常运行过程中，每个到达的 DAG 任务将通过算法 1 进行调度并执行，当节点或链路发生故障事件时，需要对受故障影响的任务进行重调度，下文分别针对节点故障事件和链路故障事件发生时的任务重调度进行分析，并提出了相应的任务重调度子算法 DaGTR-N 和 DaGTR-L。

算法 2 任务图拓扑排序算法

输入：DAG 任务 G 的拓扑 G=(V,ε)

输出:G 中子任务的拓扑排序

1. 定义已排序的子任务集合为 O=[]
2. while V≠∅ do
3. 从 V 中找到一个入度为 0 的子任务 v_0
4. 将 v_0 从 V 中移除
5. foreach v_0 的出边,i.e. $\varepsilon_{0,i}$ do
6. 将 $\varepsilon_{0,i}$ 从边的集合 ε 中移除
7. end foreach
8. 将 v_0 添加到 O 中
9. end while
10. return O

4.1 服务器节点故障场景下的任务重调度

服务器故障事件分为硬件损坏导致的长时故障和软件失败导致的短时故障。长时故障的服务重启时间可能达到数小时乃至数日,比一个时隙的大小高出很多个数量级,可以简单地认为该节点不再可用,服务重启时间为无穷大;短时故障的服务恢复时间取决于虚拟机重启的耗时,可能需要经历若干个时隙。

DaGTR-N 方法用于重调度受到服务器节点故障影响的任务。该算法包括 3 个步骤,首先找出受故障事件影响的任务,然后调整时隙资源矩阵,最后重调度受影响的所有子任务。

假设在某个时刻,服务器节点 u_f 发生故障,在其服务恢复阶段,原本预约在 u_f 上执行的子任务无法获得服务,需要重新调度,将这些子任务称为 A 类子任务。在任务图中,如果其入口到出口的路径上存在 A 类子任务,那么该路径上 A 类子任务的所有后续任务都无法按原计划执行,这些受到间接影响的子任务被称为 B 类子任务。A 类子任务和 B 类子任务都需要进行重新调度。图 4 给出了服务器节点故障场景下的任务调度示例,实心和空心圆圈分别表示当前时隙服务器所有内核的忙碌状态和空闲状态,3 台服务器分别部署了[sf_1, sf_2, sf_3]的一个子集。DAG 任务 G 包括 5 个子任务,原本的调度执行计划为:v_1 于时隙 1 在 u_1 执行;v_2 于时隙 2 在 u_2 执行;v_3 于时隙 2 在 u_3 执行;v_4 于时隙 3 在 u_2 执行;v_5 于时隙 4 在 u_1 执行。系统运行过程中,服务器节点 u_2 在时隙 2 发生故障,需要 1 个时隙重启服务,v_2 属于 A 类子任务。在时隙 2,u_1 和 u_3 都没有可用的计算资源,因此 v_2 重调度后,将于时隙 3 在 u_1 执行。子任务 v_1 和 v_3 不受故障事件的影响,不需要重调度。v_4 和 v_5 的计划执行时间虽然与 u_2 的故障重启时间不存在交集,但因为 v_2 是其前驱子任务,v_2 重调度后,v_4 和 v_5 也无法按原计划执行,需要进行重调度,因此 v_4 和 v_5 属于 B 类子任务。v_4 和 v_5 重调度后分别于时隙 4 和时隙 5 在服务器 u_2 和 u_3 上执行。子任务执行位置调整后,数据边的传输也需要调整,调整前后的数据边传输分别为黑色虚线和红色虚线,值得注意的是,数据边的传输也需要经历若干时隙并占用链路带宽资源,为了便于理解,在图 4 中未标出。

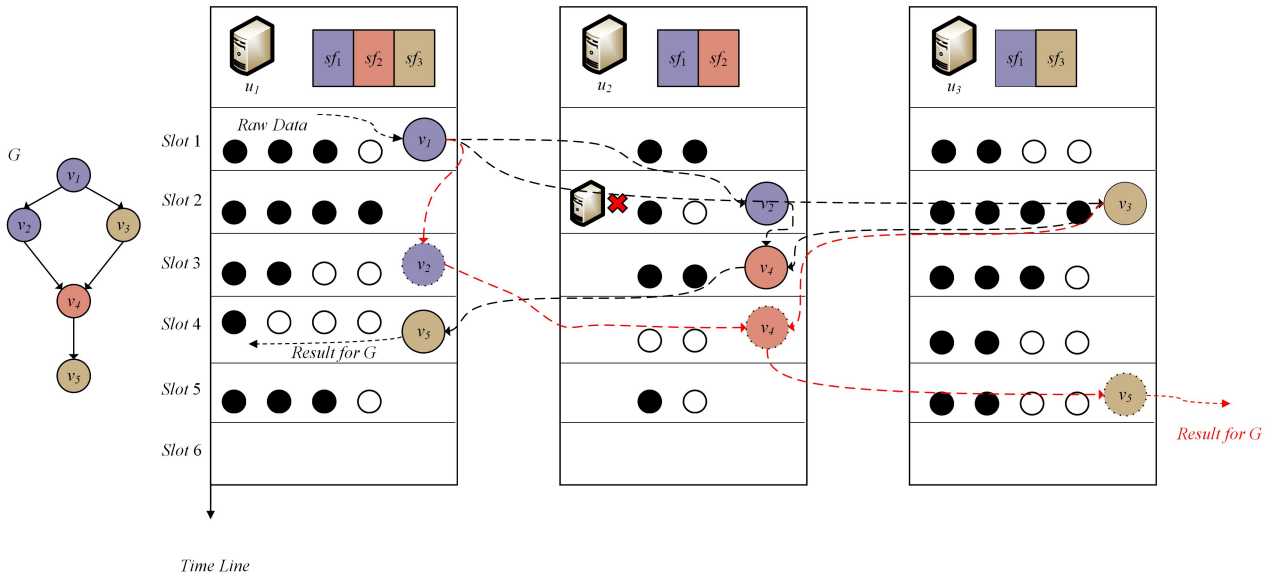


图 4 服务器节点故障场景下的任务调度示例

Fig. 4 Example of task scheduling in the scenario of server node failure

找出所有 A 类和 B 类子任务后,将这些子任务以及相关数据边预约的资源从矩阵中扣除,这些资源可以被再次分配给其他任务。随后将故障服务器在服务重启阶段的可用资源标记为 0。

所有受故障事件影响需要重调度的子任务集合为 S_f,S_f 中所有子任务所属的 DAG 任务的集合为 D_f,用 $D(v)$ 表示子任务 v 所属的 DAG 任务,那么有:

$$\forall v \in S_f, D(v) \in D_f \tag{13}$$

本文算法采用文献[21]中描述的 S-level 来对 D_f 中的所有 DAG 任务的重调度进行优先级排序。S_f 中一个子任务的 S-level 等于它到 DAG 出口子任务的距离,图 4 中,子任务 v_2 和 v_4 的 S-level 分别为 2 和 1。据此,定义 D_f 中每个 DAG 任务的 S-level 等于其所有受影响子任务中最高的 S-level,任务 G 的 S-level 等于 2。一个 DAG 任务的 S-level 越小,表明故障发生时它越接近完成,因此其重调度的优先级越高。而在一个 DAG 内部,所有受影响的子任务的重调度优先级根据

其在拓扑中的排序决定。

随后,本文算法基于贪心思想重调度受影响的子任务。对于S_f中的一个子任务v,首先找到所有部署了服务功能$SF(v)$的服务器集合U_f,然后利用资源矩阵以及式(3)—式(7)来计算v在U_f中每个服务器上执行的完成时间,选择能够使v最早完成的服务器作为其重调度目的地。子算法 DaGTR-N 的伪代码如算法 3 所示。

算法 3 DaGTR-N(Dependency-aware Greedy Task Rescheduling-Node)

输入:故障服务器u_f及其故障时段t_f^{node};故障事件发生时网络中背景任务的集合G_f;G_f中每个 DAG 任务 G 的拓扑;故障事件发生时的时隙资源矩阵

输出:故障影响的子任务的重调度策略;更新后的时隙资源矩阵

1. 定义u_f故障影响的 A 类子任务的集合为O_A,B 类子任务的集合为O_B
2. 令O_A=[ ],O_B=[ ]。
3. foreach G_f中的 DAG 任务 G do
4. foreach G 中的子任务 v do
5. if v 在u_f执行且执行时段与t_f有交集 then
6. 将 v 添加到O_A
7. end if
8. end foreach
9. end foreach
10. foreach O_A中的子任务v_A do
11. v_A所属的 DAG 任务为G_A
12. foreach G_A中的子任务 v do
13. if v 是v_A的后驱任务且 v 不在O_A和O_B中 then
14. 将 v 添加到O_B
15. end if
16. end foreach
17. end foreach
18. foreach $O_A \cup O_B$中的子任务 v do
19. 将 v 及 v 相关数据边占用的资源从资源矩阵中扣除
20. end foreach
21. 根据 S-level 排序决定所有受影响 DAG 任务的重调度优先级
22. 根据子任务 v($v \in O_A \cup O_B$)在 DAG 中的拓扑排序决定其重调度顺序
23. 将资源矩阵中故障部位在故障时段中的可用资源设为 0
24. foreach $O_A \cup O_B$中的子任务 v do
25. 找出所有部署了 SF(v)的服务器节点的集合U_f
26. 遍历U_f中所有节点,根据式(3)—式(7)及资源矩阵计算每个节点完成 v 的时间
27. 选择令 v 最早完成的节点u_x作为 v 的重调度目的地,即 A(v)=u_x
28. 用 OSPF 算法为 v 相关的数据边选择传输路径,并根据资源矩阵分配传输带宽
29. 为 v 分配u_x的计算资源并计算得到 v 在u_x上执行的时段
30. 根据带宽和计算资源分配情况更新时隙资源矩阵
31. end foreach
32. return 所有 A 类、B 类子任务的重调度结果和更新后的时隙-资源矩阵

4.2 通信链路故障场景下的任务重调度

在典型园区边缘网络中,通信链路分为交换网络中的骨干链路和直连边缘服务器的边缘链路。在如图 2 所示的拓扑中,链路$e_{1,8}$和$e_{2,9}$等为边缘链路,$e_{8,9}$和$e_{8,12}$等为骨干链路。一条边缘链路中断,将导致直连的服务器不可用,而一条骨干链路中断后,数据边可以选择其他链路传输,但是网络中的可用数据传输带宽将明显减少。

与 4.1 节中的分析类似,通信链路的故障将导致子任务无法按照原本的调度计划执行,需要对受影响的子任务进行重新调度,并为相关的数据边传输重新选择路径并分配带宽。图 5 给出了链路故障下的任务调度示例。在原本的调度计划中,数据边$\varepsilon_{1,3}$的传输路径经过链路$e_{1,8}$,$e_{8,12}$,$e_{12,7}$。在某个时刻,骨干链路$e_{8,12}$发生中断故障,$\varepsilon_{1,3}$的传输中断,并影响了后续子任务的执行。数据边$\varepsilon_{1,3}$的传输失败导致子任务v_3,v_4,v_5成为 B 类子任务,需要对它们进行重调度,并为相关的数据边重新选择传输路径并分配带宽。

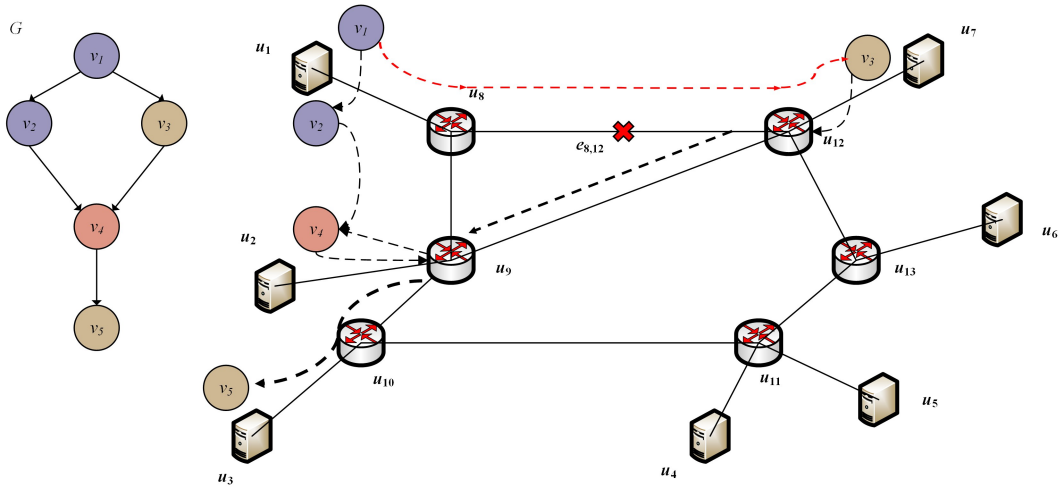


图 5 边缘骨干链路故障场景下的任务调度示例

Fig. 5 Example of task scheduling in the case of edge backbone link failure

链路故障场景下的任务重调度过程与 4.1 节的过程类似：首先找出所有受影响的子任务和相关数据边，然后根据故障情况调整资源矩阵，最后以基于贪婪的思想重调度受影响的所有子任务。DaGTR-L 的伪代码如算法 4 所示。

算法 4 DaGTR-L(Dependency-aware Greedy Task Rescheduling-Link)

输入：故障链路e_f及其故障时段t_f^{link}；故障事件发生时网络中背景任务的集合G_f；G_f中每个 DAG 任务 G 的拓扑；故障事件发生时的时隙资源矩阵

输出：故障影响的子任务的重调度策略；更新后的时隙资源矩阵

1. if 故障链路e_f是边缘链路 then
2. u_f是e_f直连的边缘服务器
3. 调用算法 3 对所有受影响的子任务进行重新调度，算法 3 的输入为u_f加上算法 4 的输入
4. else if 故障链路e_f是骨干链路 then
5. 定义e_f故障影响的 B 类子任务集合为O_B
6. foreach G_f中的 DAG 任务 G do
7. foreach G 中数据边$\epsilon_{i,j}$ do
8. if $\epsilon_{i,j}$的传输路径经过e_f并且传输时段和t_f^{link}存在交集 then
9. 将$\epsilon_{i,j}$的终点子任务v_j添加到O_B中
10. foreach v_j的后驱子任务v_j^{succ} do
11. 将v_j^{succ}添加到O_B中
12. end foreach
13. end if
14. end foreach
15. end foreach
16. foreach O_B中的子任务 v do
17. 将 v 及 v 相关数据边占用的资源从资源矩阵中扣除
18. end foreach
19. 调用算法 3 中的第 21－31 行来为所有受影响的子任务进行重新调度并更新资源矩阵(这里令$O_A=[\ ]$)
20. end if
21. return 所有受影响子任务的重调度结果和更新后的时隙资源矩阵

5 实验仿真

为了验证本文算法的有效性，基于 Python 编写了一个仿真器 EdgeSimDAG，并进行了大量的实验，仿真实验运行于配置有 8 核 1.8 GHz Inter Core i7 处理器及 16 GB 2666 MHz DDR4 内存的 ThinkPad T14 Laptop 中，主机系统为 Windows 10，软件环境为 Python 3.7。实验结果证明，DaGTR-N 和 DaGTR-L 能够有效应对服务器节点和通信链路的故障场景，有效避免了故障事件引起的任务失败。

5.1 仿真设置

仿真实验中利用图 2 中的拓扑作为实验拓扑，拓扑中包含 7 台边缘服务器，6 台 OF 交换机以及若干通信链路。假设网络中一共有 10 种不同的 SF，记作 $F=[sf_1, sf_2, \cdots, sf_{10}]$，每台边缘服务器随机部署其中的 5 种 SF。每台边缘服务器可用的计算资源表示为其可用的 CPU 内核数量，同一台服务器的 CPU 内核处理能力是一致的，实验中，每台服务器的 CPU 内核数量在(2,4,8)范围内随机取值，内核的处理频率为 1～3 GHz 之间的随机值。拓扑中每条通信链路的带宽设置为 200～500 Mbps。远程云中心的可用计算资源假设为无限的，其 CPU 处理频率设置为 3 GHz，从边缘到云端的数据传输带宽和传播时延分别设置为 200 Mbps 和 300 ms。设置每个 DAG 任务包含 8～10 个子任务，每个子任务对应唯一一种 SF。每个子任务处理所需的 CPU 周期为 0.3 ～0.5 GHz，每条数据边要传输的数据量为 5～10 MB。另外，设置时隙资源矩阵的每个时隙大小为 100 ms，即资源分配的最小单位是 100 ms。表 2 列出了仿真实验的参数设置。

表 2 仿真参数设置

Table 2 Simulation parameter setting

Parameters	Values
系统中 SF 种类数量	10
每台服务器部署的 SF 种类数量	5
服务器u_k的可用 CPU 内核数量(c_k)	[2,4,8]
服务器u_k的 CPU 处理频率(f_k)/GHz	[1,3]
云中心 CPU 处理频率(f_{cloud})/GHz	3
通信链路$e_{i,j}$的带宽($b_{i,j}$)/Mbps	[200,500]
边缘到云端的传输带宽(b_{cloud})/Mbps	200
边缘到云端的传输延迟(d_{cloud})/ms	300
DAG 任务中包含的子任务数量(n_f)	[8,10]
处理每个子任务所需的 CPU 周期/GHz	[0.3,0.5]
每条数据边传输的数据量/Mb	[5,10]
时隙资源矩阵中时隙的大小/ms	100

5.2 实验结果及分析

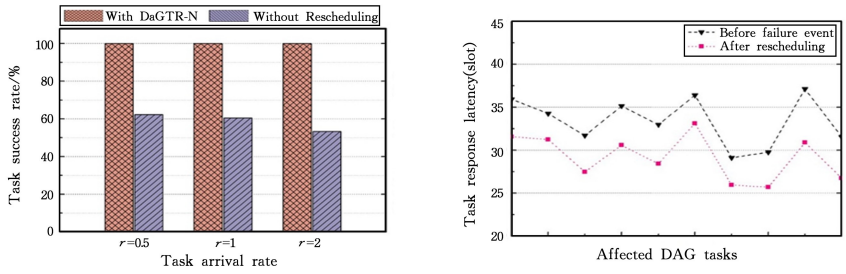
实验在 500 个时隙里对到达的 DAG 任务采用算法 1 进行调度，设定服务器短时故障所需的服务恢复时间为 10 个时隙，服务器长时故障及通信链路故障的服务恢复时间为无穷大，可以认为该段资源在后续调度中无法再提供服务。为了分别体现 DaGTR-N 和 DaGTR-L 在节点和通信链路故障场景下的有效性，下面设置了 3 组实验。

实验 1 在系统运行过程中令某个服务器节点发生硬件长时故障，且后续无法使用。然后采用 DaGTR-N 算法调度所有受故障影响的任务。

实验 2 在每个时隙，每个服务器节点以 0.1%～0.5% 的概率发生软件短时故障，为了便于分析，保证每个时隙处于故障状态的服务器数量不超过 1 台。故障事件发生时，采用 DaGTR-N 调度所有受故障影响的任务。

实验 3 在系统运行过程中令某段骨干链路发生故障中断，且后续无法使用。采用 DaGTR-L 调度所有受故障影响的任务。

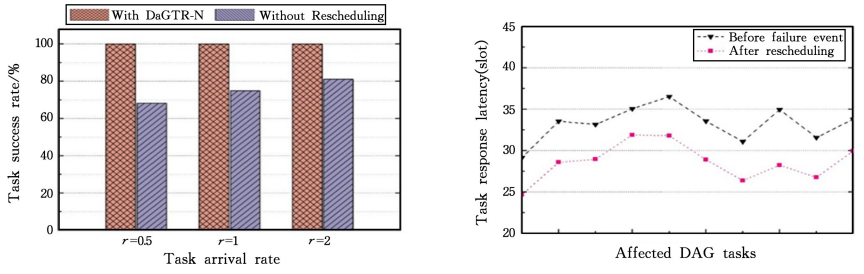
在 3 组实验中，分别设置故障事件下不对任务进行重调度的场景作为实验对照，受故障影响不进行重调度的任务将会执行失败。图 6(a)、图 7(a)、图 8(a)分别给出了 3 组实验中，不同的任务到达率 r(一个时隙中到达的 DAG 任务数量)条件下，发生故障时网络中正在执行的背景任务的执行成功率，数据是 10 次仿真后的平均值。



(a)是否采取重调度策略的任务执行成功率比较　　(b)$r=1$ 时,重调度前后的受影响任务响应延迟比较

图 6　节点长时故障事件发生时采用 DaGTR-N 算法重调度受影响的 DAG 任务

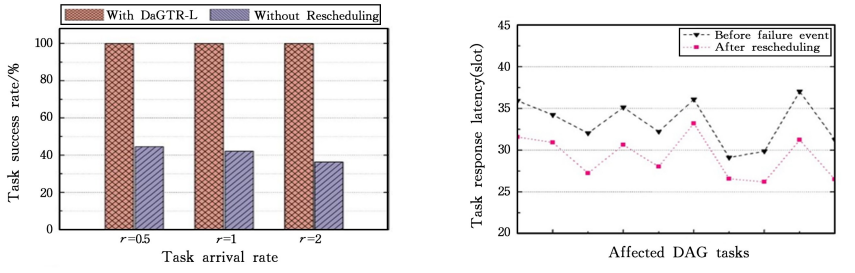
Fig. 6　Using DaGTR-N to reschedule the affected DAG tasks when long-term failure events of servers happen



(a)是否采取重调度策略的任务执行成功率比较　　(b)$r=1$ 时,重调度前后的受影响任务响应延迟比较

图 7　节点短时故障事件发生时采用 DaGTR-N 算法重调度受影响的 DAG 任务

Fig. 7　Using DaGTR-N to reschedule the affected DAG tasks when short-term failure event of servers happen



(a)是否采取重调度策略的任务执行成功率比较　　(b)$r=1$ 时,重调度前后的受影响任务响应延迟比较

图 8　链路故障事件发生时采用 DaGTR-L 算法重调度受影响的 DAG 任务

Fig. 8　Using DaGTR-L to reschedule the affected DAG tasks when failure event of links happen

从这些实验结果可以看出,所有受故障影响的任务在重调度后能够全部成功执行,也就是说重调度算法能够有效避免故障事件造成任务执行失败。另外,从图 6(a)—图 8(a)中可以看出,在同样的任务到达率条件下,链路的长时故障事件影响的正在执行的 DAG 任务比例最高,节点长时故障其次,受节点短时故障影响的任务最少。这样的结果是显然的,因为在基底网络中,一条数据边的传输路径可能经过多条链路,意味着多条传输路径可能经过同一条链路,即单条链路的复用率很高。

而图 6(b)、图 7(b)和图 8(b)则分别给出了 3 组实验中,在 $r=1$ 的条件下,所有受故障影响的 DAG 任务重调度前后的响应延迟比较。可以看出,经过算法重调度后的 DAG 任务能够更早完成,这是因为重调度算法基于贪心的思想为每个受故障影响的子任务选择能够最早完成的服务器来执行调度,因此整个 DAG 任务也能够更早地完成。

结束语　本文研究了边缘计算场景中面向资源故障的 DAG 任务的调度问题。本文首先针对边缘系统常见的服务器节点和链路故障事件进行了建模,并建立了时隙-资源矩阵的模型来细粒度地表示和分配系统资源,随后提出了故障场景下的任务重调度算法 DaGTR,该算法包括 DaGTR-N 和 DaGTR-L 两种子算法,分别用于节点和链路故障事件发生时的受影响任务调度,该算法基于贪心思想进行重调度,为每个受影响的子任务和数据边重新选择合适的服务器及传输路径。仿真结果表明,重调度算法 DaGTR 能够找出所有受故障事件影响的子任务,并将它们调度到其他正常运行的服务器执行,而且能够有效降低所有受影响 DAG 任务的响应延迟。

参 考 文 献

[1] China Internet Network Information Center. The 45th Statistical Report on the Development of Internet in China [R/OL]. Beijing, CNNIC Report, 2020. http://www.cac.gov.cn/2020-04/

27/c_1589535470378587.htm.

[2] SATYANARAYANAN M. A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets [J]. ACM SIGMOBILE Mobile Computing and Communications Review, 2015, 18(4): 19-23.

[3] WANG J, PAN J, ESPOSITO F, et al. Edge cloud offloading algorithms: Issues, methods, and perspectives [J]. ACM Computing Surveys (CSUR), 2019, 52(1): 1-23.

[4] LIANG J B, ZHANG H N, JIANG C, et al. Research Progress of Task Offloading Based on Deep Reinforcement Learning in Mobile Edge Computing [J]. Computer Science, 2021, 48(7): 316-323.

[5] LIU T, FANG L, GAO H H. Survey of Task Offloading in Edge Computing [J]. Computer Science, 2021, 48(1): 11-15.

[6] HU Y C, PATEL M, SABELLA D, et al. Mobile edge computing—a key technology towards 5G [J]. ETSI White Paper, 2015, 11(11): 1-16.

[7] LI H, LI X H, XIONG Q Y, et al. Edge Computing Enabling Industrial Internet: Architecture, Applications and Challenges [J]. Computer Science, 2021, 48(1): 1-10.

[8] BHATTCHARYA A, DE P. Computation offloading from mobile devices: can edge devices perform better than the cloud? [C] // Proceedings of the Third International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing. 2016: 1-6.

[9] SHI W S, ZHANG X Z, WANG Y F, et al. Edge Computing: State-of-the-Art and Future Directions [J]. Journal of Computer Research and Development, 2019, 56(1): 69-89.

[10] COFFMAN E G. Computer and Job-shop Scheduling Theory [J]. Oral Surgery Oral Medicine Oral Pathology, 1976, 5(2): 143-149.

[11] LIU L, TAN H S, JIANG H C, et al. Dependent task placement and scheduling with function configuration in edge computing [C] // Proceedings of the International Symposium on Quality of Service. ACM, 2019.

[12] HE K, MENG X, PAN Z, et al. A Novel Task-Duplication Based Clustering Algorithm for Heterogeneous Computing Environments [J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 30(1): 2-14.

[13] QI Q, WANG J, MA Z, et al. Knowledge-driven Service Offloading Decision for Vehicular Edge Computing: A Deep Reinforcement Learning Approach [J]. IEEE Transactions on Vehicular Technology, 2019, 68(5): 4192-4203.

[14] OO T, KO Y B. Application-aware Task Scheduling in Heterogeneous Edge Cloud [C] // International Conference on Information and Communication Technology Convergence (ICTC). IEEE, 2019: 1316-1320.

[15] LIU H, JIA H, CHEN J, et al. Computing Resource Allocation of Mobile Edge Computing Networks Based on Potential Game Theory [C] // IEEE 4th International Conference on Computer and Communications (ICCC). IEEE, 2018.

[16] COLMAN-MEIXNER C, DEVELDER C, TORNATORE M, et al. A Survey on Resiliency Techniques in Cloud Computing Infrastructures and Applications [J]. IEEE Communications Surveys & Tutorials, 2017, 18(3): 2244-2281.

[17] MARTINS J, AHMED M, RAICIU C, et al. ClickOS and the art of network function virtualization [C] // Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, 2014.

[18] MENG J Y, TAN H S, LI X Y, et al. Online Deadline-aware Task Dispatching and Scheduling in Edge Computing [J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(6): 1270-1286.

[19] MCKEOWN N, ANDERSON T, BALAKRISHNAN H, et al. OpenFlow [J]. ACM SIGCOMM Computer Communication Review, 2008, 38(2): 69.

[20] TALEB T, KSENTINI A, SERICOLA B. On Service Resilience in Cloud-native 5G Mobile Systems [J]. IEEE Journal on Selected Areas in Communications, 2016, 34(3): 1-1.

[21] KANIZO Y, ROTTENSTREICH O, SEGALL I, et al. Optimizing Virtual Backup Allocation for Middleboxes [J]. IEEE/ACM Transactions on Networking, 2017, 25(5): 2759-2772.

CAI Ling-feng, born in 1995, postgraduate. His main research interests include edge computing, cyberspace security and machine learning.

XING Chang-you, born in 1982, Ph.D, associate professor. His main research interests include software defined network, network measurement.