RT-SANE : Real Time Security Aware Scheduling on the Network Edge

Anil Singh
Department of Computer Science & Engineering, Indian Institute of Technology
Ropar, India
anil.singh@iitrpr.ac.in

Nitin Auluck
Department of Computer Science & Engineering, Indian Institute of Technology
Ropar, India
nitin@iitrpr.ac.in

Omer Rana
School of Computer Science & Informatics, Cardiff University
Cardiff, UK
ranaof@cardiff.ac.uk

Andrew Jones
School of Computer Science & Informatics, Cardiff University
Cardiff, UK
jonesacf@cardiff.ac.uk

Surya Nepal
Distributed Systems Security Group, CSIRO
Sydney, Australia
surya.nepal@data61.csiro.au

ABSTRACT

Edge computing extends a *traditional* cloud data centre model often by using a microdata centre (*mdc*) at the network edge for computation and storage. As these edge devices are in proximity to users, this results in improved application response times and reduces load on the cloud data center (*cdc*). In this paper, we propose a security and deadline aware scheduling algorithm called *RT-SANE* (Real-Time Security Aware scheduling on the Network Edge). Applications with stringent privacy requirements are scheduled on an *mdc* closer to the user, whereas others can be scheduled on a *cdc* or a remote *mdc*. We also discuss how application performance and network latency influence the choice of an *mdc* or *cdc*. The intuition is that due to a lower communication latency between the user & the *mdc*, more applications are able to meet their deadlines when run on the *mdc*. Conversely, applications with loose deadlines may be executed on a *cdc*. In order to facilitate this, we also propose a distributed orchestration architecture and protocol that is both performance & security aware. Simulation results show that *RT-SANE* offers superior real-time performance compared to a number of other scheduling policies in an Edge computing environment, while meeting application privacy requirements.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Real-time schedulability**; • **Information systems** → *Network attached storage*; • **Security and privacy** → *Distributed systems security*; • **Human-centered computing** → *Ubiquitous and mobile computing theory, concepts and paradigms*;

KEYWORDS

Internet of Things; IoT; Fog Computing; Edge Computing; Cloud Computing; Cloudlets; Micro Data Centers; Cloud Data Center; Distributed Orchestration; Real-Time; Security.

UCC'17, December 5–8, 2017, Austin, Texas, USA

ACM ISBN 978-1-4503-5149-2/17/12…$15.00
https://doi.org/10.1145/3147213.3147216

1 INTRODUCTION

Applications utilizing Internet of Things (IoT) devices are increasingly generating large volumes of data, that needs to be processed within a particular deadline [16]. Although numbers vary (across Gartner, Cisco and other market forecasts), it is estimated that by 2020, there will be multi-billion devices connected to the Internet [25]. In order to effectively analyze the vast quantities of data generated by these devices, it becomes imperative to build scalable, distributed systems that can respond to this data management and analysis challenge. Smart phones these days have better hardware specifications than desktops of not too long ago. Such devices have enabled the processing of data intensive applications closer to the data generation sources. The data offloaded by these devices may be processed at a remote data center [15], which is also the dominant mode of operation at present. However, there may be considerable network delay between the devices and the remote data center, implying that by the time the data reaches the data center it may be difficult to reach their Quality of Service (QoS) targets [1] – especially in the case of real-time applications [18]. In such applications, data analysis must be completed within a pre-specified deadline to meet user requirements. Examples of such systems include real time gaming, streaming of audio/ video content over the Internet, command and control systems within built environments or in intelligent transport, etc.

To overcome latency associated with migrating data across a network for processing, it makes sense to leverage the infrastructure at the edge of the network [3] to perform partial computation. Such infrastructure may include routers, switches, gateways, and integrated access devices used for offloading some of the computation originally meant for the data center. This Edge computing model has been described in [15, 17]. Typically, applications access the cloud through access points, which allow the data to travel across the network to the data center. A key insight here is that

such access points may be extended to provide computing and storage services at the network edge, via cloudlets/micro data centers (*mdc*s). Devices such as smart phones can communicate with micro data centers, which in turn communicate with the cloud data center (*cdc*). The *mdc*s may also communicate with each other to share application (execution) state, for example, to share the state of applications that are preempted from their local *mdc*, due to mobility, and are resumed on another *mdc* in their new coverage area.

In this paper, we propose *RT-SANE*, a security aware, real-time scheduling algorithm to support integration of an *mdc* and *cdc*, to balance performance and data privacy/security constraints of applications. The algorithm is security-aware, in the sense that private applications are executed only on the local/private *mdc* of a user. Semi-private applications may be executed on the local/private *cdc*. Public applications may be sent to a remote *mdc* or *cdc* for execution. Subject to meeting security constraints, *RT-SANE* schedules applications with tight deadlines on a local or remote *mdc*. Applications with loose deadlines may be sent to either the local or remote *cdc*.

An application where such an algorithm may be useful is "Ambient Assisted Living". Consider the case of elderly people having weak eyesight living alone in their homes. They could use a device like Google Glass to recognize people who come to visit them. A small database of images of known people could be stored at the local *mdc* of the person. This would make the image matching algorithm run with minimal latency, as an image match request does not need to be transmitted to a *cdc* for processing. Moreover, the database of known people is private to the user and does not need to be stored at the *cdc*. Storing images on the *mdc* may also have privacy limitations, based on organizations from which visitors originate. There is therefore a need to consider a more balanced approach for hosting such data between an *mdc* and a *cdc*.

The rest of this paper is organized as follows. Section II comprises a brief survey of related work. A distributed orchestration architecture and protocol is described in section III. The model and problem formulation is discussed in section IV. The *RT-SANE* algorithm is described in section V, with simulation results in Section VI. Finally, section VII concludes the paper.

2 RELATED WORK

Performing all computations at a cloud data center has shortcomings, as it places a heavy load on the cloud infrastructure. This can cause response time delays to user applications. This delay may be acceptable for applications such as web browsing, for others such as interactive gaming, this delay could be unacceptable. This is where edge computing devices could play a role. If some of the computation could be performed close to the user(s) at the network edge, the response times could greatly improve. Edge computing could also be used in environments with limited resilience (or a high failure rate), such as in a disaster recovery scenario [21]. In these applications, overcoming failures in communication networks and reduced response times are of paramount importance. Users in a "smart home", for example, may not be comfortable in sending their sensitive data to a cloud for computation. However, they may accept use of resources at the edge of the network, that may be under their control/ownership. However, there is a need for application scheduling schemes on the network edge that are security aware. Edge computing seems to be a viable alternative due to its promise of providing lower response times and better security.

An overview of edge computing is provided in [20], where authors also present several case studies that can benefit from edge computing. Some of these applications are video analysis, smart homes, smart cities, augmented reality, visual entertainment games, connected health, among others. Two scheduling algorithms for edge networks have been proposed [23, 24]. The common thread in both these works is that the authors attempt to exploit the edge of the network to perform computation. In [24], the authors propose an ILP (Integer Linear Programming) based algorithm to solve the scheduling problem. The proposed *iFogStor* system provides close to optimal results, although at a high cost, making the outcome difficult to scale to large problem sizes. In the same paper, a heuristic version of the algorithm, called *iFogStorZ*, approximates the result at an economical cost. However both [23, 24] do not take application deadlines into account.

There has been some work on mobility aware task allocation for cloud computing [12, 13]. In [12], the authors carry out a survey of scheduling algorithms in cloud computing. Furthermore, they propose an allocation algorithm that is mobility aware. In [13], the authors propose a heuristic algorithm that tries to balance the trade-off between the application makespan and the monetary cost of cloud resources. However, in both [12] & [13], real-time tasks with deadlines have not been considered.

Real-time scheduling has received significant attention in the past [9–11]. A number of algorithms have been proposed for various kinds of architectures (both uni- and multi-processor systems). However, there is limited work on real-time scheduling in edge networks [14]. To the best of our knowledge, no single algorithm for edge networks has been proposed that is security aware, and that considers real-time tasks with deadlines. We have tried to fill this void in this work.

3 DISTRIBUTED ORCHESTRATION ARCHITECTURE & PROTOCOL

3.1 Distributed Orchestration Architecture

A centralized orchestration has a number of shortcomings. First, it has a single point of failure and is less resilient to attacks, such as a Denial of Service attack. Second, it is inherently unsuitable for edge computing due to the extra latency introduced by the need to do frequent communication with the centralized orchestrator that is deployed in the cloud. To address this challenge, we introduce a fully-decentralized distributed orchestration mechanism using the underlying principal of a collaborative multi-agent system. In our model, each computing device in the network has an orchestration agent. For each job, the orchestration agent creates job specific agent instances at different nodes and they collaboratively work towards achieving the goal (i.e., completing the user job at a minimum cost within a given deadline, without violating a specified security requirement).

A conceptual architecture of our distributed orchestration mechanism for edge computing is shown in figure 1. In the figure, a user

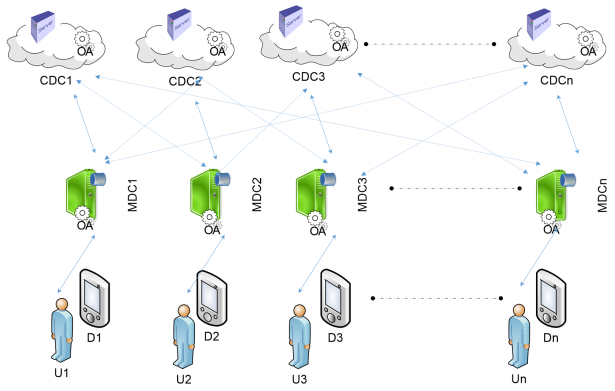

Figure 1: Distributed Orchestration Architecture (DOA) for Edge Computing

has a mobile device (D_x), which can sense data, build a computational job, receive results from job execution, and act on the results (visualization or actuation). The user job is submitted to an *mdc* capable of storing data and executing jobs with low latency. For simplicity, we assume that each user device is connected to one *mdc* (local *mdc*). Similarly, each *mdc* is connected to at least one *cdc*. We assume that a device D_x has trust in its local/home *mdc*. The home *cdc* is classified as semi-trusted, i.e., it can execute the job as requested, but cannot guarantee the privacy of data and jobs. Other non-local/remote *mdc*s and *cdc*s are untrusted.

3.2 Best Effort Orchestration Protocol

Next, we describe an orchestration protocol by considering three different cases: (a) a job executes at its local *mdc*, (b) a job executes at its local *cdc*, and (a) a job executes at a remote *mdc*. Figure 2 shows a sequence diagram for these three different cases, illustrating a best effort orchestration protocol.

In the first case, device (D_1) submits a job to its local *mdc* (mdc_1), which is able to meet all the specified requirements of cost, deadline and security. The job is executed at the local *mdc* and the result sent back to the device. As a local *mdc* is trusted, this case represents a scenario of job execution with the highest level of security. In the second case, the job submitted by D_1 cannot be executed on the local *mdc*. This could be due to the *mdc* being busy performing another job, and the job to be executed cannot meet the deadline if it waits until the completion of all scheduled jobs. In such cases, the local *OA* interacts with a *cdc* (cdc_1) to create a proxy agent, which takes over the responsibility of job completion. The result is returned to D_1 via mdc_1. This case represents a scenario where a job can execute on a semi-trusted resource and is able to meet the two other requirements (i.e., cost and deadline). In the third case, the job cannot be completed by the local *cdc* (cdc_1). This means the home *cdc* has to find other *cdc*s or *mdc*s that can complete the job. In our example, cdc_1 first contacts cdc_2, but it cannot meet the latency requirement. This means the job has to be computed closer to the device to meet latency requirements. Next, cdc_2 finds another *mdc* closer to D_1 that can meet these latency requirements. It instantiates a new proxy agent at the remote *mdc* (mdc_3) and passes the job to it. It terminates the local *OA* instance as it is no longer needed. Now, mdc_3 completes the job and returns the results directly to mdc_1, which then passes it to D_1. All proxy agent instances for a specific job are terminated once the job is completed.

4 SYSTEM MODEL & PROBLEM FORMULATION

Table 1: Table of Key Notations

C	Set of Cloud Data Centers
c_l, c_f	Local & remote cloud data centers
M	Set of micro-data centers
m_l, m_f	Local & remote micro data centers
J	Set of all jobs/applications
U	Set of users
u_i	Particular user $u_i \in U$
D	Set of Devices
D_i	Particular device $D_i \in D$
j_i	Specific job/task/application $\in J$
T_j	Set of tags assigned to jobs
T_r	Set of tags assigned to resources
t_{j1}, t_{j2}, t_{j3}	Tag for private, semi-private, public jobs
t_{r1}, t_{r2}, t_{r3}	Tag for trusted, semi-trusted, untrusted resource
$cp(m)$	capacity of *mdc* m
$cp(c)$	capacity of *cdc* c
$ted(j_i)$	Total execution cost of job j_i
$et(j_i)$	Execution cost of an interaction of j_i
$st(j_i)$	Start time of job j_i
$ct(j_i)$	Completion time of job j_i
$d(j_i)$	Deadline for task j_i
$cd(u_i, j_i, m_l)$	Communication delay for j_i between m_l and u_i
$cd(u_i, j_i, c_l)$	Communication latency for j_i between u_i and c_l
DC	Deployment cost
UT_{m_l}, UT_{m_f}	Utilization for m_l & m_f
UT_{c_l}, UT_{c_f}	Utilization for c_l & c_f

The key notations used in the system model are shown in table I. The architecture of our system consists of a set of x cloud data centers, $C = \{c_1, c_2, c_3, ..., c_x\}$. A cloud data center c is of two types: local (c_l) or remote (c_f). Each cloud data centre is connected to a set of z micro-data centers, $M = \{m_1, m_2, m_3, ..., m_z\}$. A micro-data centre *mdc* can be of two types: local (m_l) or remote (m_f). Each *mdc* has processing capacity, denoted by $cp(m)$, given in Millions of Instructions per Second (MIPS) – a MIPS rating has been used to ensure that it aligns with the use of iFogSim used here. Other alternative metrics to capture the computing capacity of a *cdc* or *mdc* may also be used. We assume that all the *mdc*s have the same processing capability, i.e., they are homogeneous. The implication of this is that each job takes the same time to execute on each *mdc*. This assumption is made to explain the model and the problem; however, the model and problem are valid beyond this assumption. There is a communication link from each $c \in C$ to each $m \in M$. Each link has a bandwidth of bw.

Let U be the set of all users, such that $U = \{u_1, u_2, u_3, ..., u_n\}$. Let D be set of all devices, such that $D = \{D_1, D_2, D_3, ..., D_n\}$.

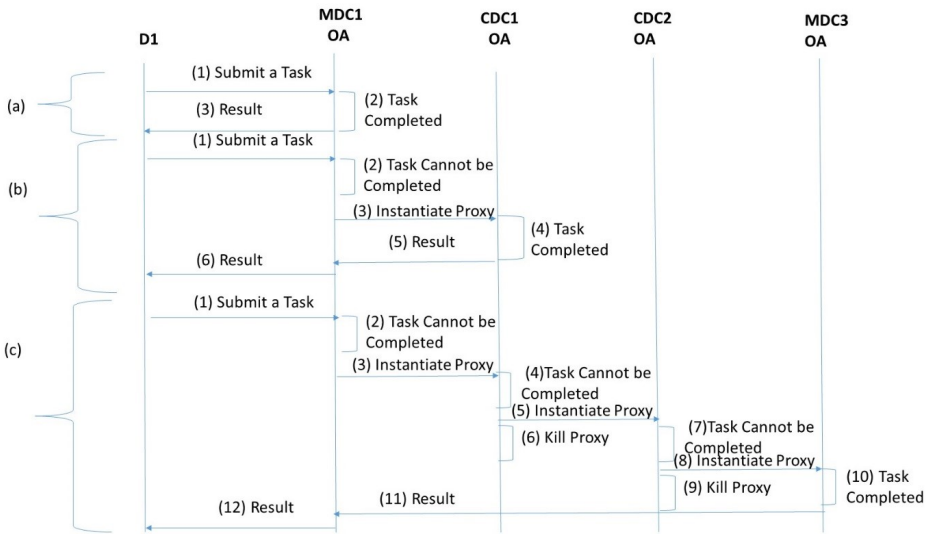

Figure 2: Sequence diagram illustrating the Orchestration Protocol

Each user u_i has an associated device D_i. Each user also has a set of jobs/applications that need to be executed on the *cdc* or *mdc*. Let J be the set of all such jobs, such that, $J = \{j_1, j_2, j_3, ...j_y\}$. Each job j_i is represented as a tuple: $< et(j_i), d(j_i), t(j_i) >$. Here, $et(j_i)$ is the execution time of the job, $d(j_i)$ is the deadline of the job, & $t(j_i)$ is the security tag assigned to the job. Since jobs have real-time processing requirements, they need to be executed before their deadline. In this work, we assume that the jobs are non-preemptive, which implies that once a job has started execution, it may not be interrupted to execute another job. Both preemptive and non-preemptive scheduling algorithms have their pros and cons, which are beyond the scope of this paper.

Table 2: Mapping based on Security Tags

	t_{r1}	t_{r2}	t_{r3}
t_{j1}	Y	N	N
t_{j2}	Y	Y	N
t_{j3}	Y	Y	Y

Each job $j_i \in J$, has one of three security tags attached to it. The set of security tags assigned to a job is given by $T_j = \{t_{j1}, t_{j2}, t_{j3}\}$. Tag t_{j1} is attached to jobs that are "private", implying that the owner of the job wants it to be executed on on his/her local *mdc*. Tag t_{j2} is attached to jobs that are "semi-private", implying that they may be executed on the user's local *cdc*. Finally, tag t_{j3} is attached to jobs that are "public", meaning they may be executed on any resource, including other *cdc*s & *mdc*s.

Similarly, execution resources also have security tags assigned to them. The set of security tags assigned to resources is given by $T_r = \{t_{r1}, t_{r2}, t_{r3}\}$. Tag t_{r1} is attached to resources that are "highly trusted", which is the user's own *mdc*. Resources that are "semi-trusted", for example, the local *cdc* are assigned a tag of t_{r2}. Finally, tag t_{r3} is attached to resources that are "untrusted", such as *mdc*s & *cdc*s which are outside the user's home network. Table II shows the possible mappings of jobs to resources. A 'Y'denotes a valid mapping, and a 'N'denotes a mapping that is not valid.

Provided the security constraints given in table II are met, given J, the jobs with "loose deadlines" are executed on the *cdc*, and the jobs with "tight" deadlines are sent to the *mdc* for execution. We discuss the proposed algorithm in greater detail in section V. Out of J, let us say that J' jobs are sent to the *mdc*s for execution, where $J' \leq J$. Trivially, $J - J'$ jobs are executed on the *cdc*. The total execution duration of a job $j_i \in J$, is given by $ted(j_i)$. This corresponds to the time for which the user will interact with the application. For example, this could be the total amount of time that a user is playing an on-line multi-player game. An interactive application, such as the one just mentioned, consists of multiple interactions between the system and the user. Let the execution time of such a single interaction on be $et(j_i)$.

The start time of j_i is denoted by $st(j_i)$. Since we consider independent jobs, all jobs can start at time 0. The completion time of j_i is denoted by $ct(j_i)$. Formally, $ct(j_i) = st(j_i) + et(j_i)$. The communication latency between job j_i of user u_i and its local *mdc*, m_l is denoted by $cd(u_i, j_i, m_l)$. We assume that bw is the bandwidth of the communication link between the user and a local *mdc*. The size of the data transmitted by the user is given by $s(j_i)$. The time to initialize the communication link is denoted by t. Also, the cost of transferring the state of the job is $sc(u_i, j_i, m_l)$. The communication cost between a user u_i, a job j_i & an m_l can now be modeled as:

$$cd(u_i, j_i, m_l) = t + sc(u_i, j_i, m_l) + \frac{s(j_i)}{bw} \quad (1)$$

The jobs assigned to the local mdc, m_l need to finish before their deadline, i.e.:

$$st(j_i) + et(j_i) + cd(u_i, j_i, m_l) \leq d(j_i) \quad (2)$$

Based on the security tags, jobs may be sent to c_l for execution. The communication delay between a job j_i, of user u_i and c_l is denoted by $cd(u_i, j_i, c_l)$. This latency may be modeled as follows:

$$cd(u_i, j_i, c_l) = t + sc(u_i, j_i, c_l) + \frac{s(j_i)}{bw} \tag{3}$$

Here, job j_i is executed on its local cloud data center c_l and must finish before the deadline. In other words:

$$st(j_i) + et(j_i) + cd(u_i, j_i, c_l) \leq d(j_i) \tag{4}$$

Adding a job j_i to an *mdc* should not result in the deadline of the currently executing/scheduled jobs to be missed. Let $J(m_l)$ be the set of all jobs that are currently running on a m_l. Let $et(j)$ be the execution time of $j \in J(m_l)$. A new job p may be chosen for execution on m_l, if and only if the following condition holds:

$$\forall j \in J(m_l), \forall m_l \in M, st(j) + et(j) + cd(u_i, j, m_l) \leq d(j_i) \tag{5}$$

Given that n is the number of jobs, let n' be the number of jobs that meet their assigned deadlines, where $n' \leq n$. We define Success Ratio (SR) as the *ratio of the number of jobs that meet their deadlines to the total number of jobs considered.* Hence, $SR = \frac{n'}{n}$. SR is an important criteria, as it influences the accuracy of running a particular application on a distributed infrastructure.

Next, we model the cost of deployment, i.e. the cost to execute jobs on m_l and c_l. This cost is represented by $DC(m_l)$ & $DC(c_l)$ respectively. $DC(m_l)$ has several components, such as the communication delay between the jobs and the *mdc*. This is represented by $cd(u_i, j_i, m_l)$. Another component is the cost to power on the *mdc* server, if it is not already running. This is given by $in(m_l)$. In case the server is on, this cost is ignored. The third component is the time that the server has to be operational for executing the jobs. This is represented as $t(m_l)$. The deployment cost for each $m_l \in M$ is given by:

$$DC(m_l) = in(m_l) + \sum cd(u_i, j_i, m_l) + t(m_l) \tag{6}$$

Let $J(m_l)$ be the set of jobs dispatched to $m_l \in M$. Let $et(j_i, m_l)$ be the execution cost of one particular job $j_i \in J(m_l)$ assigned to m_l, where $t(m_l)$ can be represented as:

$$\sum et(j_i, m_l), \forall j_i \in J(m_l) \tag{7}$$

The deployment costs of all $m_l \in M$ can be specified as:

$$DC(M_L) = \sum DC(m_l) \tag{8}$$

Likewise, the deployment cost for a local cloud data center $c_l \in C$ can be expressed as:

$$DC(c_l) = in(c_l) + \sum cd(u_i, j_i, c_l) + t(c_l) \tag{9}$$

$t(c_l)$ can be represented as:

$$\sum et(j_i, c_l), \forall j_i \in J(c_l) \tag{10}$$

The deployment costs of all $c_l \in C$ can be written as:

$$DC(C_L) = \sum DC(c_l) \tag{11}$$

Finally, the total cost of deployment on all local *cdc*s & *mdc*s is given by:

$$DC = DC(M_L) + DC(C_L) \tag{12}$$

Next, we model the utilizations of the m_l and c_l. For a particular user u_i, the utilization of an *mdc*, $m_l \in M$ can be expressed as:

$$UT_{m_l}(u_i) = \frac{\sum et(j_i, m_l)}{cp(m_l)}, \forall j_i \in J(m_l) \tag{13}$$

Similarly, the utilization of a *cdc*, $c_l \in C$ can be expressed as:

$$UT_{c_l}(u_i) = \frac{\sum et(j_i, c_l)}{cp(c_l)}, \forall j_i \in J(c_l) \tag{14}$$

Here, $J(c_l)$ is the set of all jobs assigned to *cdc*, c_l.

For a particular user u_i, the total utilization of the local *mdc* and *cdc* of a specific user u_i can be expressed as:

$$UTL(u_i) = UT_{m_l}(u_i) + UT_{c_l}(u_i)$$

The total system utilization, for all users, can now be represented as:

$$UTL(system) = \sum_{i=1}^{n} UTL(u_i)$$

Note that a similar formulation will hold for remote *mdc*s & *cdc*s.

The optimisation problem that we solve in this work can be formulated as follows:

Maximize SR, i.e. maximize $\frac{n'}{n}$ & $UTL(System)$, while minimizing $DC, \forall m \in M, \forall c \in C$, subject to the security tag based jobs → resources mapping shown in table II. This mapping ensures that the privacy concerns of users are taken into account. Maximizing SR implies maximizing the number of jobs for which the following constraints hold:

$$st(j_i) + et(j_i) + cd(u_i, j_i, m) \leq d(j_i)$$
$$st(j_i) + et(j_i) + cd(u_i, j_i, c) \leq d(j_i)$$
$$\forall j_i \in J, \forall m \in M, \forall c \in C, \forall u_i \in U$$

5 PROPOSED ALGORITHM *RT-SANE*

In this section, we discuss the proposed algorithm *RT-SANE* (Real-Time Security Aware Scheduling on the Network Edge). As explained in the last section, there are three categories of security tags assigned to jobs – t_{j1} for private jobs, t_{j2} for semi-private jobs, & t_{j3} for public jobs. Likewise, there are three security tags assigned to resources – t_{r1} for highly trusted resources, t_{r2} for semi-trusted resources, & t_{r3} for untrusted resources. For a particular user $u_i \in U$, we assume that their local *mdc*, m_l is trusted, their local *cdc*, c_l is semi-trusted, & all other *mdc*s and *cdc*s are untrusted. The set of jobs that need to be executed - J, is separated into two different scheduling queues. Queue Q_1 contains only trusted jobs, whereas queue Q_2 contains semi-trusted & un-trusted jobs. We now define *four schedulability conditions.*

- ***MDC deadline condition (C_1)***: $st(j_i)+et(j_i)+cd(u_i, j_i, m) \leq d(j_i)$, this condition is met.
- ***CDC deadline condition (C_2)***: $st(j_i)+et(j_i)+cd(u_i, j_i, c) \leq d(j_i)$, this condition is met.

- ***MDC spare capacity condition (C_3)***: $\forall j_i, \forall m$, if $et(j_i) \leq (cp(m) - \sum et(j_i, m))$, this condition is met.
- ***CDC spare capacity condition (C_4)***: $\forall j_i, \forall c$, if $et(j_i) \leq (cp(c) - \sum(et(j_i, c))$, this condition is met.

Here, m could be a local or remote *mdc*. Similarly, c is a local or remote *cdc*. The rationale for conditions C_1 & C_2 is that the algorithm executes jobs on resources only if the deadlines are met. The rationale for conditions C_3 & C_4 is that it makes sense to execute jobs on the resources, only if the resources have sufficient spare capacity available.

The algorithm *RT-SANE* works as follows. First, the quantities st, ct, cd are calculated for all $j_i \in J$. All the jobs to be scheduled are added to a scheduling queue Q. All jobs in Q are sorted in increasing order of deadlines, so that the job at the head of the queue has the smallest deadline. If multiple jobs have the same deadline, their execution order may be picked randomly. Next, for all private jobs, the scheduler tries to execute them on their local *mdc* m_l, provided the job will finish before the deadline, and provided the local *mdc* has sufficient spare capacity available (conditions C_1 & C_3 are met). If either or both conditions are not met, the job needs to wait and be re-submitted later. In this case, the distributed orchestrator can preempt the execution of the job. For all semi-private and public jobs, the scheduler first tries to execute them on the local *mdc* or on the local *cdc*. If this is not possible, due to the local *cdc* and *mdc* being overloaded, the job is executed on a remote *mdc* or *cdc*. If the deadline condition for the remote *mdc* is met, and the remote *mdc* has sufficient spare capacity available, the job is executed on the remote *mdc*. Otherwise, the job is executed on the remote *cdc*. In order to be scheduled on either an *mdc* or a *cdc*, the four conditions described above have to be met for all *mdc*s & *cdc*s.

Algorithm 1 RT-SANE

```
1:  procedure RT-SANE
2:      Calculate st, ct, cd, ∀j_i ∈ J.
3:      Populate Q with tags t_j1, t_j2 & t_j3 jobs.
4:      Arrange Q in increasing order of deadlines.
5:      ∀j_i with tag t_j1:
6:      if (C1 is met on m_l) && (C3 is met on m_l) then
7:          schedule j_i on its local mdc m_l.
8:      else
9:          re-submit job later.
10:     ∀j_i with tags t_j2 or t_j3:
11:     if (C1 is met on m_l) && (C3 is met on m_l) then
12:         schedule j_i on its local mdc m_l.
13:     if (C2 is met on c_l) && (C4 is met on c_l) then
14:         schedule j_i on its local cdc c_l.
15:     if (C1 is met on m_f) && (C3 is met on m_f) then
16:         schedule j_i on a remote mdc m_f.
17:     if (C2 is met on c_f) && (C4 is met on c_f) then
18:         schedule j_i on a remote cdc c_f.
19:     Calculate DC, UT, ∀c, m ∈ C, M.
```

6 SIMULATION RESULTS & DISCUSSION

The goal of the simulations reported in this section is to evaluate the performance of the proposed algorithm *RT-SANE* using a sample scenario, as described in Figure 1. The proposed algorithm is based on the Distributed Orchestration Architecture & Protocol discussed earlier in section III. The Distributed Orchestration Protocol has three cases: (a) a job is executed on the local *mdc*, (b) a job is executed on the local *cdc*, (c) a job is executed on the remote *cdc* or *mdc*. In terms of the security requirements, the local *mdc* is trusted, the local *cdc* is semi-trusted, and the remote *cdc*s & *mdc*s are untrusted.

6.1 Simulation Setup & Parameters

For the simulation, a scenario based on Figure 1 has been set up. There are 6 users $u_1, ..., u_6$. Each user u_i is running job j_i on device $D_i, \forall i = 1 - 6$. The job execution cost is varied uniformly from 500 – 5500 MIPS/job (this expresses the computational requirement of each job to be scheduled via *RT-SANE*). Users and their jobs are submitted to *mdc*s which can communicate to *cdc*s. Each user u_i has a local *mdc*, a local *cdc*, & remote *mdc*s & *cdc*s. The number of *mdc*s has been fixed at two. The processing capacity of the *mdc*s has been fixed at 1000 MIPS and of the *cdc*s has been fixed at 44800 MIPS.

Each *mdc* & *cdc* has a distributed Orchestration Agent (*OA*) running on it. Each *OA* creates job specific agent instances that collaboratively work together to fulfill the performance and security requirements of the jobs.

The simulations were run on iFogSim [6], which enables us to simulate different characteristics of an *mdc* and *cdc*. iFogSim is particularly relevant for this work as it focuses on the evaluation of resource management policies across fog and cloud environments. iFogSim is centered on the use of a Sense-Process-Actuate model, which makes it relevant for a number of different types of edge devices. All iFogSim simulations have been run on a machine with an Intel Xeon 2.40 GHz Processor and 12GB of RAM. In iFogSim, a class named *MultipleApps* has been created. This represents 6 different independent jobs, each having 1 module. The *MIPS* capacity requirements and job deadlines have been declared in this class. The capacity of *mdc*s and *cdc*s, the communication delay *cd*, and the assignment of modules have also been specified in this class. The function *updateAllocatedMips* (in FogDevice class) is responsible for allocating *MIPS* load for different modules. This has been modified to take into account the deadlines and *FCFS* scheduling methods, in addition to the time shared method already included in iFogSim distribution. The job priority queue stores the modules in non-decreasing order of their deadlines (head-tail) or in *FCFS* order. A function has been created in the same class to check whether a module has finished its execution or not. If it has, then it is removed from the priority queue, so that remaining jobs will get all the available *MIPS* of the *mdc*s & the *cdc*s.

The following parameters have been used in the simulations:

(1) Success Ratio (SR) = $\frac{n'}{n} \times 100$ (as described previously – i.e. percentage of completed tasks vs. those submitted).
(2) Throughput: number of jobs that are able to finish execution in a specified time interval.

(3) *MIPS* Load: This is the *MIPS* requirement of all the jobs. The *MIPS* value of each job was uniformly selected from the range (500 - 5500). The average of the *MIPS* values for all jobs was then calculated. This value was then multiplied by a factor of 1 to 6 to get a range of *MIPS* loads.
(4) Deadline Factor (*DF*): This is the range over which the deadlines of the jobs have been varied. A low value indicates tight deadlines overall in the system & vice versa. A lower bound for the job deadline $d(j_i)$ was calculated, equal to $ect(j_i)$. From this, the average deadline value of the system was calculated. This value was then multiplied by a factor of 1 to 6 to get a range of deadline factor values.
(5) Delay Factor (*DLF*): This is the range for the communication delay between the users, the *mdc* and the *cdc*s. A low value indicates less overall communication delays and vice versa. Initial delay values were 2 milliseconds from user to *mdc* and 100 milliseconds from user to *cdc*. These have then been increased by 10 milliseconds repeatedly to get the *DLF* values.

6.2 Results & Discussion

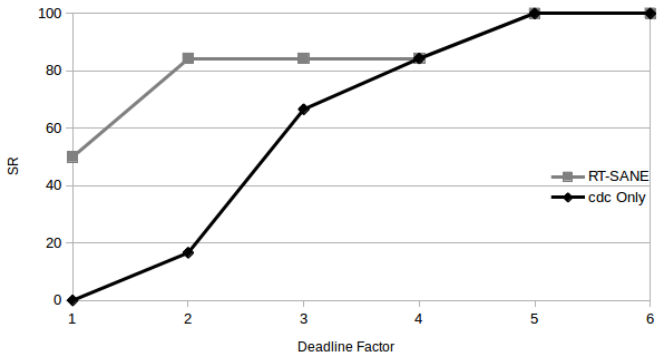

Figure 3: Effect of *DF* on *SR*

6.2.1 Effect of including Edge capability on Performance: The goal of this set of simulations is to show that using *mdc*s as an additional resource for scheduling jobs, besides the cloud, results in a higher Success Ratio. In the first simulation of this section, we compare the effect of increasing the Deadline Factor (*DF*) on the Success Ratio (*SR*) for two algorithms: *RT-SANE* & *cdc – only*. In the *cdc – only* algorithm, all jobs are directly sent to the cloud data center for execution. The overhead of doing this is that all jobs need to undergo significant communication delays. The *OA*s in *RT-SANE*, on the other hand, try to schedule the jobs on the local *mdc* first (case (a)). The number of *mdc*s was fixed at 2. The communication delay between users & the *mdc*s was fixed at 2 milliseconds, & the communication delay between users & the *cdc* was fixed at 100 milliseconds. The Deadline Factor was calculated as described earlier in section VI(a). The results of this simulation are shown in Figure 3. As illustrated, increasing the *DF* value leads to an increase in the *SR* value. This may be explained as follows. Increasing the *DF* value leads to the "loosening up" of the deadlines, i.e. their values become larger. Hence, the jobs are able to meet their deadlines, even after incurring the large communication delays to the cloud. We also observe that for lower *DF* values, the proposed algorithm *RT-SANE* offers a better *SR* than the *cdc – only* algorithm. This is because the *OA* first schedules jobs on the *mdc*s, which have a much smaller communication latency than the cloud. Hence, a larger number of jobs are able to meet their deadlines. After a particular *DF* value, both algorithms offer similar performance, as by now, the deadlines have become so "loose", that one may also schedule jobs only on the cloud, without missing the deadlines.

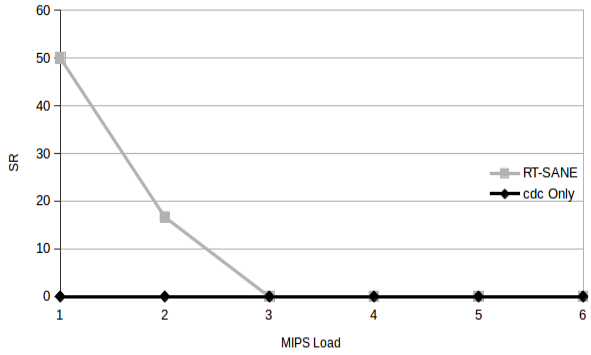

Figure 4: Effect of *MIPS* Load on *SR*

In the second simulation of this section, we study the effect of increasing the *MIPS* load on the *SR*. Again, the number of jobs was fixed at 6, & the number of *mdc*s was fixed at 2. The *MIPS* value of each job was uniformly selected from the range (500 - 5500). The *MIPS* load was calculated as described in section VI(a). The results of this simulation are shown in Figure 4. For low values of *MIPS* load, *RT-SANE* offers high *SR* values. This is because the task specific agents are able to schedule a large number of jobs at the local *mdc* (case (a)). We observe that increasing the *MIPS* load reduces the *SR*, because we are effectively increasing the computation load being placed on the *mdc*s and *cdc*s. As this load increases, initially, the local *mdc*s reach their capacity, so the *OA* needs to create proxy agents to schedule the jobs on the local *cdc*s (case(b)). As the *MIPS* load further increases, the local *cdc* also becomes overloaded, and the *OA* creates proxy agents to execute jobs on remote *cdc*s and *mdc*s (case(c)). In the *cdc – only* scheduling algorithm, none of the jobs are able to meet their deadline, even for lower *MIPS* load values. We attribute this to the tight deadlines that have been assigned to the jobs. If the deadlines are "looser", some jobs may meet their deadlines, even if they are scheduled on the cloud. We observe from this simulation, that for tight deadlines, the proposed algorithm *RT-SANE* offers a much better performance than the *cdc – only* algorithm, especially for lower *MIPS* load values. For high *MIPS* load values, both algorithms show similar performance, as the computation load is much greater than the processing capacity, & this leads to all deadlines being missed, irrespective of the algorithm.

In the third simulation of this section, we study the effect of the Delay Factor (*DLF*) on the *SR*. The communication delay between users & the *mdc*s was fixed at 2 milliseconds, & the communication delay between users & the *cdc* was fixed at 100 milliseconds. From

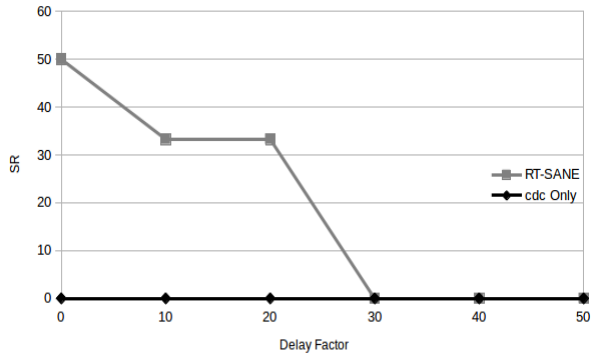

Figure 5: Effect of *DLF* on *SR*

these initial values, the *DLF* was calculated as described in section VI(a). The results of this simulation are shown in Figure 5. We observe that increasing the *DLF* leads to a decrease in the *SR*. This can be explained as follows. As we are adding more delay to the network, more jobs reach their execution destination (*mdc*s & *cdc*s) later. Hence, they end up missing their deadlines. Essentially, as we increase the *DLF*, the number of jobs for whom case (a) is satisfied becomes less. Hence, jobs need to be sent to the *cdc*, & this delay causes the jobs to miss their deadlines. We also observe that *RT-SANE* offers a better performance than the *cdc − only* option, in terms of a higher *SR*. This is because, *RT-SANE* effectively utilizes the *mdc*s for the execution of jobs, in addition to the *cdc*. Beyond a particular *DLF* value, both algorithms offer similar performance, as by now, the delays are so high that even the presence of *mdc*s does not help in meeting job deadlines. For lower *DLF* values, however, *RT-SANE* offers a much better performance.

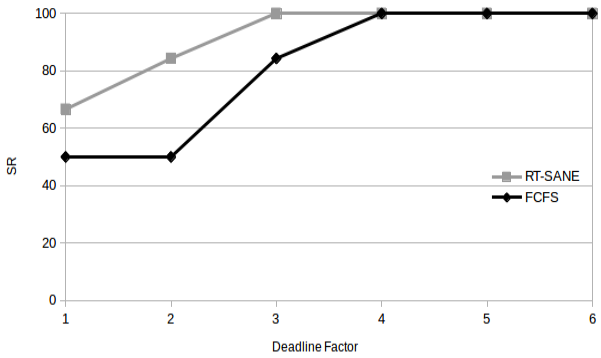

Figure 6: Effect of *DF* on *SR*(Deadline Heuristics)

6.2.2 Effect of Deadline Heuristics on Performance: The goal of this part of the simulation is to show that while scheduling jobs on the network edge (*mdc*s), using an effective heuristic results in better performance, in terms of a higher Success Ratio. In the first simulation, we observe the effect of varying the Deadline Factor *DF* on the performance for both *RT-SANE* & First Come First Serve (*FCFS*) heuristics. The number of jobs was fixed at 6, and the number of *mdc*s was fixed at 2. The communication delay between users and the *mdc*s was fixed at 2 milliseconds, & the communication delay between users and the *cdc* was fixed at 100 milliseconds. The *DF* was calculated as shown in section VI(a). The results of this simulation are shown in Figure 6. We observe that increasing the Deadline Factor (*DF*) leads to an increase in the *SR*. This is because as the *DF* is increasing, deadlines are becoming looser, and the scheduling algorithms are able to ensure that a larger number of jobs can finish before their deadline. We also observe that for lower *DF* values, *RT-SANE* offers higher *SR* values than *FCFS*. This is because it applies the earliest deadline first heuristic, i.e. jobs with earlier deadlines are being executed earlier. Hence, they have a greater chance of finishing before their deadline, versus the case in which jobs are strictly executed first come first serve. Beyond a particular *DF* values, both algorithms show the same *SR* values. By now, the deadlines are so loose that the order of execution of the jobs has no impact. From this simulation, we observe that for lower *DF* values, *RT-SANE* offers a higher *SR*, as it uses an intelligent heuristic for scheduling jobs.

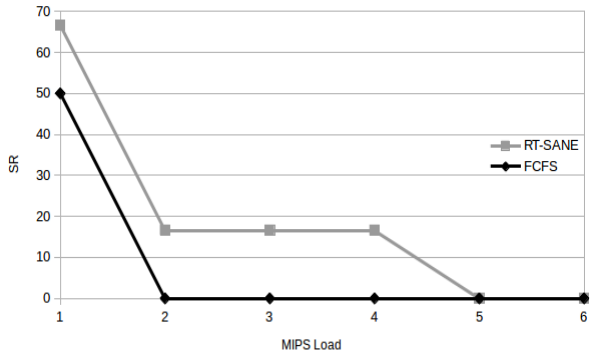

Figure 7: Effect of *MIPS* Load on *SR* (Deadline Heuristics)

In the second simulation, we study the effect of *MIPS* load on *SR*, for both *RT-SANE* & *FCFS*. The *MIPS* value of each job was uniformly selected from the range (500 - 5500). The *MIPS* load was calculated as shown in section VI(a). The results of this simulation are shown in Figure 7. We observe that as we increase the *MIPS* load, the *SR* values decreases. For lower *MIPS* load values, the job specific agents are able to execute the jobs on the local *mdc*, so the *SR* values are high. For higher *MIPS* load values, the *OA* needs to create proxy agents to execute the jobs on the local *cdc*, or on the remote *cdc*s & *mdc*s, so the *SR* values are low. However, the performance of *RT-SANE* is observed to be better than that of *FCFS*, as it is intelligently executing the earlier deadline jobs first, whereas, in *FCFS*, the jobs are scheduled in the order that they arrive. Hence, we observe that for lower MIPS load values, it makes sense to execute jobs using *RT-SANE*, as it offers a higher *SR*. For higher *MIPS* load values, the computation load placed on the system is far greater than the capacity, so the choice of heuristic becomes immaterial.

Next, we study the effect of increasing the Delay Factor *DLF* on *SR*. The communication delay between users & the *mdc*s was fixed at 2 milliseconds, & the communication delay between users

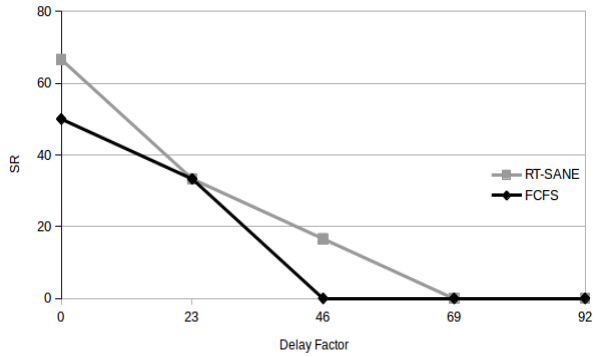

Figure 8: Effect of *DLF* on *SR* (Deadline Heuristics)

& the *cdc* was fixed at 100 milliseconds. From these initial values, the *DLF* was calculated as described in section VI(a). The results for this simulation are shown in Figure 8. Increasing *DLF* reduces the *SR*, as an increasing *DLF* leads to more delay being induced in the network, which results in jobs reaching the *mdc*s later. As *DLF* increases, the number of jobs for which case (b) holds increases, as the local *mdc* is overloaded. Eventually, the number of jobs for which case (c) holds increases, as the local *cdc* is overloaded. This result in more jobs missing their deadlines. However, *RT-SANE* demonstrates a better performance than *FCFS*, as it picks jobs with earlier deadlines to execute first. Hence, a larger number of tasks are able to finish before their deadline.

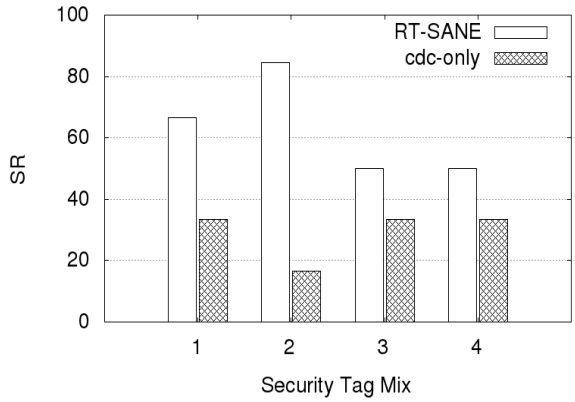

Figure 9: Effect of Security Tag Mix Cases on *SR*

6.2.3 Effect of Security on Performance: The goal of this simulation was to study the effect of security tag assignment on the system performance. Several "security tag mixes" were considered, as shown in table III. As an example, in $case_1$, $1/3^{rd}$ of the jobs were private, $1/3^{rd}$ of the jobs were semi-private, & $1/3^{rd}$ of the jobs were public. We considered 4 different cases, with different mixes for private, semi-private, & public jobs for each case. Note that all private, semi-private, & public jobs are assigned security tags of t_{j1}, t_{j2}, & t_{j3} respectively. The number of jobs was fixed at 6, & the number of *mdc*s was fixed at 2. The communication delay between users & the *mdc*s was fixed at 2 milliseconds, & the communication delay between users & the *cdc* was fixed at 100 milliseconds. The results for this simulation are shown in Figure 9. In general, *RT-SANE* offers a better *SR* than the $cdc-only$ approach, partly because of the earliest deadline heuristic employed in *RT-SANE*, that ensures that a larger number of jobs meet their deadlines, & partly because of the fact that the private jobs cannot be executed on the cloud in the $cdc-only$ approach. When we go from case 1 to case 2, we observe that the *SR* for *RT-SANE* goes up. This is because, the number of semi-private & public jobs has been reduced. So, a lesser number of jobs are going to the cloud for execution. However, the *SR* value for $cdc-only$ reduces. This is because the number of private jobs has now gone up, & such jobs cannot be executed on the *cdc*. We observe from this simulation that the proposed algorithm *RT-SANE* effectively handles the security constraints of jobs, while offering a good *SR*, something which the $cdc-only$ algorithm is unable to do.

Table 3: Various "Security Tag Mixes" Considered

	Frac. (t_{j1})	Frac. (t_{j2})	Frac. (t_{j3})
$Case_1$	1/3	1/3	1/3
$Case_2$	2/3	1/6	1/6
$Case_3$	1/6	1/6	2/3
$Case_4$	1/6	2/3	1/6

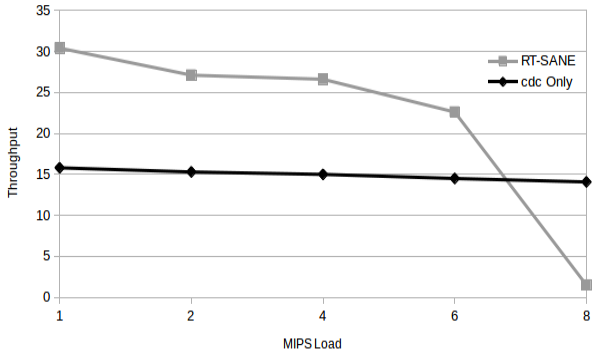

Figure 10: Effect of *MIPS* Load on Throughput

6.2.4 Effect of MIPS Load on Throughput: In the last simulation, we consider the effect of *MIPS* Load on another system metric – throughput. In this case, we assume that the deadlines are very "loose", so the main goal is to maximize the system throughput. A value of 10 milliseconds has been taken as the unit time. The number of jobs was fixed at 6, and the number of *mdc*s was fixed at 2. The communication delay between users & the *mdc*s was fixed at 2 milliseconds, & the communication delay between users and the *cdc* was fixed at 100 milliseconds. The *MIPS* load for jobs was calculated as described in section VI(a). The results for this simulation are shown in Figure 10. We observe that *RT-SANE* offers better throughput values than the $cdc-only$ approach. This is due

to the presence of *mdc*s, which are additional resources that can be allocated to the jobs (case(a)). This results in a larger number of jobs finishing per unit time. We also observe that interestingly, as the *MIPS* load increases, although the throughput values for $cdc - only$ are lower, the degradation is graceful. This is because the *cdc* has a much larger capacity than the *mdc*s. Hence, it can handle an increase in the *MIPS* load. The *mdc*s, on the other hand, due to their limited capacity, cannot handle the *MIPS* load increase that well. However, since the user to *mdc* delay is low, they play a part in *RT-SANE* by offering better system throughput. We observe that the proposed algorithm *RT-SANE* offers a higher throughput than the $cdc - only$ algorithm, especially for low *MIPS* load values.

7 CONCLUSIONS

One of the bottlenecks of processing data on the cloud is the large communication latency to user devices. This latency may be detrimental to real-time applications with stringent deadlines. This is exactly where edge devices can make an impact. As compared to sending all applications to the cloud for execution, scheduling applications with tight deadlines on edge devices can offer better performance, in terms of a higher Success Ratio, and support security concerns associated with moving data to a cloud data centre. The proposed algorithm, *RT-SANE*, is both security & performance aware. Private jobs are sent only to the local *mdc* for execution. Semi-private jobs may be executed at the local *cdc*, and public jobs may be executed on remote *mdc*s or *cdc*s. Subject to specific security constraints, jobs with tight deadlines are scheduled on the *mdc*s. Only if the *mdc*s are overloaded, are the jobs sent to the *cdc* for execution. In order to facilitate this, we have also proposed a Distributed Orchestrator based architecture and protocol. Simulation results (carried out using iFogSim) demonstrate that the proposed algorithm offers superior performance in terms of a higher Success Ratio, by taking into account the multi-tier nature of the Edge/Cloud architecture, while also meeting the security needs of the applications.

As part of future work, we plan to incorporate more complex precedence constrained (workflows) in our job model. Additionally, we are also interested in considering the case of "heterogeneous" *mdc*s & *cdc*s.

ACKNOWLEDGMENT

The work reported in this paper was partially enabled by the European Union's Horizon 2020 research and innovation programme under grant agreement No 643963 (SWITCH Project).

REFERENCES

[1] Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are, Cisco White Paper, 2015.
[2] A. V. Daesterjerdi and R. Buyya, Fog Computing: Helping the Internet of things to realize their potential, IEEE Computer, vol. 49, no. 8, pp. 112-116, 2016.
[3] L. F. Bittencourt, O. Rana, and I. Petri, Cloud Computing at the Edges, Springer International Publishing, 2016, pp. 3-12.
[4] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, Fog Computing: A platform for Internet of things and analytics, Springer International Publishing, 2014, pp. 169-196.
[5] B. Jennings and R. Stadler, Resource management in clouds: survey and research challenges, Journal of Network and Systems Management, vol. 23, no. 3, 2015, pp. 567-619.
[6] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, iFogSim: A toolkit for modeling and simulation of resource management techniques in Internet of things, edge and fog computing environments, CORR abs, vol. 1606.02007, 2016. [Online]. Available: http://arxi.org/abs/1606.02007.
[7] L. F. Bittencourt, E. R. M. Madeira, and N. L. S. Da Fonseca, Scheduling in hybrid clouds, IEEE Communications Magazine, vol. 50, no. 9, 2012, pp. 42-47.
[8] J. D. Montes, M. Abdelbaky, M. Zhou, and M. Parashar, Cometcloud: Enabling software-defined federations for end-to-end application work-flowsâĂİ, IEEE Internet Computing, vol. 19, no. 1, 2015, pp. 69-73.
[9] C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM, vol. 20, no. 1, 1973, pp. 46-61.
[10] Z. Guo and S. Baruah, A neurodynamic approach for real-time scheduling via maximizing piecewise linear utility, IEEE Transactions on Neural Networks and Learning Systems, vol. 27, no. 2, 2016, pp. 238-248.
[11] J. Singh, S. Betha, B. Mangipudi, N. Auluck, Contention aware energy efficient scheduling on heterogeneous multiprocessorsâĂİ, IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 5, 2014, pp. 1251-1264.
[12] B. A. Hridita, M. Irfan & M. S. Islam, Mobility aware task allocation for mobile cloud computing, International Journal of Computer Applications, Vol. 137, No. 9, 2016 pp. 35-41.
[13] X. D. Pham & E. N. Huh, Towards task scheduling in a cloud fog computing system. The 18^{th} Asia-Pacific Network Operations and Management Symposium, APNOMS, Kanazawa, Japan, October 5-7, 2106, pp. 1-4.
[14] M. Shojafar, N. Cordeschi and E. Baccarelli, âĂIJEnergy-efficient adaptive resource management for real-time vehicular cloud servicesâĂİ, IEEE Transactions on Cloud Computing, preprint, April 6, 2016, pp. 1-14.
[15] L. F. Bittencourt, M. M. Lopes, I. Petri and O. Rana, Towards virtual machine migration in fog computing, The 10^{th} International conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), November 4-6, 2015, Krakow, Poland, pp. 1-8.
[16] F. Xia, L. T. Yang, L. Wang and A. Vinel, Internet of Things, Editorial, International Journal of Communication Systems, Vol. 25, 2012, pp. 1101-1102.
[17] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh and R. Buyya, Fog computing: principles, architectures, and applications, Internet of Things: Principles and Paradigms, R. Buyya and A. Dastjerdi (eds), Morgan Kaufmann, ISBN: 978-0-12-805395-9, Burlington, Massachusetts, USA, May 2016.
[18] J. W. S. Liu, âĂIJReal-Time SystemsâĂİ, Prentice Hall, April 2000, 592 pages.
[19] S. Sharif, P. Watson, J. Taheri, S. Nepal, and A. Zomaya, Privacy-aware scheduling SaaS in high performance computing environments, IEEE Transactions on Parallel & Distributed Systems, vol. 28, no. 4, April 2017, pp. 1176-1188.
[20] W. Shu, J. Cao, Q. Zhang, Y. Li, and L. Xu, Edge computing: vision and challenges, IEEE Internet of Things Journal, vol. 3, no. 5, October, 2016, pp. 637-646.
[21] M. Satyanarayanan, G. Lewis, E. J. Morris, S. Simanta, J. Boleng, K. Ha, The role of cloudlets in hostile environments, IEEE Pervasive Computing, October, 2013, pp. 40-49.
[22] M. Satyanarayanan, Augmenting cognition, IEEE Pervasive Computing, April, 2004, pp. 4-5.
[23] S. Shekhar, A. D. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale, INDICES: Exploiting edge resources for performance aware cloud hosted services, First IEEE/ACM International Conference on Fog & Edge Computing, Madrid, Spain, May 14, 2017, pp. 75-80.
[24] M. I. Naas, P. R. P. Orange, J. Boukhobza, L. Lemarchand, iFogStor: an IoT data placement strategy for fog infrastructure, First IEEE/ACM International Conference on Fog & Edge Computing, Madrid, Spain, May 14, 2017, pp. 97-104.
[25] D. Evans, The Internet of Things, how the next evolution of the Internet is changing everything, Cisco White Paper, April 2011.