An Inter-Cloud Meta-Scheduling (ICMS) Simulation Framework: Architecture and Evaluation

Stelios Sotiriadis, Nik Bessis, Ashiq Anjum, and Rajkumar Buyya

Abstract—Inter-cloud is an approach that facilitates scalable resource provisioning across multiple cloud infrastructures. In this paper, we focus on the performance optimization of Infrastructure as a Service (IaaS) using the meta-scheduling paradigm to achieve an improved job scheduling across multiple clouds. We propose a novel inter-cloud job scheduling framework and implement policies to optimize performance of participating clouds. The framework, named as Inter-Cloud Meta-Scheduling (ICMS), is based on a novel message exchange mechanism to allow optimization of job scheduling metrics. The resulting system offers improved flexibility, robustness and decentralization. We implemented a toolkit named "Simulating the Inter-Cloud" (SimIC) to perform the design and implementation of different inter-cloud entities and policies in the ICMS framework. An experimental analysis is produced for job executions in inter-cloud and a performance is presented for a number of parameters such as job execution, makespan, and turnaround times. The results highlight that the overall performance of individual clouds for selected parameters and configuration is improved when these are brought together under the proposed ICMS framework.

Index Terms—Cloud computing, interoperable clouds, inter-clouds, meta-scheduling systems

1 Introduction

The concept behind cloud computing is to provide an on demand scalable and agile infrastructure. Its biggest advantage is the service elasticity that offers scaling of the cloud resources based on user demand [7]. In this work, we focus on inter-cloud (IC) that is an infrastructure that exploits communication across multiple clouds to support diverse and large number of user requests. Inter-cloud aims to increase cloud service elasticity and scalability while minimizing the operational costs. It allows the formation of a collaborative partnership for service exchange under a mutually agreed management while ensuring a certain level of Quality of Service (QoS). In particular, an inter-cloud facilitates communication by acting as a gateway and broker between different cloud providers. In this work we propose an inter-cloud framework that optimises the performance of an infrastructure that may comprise of multiple clouds.

In order to realize it, meta-scheduling may play an important role in the way resources are managed and requests are processed [1]. Specifically, a meta-scheduler could select available resources from multiple clouds taking into account appropriate Service Level Agreements (SLAs), operating conditions (e.g. cost, availability) and performance criteria [4]. This requires resources from multiple clouds to be orchestrated in such a way that tasks are efficiently executed. Our goal is to gain advantage of already developed solutions for large-scale meta-scheduling approaches and implement an Inter-Cloud Meta-Scheduling (ICMS) framework that can improve performance metrics including task execution times, latencies and makespan times by exploiting resources from multiple clouds.

The work is motivated from the future of Internet computing as described in [6]. Specifically, the authors note that today there are different cloud providers that address different needs and may offer different functionality, yet they share the same characteristics in terms of how resources are being provisioned and consumed. These clouds share similarities in structure and architecture. Inter-cloud models should allow tasks to be exchanged in order to achieve better QoS levels by exploiting the resources from a number of cloud providers by employing novel meta-scheduling approaches. In this work we address the limitation in the current cloud implementations that they do not offer support for task federation.

In contrast to other efforts, as described in [2] and [4], we propose a more inclusive design that provides task federation through a decentralized meta-scheduling solution. Each cloud infrastructure may have their own local scheduler which may not have information about resources in other clouds. This work extends the initial effort in [15] by presenting the complete architecture along with new algorithms and the messaging model of ICMS. Further, the experimental study demonstrates an extended use of performance metrics based on new algorithms and use cases

- *S. Sotiriadis is with the Technical University of Crete (TUC), Chania, Crete, Greece. E-mail: s.sotiriadis@intelligence.tuc.gr.*
- *N. Bessis is with the Department of Computer Science, Edge Hill University, United Kingdom. E-mail: Nik.Bessis@edgehill.ac.uk.*
- *A. Anjum is with the University of Derby, Kedleston Road, Derby DE22 1GB, United Kingdom. E-mail: a.anjum@derby.ac.uk.*
- *R. Buyya is with the University of Melbourne, Australia. E-mail: rbuyya@unimelb.edu.au.*

Manuscript received 23 July 2014; revised 27 Dec. 2014; accepted 25 Jan. 2015. Date of publication 3 Feb. 2015; date of current version 2 Feb. 2018.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2015.2399312

for evaluation purposes. Also, the new algorithms and performance evaluation experiments have been produced in the SimIC [16] that realizes inter-cloud algorithms along with the messaging model. Its architecture is based on CloudSim [21] yet it implements and extends its features from the perspective of processing batch jobs in meta-scheduling systems. Having said that, the paper is organized as follows, Section 2 presents a discussion of related works. Section 3 details the proposed ICMS framework and Section 4 presents the SimIC simulation toolkit and the experimental configuration. Section 5 details the performance results and evaluation. The work concludes in Section 6 with a summary and a discussion of the future directions.

2 Related Works

The inter-cloud has been characterized as a large-scale resource management system comprising of multiple autonomous clouds [5]. These independently managed clouds may be homogenous or heterogeneous, yet in an inter-cloud infrastructure they will need to function under a single federated management entity. This section focus on a literature review of the meta-scheduling approaches developed for large-scale systems that may exhibit similar characteristics to inter-clouds. In detail, we focus on the algorithms with regards to inter-clouds.

The work of [8] presents a decentralized dynamic algorithm named Estimated Load Information Scheduling Algorithm (ELISA). The algorithm estimates the queue length of neighboring processors and then reschedules the loads based on estimates. The method aims to increase the possibilities to gain load balancing by estimation based on updated information after large time intervals. Yet, the method is not adaptable to inter-cloud as the algorithm requires lengths of queues of neighboring hosts; consequently it exposes internal information. In [9] authors demonstrate a distributed computing scheduling model. The key idea of the proposed meta-scheduler is to redundantly distribute each service to multiple sites, instead of sending the service to the most lightly loaded. We envision that inter-clouds will mainly be used for highly loaded scenarios; therefore this method will decrease the overall performance.

The work of [10] presents a model for connecting various Condor work pools that yield to a self-organizing flock of Condors. The model uses the Condor resource manager to schedule services to idle resources. This method, similar to [8], includes comparison of queues, so makes local information to be exposed and it is considered not adoptable to inter-clouds. The authors conclude that it performs better for lightly loaded sites and thus as in [9] this will also decrease the overall performance. Authors in [11] present a scheduling infrastructure called OurGrid which is based on the Bag-Of-Tasks applications. OurGrid is a collection of peers constituting a community. This is a decentralized solution based on site reputation and debts. As debts grow services could become less prioritized, thus could lead to starvation, which in turn could affect inter-cloud performance. In [12] authors discuss a market-based resource allocation system. The scheduling mechanism in this system is based on auctions. Specifically, each resource provider or owner runs an auction for their resources. However, this does not guarantee an optimized inter-cloud solution as resources can be under-utilized due to meta-schedulers that might bid always for a specific set of resources.

In [29], authors describe two scheduling algorithms, namely Modified ELISA (MELISA) based on [8] and load balancing on arrival. Both algorithms are based on the distributed scheme of sender-initiated load balancing. To improve MELISA performance, the authors conclude that the load balancing on arrival method will balance the high service arrival rates. However, this solution includes exchanging of local queues as discussed in [8], thus it is inefficient with regards to inter-clouds. The delegated matchmaking (DMM) approach presented by [13] is a novel delegated technique, which allows the interconnection of several grids without requiring the operation under a central control point. Their simulation results show that DMM can have significant performance and administrative advantages. However, this work raises heterogeneity issues in large-scale distributed settings.

In [17] authors present a model for an InterGrid system that enables distributed resource provisioning from local to global scale. In [18], authors evaluate the performance analysis of the InterGrid architecture by using various algorithms e.g. conservative backfilling. The results show that the average response time has improved in the aforementioned evaluated scheduling algorithms. Yet, [19] suggest that the approach reflect an economical view as business application is the primary goal. In [19], authors present a decentralized model for addressing scheduling issues in federated grids. This solution proposes the utilization of GridWay, as a meta-scheduler to each grid infrastructure. The authors assume a complete setting in terms of meta-brokers knowledge for each other, thus makes it appropriate for small-scale settings and not for large-scale inter-clouds.

In [20] is presented the problem of broker selection in multiple grid scenarios by describing and evaluating several scheduling techniques. In particular, system entities such as meta-brokers are represented as gateways. Authors claim that performance is not penalized significantly; however resource information accuracy may be lost. This work did not address these meta-scheduling features in inter-clouds. The work of [23] introduces a decentralized dynamic scheduling approach called community aware scheduling algorithm (CASA). The CASA that based on [28] contains a collection of sub-algorithms to facilitate service scheduling across distributed nodes. The message distribution is based on the probability to find a resource, thus requires training of the system to define probabilities. In this study, ICMS defines algorithms for dynamic scheduling that goes beyond exchanging local scheduling queues. Finally, in [3] authors present a scalable cloud system modeled around the Amazon EC2 architecture, with a workload model that offers fluctuating traffic characteristics. Table 1 shows a summary of large-scale scheduling approaches by extending the work performed in [22]. It should be mentioned that in [6] authors present a detailed theoretical comparison among these approaches.

An important characteristic of our approach is the message exchanging feature that is considered as a key requirement by most of the decentralized approaches, as reported in [9], [10], [11], [12], and [19]. However, most of these

TABLE 1
Summary of Large-Scale Scheduling Approaches

Approach	Advantages	Disadvantages
Works of [8] and [29] demonstrate ELISA and MELISA that calculate the neighboring nodes load by considering job arrival rate and node loads. Jobs are transferred based on the comparison of nodes load and not queue length.	Distributed algorithm based on the centre-initiated load balancing. MELISA performs better in large scale systems compared to MELISA.	Adaptability for dynamics cannot be guaranteed and privacy issues exposed due to queues exchanging (comparison of nodes load), and virtualization capabilities are not included.
In [9], the scheduler redundantly distribute each job to multiple sites, instead of sending the job to the most lightly load though backfilling.	Increases the possibility of effective backfilling and brings better fairness.	Performs best for low loaded sites, lower overall performance for large-scale systems and no virtualization capability.
In [10], the approach connects various Condor pools which yield to a self-organizing flock of condors. It schedules jobs to idle resources by using Condor resource manager and invokes flocking mechanism only for busy machines.	It uses the Condor resource manager for scheduling to idle resources and flocks can reduce the maximum job waiting time in the queue.	Pools are characterized to suitable/not suitable; as a result unfairness will lead to starvation, also comparison of queue lengths exposes privacy issues and virtualization is not determined.
In [11], scheduling executed by site reputation and resource availability, and brokers schedule jobs through arrangements and priorities to peers where each peer can maintain ranking of all known peers.	Total decentralized solution where peers keep track of local balance for each known peer based on past interactions.	As debts grow, jobs become less prioritized, thus solution could lead to starvation. Also resources can be under-utilized due to meta-schedulers bidding for specific resources.
In [13], the work temporarily binds local resources to remote resources, when a user cannot be satisfied at the local level, through delegated matchmaking (DMM). Remote resources are added transparently.	Improved performance by reducing administrative overhead, also no local operation of central control point.	Dynamics of the system are ignored as a steady state is assumed during simulation. Also, heterogeneity and virtualization issues are not fully considered.
In [18], the target is InterGrid infrastructure where authors interlink grid islands using peering arrangements and gateways to allow a cross collaboration among various grids.	It evaluates the performance of four complex algorithms and shows an improvement in average response times.	The system dynamics may affect connections of grid islands (e.g. failures could happen during communication) and also brokers are self-interested and not global.
In [19], a meta-scheduler called GridWay sits on top of each grid infrastructure on the federated grid. Four algorithms have been developed and can be executed in the GridWay.	No requirements for information of remotes nodes and it consider past performance requirements to forecasts new objectives.	Only adoptable for specific information system as requires training mechanism for forecasting performance, also overhead during training may be increased.
In [20], a meta-broker selection process is shown for multiple grid interoperating cases. The scheduling policy consists of the bestBroker-Rank policy.	Improves workloads and resource utilization as well as load balancing among different grids.	The method assumes complete and detailed resource information sharing in a stable infrastructure.
In [23] shows a dynamic scheduling approach called CASA which functions as a scheduling decision to job schedule across decentralized distributed nodes.	Could lead to the same amount of executed jobs in centralized as in decentralized.	Job distribution is based on a probability to find a resource, thus requires training of the system to define probabilities.

approaches do not detail the whole request-response procedure. For example, [24] suggests that messages are exchanged among components in order to make cooperative scheduling decisions. Since the rejected responses are returned an increased message overhead is observed. Similarly, [19] suggests an algorithm that allows rejected messages to return in the case a grid does not have the required slots for allocation. Authors in [10] suggest that a node that receives a message becomes aware of available resources in the pool. This includes messages that are exchanged in all the resources available in the resource pool. In contrast, [11] considers a broadcasting approach where a resource does not always require to reply back. However, majority of the current performance optimization approaches overlook the benefits that may derive from a more fine-grained message exchanging approach. A more detailed discussion of the message exchanging mechanisms in distributed systems is presented in [14].

3 The Inter-Cloud Meta-Scheduling (ICMS) Framework

The ICMS is the means to represent the inter-cloud service distribution that allows the integration of modular policies. The ICMS is organized in a layered structure as detailed in Fig. 1. The primary functionalities are divided in three layers namely, the service submission, the distributed

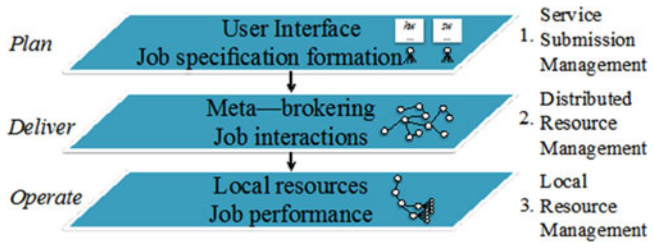

Fig. 1 The three-layered structure of ICMS.

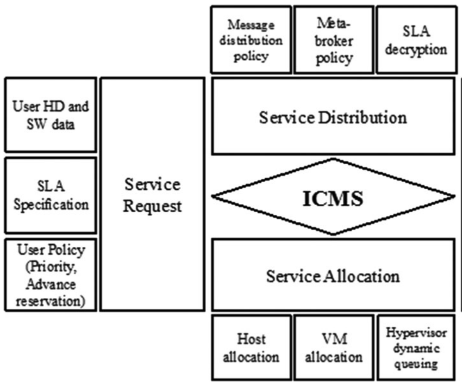

Fig. 2. The ICMS modular structure.

resource and the local resource management layers. In layer 1, a pre-defined topology includes users that forward requests to layer 2. The latter includes a random topology based on random interconnections of distributed meta-brokers (represented as nodes) to exchange services. The service distribution is based on messages that are exchanged among meta-brokers. The ICMS supports a dynamic workload management to allow decision-making for services distribution on the meta-brokering level as detailed in [15]. As indicated earlier, we focus on IaaS so each service encompasses a request for a Virtual Machine (VM) with regards to the computational power and other related parameters (i.e., number of CPUs, CPU cores, memory, storage, and bandwidth). Each message includes a description of such information.

Layer 3 contains a topology that involves the formation of low-level infrastructure and its entities such as local-brokers, data stores, hosts and VMs. It should be mentioned that our assumption is that clouds follow a standard setting (e.g. follow the open cloud computing interfaces) that includes a local resource broker that controls interactions with the datacenter hypervisor that in turn sandboxes it to a VM. Policies for scheduling and local resource management are implemented in the local resource broker. As shown in Fig. 1, the layers include the key elements of the service life cycle that are to plan, deliver and operate.

Layer 1, the service submission management layer, is responsible to create the service configuration by translating user requirements to system specification. The output is in a form that is recognized by the inter-cloud entities. Layer 2, the distributed resource management layer, collects service submissions and descriptions, extracts information regarding performance criteria (e.g. service size) and forwards it to the appropriate execution entity. This entity could be either a local resource queue or a remote meta-broker that further distributes the service to interconnected brokers. Layer 3, the local resource management layer, offers the service execution environment. Here, services are forwarded to the lowest level of the infrastructure (local resource management system—LRMS) and sandboxed in VMs. Prior to this, each service is queued into the LRMS queue where a scheduling algorithm allocates services to resources depending on the configuration of the scheduler (e.g. first come first service, shortest service first etc.). The whole ICMS is based on a group of modular policies and each of which realizes the layered structure and the dynamic requirements.

Fig. 2 illustrates the configuration of the four modules of the ICMS conceptual architecture, namely Service Request, Service Distribution, Service Availability and Service Allocation. First, the "Service Request" module includes the user specification and the service formation process. Each service request is recorded into a service level agreement representation. SLAs describe service requirements e.g. service CPU etc. along with a user policy for priorities or advance reservation mechanisms for prioritized users. The "Service Distribution" module contains the message distribution, the meta-brokering and the SLA policy as in layer 2.

In addition, the module incorporates a mechanism for interpreting and translating the content of the SLA. The "Service Availability" module contains the SLA matchmaking, dynamic workload and local resource policies as in layer 3. This includes that each local-broker could define the internal resource usage (by evaluating current executions) in order to decide whether this is capable to execute the service locally. Finally, the "Service Allocation" module includes the hypervisor scheduler, the host allocation and VM allocation policies as in layer 4. The hypervisor is responsible for a) the sharing of host's computational power between the VMs (host scheduling), b) the sharing of VM allocation of computational units (VM allocation) and c) the management of the hypervisor that queues the services in hosts.

The communication between the ICMS modules is achieved by utilizing a novel message exchanging procedure that allows services to be exchanged as events that are sent and received between meta-brokers by following the Message Exchanging Optimization (MEO) model [14], [26]. The assumption here is that we have a decentralized topology of meta-brokers to allow event request-response during regular time intervals. The following steps demonstrate that process.

1) The service distribution starts when a number of services are submitted to a meta-broker. Each service request contains a set of requirements such as time intervals (e.g. waiting time, interval etc.) and computational units (CPU, memory, bandwidth, etc.). In addition, each service request includes priorities and advance reservation features for allowing specific services to be executed on specific types of resources.
2) Each service request is stored in a list. Each list row has a message with key characteristics including the deadline and the service length as the mean for calculating resource availability on remote resources. The service requests are dispatched during regular intervals.
3) The service requester defines the interval deadline, which defines a delay limit and the size of the list. For large lists the deadline could be higher as the time needed to dispatch is higher. This also considers the cost of communication among entities. So a small deadline results in a small number of submissions, while a large one could lead to heavy submissions. The ICMS default interval is configurable to meet the needs of an experiment. Further details are

provided in the experimental analysis section where a detailed discussion of ICMS configuration is shown.

4) The service requester collects addresses of interconnected nodes from an internal catalogue of resources. These nodes are meta-brokers that are used to receive the requests dispatched from the service lists.
5) The service requester sends a service request as a message consisting of quality of service requirements (e.g. deadline, service length etc.). The message includes the ranking criteria (e.g. turnaround, energy consumption level); so all brokers will use the same resource selection criteria. It should be noted that identification tags define a message. During communication, the tags are set to unique values to characterize the group of messages.
6) A broker collects a single service request and performs an internal resource availability check according to the ranking criterion. Then it generates a priority of services which is stored in a temporary ranking list. If the list is empty, the broker cannot execute the services and it will not respond back. In any other case, the list with the ranked services is forwarded back to service requester.
7) Each service request is ranked based on a scheduling function and a decision is taken accordingly.
8) In case of service availability (each service of the list can be executed locally) the broker generates a list with services.
9) In case of non-availability (e.g. broker cannot execute all or few of the services contained in the list) a further service distribution request will be re-performed using steps 1 to 6.
10) In case of complete non-availability the broker will cease communication and therefore, responses are not sent back.
11) A new list consisting of service requests which is ranked in a descending order is created. This forms the criteria for selecting services at the next resource management level. In the case that the broker acknowledges that the service request(s) will be executed on a remote machine, the broker re-directs messages to interconnected nodes. All messages are assigned with updated time deadlines.
12) The ranked lists are collected from the service requester that compares and decides whether a remote resource will be selected for execution or not.
13) The procedure ends and each service request is sent to a local or remote resource.

This concludes the steps of communication, in the next section we focus on the definition of the service submission and service execution features.

3.1 The Definition of Service Submission in ICMS

Let assume that there are M meta-brokers that form a decentralized inter-cloud where $M \in \{m_1, m_2, \ldots, m_n\}$. Each meta-broker does not have a value but is associated with the name of interacted cloud. For instance cloud 1 has a meta-broker named as meta-broker$_1$. The number n equals the number of participating clouds in an inter-cloud, thus each cloud has at least one meta-broker. Each service request is defined as j_i and is assigned to a meta-broker m_i and contains a number of physical characteristics named as CPU cpu_i, memory mem_i, cores cor_i, storage $stor_i$ and bandwidth bw_i. Each j_i is a set of tuples where each request encapsulates a ($j_i = cpu_i$, cor_i, mem_i, $stor_i$, bw_i). It should be mentioned that a service request is an IaaS encapsulation and it is defined in a similar manner to the Amazon EC2 service specification [25]. The cpu_i and the cor_i define the clock rate as $ClockRate_i = cpu_i * cor_i$. It includes also the cycles per instructions for each service named as cpi_{j_i} to calculate the required execution time. This will help us to quantify the service size in terms of traditional jobs length. Further to this, the meta-broker defines a metric to characterize each submission, e.g. the cycles per instruction (CPI) and the execution time of the j_i. The $cpi_{j_i}$is defined as $cpi_{j_i} = cycles_{j_i}/instructions_{j_i}$[6]. The execution time $exec_{j_i}$ is calculated as $exec_{j_i} = instructions_{j_i} * cpi_{j_i}/ClockRate_{j_i} * 10^5$. In this paper, we also define the millions of instruction per second ($mips_{j_i}$) to describe an additional service length metric calculated as $mips_{j_i} = ClockRate_{j_i}/cpi_{j_i} * 10^{-6}$. Both cpi_{j_i} and $mips_{j_i}$define the service size with regards to the specified user submission. Each meta-broker m_i is assigned with a latency lat_{m_i}that defines the delay of the broker to execute a service request including the time needed for coordination and internal communication.

The total service execution time is the sum of the latencies of the meta-broker m_i to the execution time, $TotalJobExecTime_{j_i} = Latency_{j_i} + exec_{j_i}$. The latency of the m_i is $Latency_{m_i} = Latency_{m_i} + ComLatency_{m_i}$. The $ComLatency_{m_i}$ defines the time needed to communicate with the local resource to extract addresses for further distribution. Each meta-broker m_i has a load of services and these are described as the throughput, where $Troughpup_{m_i}$ is equal to the count of m$_i$ in the inter-cloud.

The ICMS calculates the utilization of m_i, e.g. the usage levels, $Utilization = (Troughput_{m_i}/Troughput_{j_i})$. For example, the utilization of the meta-broker m_i is the division of the throughput of the served jobs to the total throughput of the jobs that could be served. Finally, the service performance is described as the execution time of the VM that sandboxes the service and is calculated as $Performance_{j_i} = Performance_{VMj_i} = 1/exec_{j_i}$.

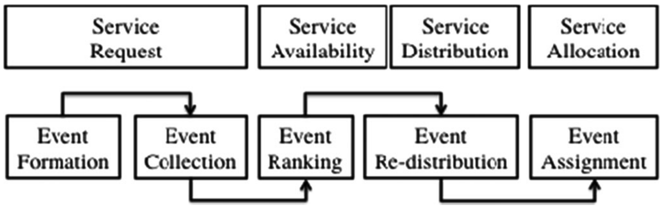

Fig. 3. The sequence diagram of the algorithmic model.

3.2 The Algorithmic Structure

Our approach includes request and response entities to implement the whole set of service execution life cycle. Fig. 3 shows the relationships of the algorithms. It should be mentioned that events are the steps that happen in the life of cloud service requests. The process starts with the event formation and collection algorithms.

Each event algorithm (along service configurations) is then ranked and distributed in the inter-cloud. The event assignment algorithm allocates the service computational units. The structure and the relationship of the algorithms follow the sequence below. First, the event collection, formation and sending procedure of the request entity takes place. This is followed by the event gathering, identification of specification, ranking and response procedure of a broker. Finally the event redistribution procedure and the event collection, ranking and assignment processes take place.

3.2.1 The Event Formation Algorithm

The process starts with the event-formation algorithm. The ICMS sets a time interval $\text{int}_{collection}$ and a criterion for the events to be ranked at a later stage. The assumption is that the meta-brokers have the same uptime and are interlinked in a decentralized topology. Users submit service requests to one or many meta-brokers in an inter-cloud. During a time interval int_α for a submission α where $\text{int}_\alpha \leq \text{in}_{collection}$ the meta-broker collects all the events including characteristics such as cpu_i, mem_i, cor_i, $stor_i$, bw_i, cpi_{j_i}, $ClockRate_i$. For all j_{m_i} it creates a list L_i where each row contains the characteristics of the service request. The algorithm then sets a tag value represented as $t_i \in \Re$ and creates the message including the L_i, t_i and a performance criterion. The default ICMS configuration includes the total service execution time given $TotalJobExecTime_{j_i} = Latency_{j_i} + exec_{j_i}$, the turnaround time $TurnTime_{cloud} = (instructions_{j_i} * cpi_{j_i}/ClockRate_{j_i} * 10^5) + Comlatency_{m_n}$ and the makespan $Makespan_{j_i} = exec_{j_i} + Latency_{m_n}$. The makespan defines the time from start to finish, and the turnaroundtime is the total execution time of the schedule.

The algorithm opens the profile of the entity $prof_i$ for $\forall m_n \in prof_i$ it sends a message to the dedicated address. It sets an interval int_{dist} that is the distribution interval time. For a time $time_i \in \Re$ where $time_i \leq \text{int}_{dist}$ compares the tag t_i for validation and collect responses by a classification function. The latter is defined by the performance criterion of the previous step. As soon as the classification event concludes the algorithm updates the list L_i and sends back an msg_{m_i} only if $sizeL_i \neq 0$. Algorithm 1 demonstrates the service formation algorithm. The operations are defined as follows: get for the collection procedure of service data, set as the operation to set the required service specification, create for the operation to create a list, open as the operation to open a profile, size as the method to return the size value of the profile, send as the method to send a message to an address defined as ad, run as the method to run an algorithm, wait as the method to wait for an interval to expire and update as the method to update a list.

3.2.2 The Event Collection Algorithm

This algorithm configures an interval value for collecting events from the source (e.g. users) and creates a list using the incoming service request specification. Initially, the algorithm sets a termination and redistribution flag (f_{trm}, f_{red}) to recognize whether this is the termination point or the redistribution. For all $\forall msg_{m_i}$ and $t_i \in \Re$ identifiable, it decomposes the message msg_{m_i} and collects the list L_i by running the performance criterion classification function that updates the list L_i. If $sizeL_i > 0$, then L_i compares the intervals of the service requester and responder meta-broker. If $\text{int}_{res} < \text{int}_{req}$ then it sets the tag to an indicator for returning messages (to perform validation).

Algorithm 1. Event Formation

Require:	res	the requesting resource
	$\text{interval}_{\text{collection}}$:	the interval time to collect service messages
	time:	the current time instance
	i	the service submitted by a source
	clocks_i	the service required clocks
	CPI_i	the service required CPI
	cores_i	the service required cores
	bw_i	the service required bandwidth
	h_i	the service required duration
	L_i	the list with the service specification data
	tag	the tag value of the message (e.g. q)
	msg	the message contains the L_i and the tag
	f_i	the profile of the entity i
	ad	the address of a node included in the f_i
	e	the tag value for incoming messages
	$\text{interval}_{\text{distribution}}$	the interval time to collect distribution messages
	response	the notification of the responder
	criterion	the performance ranking criterion defined by the entity i
Algorithms:	Ranking algorithm	the event ranking algorithm that accepts the criterion as an input value for service classification
	Assignment algorithm	the assignment algorithm that accepts the L_i as input value to determine the next phase of resource allocation

```
1.  set $\text{interval}_{\text{collection}}$, criterion
2.  while time < $\text{interval}_{\text{collection}}$ wait
3.    for all i
4.      get($\text{clocks}_\text{i}$, $\text{CPI}_\text{i}$, $\text{cores}_\text{i}$, $\text{bw}_\text{i}$, $\text{h}_\text{i}$)
5.      set i[$\text{clocks}_\text{i}$, $\text{CPI}_\text{i}$, $\text{cores}_\text{i}$, $\text{bw}_\text{i}$, $\text{h}_\text{i}$]
6.      create($\text{L}_\text{i}$[i])
7.    end for
8.  set tag ← q
9.  create(msg[$\text{L}_\text{i}$,tag, criterion]
10. open($\text{f}_\text{i}$)
11. for all $\text{f}_\text{i}$.size()
12.   ad ← get($\text{f}_\text{i}$[k])
13.   send(msg, ad)
14. end for
15. set $\text{interval}_{\text{distribution}}$
16. if time < $\text{interval}_{\text{distribution}}$ and
17.   if tag = e then
18.     get(response)
19.     run(Ranking algorithm(criterion))
20.     update($\text{L}_\text{i}$[i])
21.   end if22. for all $\text{L}_\text{i}$.size()
22.   run(Assignment algorithm($\text{L}_\text{i}$))
23. end for
```

Algorithm 2. Event Collection

Require:	i	the requesting node
	i′	the responding node
	msg_i	the incoming message from requester i
	flag	the flag variable
	trm	the termination flag
	rds	the redistribution flag
	q	the tag indication for incoming message from requester
	w	the tag indication for incoming message from redistributor
	$int_{responder}$	the interval of the responder
	$int_{requester}$	the interval of the requester
	e	the tag indication for returning messages
Algorithms:	Ranking algorithm	the ranking algorithm that accepts the criterion as value
	Redis-tribution algorithm	the redistribution algorithm

```
1.  set flag ← {trm, rds}
2.  for all $msg_i$ and (tag = q or tag = w)
3.    decompose ($msg_i$)
4.    get $L_i$
5.    run(Ranking_algorithm(criterion))
6.    update($L_i$)
7.    if $L_i$.size > 0 then
8.    if $int_{responder} < int_{requester}$ then
9.      set tag ← e
10.     ad ← i
11.     send(msg, ad)
12.    end if
13.  else
14.    if $f_i$.size = 0 then
15.      flag = trm
16.    else then
17.      flag = rds
18.    end if
19.    case: flag = trm
20.      terminate($msg_i$)
21.      destroy($L_i$)
22.    case: flag = rds
23.      open($f_{i'}$)
24.      for all $f_i$.size()
25.          run(Redistribution algorithm($f_{i'}$))
26.        end for
27.      end case
28.    end if
29.  end for
```

The broker sends the event back to the service requester by sending msg_{m_i} that includes the newly formedL_i. In the case of $size f_i = 0$ it sets the f_{trm} flag on, else it sets it to f_{red} flag off. Specifically, for the first case the algorithm terminates the L_i, while for the second case it opens the local profile $prof_i$ and runs the redistribution algorithm in order to find a new resource for service execution. This allows a decentralized behaviour of the ICMS as we assume that there are multiple levels of interconnected meta-brokers. In addition it characterizes the responder meta-broker either as a termination point or as an intermediate node on communication. Algorithm 2 demonstrates the event collection procedure. The operations are defined as follows: "decompose" for a message decomposition operation, "get" for the collection procedure of service data, "rank" for the ranking procedure, "set" as the operation to set the required service specification, "update" as the method to update a list, "size" as the method to return the size value of the profile, "send" as the method to send a message to address, "terminate" as the method to terminate a message at the responder, "destroy" as the method to delete a list namely as L_i at the responder, "open" as the method to open a profile, and finally the "run" as the method to execute an algorithm or an operation.

The event collection algorithm facilitates the assembly procedure for incoming messages and the formation of the ranked list. The algorithm identifies messages for service delegation by identifying port tags (key: $\mathrm{tag} = \mathrm{q}$, for incoming message for requester and $\mathrm{tag} = \mathrm{w}$, for incoming message for further redistribution/decentralization).

3.2.3 The Event Ranking Algorithm

The event ranking algorithm defines the criteria for service classification in the request or response from a meta-broker. To quantify such action we aim to minimize a function that calculates a set of metrics (known as rankings). In this paper, we define a number of parameters to calculate rankings such as: execution times, total times as well as energy consumption and service cost metrics as detailed in Algorithm 3. The operation includes the size as the method to return the size value of the profile.

The degree criterion defines the degree of the decentralized meta-broker topology as presented in [14], [26]. In addition, we have implemented the consumption per entity cost for monitoring energy utilization (e.g. at datacentre and host level). At last, we included the cost functions for defining the message and delay cost.

3.2.4 The Event Redistribution Algorithm

This algorithm describes the process of a meta-broker m_n to redistribute the event request to its interconnected meta-brokers. The message redistribution algorithm implements the event relocation procedure in the case of further event dissemination. The procedure alters the tag values of messages and forwards each one to a node belonging to a personalized profile list. For all incoming msg_i that have a flag f_i and $f_i = f_{red}$ it opens the $prof_i$ and collects the address of the linked meta-brokers. It sets the tag $t_i \in \Re$ to an indicator $q \in \Re$ for outgoing messages from redistribution. After for $\forall j_{m_i}$ it creates a list L_i with each row containing the characteristics of the service and creates the message msg_i that includes L_i, t_i and the performance criterion. The algorithm defines an interval $time_i$ where $time_i \prec \mathrm{int}_{red}$, so during that time it sends messages to other meta-brokers. Algorithm 4 demonstrates the event redistribution algorithm.

A key aspect is that the algorithm operates under the initial deadline value in order to be terminated in cases of interval violations. The algorithm allows messages to be forwarded only if there is no availability in the local resource pool. In this case, messages are reformed and transferred to remote entities for requesting resource availability according to specific criterion.

Algorithm 3. Event Ranking

Require:	i	the requesting or responding node
	Rank	the output of the criterion
	instr	the number of instructions
	cycles	the number of service cycles
	h	the uptime of the service in host
	dl	the delay of the entity
	int	the interval of an entity (e.g. int_i is the interval of requester)
	udl	the decision making time (e.g. udl_y)
	watts	the watts of the host entity
	consPerKW	the consumption per kW rate of the entity
	Coef	the coefficient value of the entity
	Nomsg	the total number of messages (e.g. from entity i to y is $Nomsg_{i:y}$)

```
1.  if criterion ← ET (Execution Time)
2.      Rank = instr*cycles
3.  end if
4.  if criterion ← TT (Total Time)
5.      Rank = (instr*CPI*1/CPU)*1.cores*h
6.  end if
7.  if criterion ← LA (Latency)
8.      Rank = dl + dli′
9.  end if
10. if criterion ← DE (Degree)
11.     Rank $\sum dl_i + \sum dl_{i'}$
12. end if
13. if criterion ← TuT (Turnaround Time)
14.     Rank = ET + LA
15. end if
16. if criterion ← MS (Makespan)
17.     Rank = ET + $udl_{i'}$
18. end if
19. if criterion ← CPE (Consumption per entity)
20.     Rank = (watts*TT*$10^{-3}$)*consPerKW*coef
21. end if
22. if criterion ← CPH (Consumption per host)
23.     Rank = watts *h*$10^{-3}$
24. end if
25. if criterion ← MeC (Message Cost)
26.     Rank = (size($L_i$) + size($L_{i'}$))*(1/bw)
27. end if
28. if criterion ← DeC (Delay Cost)
29.     Rank = ($Nomsg_{i:i'}$ +(($Nomsg_{i':i}$)/ $Nomsg_{i:i'}$))/$int_i$
30. end if
31. if criterion ← PR (Probability Cost)
32.     Rank = $dl_{entity}/int_{entity}$
33. end if
```

3.2.5 The Event Assignment Algorithm

The event assignment algorithm determines the next phase of the resource allocation. Here the events have been concluded and the service request is sandboxed in a VM. Algorithm 5 implements the allocation of services in entities (thus to their local hosts' scheduler). The algorithm collects the execution results after the completion of a service request. In particular for all service requests $\forall j_i$ allocates each of which to the LRMS. The operations include: set the tag to allocate the service to resource and the send (LRMS) to send procedure of service data into LRMS.

Algorithm 4. Event Redistribution

Require:	msg	the requesting message
	i	the requesting or responding node
	msg_i	the incoming message from requester i
	L_i	the list with the service specification data
	$f_{i'}$	the profile of the entity
	flag	the flag variable
	rds	the redistribution flag
	p	the tag indication for outcoming message from redistributor
	int	the interval of the requester
	t	the time instance

```
1.  for all msg where flag = rds
2.      open($f_{i'}$)
3.      get(ad)
4.      set tag ← p
5.      while Rank ← o then
6.          create($L_{i'}[j_i]$)
7.      end while
8.      create(msg[$L_{i'}$,tag, criterion]
9.      while t < $int_i$ then
10.         send(msg,ad)
11.     end while
12. end for
```

Algorithm 5. Service Assignment

Require:	i	the requesting or responding node
	j	the service
	j_{set}	the set of services in not i
	a	the value to define assignment
	res	the performance results of the service assignment

```
1. for all j ∈ $j_{set}$
2.     set tag ← a
3.     allocate (j,ad)
4.     send(LRMS)
5. end for
```

The procedure first collects the user service request from the SLA and selects the VM allocation policy according to the LRMS specification. The default queues implemented in ICMS are the First Come First Served (FCFS), Shortest Service Frist (SJF), Earliest Deadline First (EDF) and Priority Scheduling (PS). For all services $\forall j_i \in queue_{LRMS}$ the hypervisor policy controls the current workload w_i and calculates the total delay that includes the turnaround time and the hypervisor processing time $TotalDelay = TurnTime + hypervisorDelay$. Each service j_i is queued into $queue_{LRMS}$ by adding a key_i, j_i as a pair. For an interval int_i or for a specific queue length $sizeQueue_{LRMS} \leq \beta$ where $\beta \in \Re$, it schedules the service requests and allocates host computational units based on a host allocation policy. Finally, it updates the current workload w_i. This concludes the ICMS description.

4 Simulating the Inter-Cloud (SimIC) Toolkit

This section illustrates the description of SimIC v2.0 (Simulating the Inter-Cloud version 2) that is a novel simulator used to implement the inter-cloud functionality. SimIC is a discrete event simulation toolkit based on the process oriented simulation package of SimJava [16]. SimIC is used to simulate an inter-cloud facility where multiple clouds collaborate for service request distribution in a simulation setup. The package encompasses the ICMS algorithms including users, meta-brokers, local-brokers, datacenters, hosts, hypervisors and virtual machines (VMs). In SimIC, the message initialization begins at time instance 1, and then a message is created at state 1. After this, state 2 collects the message (get from out port State 1) and sends the message to in port State 3. During this time, the instance passes from time 2 to 3 and finally to time instance 4. Finally, the message is terminated (or initialized) from another state in order to continue the information exchanging. A more detailed discussion of the tool is presented in [16] that illustrate the layered structure of the tool and internal processes.

4.1 The SimIC Technical Features

The SimIC has been developed using the Java 2 Platform (JDK 1.6). It is based on the process event simulation API of the SimJava version 2 [6]. Its high level structure is based on the entities of CloudSim [21]. We have extended these in order to implement meta-scheduling capabilities and batch job simulation.

4.2 The SimIC Layered Architecture

The entities and their functionality are organized in a three-layer structure. This includes the entity layer, the queuing, behaviour and tagging layer, and the performance and tracing layer. Specifically, Layer 1 includes the entities representing the objects of the system. In a SimJava simulation, each feature is represented by a sim_entity class that encapsulates the core functionality. Each SimIC class defines the actual behaviour (layer 2) of entities that are the ICMS resources. The core classes are User, Meta-broker, Local-broker, Datacenter, Hypervisor, Hosts, VMs and Bucket. The initialization process begins when a user starts communication with the meta-broker through a user interface. Like a meta-scheduling system. The latter acts on behalf of the user and forwards the request to low level resources (either local or remote sites). This procedure is executed by a local-broker.

Layer 2 represents the core features of SimIC including the utilization of ports, functionalities and constraints that demonstrate the actual behaviour of the system entities. Each class contains at least one port for input or output messages to other linked entities. In addition, it incorporates mechanisms for collecting messages, taking decisions (based on policies) and forwarding to an entity for request delegation and execution. Each entity is defined by constraints to govern its actions. The actual communication is based on the tags that are assigned to messages during exchange. These tags are the means of identifying the origin of a message and the operations expected from a responder. Additionally, queuing refers to the orchestration of events (that are service messages) according to different LRMS (FCFS, SJF, PS).

Layer 3 relates to the performance monitoring and tracing operations of the system entities. The performance measures include execution time of the VM, turnaround time of service, makespan of the service, throughput of services in an entity, host utilization levels, VM utilization levels, service latencies and VM uptime times. Most of these metrics could be utilized by different entities in order to measure the performance of SimIC at different instances, for example throughput of a datacentre or latency at a hypervisor.

4.3 The SimIC Entities

SimIC automates service request distribution among decentralized meta-brokers. Meta-brokers are placed on top of each cloud in order to communicate with other brokers to produce a distributed and interoperable cloud infrastructure (similar to grid computing). In SimIC each request is treated as unique. For example, a user requests for a VM, suppose with 0.25 of 1 host performance and executes a set of programs with 100*106 instructions, and CPI (cycles per instructions) of 3 (300 cycles/100 instructions) in a machine with clock rate of 1,000 MHz (0.25 of 4,000 MHz of Host with single core). The performance indicators of the VM are calculated as follows. The execution time is given by $ExecTime_{VM} = Instruction \times CPI \times 1/cpu \times 1/cores$. Thus, the result is calculated as follows: $ExecTime_{VM} = 100\times 10^6 \times 3 \times 1/1{,}000 \times 1 = 3 \times 10^5\,\text{ns} = 0.3\,\text{ms}$. The performance of the VM is calculated at 3.33 based on $Performance_{VM} = 1/ExecTime_{VM} = 3.33$. Next, we present a description of the SimIC entities that implement ICMS functionality.

The *UserCharacteristics* class instantiates the service information for each of the users by incorporating hardware and software requirements that has been previously defined in two different files. Each service j_i is assigned to a meta-broker m_i. It contains a number of physical characteristics named as CPU cpu_i, memory mem_i, cores cor_i, storage $stor_i$ and bandwidth bw_i. The *ServiceCharacteristics* class calculates an initial performance request based on the performance estimation that is calculated by the number of MIPS as given by the formula $ClockRate/CPI \times 10^6$. The *OutputUserRequirements* class generates a dynamic user profile that includes a variety of hardware, software (heterogeneous requirements) and initial performance request measurements.

The *User* class is responsible in forwarding requests (namely as j_{m_i}) to resources, wherein each request is scheduled after a specific processing delay to a dedicated meta-broker. The *Meta-broker* class implements the interoperability functionality of SimIC ($M \in \{m_1, m_2, \ldots, m_n\}$). Specifically, each meta-broker is interconnected with one or more meta-brokers depending on a simulation experiment. The *Bucket* class represents the terminal entity that collects the unexecuted services and keeps logs related to services. These could be either re-directed to an inter-cloud after a regular interval or terminated if there is an SLA mismatching. Termination and re-distribution flags (f_{trm}, f_{red}) are used to decide whether this is the termination or the re-distribution point.

The *local-broker* (that is the internal cloud broker) class defines an SLA matchmaking process for deciding whether the specification of user requirements could be addressed

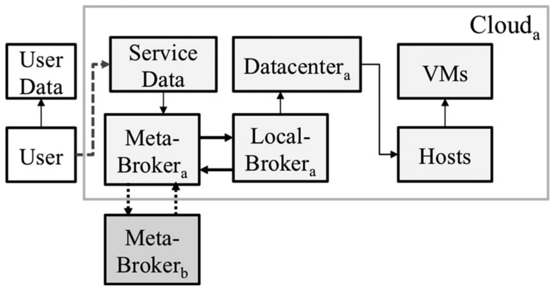

Fig. 4. Description diagram of the SimIC entities performance evaluation.

TABLE 2
The SimIC Configuration

Username	StS	MaL	OlS	NiS	NiB
Memory	4,000	6,000	2,000	2,000	8,000
CPU-cores	4	4	2	2	4
CPU-speed	4,000	4,000	2,000	2,000	10,000
Storage-HD	10,000	10,000	10,000	10,000	10,000
BW	10,000	10,000	5,000	10,000	10,000
Instructions	10^*10^8	12^*10^8	15^*10^8	16^*10^8	16^*10^8
CPI	1	4	3	3	3

by a local resource. The datacentre's current performance is dynamically calculated for measuring the available computational power. This is realized by a messaging policy, as for all msg_{m_i} and the $t_i \in \Re$ is identifiable, the algorithm decomposes the message $msg_{m_i}$and collects the list L_i by running the performance criterion classification function that updates the list L_i. This includes the validation process of tag t_i. The *OpenHost* class imports each host characteristic from a file by allowing SimIC to access host hardware characteristics while OpenHostsList opens a list from a file that contains the individual hosts dedicated to a specific cloud.

The *Datacenter* class accepts events for VMs deployment in the cloud that are determined by a hypervisor. This class implements functionality for calculating costs and energy consumption. It passes all events to a local policy enforcement engine. The *Hyper* class represents a hypervisor and is responsible for collecting requests for VMs from the datacenter class by accessing the host and VM allocation policies. This class queues each service j_i into a $queue_{LRMS}$ by adding a key_i, j_i as a tuple. The *HyperCall* class generates an internal thread to release the services that have been scheduled in the queue according to the LRMS algorithm (int_i and $sizeQueue_{LRMS} \leq \beta$).

The *HostCharacteristics* class imports each specific host computational capacity as defined in a file. The *Hosts* class represents a static computing machine. The class gets an event from Hyper for requesting an instance of the host characteristics. Eventually, this adds an additional delay to the hypervisor decision making process when allocating a VM. This is the latency of the host for starting the service execution. The *VM* class sandboxes the user profile. A more detailed discussion of the SimIC implementation is presented in [16].

To calculate the total communication delay of messages we split the latencies incurred at different stages of a service request execution life cycle in an inter-cloud. Thus, let's assume that a number of entities E with $E = \{e_1, e_2, \ldots, e_n\}$ are linked as a directed graph to form a topology in an inter-cloud. For each communication a message msg_i is sent containing the service requirements j_i and a tag $tag_i \in \Re$. The assumption is that a trail is generated from one entity to another in such a way that the weight of the trail $w_{e_i} : w_{e_n}$ is calculated as the latency of the message msg_i to reach entity e_n. The cumulative latency of the user to VM communication is calculated as follows: $Latency_{mrb} = \sum_{mbr_i \in E} \deg(meta\text{-}broker_i)$, $Latency_{dc_i} = (\frac{1}{2} \times \sum_{dc_i \in E} \deg(dc_i) + \sum_i \deg(dc_i)) \times coef$ thus, $Latency_{user-vm} = Latency_{mb_i} + Latency_{dc_i} + Latency_{hyper_i}$

So, for each message msg_i that is sent from entity e_1 to e_n, the messaging factor (MF) defines a metric for the cost of message distribution. In a bi-directional graph formation this is calculated as the division of the sum of the messages received by the sum of the messaged sent as $MF = \sum_{j=0}^{\vartheta} msg_j / \sum_{j=0}^{\eta} msg_i$ where $msg_i \in \{msg_1, msg_2, \ldots, msg_n\}$, $msg_j \in \{msg_1, msg_2, \ldots, msg_j\}$ and $L_i \neq 0$. Here η represents the maximum number of requested messages and ϑ is the maximum number of received messages. To conclude, this section presented SimIC, a toolkit that allows system architects to configure a variety of inter-clouds in terms of entities and policies. The toolkit contains a number of scheduling algorithms and features for achieving configurable service execution. Fig. 4 demonstrates the various actors and the interactions among the SimIC entities. A more detailed discussion and explanations of the various entities along with their relationships are presented in [16]. Next section demonstrates an experimental analysis and evaluation of the ICMS.

5 Performance Evaluation

The experimental setup implements the messaging approach of [14] and involves the comparison of two approaches, namely a centralized inter-cloud and the decentralized ICMS model of inter-cloud being followed in this paper. In centralized approach the assumption is that there is a bi-directional communication among all nodes in a cloud. In this approach, we first focus on demonstrating that there is no experimental bias. We achieve this by running a number of tests, which show that a centralized IC does not affect cloud performance. Then we configure an ICMS based IC for service executions which is similar to the centralized setup. Finally, we show performance analysis of ICMS considering service request arrivals and load distributions in both static and dynamic modes. Our experimental results show that ICMS with dynamic workload management outperforms static mode when all resources are available. Our simulations implement the next experimental setting where five users submit requests in cycles, as it is shown in Table 2. The hosts specification includes a total of 166 cores per cloud with an average of 10^3 MHz CPU, 10 GB RAM, 10^4 GB storage and 10 mbps bandwidth per host.

5.1 Cloud versus Inter-Cloud Settings

The first experiment aims to demonstrate that IC does not affect performance of a cloud, thus we compare with a

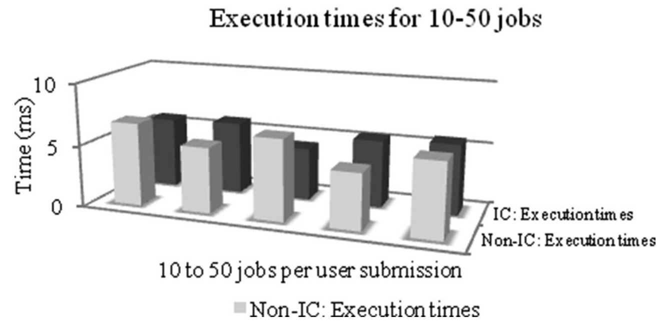

Fig. 5 Makespan for 10 to 50 services (non-IC versus IC).

similar hardware setup. We show that IC performs better than or equal to non-IC setting where both cases have exactly the same computational capacity. This means that a cloud that is non-IC based has exactly the same capacity (CPU, memory, storage and bandwidth) as with an IC based cloud that is made from two clouds, cloud 1 and cloud 2. The experimental analysis involves constant submissions of intervals of 2 ms. Cloud 1 includes five users that submit 10 to 50 service requests. Each time a request arrives in the hypervisor, a new VM is generated according to the available resources.

In both cases (cloud and IC) the utilization model of [5] is applied. This involves that resources will be allocated if they are available until the utilization reaches its peak (100 percent). IC distributes the jobs based on the MEO and centralized distribution approaches as discussed in [14]. In both cases (cloud and IC) all services are executed in local clouds, as there is no option for further service distribution. As the number of service requests increases, the IC will increase the makespan value due to the latency (set to 2 ms) that is caused by the message exchanges. In particular, the study sets this value to a low number (2 ms) as the assumption is that cloud 1 divided over two.

Fig. 5 shows the makespan values for 10 to 50 service requests per user to each cloud. Both trend lines show similar variations, which means that non-IC and IC follow similar makespan trends. Fig. 6 shows the execution times (10 to 50 services) for both IC and non-IC cases. The average execution time for a single service request in the non-IC case is 5.79 ms while for IC case is 5.38 ms. This shows that IC achieves better execution time due to the better allocation of the resources. The improvement of IC is calculated at 7 percent (percentage of the division of the difference of higher to lower value, to the higher value).

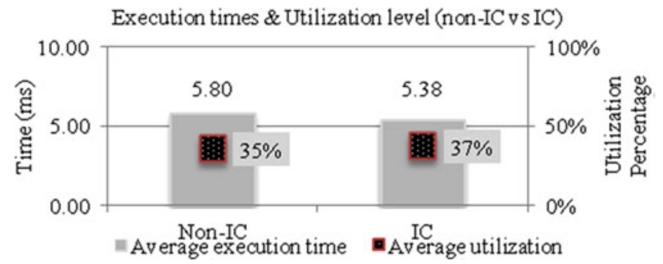

Fig. 6. Execution times for 10 to 50 services (non-IC versus IC).

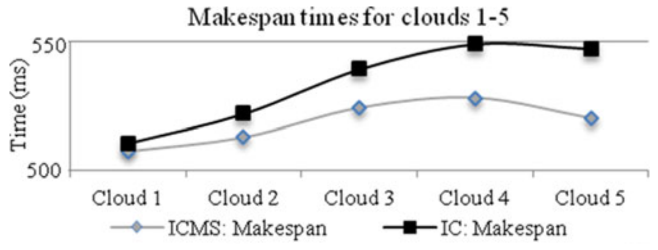

Fig. 7. Average execution time and average utilization for 10 to 50 services (non-IC versus IC).

Fig. 7 shows the average execution time and average utilization rates for both cases. It indicates that the average utilization of the IC case is 37.2 percent and the non-IC case is 35.4 percent. The average execution time shows decreasing value for the IC case. To conclude, IC increases utilization levels because it executes more service requests by decreasing the IC average execution times. The values are calculated based on the formulas of Section 3.1 and is related to the throughput value of services. In detail, the values are relatively low due to the low number of service requests with respect to the cloud resources.

5.2 The Inter-Cloud versus ICMS Setting (1 Service Request Submission Per Cloud)

We present two cases for 1 and 50 user submissions per cloud and we monitor the performance in both cases. The experiment demonstrates that ICMS performs better than or equal to the IC setting (with augmented datacenter view) with both having the same host configuration (five clouds). This increase in performance is due to the service distribution and meta-scheduling approach being followed in the ICMS framework. In IC each meta-broker has a complete knowledge of the actual cloud infrastructure (e.g. datacenter characteristics, Hosts, VMs) as it communicates with other cloud brokers for information exchange. In contrast, the ICMS approach has a partial knowledge of the infrastructure and follows the decentralized message distribution as it is discussed in [14]. This offers a higher level of abstraction for the entire cloud because a set of users are only mapped to a restricted set of meta-brokers at a time.

5.2.1 The Inter-Cloud versus ICMS Setting for 1 Service Submission per Cloud

The experiment includes an IC of five clouds that have the same host specification with the ICMS and the topology is considered as decentralized. Specifically we first assume that each cloud meta-broker can access the next in the list. For example, meta-broker 1 sends a service request to meta-broker 2, then meta-broker 2 to meta-broker 3, etc. For each service request that is submitted, if cannot be executed in the local cloud, it is always forwarded to remote cloud(s). The availability is set so that each service can be executed in the next cloud (e.g. service 1 to cloud 2, service 2 to cloud 3 etc.). In the centralized case (IC) the assumption is that all clouds can access all other clouds directly. Fig. 8 shows the makespan times for one service submission per user with 1 ms interval.

It is apparent that the values are decreasing for the case of ICMS. The average value is calculated to 519 ms, while

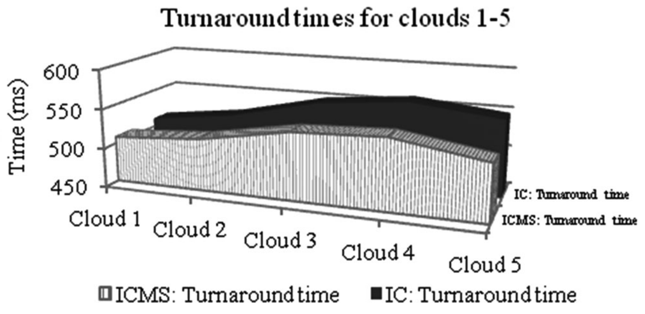

Fig. 8. Makespan times for one service per user (IC versus ICMS).

the IC is measured to 534 ms; that shows an improvement of 15 ms in the average values. The improvement factor for this case is calculated to be 3 percent. Similarly to makespan values, the turnaround times for one service submission for both cases are as follows. The average value for ICMS is 524 ms and for IC it is 539 ms. On this basis, the centralized case increases the turnaround times mainly because of transferring services among datacenters. An improvement of 5 percent in turnaround times is observed in cloud 5 where the same submission requires 521 ms in ICMS and 548.4 ms in IC. Fig. 9 shows the performance comparison of both cases in terms of response ratios. Specifically, the response ratio is calculated as the difference of the highest value of the metric from the lowest value of the same metric divided with the highest value e.g. $(x - y)/x \times 100\%$ when x and y represent $latency_{user\text{-}vm}$. The figure shows that the performance increases for ICMS as more users submit requests in a linear manner. Yet as requests are transferred to remote clouds (e.g. cloud 2, cloud 3 etc.) the ICMS performance decreases with regards to the performance as in the original ICMS setup.

5.2.2 *The Inter-Cloud versus ICMS Setting for 50 Service Submissions per Cloud*

This experiment demonstrates the simulation results for high workload submissions (50) per user. As more services are exchanged resource availability becomes more limited and allocation management becomes more complex. In order for the results to be comparable the study takes into account clouds with exactly the same utilization levels (e.g. for this experiment clouds 3 and 4 offer the same utilization of 20 percent and clouds 2 and 5 with utilization of 6 percent). Services that cannot be executed due to non-resource availability or SLA mismatching are dropped, as the dynamic workload is inactive. This means that we do not re-schedule jobs to resources.

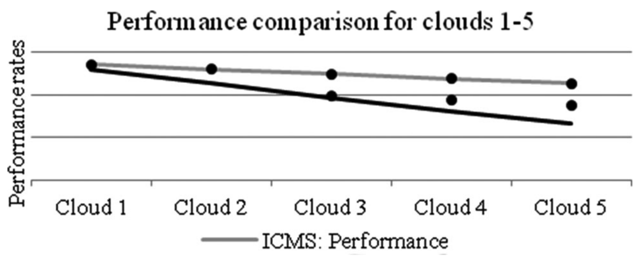

Fig. 9. Comparison of performance (response ratios) of ICMS-IC.

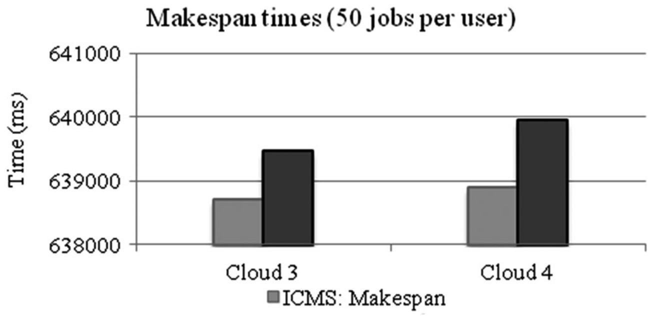

Fig. 10. Makespan times for 50 services per user for clouds with same utilization (clouds 3, 4).

Fig. 10 shows that the makespan times for 50 services per user have slightly improved results for ICMS and clouds 3 and 4 (same utilization levels). The average makespan time for IC (clouds 3 and 4) is 639,706 ms while the same metric value for ICMS (clouds 3 and 4) is 638,806 ms (900 ms difference). Fig. 11 shows the makespan for clouds with low utilization of 6 percent (clouds 2, 5). Again, ICMS algorithms offer lower makespan times when compared to IC. To conclude, both cases (1 and 50 users) show that the ICMS achieves better makespan and turnaround times. This will affect the resource utilization and resource usage as the total scheduling and execution time of services is reduced.

5.3 The ICMS Setting: Low and High Delays and 40 to 100 Percent Resource Availability

This experiment demonstrates the dynamic workload management for an ICMS case. The decentralized ICMS sends service requests to different clouds by incorporating dynamic distribution. This experiment executes requests having a combination of 1 to 4 ms delay and 40 to 100 percent resource availability. The percentage is related to the ability of a cloud to execute the specific service task; e.g. the 40 percent availability is selected as it demonstrates a cloud with low resource availability. The next list is a mixture of different combinations in the experiment.

i) 1-40%: delay 1 ms, availability 40 percent.
ii) *4-40%: delay 4 ms, availability 40 percent (where * indicates that delay is 4 times higher than case i).
iii) 1-100%: delay 1 ms, availability 100 percent.
iv) *4-100%: delay 4 ms, availability 100 percent (where * indicates that delay is 4 times higher than case iii).

Fig. 12 shows the makespan times of ICMS for each of the four cases. It is shown that when the availability is 40 percent ICMS distributes service requests to all clouds;

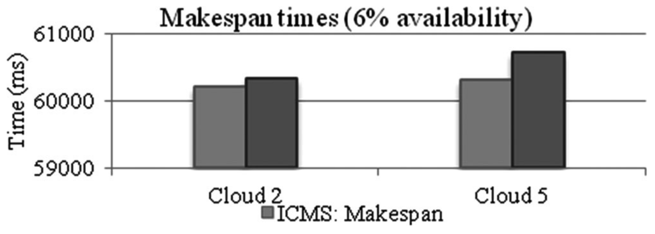

Fig. 11. Makespan times for 50 services per user for clouds with same utilization of 6 percent (clouds 2, 5).

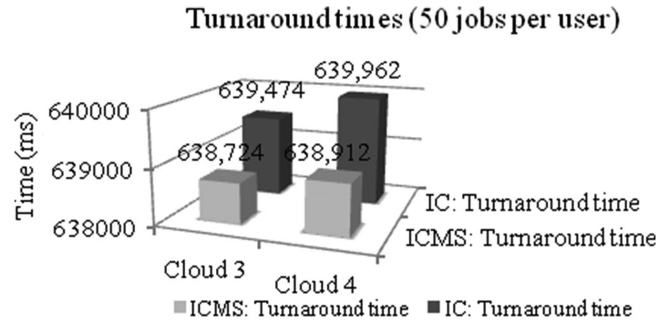

Fig. 12. Makespan times for ICMS cases.

however in the case of 100 percent availability, cloud 4 executes most of the service requests. This is because of the high number of hosts that are available in cloud 4, which increase the available computational power. Fig. 13 shows the overall resource utilization levels of clouds 1 to 5 for all four cases. Specifically the highest utilization is found in case 3 that details execution of all the services with high resource availability. It is also demonstrated that case 2 that involves experimentation with high delays it decreases the utilization levels. This is because of the increasing delay when continuous submissions occur.

5.4 The ICMS Setting: A Mixing of User Submissions and 100 Percent Resource Availability

This experiment demonstrates the ICMS performance for a combination of user submissions. In the first case, 50 service requests are submitted to cloud 1, while in the second case 10 service requests are submitted to each cloud by a user (total of 50 service requests). Fig. 14 shows the percentage of successful executions when comparing one user per cloud and all users in cloud 1. It is shown that the user requests distribution in different clouds offers better percentages of successful service requests execution. Thus, this shows that in an IC, the spreading of users in different clouds could assist in achieving higher percentages of successful executions.

Fig. 15 shows the makespan and turnaround times for services served by five clouds (1 to 5). It is shown that for high number of service request submissions, ICMS makes a

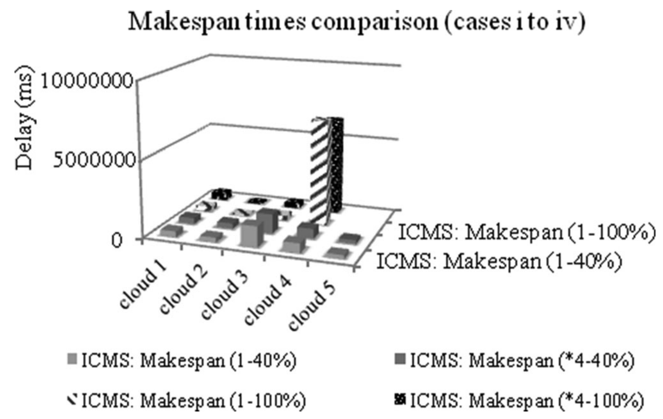

Fig. 13. Overall utilization levels for ICMS cases.

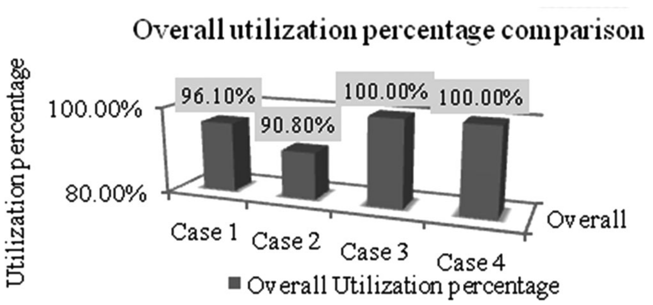

Fig. 14. Successful execution percentages for cases 5a and 5b.

better distribution by allocating resources more efficiently (based on the lower makespan times). In addition, turnaround times for higher workloads have been sufficiently decreased. For example, makespan times for cloud 2 shows an improvement rate of 4.9 percent.

6 Concluding Remarks and Future Work

This work presents the ICMS, a framework that allows inter-cloud service distribution. We have developed this framework to address the issue of large scale service request distributions in IC that cannot be achieved from current approaches. Our experimental results support the following conclusions: (a) ICMS has an improved makespan time and reduced turnaround time, (b) ICMS outperforms standard IC in terms of remote cloud invocations and (c) ICMS improves performance each time a new service request is submitted to IC. Future directions involve the extension of SimIC in terms of VM migration policies. Further experiments with more clouds would have given a better reflection of the performance improvements. In addition, we aim to work on a message passing interface system for queuing host processors for information processing during interactions.

We aim to explore the security issues during communication between IC in order to enhance the effectiveness of our ICMS framework. Also, a future research direction will be to test the system in terms of high variability in inter-intervals and service times in order evaluate probability based distribution of services in inter-cloud. Finally, we aim to extend ICMS to support real cloud platforms. In particular in [27] we developed a platform service to retrieve data from clouds e.g. instances, images and resources in OpenStack systems and we implement an inter-cloud metabroker that acts as mediation service. In particular, the platform service does not target to change internal cloud system processes but to utilize available interfaces by enabling

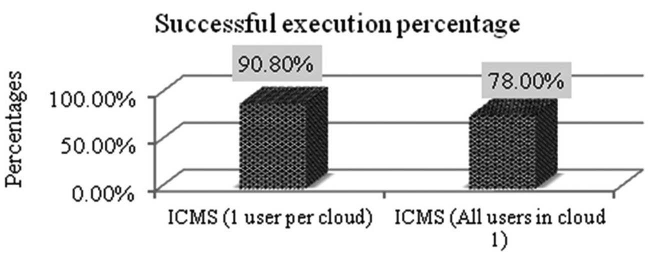

Fig. 15. Makespan and turnaround times for both cases.

remote management of inter-cloud services in a unified manner. A future direction is to include the whole algorithmic model included in ICMS in within the platform service and to explore experimentation results for heterogeneous platforms.

Acknowledgments

S. Sotiriadis is the corresponding author.

References

[1] N. Bessis, S. Sotiriadis, F. Xhafa, F. Pop, and V. Cristea, "Meta-scheduling issues in interoperable HPCs, grids and clouds," *Int. J. Web Grid Serv.*, vol. 8, no. 2, pp. 153–172, 2012.

[2] Global Inter-Cloud Technology Forum, "Use cases and functional requirements for inter-cloud computing," GICTF White Paper, Aug. 9, 2010, [Online]. Available: http://www.gictf.jp/doc/GICTF_Whitepaper_20100809.pdf

[3] I. Moschakis and H. D. Karatza, "Parallel job scheduling on a dynamic cloud model with variable workload and active balancing," in *Proc. 16th Panhellenic Conf. Inform.*, Oct. 5–7, 2012, pp. 93–98.

[4] A. Folling, C. Grimme, J. Lepping, and A. Papaspyrou, "Decentralized grid scheduling with evolutionary fuzzy systems," in *Proc. Job Scheduling Strategies Parallel Process.*, 2009, pp. 16–36.

[5] R. Buyya, R. Ranjan, and R. N. Calheiros, "InterCloud: Utility-oriented federation of cloud computing environments for scaling of application services," *Algorithms Archit. Parallel Process*, vol. 6081/2010, no. LNCS 6081, pp. 13–31, 2010.

[6] S. Sotiriadis, "The inter-cloud meta-scheduling framework," Ph.D. dissertation, 2013, [Online]. Available at http://derby.openrepository.com/derby/handle/10545/299501

[7] S. Sotiriadis, N. Bessis, and N. Antonopoulos, "Towards inter-cloud schedulers: A survey of meta-scheduling approaches," in *Proc. 6th IEEE Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, Oct. 26–30, 2011, pp. 59–66.

[8] L. Anand, D Ghose, and V. Mani, "ELISA: An estimated load information scheduling algorithm for distributed computing systems," *Comput. Math. Appl.*, vol. 37, no. 8, pp. 57–85, 1999.

[9] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "Distributed job scheduling on computational grids using multiple simultaneous requests," in *Proc. 11th IEEE Int. Symp. High Perform. Distrib. Comput.*, Jul. 23–26, 2002, p. 359.

[10] A. R. Butt, R. Zhang, and Y. C. Hu, "A self-organizing flock of condors," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2003, pp. 15–21.

[11] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "OurGrid: An approach to easily assemble grids with equitable resource sharing," in *Proc. 9th Workshop Job Scheduling Strategies Parallel Process*, 2003, pp. 1–20.

[12] K. Lai, B. A. Huberman, and L. Fine, "Tycoon: An implementation of a distributed market-based resource allocation system," HP Labs, Tech. Rep., 2004.

[13] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny, "Inter-operating grids through delegated matchmaking," in *Proc 2007 ACM/IEEE Conf. Supercomput.*, Nov. 2007, pp. 1–12.

[14] N. Bessis, S. Sotiriadis, F. Pop, and V. Cristea, "Using a novel message-exchanging optimization (MEO) model to reduce energy consumption in distributed systems," *Simul. Model. Pract. Theory*, vol. 39, pp. 104–120, Dec. 2013.

[15] S. Sotiriadis, N. Bessis, P. Kuonen, and A. Antonopoulos, "The inter-cloud meta-scheduling (ICMS) framework," in *Proc. 27th IEEE Int. Conf. Adv. Inform. Netw. Appl.*, Mar. 25–28, 2013, pp. 64–73S.

[16] S. Sotiriadis, N. Bessis, and A. Antonopoulous, "SimIC: Designing a new inter-cloud simulation platform for integrating large-scale resource management," in *Proc. 27th IEEE Int. Conf. Adv. Inform. Netw. Appl.*, Mar. 25–28, 2013, pp. 90–97.

[17] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour, "Enhanced algorithms for multi-site scheduling," in *Proc. 3rd Int. Workshop Grid Comput.*, pp. 219–231, 2002.

[18] M. D. De Assuncao and R. Buyya, "Performance analysis of allocation policies for interGrid resource provisioning," *Inform. Softw. Technol.*, vol. 51, no. 1, pp. 42–55, 2009.

[19] K. Leal, E. Huedo, and I. M. Llorente, "A decentralized model for scheduling independent tasks in federated grids," *Future Generation Comput. Syst.*, vol. 25, no. 8, pp. 840–852, 2009.

[20] I. Rodero, F. Guim, J. Corbalan, L. Fong, and S. M. Sadjadi, "Grid broker selection strategies using aggregated resource information," *Future Generation Comput. Syst.*, vol. 26, no. 1, pp. 72–86, 2010.

[21] R. N. Calheiros, R. Ranjan, A. Beloglazov, A. F. C. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw.: Pract. Exp.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[22] S. Sotiriadis, N. Bessis, F. Xhafa, and N. Antonopoulos, "From meta-computing to interoperable infrastructures: A review of meta-schedulers for HPC, grid and cloud," in *Proc. 26th IEEE Int. Conf. Adv. Inform. Netw. Appl.*, Mar. 26–29, 2012, pp. 874–883.

[23] Y. Huang, N. Bessis, P. Norrington, P. Kuonen, and B. Hirsbrunner, "Exploring decentralized dynamic scheduling for grids and clouds using the community-aware scheduling algorithm," *Future Generation Comput. Syst.*, vol. 29, pp. 402–415, 2013.

[24] H. H. Mohamed and D. H. J. Epema, "Experiences with the Koala co-allocating scheduler in multiclusters " in *Proc. 5th IEEE Int. Symp. Cluster Comput. Grid*, 2005, pp. 784–791.

[25] Amazon elastic compute cloud (amazon EC2), (Dec. 15, 2013). [Online]. Available: http://aws.amazon.com/ec2/

[26] N. Bessis, S. Sotiriadis, F. Pop, and V. Cristea, "Optimizing the energy efficiency of message exchanging for service distribution in interoperable infrastructures," in *Proc. 4th IEEE Int. Conf. Intell. Netw. Collaborative Syst.*, Sep. 19–21, 2012, pp. 105–112.

[27] S. Sotiriadis, N. Bessis, and E. Petrakis, "An inter-cloud architecture for future internet infrastructures," in *Adaptive Resource Management and Scheduling for Cloud Computing*, F. Pop and M. Potop-Butucaru, Eds. New York, NY, USA: Springer, 2014, pp. 206–216.

[28] Y. Huang, N. Bessis, A. Brocco, S. Sotiriadis, M. Courant, P. Kuonen, and B. Hisbrunner. "Towards an integrated vision across inter-cooperative grid virtual organizations," in *Proc. 1st Int. Conf. Future Generation Inform. Technol* , 2009, pp. 120–128.

[29] R. Shah, B. Veeravalli, and M. Misra, "On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 12, pp. 1675–1686, Dec. 2007.

Stelios Sotiriadis is a research collaborator at the Technical University of Crete (TUC) and a member of the Intelligent Systems Laboratory of the School of Electronic and Computer Engineering, Technical University of Crete, Chania, Crete, Greece. He works for the Future Internet Social Technological Alignment Research (FI-STAR FP7), which is an FI-PPP programme. His research interests are related to cloud computing, inter-cloud, future internet (fi) applications, meta-scheduling systems, internet of things (iot) and openstack systems. He has published more than 50 papers and received two best paper awards. His personal profile is online on *www.sotiriadis.gr*.

Nik Bessis is currently a full Professor and the Head of the Department of Computer Science at Edge Hill University, UK. He is a fellow of HEA, BCS and a senior member of IEEE. His research is on social graphs for network and big data analytics as well as developing data push and resource provisioning services in IoT, FI and inter-clouds for a number of settings including disaster management. He is involved in a number of funded research and commercial projects in these areas. Prof Bessis has published over 250 works and won 4 best papers awards. He is an editor of several books and an editor-in-chief of a refereed international journal.

A. Anjum is a reader in distributed computing in the School of Computing and Mathematics at the University of Derby. His areas of research include distributed and parallel systems, cloud computing and scalable methods to mine large and complex data sets. He has worked on various research projects, funded by European, American, and Asian funding agencies. He has been part of the EC funded projects in Grid and distributed systems, machine learning, and data mining such as Health-e-Child (IP, FP6), neuGrid (STREP, FP7), and TRANSFORM (IP, FP7), where he has been dealing with resource management of large-scale systems, performance monitoring and optimization, data mining, and service orchestration.

R. Buyya received the BE and ME degrees in computer science and engineering from Mysore and Bangalore Universities in 1992 and 1995, respectively; and a doctor of philosophy (PhD) degree in computer science and software engineering from Monash University, Melbourne, Australia in 2002. He is a professor of computer science and software engineering; future fellow of the Australian Research Council; and the director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft Pty Ltd., a spin-off company of the University, commercialising its innovations in grid and cloud computing. He served as founding editor-in-chief (EiC) of the *IEEE Transactions on Cloud Computing* (TCC).