Utility-Centric Networking: Balancing Transit Costs With Quality of Experience

Truong Khoa Phan, David Griffin, Elisa Maini, and Miguel Rio

***Abstract*— This paper is focused on techniques for maximizing utility across all users within a total network transit cost budget. We present a new method for selecting between replicated servers distributed over the Internet. First, we introduce a novel utility framework that factors in quality of service metrics. Then we design an optimization algorithm, solvable in polynomial time, to allocate user requests to servers based on utility while satisfying network transit cost constraints, mapping service names to service instance locators. We then describe an efficient, low overhead distributed model which only requires knowledge of a fraction of the data required by the global optimization formulation. Next, a load-balancing variant of the algorithm is explored that substantially reduces blocking caused by congested servers. Extensive simulations show that our method is scalable and leads to higher user utility compared with mapping user requests to the closest service replica, while meeting network traffic cost constraints. We discuss several options for real-world deployment that require no changes to end-systems based on either the use of SDN controllers or extensions to the current DNS system.**

***Index Terms*— Utility function, server selection, name resolution, optimization.**

I. Introduction and Motivation

As THE Internet becomes the enabler for an increasing variety of services with a wide spectrum of requirements, pressure is being put onto the Internet ecosystem to facilitate service placement and to select the best replica for each user request. This replication always involves multi-stakeholder trade-offs between costs and quality of service (QoS).

In this paper we are addressing the problem of enabling a high Quality of Experience for users by selecting service replicas that provide high utility while at the same time keeping the solution within an acceptable total cost, where cost is composed of transit costs for an ISP incurred by inter-domain traffic. In essence we are maximizing utility across all users within the bounds of a total inter-domain traffic cost.

There are many drivers for service replication, including server resilience, network diversity, and proximity of servers to users. Deploying services close to the users allows application providers to improve on QoS metrics such as latency and throughput. Some frameworks, such as fog computing [1], even attempt to put service instances at the extreme edge of the network in locations such as access points.

Service quality has two major sets of component metrics, relating to computation and networking parameters. Servers need to be properly provisioned for the arrival rate and holding time of user requests otherwise users will not be served or blocked. The service selection system needs to take into account both computation and networking factors to optimize its selection.

Resolution involves converting a service name to a specific network locator for the selected replica. Our work assumes that the user's ISP is in the best position to make this selection (either through name resolution or software defined networking). The ISP has accurate information regarding the user's position in the network and detailed knowledge of the current network status. Furthermore, implementation of the resolution process by the ISP allows the ISP to apply its own network policies in the selection process, for example to manage inter-domain transit traffic load and hence costs. A centralized approach would, in theory, allow global optimization but is unrealistic at global scales. A central entity would not have access to information on the detailed user location, the network topology or current network status. Furthermore, it would be incapable of implementing ISPs' specific traffic policies as it would have to arbitrate between conflicting policies of different ISPs which would be problematic from a business point of view. For those reasons, our server selection model can be implemented in a similar way to PaDIS [2] which allows ISPs to better assign users to servers by exploiting its own knowledge of network conditions and user locations.

The main contributions of the work described in this paper are as follows:

- First, we introduce a novel *utility function* to model user perception of service performance relating to one or more underlying QoS metrics. Application providers are able to define the parameters of the utility functions per service, according to the requirements of that service. A key novel feature of our utility function is the ability to define the threshold beyond which user perception of quality improvements will not improve further.
- Secondly, we design a *centralized optimization algorithm that can be solved in polynomial time* that allows ISPs to redirect their users to the best replica of the service, allowing a trade-off between traffic costs and users' QoS. The model allows to optimize for multiple services simultaneously.
- Finally, we propose a *low overhead distributed model* that allows ISPs to run a local version of the selection

Manuscript received September 30, 2016; revised September 8, 2017; accepted November 6, 2017; approved by IEEE/ACM Transactions on Networking Editor T. Hou. Date of publication December 25, 2017; date of current version February 14, 2018. This work was supported in part by FP7 FUSION under Grant 318205, in part by the U.S. Army Research Laboratory and U.K. Ministry of Defence under Grant W911NF-16-3-0001, in part by H2020 5G-MEDIA under Grant 761699, and in part by CHIST-ERA CONCERT projects under Grant I1402. *(Corresponding author: Truong Khoa Phan.)*

T. K. Phan, D. Griffin, and M. Rio are with the Department of Electronic and Electrical Engineering, University College London, London WC1E 6BT, U.K. (e-mail: t.phan@ucl.ac.uk; d.griffin@ucl.ac.uk; miguel.rio@ucl.ac.uk).

E. Maini is with Vodafone, Newbury RG14 2FN, U.K. (e-mail: elisa.maini@vodafone.com).

Digital Object Identifier 10.1109/TNET.2017.2780257

1063-6692 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

algorithm without the need for global knowledge of all service replicas and network conditions. We demonstrate that the algorithm always converges to a stable state and can perform load balancing to significantly improve the overall performance.

This paper is organized as follows: We start by surveying related work in Section II. Section III introduces the utilitarian server selection model and is followed by the formulation of the centralized optimization problem (Section IV) and a distributed model (Section V). Section VI presents the algorithm evaluation results and Section VII presents a small-scale testbed deployment and discusses how the model can be deployed in practice. We draw our conclusions on the work in Section VIII.

II. Related Work

A. Utility Function

In general, utility is a measure of user satisfaction and is a function of received QoS. Most of the work in the literature considers a utility function which is, in general, a non-decreasing function of the effective transmission rate (bandwidth) [3]–[5] or signal-to-interference ratio (SIR) of ongoing connections [6]. In this work, we consider utility as a non-increasing function of latency, although additional performance metrics will be incorporated in future work.

Differing from most of the work in the literature, our utility function has a special initial region $[0, T_{min}]$ in which the utility score is unchanged. This feature better captures the nature of services (see the example in section III-A) and, as we show in Section VI-B, optimization based on such a utility function improves overall system performance. To the best of our knowledge, there are no or few practical solutions of the utility work deployed in actual networks due to their complexity [5], [6]. For instance, the proposed algorithms in [5] require the modification of TCP stacks at the end-hosts, however they also do not guarantee the algorithms convergence in a general network.

Our work does not require any change in the end-host TCP stacks and, therefore, can be more easily deployed in real networks (see section VII for more detail).

B. Server Selection

Our work is related to recent work on optimizing performance-cost for server selection [7], [8]. For example, Wendell *et al.* [7] introduce DONAR - a decentralized replica-selection system that considers client locality, server load, and policy preferences. Like DONAR our model can also balance client requests across replicas according to a manually set capacity cap at each replica, but in addition we also adopt a new strategy called *maximizing the minimum spare capacity* which automatically allocate resources between replicas in a fair way. Zhang *et al.* [8] focuses on optimizing cost and performance in online service provider networks. The objective is to search for an optimal "sweet-spot" in the performance-cost Pareto front. Auspice [9] uses a heuristic placement algorithm to determine the locations of active replicas so as to minimize client-perceived latency.

The work described in these papers trades off latency, cost and load balancing. However, by simply minimizing latency, selection of the lowest latency replica is always preferred. Our utility function models the fact that the user perception of utility is not increased by reducing latency below a certain threshold, which depends on service type. This is a less-greedy approach, resulting in a fairer solution by improving performance for all users without reducing utility for individual users, as we demonstrate in Section VI-B.

In the distributed version of our algorithm we rely solely on updates from servers to resolvers and no messages need to be exchanged between servers or between resolvers. This significantly reduces communication overhead and system complexity compared to other approaches in the literature. Our algorithm can find the optimal solution in polynomial time and can optimize for multiple services simultaneously. Moreover, in the case of insufficient resources at servers, the algorithm allows a trade-off between blocking probability and total utility.

III. Utilitarian Server Selection

The goal of our *utilitarian server selection* algorithm is to provide the highest QoS for the greatest number of users [10] within the bounds of a total budget on transit traffic costs. Our framework unifies the objectives of several stakeholders and the QoS of the end users. In our approach, the stakeholders involved in service selection are as follows:

- Execution zones (EZ): These are the entities running the computational infrastructure hosting the distributed application. Note that in the remainder of the paper the term server is also used as an equivalent of EZ.
- Application (Service) Providers: These are the organizations that run applications over a distributed set of execution zones.
- Internet Service Providers (ISP): These will implement resolution algorithms to resolve users queries, mapping service names to locators. The algorithms as described in this paper are candidates for deployment in ISP resolvers that will trade-off the QoS of their users with the costs incurred by inter-domain transit traffic.

Given these stakeholders, we next present an overview of the server selection process and the role of the utility function in the optimization algorithm.

A. Illustrative Example

We have observed that for several services, *the user perception of QoS does not improve further beyond a certain threshold*. For instance, people do not perceive any degradation in the quality of voice services if the *latency is equal to or less than* 20 *ms* [11]. For web services, a response time of 100 ms (including network latency and processing time) is the limit for users to perceive that the system is reacting instantaneously [12]. For video streaming no further improvement in quality will be achieved by increasing the bandwidth beyond the maximum bit rate of the video stream. Based on these observations, we illustrate the capped nature of the utility function by means of a simple example in Fig. 1. There are two users requiring a voice service which is available in both EZ_1 and EZ_2. However, each EZ has capacity to serve only one user at a time. Assuming there is sufficient bandwidth on all links, the classical closest-based algorithms [7], [8], [13] would minimize average latency and result in

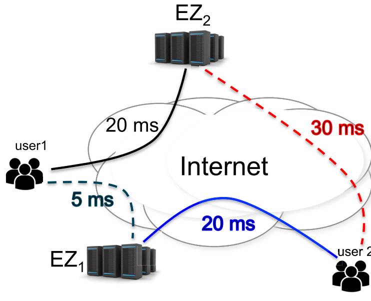

Fig. 1. Utility based vs. closest based selection.

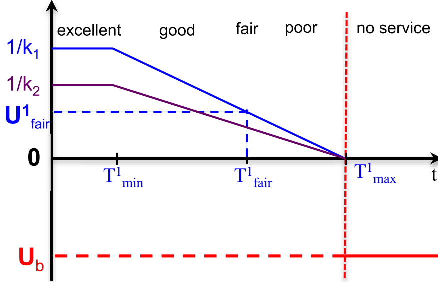

Fig. 2. Utility function vs. latency.

(user $1 \to EZ_1$) and (user $2 \to EZ_2$) (shown with dashed lines) with an average latency of $(5+30)/2 = 17.5$ ms. This means user 1 will experience excellent quality while user 2 experiences degradation (compared to the threshold of 20 ms for voice services). However, by using a utility-based selection algorithm, a better solution can be achieved (user $1 \to EZ_2$) and (user $2 \to EZ_1$) where *both users receive the highest QoS, with a latency of* 20 *ms*. It should be noted that this is a simple example with only two users and two EZs, for which the solution is trivial. In a real-world scenario with thousands of users and EZs the optimal solution is non-trivial, which the algorithms proposed in this paper aim to solve with both centralized and distributed versions.

B. Utility Function

Our general utility function is based on practical research on quality of service utility [14], [15] and prior investigations into mean opinion scores [16]. Our interval data points map to user ratings of *excellent*, *good*, *fair*, *poor* and *no service or blocked* (Fig. 2) [17].

Using our utility framework, application providers specify the utility function by two thresholds: T_{min} and T_{max}. Note that utility is not restricted to only latency, in future work, we will extend the utility function to be a combination of several QoS metrics such as latency, bandwidth, loss, etc. As shown in Fig. 2, we use a non-increasing piecewise linear utility function that is characterized by:

- We use $k \geq 1$ to set user priority. Two users of the same service are shown in Fig. 2. The user with lower k ($1 \leq k_1 < k_2$) gets a higher utility for the same latency value. As a result, the algorithm gives higher priority for users with low k to connect to closer EZs.
- If $t \leq T_{min}$: depending on the service type, an appropriate value of T_{min} is selected meaning that even if the latency is below this value, the improvement is not perceived by the users of that service, thus the utility is unchanged ($U_{max} = 1/k$).
- If $T_{min} < t \leq T_{max}$: QoS is within an acceptable range ($0 \leq U < 1/k$). User satisfaction reduces as the latency increases. We also define an optional parameter $T_{fair} \in [T_{min}, T_{max}]$ as the point from which users are aware of reduced performance but it is still within an acceptable range. Note that T_{fair} is used to simply qualify where is the point that has a fair QoS, and it does not change the slope of the utility graph which is only affected by T_{min}, T_{max} and k.
- If $T_{max} < t$: the request is *blocked* (*no service*) because the latency is beyond the acceptable range. More details on blocked requests are presented in Section IV-A.

Based on this utility function, the *utilitarian server selection* solution for the problem in Fig. 1 will be (user $1 \to EZ_2$) and (user $2 \to EZ_1$) where both users receive maximum utility U_{max} with $t = T_{min} = 20$ms.

IV. Centralized Optimization

We use linear programming (LP) to formulate the server selection problem which maximizes the total utility of all users while taking into account constraints on the data transit cost and the load balancing strategy. A central prerequisite for our model is the existence of a forecasting demand component that provides an input to the optimization algorithm. Although client demand varies with time, work in the literature [7] points to a reasonably stable demand within 10 minutes intervals. Note that, we aggregate individual users with the same preference to form a group. This can be done, for example, according to users' postal codes [7] or by users' IP prefixes [8]. Aggregation of this kind reduces the quantity of input variables for the optimization and also stabilizes request rates per-group [7]. For simplicity, we use the term "user i" to represent "user group i".

First, we define the utility function for each pair of (user i, service j) described in Section III-B as follows:

$$u_{ij} = \begin{cases} 0 < \dfrac{1}{k_{ij}} \leq 1 & \text{if } t_{ij} \leq T_{min}^{ij} \\ \dfrac{-t_{ij} + T_{max}^{ij}}{k_{ij}(T_{max}^{ij} - T_{min}^{ij})} \geq 0 & \text{if } T_{min}^{ij} < t_{ij} \leq T_{max}^{ij} \\ U_b < 0 & \text{otherwise} \end{cases}$$

A constant $k_{ij} \geq 1$ is used to indicate the priority level for each pair of user i and service j. By selecting values for k_{ij}, T_{min}^{ij} and T_{max}^{ij}, we can draw the utility graph as in Fig. 2. When the latency is larger than T_{max}^{ij}, the request is considered to be blocked. We set U_b to be a small negative value to indicate the utility of a blocked request. More details on how to set value for U_b are presented in Section IV-A.

Given the key notations in Table I, we use linear programming to formulate the utilitarian server selection problem.

$$\max[\sum_{(i,j)\in\mathcal{D}} u_{ij}] \tag{1}$$

$u_{ij} \leq 1$ is utility value of user i requesting service j. The objective function (1) is to maximize the total utility over all

TABLE I
KEY NOTATIONS (IN ALPHABETICAL ORDER)

b_{ij}	bandwidth required by user i for service j
$COST$	the maximum transit cost (budget)
c_{iz}	unit transit cost between user i and EZ z
$\mathcal{D}$	set of user requests $\mathcal{D} = \{(i,j), \forall i \in \mathcal{I}, j \in \mathcal{J}\}$
d_{ij}	number of session slots of service j requested by user i
(i,j,z)	user i, service j and EZ z
$\mathcal{I}$	set of user $\mathcal{I} = \{i\}$
$\mathcal{J}$	set of services $\mathcal{J} = \{j\}$
k_{ij}	important parameter of a pair (user, service)
l^j_{iz}	latency between user i and EZ z for service j
MMSC	Maximizing the minimum spare capacity
S^j_z	quantity of available session slots for service j at EZ z
t_{ij}	average latency for user i to access service j
U_b	utility value of a blocked user
u_{ij}	utility of user i when receiving service j
x^j_{iz}	fraction of user group i receiving service j from EZ z
y_{ij}	variable used to compute utility
$\mathcal{Z}$	set of execution zones (EZ) $\mathcal{Z} = \{z\}$

users. Given the objective, we add the following constraints to the formulation.

$$\sum_{z \in \mathcal{Z}} x^j_{iz} = 1 \quad \forall (i,j) \in \mathcal{D} \tag{2}$$

$x^j_{iz} \leq 1$ is the fraction of user i receiving service j from EZ z. Constraint (2) ensures that all requests of the user i for the service j have to be served by at least one EZ.

$$\sum_{i \in \mathcal{I}} d_{ij} x^j_{iz} \leq S^j_z \quad \forall j \in \mathcal{J}, z \in \mathcal{Z} \tag{3}$$

S^j_z is the maximum number of session slots of service j which can be served by EZ z. We use session slot as a unit of measurement representing how many user requests can be accommodated simultaneously. d_{ij} is the demand volume of user i for service j. The capacity constraint (3) guarantees the available session slots of z is sufficient to serve user requests.

$$t_{ij} = \sum_{z \in \mathcal{Z}} l^j_{iz} x^j_{iz} \quad \forall (i,j) \in \mathcal{D} \tag{4}$$

l^j_{iz} is the network latency for user i to get service j from EZ z. Hence t_{ij} in equation (4) is the average latency for the user i to get the service j. We consider a full mesh connection between users and EZs. We then remove all pairs of (user i and EZ z) if the latency $l^j_{iz} > T^{ij}_{max}$. This step guarantees that, even without adding explicit constraints in the LP model, the latency for any user i to connect to any EZ z to get service j is always less than or equal to T^{ij}_{max}.

$$y_{ij} \geq 0 \quad \forall (i,j) \in \mathcal{D} \tag{5}$$

$$y_{ij} \geq t_{ij} - T^{ij}_{min} \quad \forall (i,j) \in \mathcal{D} \tag{6}$$

y_{ij} is a temporary variable used to compute the utility function. Constraint (5) - (6) ensure that $y_{ij} \geq 0$ if $t_{ij} \leq T^{ij}_{min}$, otherwise $y_{ij} \geq t_{ij} - T^{ij}_{min} > 0$.

$$u_{ij} = \frac{T^{ij}_{max} - T^{ij}_{min} - y_{ij}}{k_{ij}(T^{ij}_{max} - T^{ij}_{min})} \quad \forall (i,j) \in \mathcal{D} \tag{7}$$

Equation (7) is used to model the utility function u_{ij} defined in Section III-B. There are two possibilities:

- If $t_{ij} \leq T^{ij}_{min}$, based on constraints (5) - (6), y_{ij} can be any values that are greater or equal to 0, however, due to the objective function maximizing the total utility, the minimum value of y_{ij} should be chosen. In other words, y_{ij} is set to 0 and thus, $u_{ij} = \frac{1}{k_{ij}}$ (the maximum utility, when $t_{ij} \leq T^{ij}_{min}$).

- If $t_{ij} > T^{ij}_{min}$, based on constraints (5) - (6), the formulation will set $y_{ij} = t_{ij} - T^{ij}_{min}$ and thus $u_{ij} = \frac{-t_{ij} + T^{ij}_{max}}{k_{ij}(T^{ij}_{max} - T^{ij}_{min})}$.

$$\sum_{z \in \mathcal{Z}} \sum_{(i,j) \in \mathcal{D}} c_{iz} b_{ij} x^j_{iz} \leq COST \tag{8}$$

$$x^j_{iz} \in [0,1], u_{ij} \leq 1 \quad \forall (i,j) \in \mathcal{D}, z \in \mathcal{Z} \tag{9}$$

Constraint (8) limits the data transit cost. As shown in [8], [13], the linear transit cost we use here is also a good approximation to the 95-th percentile transit cost.

The optimization formulation presented above is a pure linear programming model as there are no integer variables as constraints (9); hence it can be *solved efficiently in polynomial time*. The number of variables x^j_{iz} in the LP problem is $|\mathcal{I}| \times |\mathcal{Z}| \times |\mathcal{J}|$ where $|\mathcal{I}|$ is the number of users, $|\mathcal{Z}|$ is the number of EZs and $|\mathcal{J}|$ is the number of service types. Since $|\mathcal{Z}|$ and $|\mathcal{J}|$ are usually much smaller than $|\mathcal{I}|$, the worst case complexity of the LP problem is $O(|\mathcal{I}|^{3.5})$ [8]. We report the execution time of the algorithm in Section VI-C.

A. Blocked User Requests

When there are insufficient resources (either due to reaching the EZ capacity or transit cost budget) constraints (3) and (8) can be violated resulting in no feasible solution. We relax this by allowing user requests to be *blocked* when there are insufficient resources. To model this, we define a *virtual EZ* which has a large capacity such that the constraint (3) can never be violated. The transit cost between users and the virtual EZ is zero. The latency between all users to this virtual EZ is set at a value which is larger than T_{max}, therefore the utility for a blocked request is $U_b < 0$ (Fig. 2). We evaluate different values of U_b and show its impact on the number of requests to be blocked (Section VI-A.2). Intuitively, the closer to 0 the value of U_b is, the greater the probability for requests to be blocked due to a smaller difference in utility between a request at T_{max} ($U_{T_{max}} = 0$) and a blocked one ($U_b < 0$). By using a virtual EZ the LP always finds a feasible solution because the constraints (3) and (8) cannot be violated, but the total utility could be extremely small, or negative. *The requests that have been assigned to the virtual EZ are considered to be blocked*.

B. Load Balancing

The linear program can be adapted to support load balancing between EZs, which we name *maximizing the minimum spare capacity (MMSC)* strategy. Spare capacity is the available capacity at an EZ, specified as a percentage of the total capacity: an EZ with a capacity of 100 slots where 70 slots have been used, has a spare capacity of 30%. Assume that there are two EZs, both with a capacity of 100 slots, and that there are two possible solutions. In solution 1, EZ_1 and EZ_2

both have a spare capacity of 30%. In solution 2, EZ_1 and EZ_2 have spare capacities of 20% and 40%. The minimum spare capacity in solution 1 is 30% while in solution 2 it is 20%. As the MMSC strategy tries to *maximize the minimum spare capacity over all possible solutions*, solution 1 will be chosen.

Because the MMSC algorithm tries to spread the load over all EZs, there is a trade-off between load balancing and the perceived utility. We present here a strategy that sets priority on performing load balancing first, and optimizing the total utility in the next step.

1) Step 1: Minimizing Blocking Probability: As the virtual EZ has very large capacity, the MMSC algorithm first *minimizes the blocking probability*, otherwise all requests would be forwarded to the virtual EZ (as this solution would maximize the minimum spare capacity). To do so, we replace the objective function (1) with:

$$Q_{MIN} = \min \sum_{(i,j)\in\mathcal{D}} d_{ij} x_{iz}^{j} \quad z = virtualEZ \tag{10}$$

to minimize the number of requests going to the virtual EZ, i.e. blocked.

2) Step 2: Maximizing the Minimum Spare Capacity: From the set of feasible server selection solutions in *Step 1*, we choose the one in which the minimum spare capacity is maximized. To implement this, we replace the objective function (10) in *Step 1* with:

$$S_{MAX} = \max \quad S_{min} \tag{11}$$

and add the following constraints:

$$s_z = 1 - \sum_{(i,j)\in\mathcal{D}} d_{ij} x_{iz}^{j} / \sum_{j\in\mathcal{J}} C_z^j \quad \forall z \in \mathcal{Z} \tag{12}$$

$$S_{min} \le s_z \quad \forall z \in \mathcal{Z} \tag{13}$$

$$\sum_{(i,j)\in\mathcal{D}} d_{ij} x_{iz}^{j} \le Q_{MIN} \quad z = virtualEZ \tag{14}$$

Equation (12) is used to compute the spare capacity (i.e. $1 - total\ load$) of each EZ, and the minimum spare capacity S_{min} is calculated by constraint (13). In constraint (14), Q_{MIN} is the minimum blocking probability found in the objective function in *Step 1*. The objective function (11) maximizes the minimum spare capacity over all EZs.

3) Step 3: Maximizing the Total Utility: In this step, we remove constraint (13) and add the following:

$$s_z \ge S_{MAX} \quad \forall z \in \mathcal{Z} \tag{15}$$

where S_{MAX} is a constant and is found from (11). To maximize the total utility, we use a model with objective function (1) and constraints (2)-(9), (12), (14), (15). We show that this MMSC strategy has several advantages over the simple distributed server selection model in Section VI-C.

V. Distributed Model

Although the centralized optimization model can be solved in polynomial time, it is impractical in real deployments as a single global resolver would be required to collect information from all EZs and networks and would also handle all resolution requests from all users. Designing an efficient distributed algorithm is a classical problem [7], [13], [18], which needs to satisfy the following general requirements:

1) *Low overhead:* a small number of control messages should be exchanged.
2) *Convergence:* the algorithm should always converge to a stable solution.
3) *Efficiency:* the solution of the distributed algorithm is close to the centralized one.

Existing approaches in the literature can be used to help satisfy requirements (2) and (3) by using optimization decomposition methods [7], sub-gradient methods [18] or alternating direction method of multipliers [13]. However, they result with high complexity formulations and require high control overhead. Potentially, control messages can be exchanged in both direction between: resolvers - resolvers, resolvers - EZs, and EZs - EZs. In this work, we propose a novel distributed model satisfying all the three aforementioned requirements. Compared with existing work, our model is simpler (polynomial time solvable) and low overhead (only one-way control messages from EZs to resolvers are needed). In addition, the messages exchanged are simple, as we describe later.

TABLE II
KEY NOTATIONS IN DISTRIBUTED ALGORITHM

$A_i^z(k)$	# slots used by R_i from EZ z at epoch k
C^z	capacity (total session slots) at EZ z
k	epoch (iteration) number
M	# resolvers that share an EZ
N	# EZs that one resolver can see in its visibility set
R_i	resolver i ($0 \le i < M$)
$S_i^z(k)$	# slots can be seen by R_i from EZ z at epoch k
Z	set of execution zones

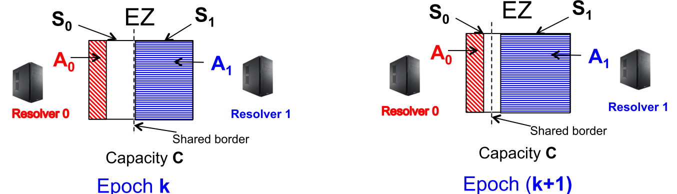

Fig. 3. Example: EZ z is shared by two resolvers R_0 and R_1.

A. Distributed Algorithm

We divide the time into **intervals** in which we assume the traffic demand is unchanged (e.g. 10 minutes as observed in [7]). Each interval is sub-divided into **epochs** and the distributed algorithm is run at the beginning of each epoch. We call the subset of EZs that are closest to a resolver and can be seen by that resolver the *visibility set*.

We introduce the notations used in the distributed algorithm in Table II. Considering an EZ z with total available session slots C^z which are shared by M resolvers. A resolver i can use at most S_i slots from the EZ z ($S_i \le C^z$). A_i is the actual slots used by the resolver i ($A_i \le S_i \le C^z$). A visualization of those notations are shown in Fig. 3.

At an epoch $k \ge 0$, let resolver $R_i(0 \le i \le M-1)$ see $S_i^z(k) \le C^z$ session slots from the shared EZ. To guarantee the capacity constraint, we have $\sum_{i=0}^{M-1} S_i^z(k) \le C^z$. Let $A_i^z(k) \le S_i^z(k)$ be the number of session slots that the resolver

R_i allocates for its users to connect to EZ z at epoch k. The algorithm, step-by-step, at each resolver is then as follows:

1) *At the beginning of each interval*: collect the estimated user demand and network metrics. We assume that these values do not change during an interval.
2) *At the beginning of each epoch*: each EZ announces the latest capacity (C^z) and the total-in-allocation session slots ($\sum_{i=0}^{M-1} A_i^z(k)$) to all resolvers that share it.
3) Each resolver updates available session slots that it can see ($S_i^z(k+1)$) in the next epoch based on the information received from the EZs in its visibility set:

$$S_i^z(k+1) = A_i^z(k)\left[1 + \frac{C^z - \sum_{i=0}^{M-1} A_i^z(k)}{\sum_{i=0}^{M-1} A_i^z(k)}\right] \quad (16)$$

   if $\sum_{i=0}^{M-1} A_i^z(k) = 0$, we set $S_i^z(k+1) = C^z$. In other words, resolver i will see full capacity of the EZ z if no other resolvers send request to that EZ.
4) Given new available session slots from EZs in step (3), the resolvers execute the linear program in Section IV (*polynomial execution time*) to find server selection solutions for their local users.

By using the equation (16), we show that our algorithm satisfies all three of the previously mentioned requirements of a distributed algorithm:

- *Low overhead:* only *one-way messages from EZs to resolvers* are required: each EZ sends message containing its capacity (C^z) and the total in-use session slots ($\sum_{i=0}^{M-1} A_i^z(k)$) by all resolvers sharing it. Each resolver then uses this updated information and the local user demand and its used session slots in the previous epoch ($A_i^z(k)$) to find a new solution.
- *Convergence:* we show, both by mathematical proof (Appendix A) and simulations (Section VI-C.2) that local decisions always converge within a handful of iterations.
- *Efficiency:* each resolver uses the linear program in Section IV, thus it guarantees that the resolver can find the optimal solution based on its visibility set in polynomial time. We show by simulations (Section VI-C) and by mathematical analysis (Appendix B) that the distributed algorithm is efficient and is close to the centralized one when visibility set is large enough.

In addition, we show that the equation (16) also achieves the *fair share on demand* requirement. As shown in Fig. 3, at the epoch k, the resolver R_0 just uses a small fraction of its shared available session slots ($A_0 < S_0$) while R_1 requires all the slots that it can see ($A_1 = S_1$). Therefore, in the epoch $(k+1)$, we should move the shared border to the left (but do not touch the red area - the allocated slots of R_0) so that there will be more free space for R_1 to forward its requests to the EZ if needed. This can be done automatically by using equation (16) (see the example in V-B).

Initially, when services are first started, each EZ announces its available session slots to all resolvers that can see it. Given the available session slots and the local user demand, each resolver executes the linear formulation in section IV to find a solution. In this initial step, some EZs can be overloaded as they are shared by many resolvers, but there is no message between resolvers to say that. However, by using the equation (16) to update available capacity at EZs after each epoch, the capacity constraints are not violated after the initial step. We present a simple example to make the algorithm clear.

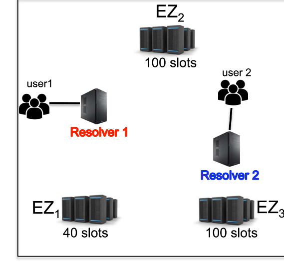

Fig. 4. Example of distributed algorithm.

B. Illustrative Example

Assume that user 1 and user 2 each requires 100 session slots. The capacities of the EZs are as shown in Fig. 4. The latencies between resolvers, users and EZs are as follows:

- $l(R_1, EZ_2) < l(R_1, EZ_1) < l(R_1, EZ_3)$
- $l(R_2, EZ_3) < l(R_2, EZ_2) < l(R_2, EZ_1)$
- $T_{min} < l(u_1, EZ_2) < l(u_1, EZ_1) < l(u_1, EZ_3) \leq T_{max}$
- $T_{min} < l(u_2, EZ_2) < l(u_2, EZ_3) < l(u_2, EZ_1) \leq T_{max}$

Recall that depending on the *visibility set* size, we can have different solutions to the server selection problem. Using the above network metrics, we consider two scenarios:

- Scenario 1 (*visibility set size is* 1): resolver 1 can only see EZ_2 (as EZ_2 is the closest EZ of R_1) and resolver 2 can only see EZ_3. Therefore, the solution will be: resolver 1 sends all 100 requests to EZ_2 and similarly, all requests of resolver 2 go to EZ_3. This solution does not change (stable solution) if the user requests are unchanged.

- Scenario 2 (*visibility set size is* 2): resolver 1 can see (EZ_1 and EZ_2) and resolver 2 can see (EZ_2 and EZ_3). Assume that the requests do not change, we present results for each resolver within 2 epochs (or 2 iterations of the distributed algorithm).

- Epoch $k = 0$:
  - Resolver 1 sees from EZ_1: $S_1^1(0) = 40$, and from EZ_2: $S_1^2(0) = 100$. As $l(usr_1, EZ_2) < l(usr_1, EZ_1)$, it forwards all 100 requests to EZ_2.
  - Resolver 2 sees from EZ_2: $S_2^2(0) = 100$, and from EZ_3: $S_2^3(0) = 100$. As $l(usr_2, EZ_2) < l(usr_2, EZ_3)$, it assigns all 100 requests to EZ_2.

  The total allocated session slots at EZ_2 is 200, and the EZ_2 is overloaded at epoch 0.
- Epoch $k = 1$:
  - Resolver 1 is updated with the current available slots it can see using the equation (16):
    - $S_1^1(1) = C^1 = 40$ (as $A_1^1(0) + A_2^1(0) = 0$)
    - $S_1^2(1) = 100 \times (1 + \frac{100-200}{200}) = 50$

  Solution after epoch 1 is: 40 requests go to EZ_1; 50 requests go to EZ_2 and 10 requests are blocked (as there are insufficient session slots).
  - Resolver 2 is updated with the current available slots it can see using the equation (16):
    - $S_2^3(1) = C^3 = 100$ (as $A_1^3(0) + A_2^3(0) = 0$)
    - $S_2^2(1) = 100 \times (1 + \frac{100-200}{200}) = 50$

  Solution after epoch 2 is: 50 requests go to EZ_2; 50 requests go to EZ_3 and no requests are blocked.

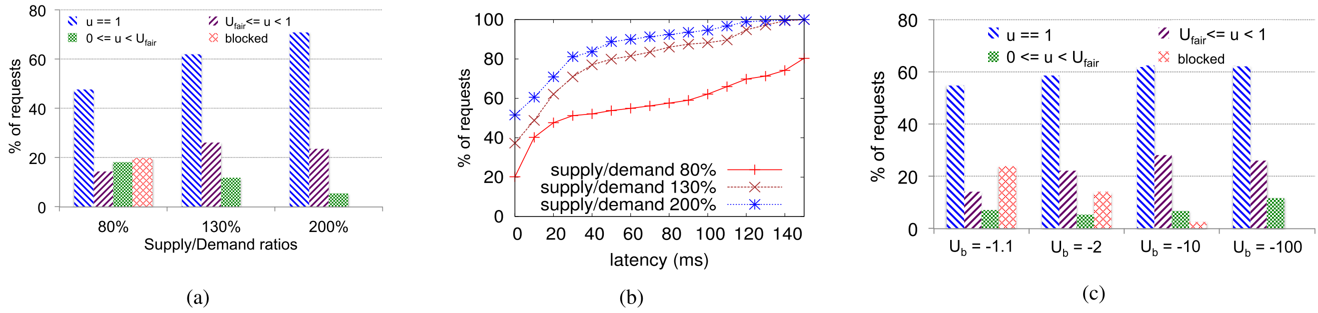

Fig. 5. Different values of $\frac{supply}{demand}$ ratios and U_b. (a) Utility with $\frac{supply}{demand}$ ratios. (b) CDF latency with $\frac{supply}{demand}$ ratios. (c) Utility with different values of U_b.

It is clear that, after epoch 1, due to equation (16), no EZs are overloaded. In this example, the solution does not change after 2 epochs provided that the user demands do not change. This means that the distributed algorithm *converges to a stable solution*. On the other hand, session slots are assigned proportionally to the requirement of each resolver. For instance, in the stable solution, R_1 and R_2 both can use 50 slots (fair share) from EZ_2. This is because in epoch 0, both R_1 and R_2 require 100 slots but EZ_2 only has a capacity of 100 slots, and the share ratio between R_1 and R_2 will be $\frac{100}{100}$ and each resolver uses 50 slots. We call this feature *fair share on demand*.

Another observation is that, because resolvers do not talk together and each of them greedily grabs available session slots, 10 requests from resolver 1 are blocked. We show that by using the *maximizing the minimum spare capacity* (MMSC) strategy, we reduce the greediness at resolvers and obtain a better solution. With the above example, we show a solution with MMSC strategy as follows:

- Epoch 0:
  - Resolver 1 sees from EZ_1: $S_1^1(0) = 40$, and from EZ_2: $S_1^2(0) = 100$ and assigns 28 requests to EZ_1 and 72 requests to EZ_2 (because the spare capacity at EZ_1 and EZ_2 is $\frac{40-28}{40} = 30\%$ and $\frac{100-72}{100} = 28\%$, respectively).
  - Resolver 2 sees from EZ_2: $S_2^2(0) = 100$, and from EZ_3: $S_2^3(0) = 100$ and assigns 50 requests to EZ_2 and 50 requests to EZ_3.
  The total allocated session slots at EZ_2 is 122, and the EZ_2 is overloaded at the initial step.
- Epoch 1:
  - Resolver 1 is updated with the current available slots it can see using the equation (16):
    - $S_1^1(1) = 28 \times (1 + \frac{40-28}{28}) = 40$
    - $S_1^2(1) = 72 \times (1 + \frac{100-122}{122}) = 59$
  The solution after epoch 1 is: 40 requests go to EZ_1; 59 requests go to EZ_2 and 1 request is blocked.
  - Resolver 2 is updated with the current available slots it can see using the equation (16):
    - $S_2^3(1) = 50 \times (1 + \frac{100-50}{50}) = 100$
    - $S_2^2(1) = 50 \times (1 + \frac{100-122}{122}) = 41$
  The solution after epoch 1 is: 29 requests go to EZ_2; 71 requests go to EZ_3 and zero requests are blocked.

As a result, we reduce the number of requests blocked at resolver 1 from 10 to only 1 by using MMSC. We further evaluate the benefit of the MMSC strategy in section VI-C.

VI. Performance Evaluation

In this section, we present the results of extensive simulations of our algorithms operating on a large-scale network dataset. First, we evaluate the algorithms with different parameters: $\frac{supply}{demand}$ ratios, U_b on blocking probability and visibility set sizes for the distributed algorithm. Next, we compare our novel utilitarian server selection (USS) with classical closest-based and min cost-based server selection algorithms. Then, we evaluate the distributed algorithm with and without *maximizing the minimum spare capacity* (MMSC) strategies, and compare with the centralized one. Next, we show the impact of a mismatch between supply and demand on the server selection solution. And finally, we discuss the impact of inaccuracies in demand forecasting on our algorithm. We solve the linear program model using IBM CPLEX solver [19]. All computations were carried out on a computer equipped with a 3 GHz CPU and 8 GB RAM.

We use a dataset with 2508 data centers distributed in 656 cities around the world [20]. For the distributed model, we assume that each city has one resolver. Since data centers in a city are geographically close to one other, we group them as a single execution zone (EZ) whose capacity is proportional to the number of data centers in that city. We assume that the services are available in all EZs. As real transit costs are commercially sensitive and therefore difficult to obtain from ISPs, we have adopted a simple model for the transit costs to different EZs based on a snapshot of the actual Amazon EC2 regional charging model. The user demand is proportional to the population of each city [21]. Latency between users and execution zones are computed based on Haversine distance between two points around the planet's surface [22].

A. Parameters of the Algorithm

1) Supply/Demand Ratios: We first find server selection solutions for different $\frac{supply}{demand}$ ratios with the centralized algorithm - without *maximizing the minimum spare capacity (non-MMSC)*. We set $T_{min} = 20$ ms, $T_{fair} = 100$ ms, $T_{max} = 150$ ms and $k = 1$ for all pairs of (group user, service). In Fig. 5a - 5b, we show the utility and the cumulative distribution function (CDF) of latency for three $\frac{supply}{demand}$ ratio scenarios: 80%, 130% and 200%. $\frac{supply}{demand} = 80\%$ means that the total available capacity in all EZs is scaled down to equal to 80% of the total requests. As a result, 20% of the requests will be blocked while maximizing total utility of the served requests. In the CDF of latency in the 80% scenario (Fig. 5a), only 80% of requests receive service within less than T_{max}

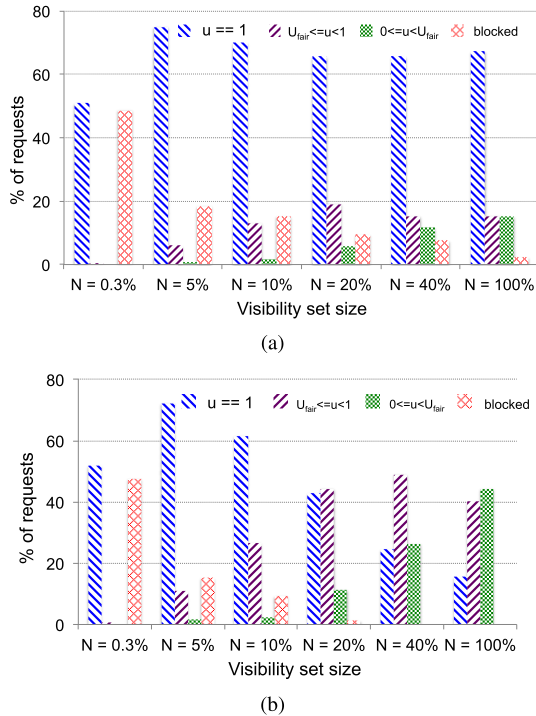

Fig. 6. Utility with different visibility set sizes. (a) non-MMSC strategy. (b) MMSC strategy.

and the remaining requests are blocked. For the two other scenarios (130% and 200%), since there are sufficient session slots, no user requests are blocked. Obviously, the greater the supply of session slots the better the solution (Fig. 5a - 5b).

2) Utility of a Blocked User: As explained in Section IV-A, our algorithm allows some user requests to be blocked while still maximizing the total utility. By selecting different values of utility for blocked requests ($U_b < 0$), we obtain solutions with different blocking probabilities. We show in Fig. 5c the results for the *centralized algorithm (non-MMSC)* with different values of U_b. When U_b is close to 0, e.g. $U_b = -1.1$ or $U_b = -2$, blocking user requests does not incur a large penalty in the total utility. A significant number of requests are blocked despite total utility being maximized. When U_b is much smaller (e.g. $U_b = -100$), blocking a single request can dramatically reduce the total utility, thus the algorithm tries to avoid as many requests being blocked as possible. In Fig. 5c, when $U_b = -100$, no user request is blocked.

In the remaining evaluation, if not stated otherwise, default values are used as follows: $\frac{supply}{demand}$ ratio = 130%, $U_b = -100$, $T_{min} = 20$ ms, $T_{fair} = 100$ ms, $T_{max} = 150$ ms, $k = 1$ and the max budget is 320.

3) Visibility Set Size: In a distributed manner, each resolver only sees its local user demand and the subset of EZs in the *visibility set*. We vary the size of the visibility set by changing parameter N, the percentage of the total 656 execution zones that can be seen by a resolver. For example, $N = 0.3\%$ means that each resolver can see its two closest EZs.

Intuitively, as N increases, more EZs and, hence, more session slots are available for a resolver to allocate user requests. As shown in Fig. 6, with both MMSC and non-MMSC strategies, the percentage of blocked requests reduces as we increase N. With the MMSC strategy, resolvers are less greedy in allocating high-QoS execution zones for their users. As a result, more session slots are available which reduces blocking probability for users in "poor resource" areas. Therefore, the MMSC strategy (Fig. 6b) performs better than the non-MMSC one (Fig. 6a). Note that in the centralized algorithm, there are no blocked user requests when the $\frac{supply}{demand}$ ratio is 130%.

B. USS vs. Closest and Min Cost Algorithms

1) USS vs. Closest Selection Algorithms: Given the parameters in VI-A.2, Figs. 7a - 7b compare the results of our USS algorithm versus the classical closest algorithm, as used in other work, for example [7]. The closest algorithm tries to allocate user requests to nearby EZs with available session slots; if the nearest EZ does not have available session slots, the algorithm considers the next closest and so on. Requests are only blocked in the case that there are no available session slots in any EZ within a T_{max} latency range. The latency of all user requests is then mapped to utility according to the utility function for voice services ($T_{min} = 20$ ms, $T_{fair} = 100$ ms and $T_{max} = 150$ ms [23]) (Fig. 7a). As can be seen in Figs. 7a - 7b, the USS algorithm performs better with less blocking probability. This is because the USS algorithm is less greedy, providing more flexibility for requests to connect to any of multiple servers within the T_{min} latency range that provide maximum utility. Taking a closer look at the CDF of latency (Fig. 7b), more requests receive low latency in the closest algorithm; however, more requests are also blocked, due to its greedy behavior.

2) USS vs. Min Cost Selection Algorithm: We show in Fig. 7c - 7d the comparison between the USS and the min cost server selection algorithms. The min cost algorithm uses a similar linear program formulation as the USS, but with an objective function of minimizing transit cost. Therefore, user requests being forwarded to EZs in the same domain is favored over those in remote domains, to reduce transit costs on inter-domain links. As a result, a large fraction of users receive low latency and, hence, high utility. However, for some domains, the supply is not sufficient for their users, therefore we can see around 15% of user requests being blocked. The USS algorithm is focused on maximizing total utility over all users and blocking probability is low.

C. Distributed Algorithm

We show in Fig. 8 the average utility score per successful request of the centralized and the distributed algorithms with different visibility set sizes.

Regarding execution time, the centralized algorithm with full knowledge of execution zones and user demands takes approximately 2 minutes to find an optimal solution, while the distributed algorithm only requires a few seconds to finish.

We first compare the MMSC ("cent. (MMSC)") and the non-MMSC ("cent. (non-MMSC)") strategies for the centralized algorithm in Fig. 8. The MMSC strategy maximizes the minimum spare capacity over all EZs, thereby distributing load more evenly. As a result, for the centralized algorithm, MMSC performs worse in terms of QoS in comparison with the non-MMSC strategy which only tries to maximize total utility. However, as we show later, the MMSC strategy is useful in the distributed algorithm.

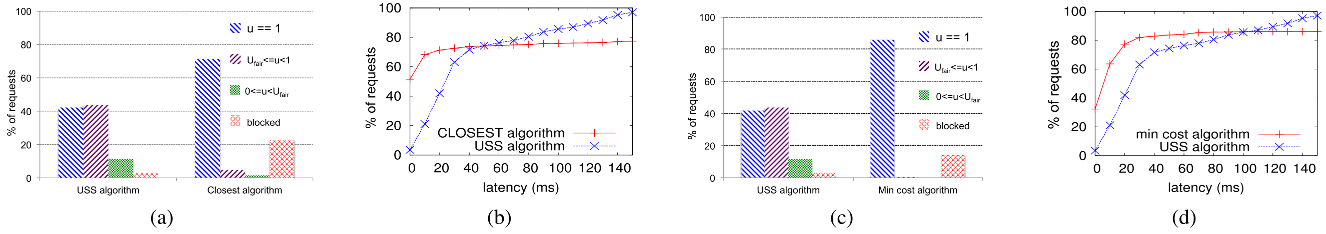

Fig. 7. Voice: USS vs. closest and min cost algorithms. (a) Utility - USS vs. closest alg. (b) Latency - USS vs. closest alg. (c) Utility - USS vs. mincost alg. (d) Latency - USS vs. mincost alg.

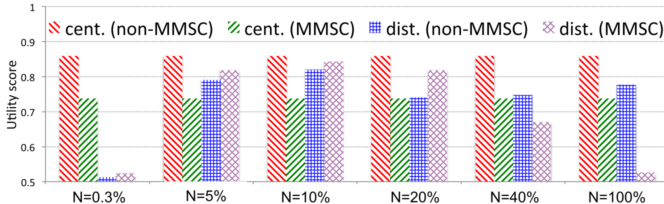

Fig. 8. Distributed vs. centralized algorithms.

1) MMSC vs. Non-MMSC: Fig. 6 shows that the MMSC strategy is better than non-MMSC in terms of blocking probability. In terms of utility score, the distributed MMSC algorithm improves QoS until $N = 10\%$ (Fig. 8). However, when $N > 10\%$ we see a reduction in utility score. This is due to two reasons. First, there is more resource contention as N increases (EZs are shared by more resolvers). Second, the MMSC strategy prioritizes reducing the load at execution zones over maximizing utility. As more EZs are visible to a resolver, requests are distributed over multiple EZs, which has the side effect of reducing user utility in some cases. On the other hand, the distributed non-MMSC algorithm tries to maximize user utility, thus QoS depends on the quantity of resources available to a resolver according to its visibility set size. In general, the larger the visibility set, the better the QoS the distributed non-MMSC can achieve. Note that when $N = 5 - 10\%$, the non-MMSC algorithm achieves a good overall utility score (Fig. 8) for non-blocked requests, however, as can be seen in Fig. 6, many requests are blocked due to insufficient resources.

2) Convergence of Distributed Algorithm:

Synchronous Mode: To evaluate the convergence of the *distributed algorithm (MMSC)*, Fig. 9a shows the quality of the solution after 10 epochs. After the first epoch, we already have a reasonably good solution and the quality of solution improves over the next epochs. After 4 epochs, the solution is close to the stable state, which is achieved after 8 epochs. These simulation results, along with the mathematical proof in Appendix A confirm that the distributed algorithm always converges to a stable state.

Asynchronous Mode: In this evaluation we investigate the situation where resolvers do not simultaneously receive session slot availability updates from the EZs. At each epoch, we randomly select 20% of the resolvers to use the old value of available session slots. Fig. 9b compared the CDF of load of EZs in synchronous and asynchronous modes using the distributed algorithm with MMSC strategy and visibility set size $N = 20\%$. As we have shown in Section V no EZs are overloaded after the initial epoch (epoch 0) with the synchronous mode of the distributed algorithm. In the asynchronous mode, however, we observe that 10% of EZs are overloaded at epoch 1 (the maximum load is 104% - Fig. 9b). This is a limitation of the distributed algorithm when working in asynchronous mode. However, as shown in Fig. 9b, the number of overloaded EZs is reduce in next epochs.

In Fig. 9c we examine the impact of asynchronous mode of operation on the latency experienced by the users. The synchronous mode of operations provides slightly better results with only 3.5% of session slots being blocked compared to 4.5% with the asynchronous mode. However, these results show that the algorithm can work well in a distributed asynchronous environment.

3) Load Balancing: We show in Fig. 10 the load of all EZs using the *distributed algorithm: MMSC vs. non-MMSC* with $N = 100\%$. The x-axis shows the "id" of the EZ: we have 656 EZs with "id" from 0 to 655. The y-axis is the load in percentage of EZ capacity (used session slots / capacity). Fig. 10a shows the load for the non-MMSC algorithm. EZs have a wide range of load (from 0% to 100%) as the objective of the non-MMSC algorithm is to maximize total utility rather than balance load. Figs. 10b - 10d show the results when applying the MMSC strategy after $1, 3$ and 5 epochs. It can be seen that the algorithm converges to a constant load across all EZs. Since we use $\frac{supply}{demand} = 130\%$, the algorithm should converge to a state where each EZ uses $\frac{100}{130} \simeq 78\%$ of their capacity as shown in Fig. 10d, after 5 epochs.

D. Mismatch Between Supply and Demand

To evaluate the impact of mismatch between supply and demand, we first run the *centralized model* (section IV) but without the capacity and the cost constraints in order to find the baseline capacity required at each EZ for an optimal server selection solution. Then we scale these values to achieve 130% $\frac{supply}{demand}$ ratio. We call this configuration the *perfect allocation*. Next, we create different levels of mismatch between supply and demand by varying a parameter "$X\%$ rand." (Fig. 11). This means that we reallocate $X\%$ of capacity, in terms of session slots, from each EZ to a common pool which is then distributed evenly across all EZs so that total capacity across the entire population of EZs remains the same under perfect allocation and "$X\%$ rand." scenarios. "0% rand." is equivalent to the perfect allocation while in "100% rand.", there is a uniform distribution of sessions slots between all EZs.

Fig. 11 shows evaluation results for the *distributed algorithm* with visibility set size $N = 5\%$ with different values for "$X\%$ rand.". Under perfect allocation ("0% rand."), the distributed algorithm performs well with no blocked requests. It is clear that as $X\%$ increases and the level of mismatch between localised supply and demand increases, more requests

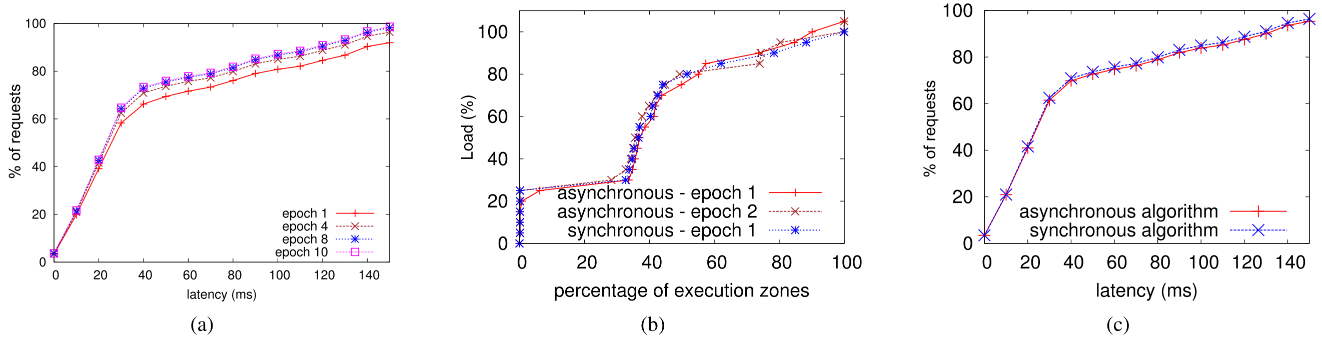

Fig. 9. Convergence of distributed algorithm. (a) Convergence of distributed algorithm. (b) Load - synchronous vs. asynchronous. (c) Latency - synchronous vs. asynchronous.

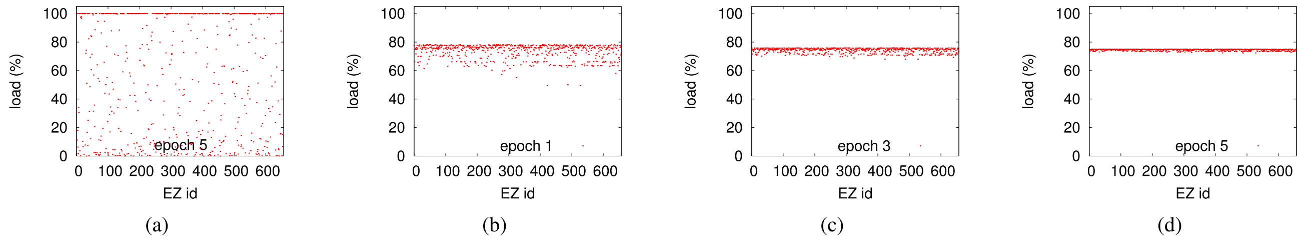

Fig. 10. Load at EZs for non-MMSC and MMSC strategies. (a) non-MMSC (epoch 5). (b) MMSC (epoch 1). (c) MMSC (epoch 3). (d) MMSC (epoch 5).

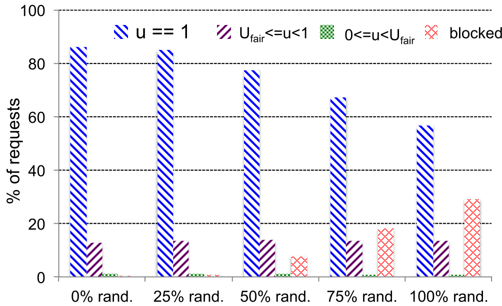

Fig. 11. Utility with different mismatch levels.

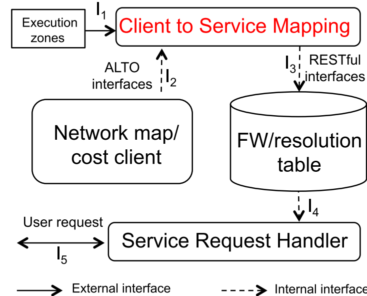

Fig. 12. Resolver architecture.

are blocked. It is noted that because we use a $130\% \ \frac{supply}{demand}$ ratio, the scenario “25% rand.” is within an acceptable range of mismatch and the solution is close to the “0% rand.” case.

E. Impact of Inaccuracy in Demand Forecast

A central prerequisite for our model is the existence of a forecasting demand component that provides input to the optimization algorithm. We discuss in this section the robustness of our solution to inaccuracy in forecasting demand. In Fig. 11 “0% rand.” represents the perfect match between forecasted demand and actual capacity with a visibility set size of $N = 5\%$. In this case there are no blocked requests; however, compared to Fig. 6b, with $N = 5\%$, there are around 17% blocked requests. This is between the cases of “50% rand.” and “75% rand.” in Fig. 11. This shows that our previous simulation environments were undertaken in range of $50\% - 75\%$ inaccuracy of the perfect resource allocation in EZs, or, put another way, the forecasting demand was $50\% - 75\%$ inaccurate. As shown in Fig. 6b, to achieve zero blocking when forecasted demand is $50\% - 75\%$ inaccurate we need a visibility set size of $N = 40\%$.

VII. Deployment Considerations

A. System Implementation

In this section, we describe the implementation of our optimization algorithm and its integration into a small-scale testbed[1] over the Internet to show the proof of concept trade-off between service utility and network cost in a real network. Fig. 12 gives an overview of the implementation architecture of the deployed resolver. The optimization algorithm is integrated in the “Client to Service Mapping” component. This component collects network state information (e.g. latency and transit cost) via interface I_2 which takes the form of IETF-ALTO interface according to RFC 7285. It also connects to execution zones via interface I_1 to retrieve the number of available session slots. The optimization algorithm is executed periodically and updates new entries (via RESTFUL web service interface I_3) to the FW (forwarding)/resolution table, which is read by the Service Request Handler. The Service Request Handler is responsible for handling client requests received on the I_5 interface. To this end, it accesses the

[1] http://www.fusion-project.eu/

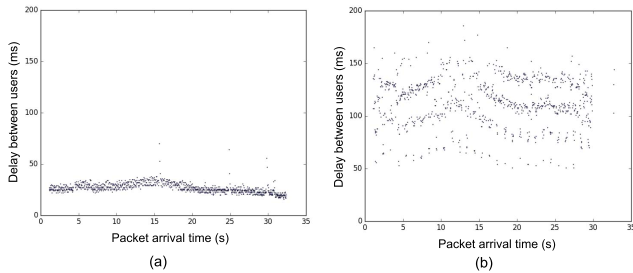

Fig. 13. Maximizing utility vs minimizing cost. (a) Maximizing utility. (b) Minimizing cost.

FW/resolution table using I_4 interface to find the appropriate server in which the client should connect. Further details on the implementation can be found in [24, Sec. 5].

1) System Setup: We set up two execution zones (EZs): (1) The Virtual Wall testbed of iMinds[2] located in Ghent, Belgium; and (2) The private network of Spinor[3] located in Munich, Germany. In this specific scenario, we have two users located in the Spinor network. We assume that the network cost between the users and the EZ in Spinor is high while it is cheaper for the users to connect the EZ in Virtual Wall. We deploy the Shark 3D application by Spinor on those EZs.

2) Experiment Results: The focus of this evaluation is on the latency experienced by users in a multi-user scenario. We collect the latency the users experience depending on the resolution result of the resolver. Note that we measure the latency at the Shark 3D application level which includes both network latency and processing time at the application level. In Fig. 13a, we show the latency recorded over 30 seconds of the experiment in case of maximizing the utility regardless the cost. As a result, both users experience low latency as they connect to the service deployed at Spinor which is close to them. On the other hand, when we try to minimize the cost (Fig. 13b), the user requests are resolved to use the service at the Virtual Wall which results in higher latency.

With this proof of copncept deployment we have shown the operation of the algorithm in a real-world testbed and that different policies of maximising utility and minimising cost results in different behavior of the resolver. Further experimental results can be found in [25].

B. Discussion on Real World Deployment

Utility based networking solutions have suffered in the past from difficult implementation road-maps, for example see [5], [6]. Our proposal has the potential for a much simpler deployment path compared to prior utility-based solutions since it only requires some changes in a single ISP. This can be achieved in two ways: through software defined networking (SDN) or with simple additions to the domain resolution system.

With SDN our algorithm can be deployed as part of a centralized controller within an ISP. If the controller is aware of the location of the replicas, it can redirect flows targeted to an anycast addresses to the desired EZ instance. This would work by allocating a particular anycast address to each service, as is commonly done today.

[2]http://doc.ilabt.iminds.be/ilabt-documentation/virtualwallfacility.html
[3]www.spinor.com

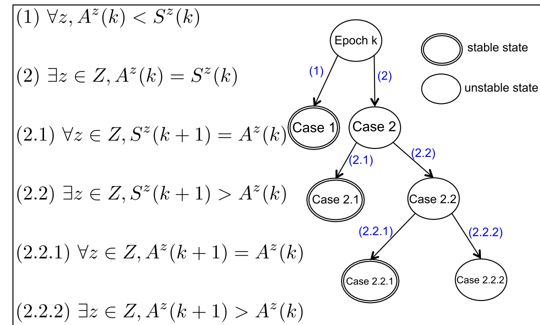

Fig. 14. Convergence of distributed algorithm.

Using name resolution is even easier as we have implemented in Section VII-A. Moreover, there has been a wide range of proposals to improve name resolution in the Internet. They can be broadly classified in three types [26]: indirection, name based routing and name resolution. The ideas in this paper can be applied to all of these with varying degrees of difficulty. Indirection proposals like [27] can use utilitarian server selection at redirection time whilst name based routing techniques such as [28]–[31] can do it at packet forwarding time. This will be straightforward for content based services. However, if the service is stateful and requires packets of the same flow to always reach the same server, both of these solutions need appropriate mechanisms to guarantee consistent resolution decisions for the duration of a session. Name resolution has also been the topic of several research papers [9], [32] and this is where utilitarian server selection can be applied with fewer changes needed to the Internet architecture, since no modifications of end systems or applications are required.

VIII. Conclusions and Future Work

This paper presented *utilitarian server selection*, a new method to implement service instance selection that can work both in centralized and distributed manner, with the aim of maximising user utility within an upper budget of transit network costs. These ideas can be deployed in several ways, such as through the use of SDN or through DNS, without requiring changes to client/server interfaces. As further work, we are planning to extend the utility function to support more QoS metrics. We are also implementing a working DNS server that implements our extensions.

Appendix A
Convergence of Distributed Algorithm

We present in this section a proof that the distributed algorithm always converges to a stable state. We show the proof by considering all possible cases that can happen over two consecutive epochs $(k) \rightarrow (k+1)$ of the distributed algorithm. We show that the algorithm will always converge to one of the stable states as shown Fig. 14. When the algorithm reaches a stable state the solution will not change over subsequent iterations unless the input data (latency, demand, etc.) is updated. Note that Fig. 14 shows two consecutive epochs k and $(k+1)$ and the transition between $(k+1)$ and $(k+2)$ can be seen as a new diagram of k' and $(k'+1)$ where $k' = k+1$, and so on.

Remark 1: $\mathbf{A_i^z(k) \leq S_i^z(k)}$ $\forall$ resolver i, epoch k, EZ z (see the notations in Table. II): each resolver cannot use more resources than it can see from an EZ (capacity constraint).

Theorem 1: $\sum_{\mathbf{i=0}}^{\mathbf{M-1}} \mathbf{A_i^z(k) \leq C^z} \quad \forall\ k \geq 1$

This theorem means that when M resolvers share an EZ, the total slots used by those M resolvers does not exceed the capacity C of the shared EZ. We prove this theorem based on the equation (16).

Proof: considering an EZ z which has capacity C and is shared by M resolvers, we present the distributed algorithm step-by-step as follows:

- *At epoch* $k = 0$, all M resolvers see C available slots from the shared EZ z (hereafter, we omit the notation z):

- At resolver R_0:

$$S_0(0) = C; A_0(0) \leq S_0(0) \text{ (as \textit{remark 1})}$$

- At resolver $R_i(0 < i \leq M-1)$:

$$S_i(0) = C; A_i(0) \leq S_i(0) \text{(as \textit{remark 1})}$$

The total session slots used by M resolvers at epoch 0 is:

$$\sum_{i=0}^{M-1} A_i(0) \leq \sum_{i=0}^{M-1} S_i(0) = M \times C \geq C$$

Therefore, it can make the EZ overloaded at epoch 0.

- *At epoch* $k = 1$:

- Resolver R_0 updates new available slots it can see at epoch $k = 1$ using equation (16):

$$S_0(1) = A_0(0) + \frac{A_0(0)[C - \sum_{i=0}^{M-1} A_i(0)]}{\sum_{i=0}^{M-1} A_i(0)};$$
$$A_0(1) \leq S_0(1) \text{ (as\textit{remark 1})}$$

- Resolver $R_i(0 < i \leq M-1)$ updates new available slots it can see at epoch $k = 1$ using equation (16):

$$S_i(1) = A_i(0) + \frac{A_i(0)[C - \sum_{i=0}^{M-1} A_i(0)]}{\sum_{i=0}^{M-1} A_i(0)};$$
$$A_i(1) \leq S_i(1) \text{ (as \textit{remark 1})}$$

By summing the left and the right hand sides of the above equations, we have the total session slots of EZ z seen by M resolvers at epoch $k = 1$:

$$\sum_{i=0}^{M-1} S_i(1) = \sum_{i=0}^{M-1} A_i(0) + \sum_{i=0}^{M-1} A_i(0) \frac{C - \sum_{i=0}^{M-1} A_i(0)}{\sum_{i=0}^{M-1} A_i(0)} = C$$

Therefore, the total session slots allocated by M resolvers at epoch 1 will be:

$$\sum_{i=0}^{M-1} A_i(1) \leq \sum_{i=0}^{M-1} S_i(1) = C$$

and there is no overload at the EZ.

- *At epoch* $k > 1$:

- Resolver R_0 updates new available slots it can see at epoch $k > 1$ using the equation (16):

$$S_0(k) = A_0(k-1) + \frac{A_0(k-1)[C - \sum_{i=0}^{M-1} A_i(k-1)]}{\sum_{i=0}^{M-1} A_i(k-1)};$$
$$A_0(k) \leq S_0(k)$$

- Resolver $R_i(0 < i \leq M-1)$ updates new available slots it can see at epoch $k > 1$ using the equation (16):

$$S_i(k) = A_i(k-1) + \frac{A_i(k-1)[C - \sum_{i=0}^{M-1} A_i(k-1)]}{\sum_{i=0}^{M-1} A_i(k-1)};$$
$$A_i(k) \leq S_i(k)$$

Similarly, by summing the left and the right hand sides of the above equations:

$$\sum_{i=0}^{M-1} S_i(k) = C \quad \forall\ k \geq 1$$

and:

$$\sum_{i=0}^{M-1} A_i(k) \leq \sum_{i=0}^{M-1} S_i(k) = C \quad \forall\ k \geq 1 \qquad \blacksquare$$

Theorem 2: $\mathbf{S_i(k) \geq A_i(k-1)}$ $\forall i \in [0, M-1], k > 1$.

The theorem says that the available slots that resolver R_i can see in the next epoch should be greater or equal to the amount that R_i has used in the current epoch. That is to say, at epoch $(k+1)$ the shared border can shift to the left but does not cross the used (red) area of R_0 (Fig. 3).

Proof: recall that resolver $R_i(0 \leq i \leq M-1)$ updates new available slots it can see at epoch k using the equation (16):

$$S_i(k) = A_i(k-1) + \frac{A_i(k-1)[C - \sum_{i=0}^{M-1} A_i(k-1)]}{\sum_{i=0}^{M-1} A_i(k-1)} \quad (17)$$

As *theorem 1*, $C - \sum_{i=0}^{M-1} A_i(k-1) \geq 0\ \forall\ k > 1$, thus:

$$\frac{A_i(k-1)[C - \sum_{i=0}^{M-1} A_i(k-1)]}{\sum_{i=0}^{M-1} A_i(k-1)} \geq 0 \quad (18)$$

From (17) and (18) we have:

$$S_i(k) \geq A_i(k-1)\ \forall\ i \in [0, M-1], k > 1 \qquad \blacksquare$$

We now show the proof of the distributed algorithm's convergence based on *remark 1, theorem 1* and *theorem 2*. Considering inside one visibility set (Fig. 15), the resolver i can see N EZs. At epoch $k > 0$, let $S^z(k) \leq C^z$ ($z \in [0, N-1]$) be the available session slots that the resolver i can see from EZ z (Fig. 15a). R_i cannot use all capacity of those EZs because they are also shared by other resolvers. Based on *remark 1*, we have $A^z(k) \leq S^z(k)\ \forall z \in [0, N-1]$. We consider 2 possible cases:

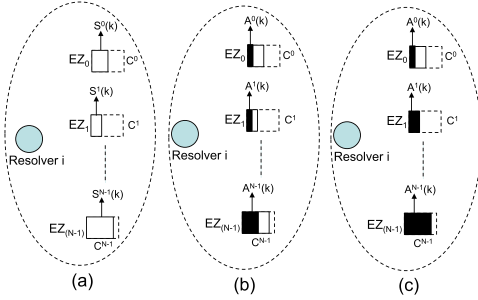

Fig. 15. Resource allocation at each visibility set.

Case 1: $A^z(k) < S^z(k)\ \forall z \in [0, N-1]$ (*strictly less than* - Fig. 15b): the available session slots at all EZs are strictly more than the requirement of the resolver. In the next epoch, the available slots of those EZs can increase or decrease. If it increases, the resolver sees more available slots but the solution does not change as increased capacity does not help to improve the utility because if it did, the *strictly less than* case will not happen. In case the available session slots reduces in the next epoch, we have $S^z(k+1) \geq A^z(k)\ \forall z \in [0, N-1]$ (*theorem 2*). Therefore, in the next epoch, the resolver will keep the same solution (it is still feasible) because there is no better solution. Thus, **case 1 is a stable state** (Fig. 14).

Case 2: $\exists\ z \in Z, A^z(k) = S^z(k)$ (Fig. 15c with $z = 1$ and $z = N-1$). As *theorem 2*, in the next epoch, we have $S^z(k+1) \geq A^z(k)$. We consider 2 cases:

- *Case 2.1*: $S^z(k+1) = A^z(k)\ \forall z \in Z$. That is, in Fig. 15c we have $S^1(k+1) = A^1(k) = S^1(k)$ and $S^{N-1}(k+1) = A^{N-1}(k) = S^{N-1}(k)$. This means that at epoch $(k+1)$, the available slots the resolver can see from EZ_1 and EZ_{N-1} do not change. This only happens when other resolvers who also share EZ_1 and EZ_{N-1} (not shown in Fig. 15) see that those EZs provide a good solution and they do not want to move requests to any other EZ. And this property is held over the next epochs unless the input data (e.g. latency, demand pattern, etc.) change. Therefore, the same (feasible) solution is kept as no other solution can improve the utility. That means **case 2.1 is a stable state** (Fig. 14).
- *Case 2.2*: $\exists\ z \in Z, S^z(k+1) > A^z(k)$. Because there are more slots at the EZs that the resolver may be interested in, solution in the next epoch can be changed. We consider 3 possible cases:
  - *Case 2.2.1*: $\forall\ z \in Z, A^z(k+1) = A^z(k)$. This means that increased capacity at EZs does not help to improve the utility. In other words, we cannot improve the current solution and **case 2.2.1 is a stable state** (Fig. 14).
  - *Case 2.2.2*: $\exists z \in Z, A^z(k+1) > A^z(k)$. The solution changes as new available session slots at EZs can give a better solution. Because the total number of user requests does not change, increasing allocation in some EZs means that the new solution needs to decrease the number of slots used in other EZs.
  - *Case 2.2.3*: $\exists z \in Z, A^z(k+1) < A^z(k)$. Even increasing available session slots, the number of slots used in those EZs can decrease because there is an increase of allocation in other EZs (note again, the total number of user requests does not change). We can see that this case is equivalent to the *case 2.2.2*: increasing slots used in some EZs, and decreasing slots used in other EZs. Therefore, we can consider only one *case 2.2.2* for both of them.

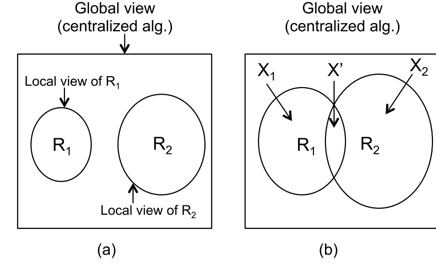

Fig. 16. Two scenarios of resolver R_1 and R_2. (a) Non-overlapping. (b) R_1 and R_2 are overlapped.

It is noted that *case 2.2.2* is not a stable state, meaning that resolver can change solution in the next epoch. From Fig. 14, we can observe that the distributed algorithm will not converge if and only if the algorithm is trapped in *case 2.2.2* forever. This means that there is a self loop or $A^z(k+1)$ has increased to infinity. The latter case cannot happen because the allocation is bounded as *theorem 1*. We consider an example: at epoch $k+1$, the solution increases the number of slots used at EZ_i and reduces some from EZ_j because this helps to improve the utility. The self loop will happen if at some points (e.g. at epoch $k' > k+1$), the solution increases the slot used at EZ_j and reduces some from EZ_i. This means that using more slots of EZ_j instead of EZ_i helps to improve the utility, which contradicts the first statement: using more slots at EZ_i and less at EZ_j can get better utility. Therefore, the self loop in *case 2.2.2* cannot happen. In other words, at some points, the distributed solution has to get out of *case 2.2.2* and as shown in Fig. 14, the distributed algorithm always converges to a stable solution. ■

APPENDIX B
COMPARISON BETWEEN THE RESULTS OF THE CENTRALIZED AND DISTRIBUTED ALGORITHMS

There are some remarks regarding the centralized and the distributed algorithms:

1) Both of them use the same linear programming formulation. The only difference is that each resolver in the distributed algorithm has only a partial view of the global dataset (user requests and EZs' capacity).
2) The distributed algorithm always converges to a stable state. In other words, each resolver will use a fixed amount of EZs' shared capacity in the stable state.

Based on those remarks, we show an analysis on the gap between the distributed and the centralized algorithms. We consider the case of two resolvers but this analysis can be easily extended to a general case of M resolvers. Assume that there are two resolvers R_1 and R_2, each has their own local view of resources depending on the visibility set size as shown in Fig. 16. Two scenarios can happen, as follows:

- In Fig. 16a, the views of the two resolvers do not overlap. If the visibility set size is large enough, the resolver can obtain the same global optimal solution as given by the centralized algorithm (remark 1). On the other hand, if the visibility set size is too small, many user requests will be blocked. This phenomenon can be observed in Fig. 6 when the visibility set size $N = 0.3\%$.

- If the views of the two resolvers overlap, let X_i be the EZ resources dedicated to R_i and X' be the shared resource between the two resolvers (Fig. 16b). Assume that $\alpha_i \in [0, 1]$ is the fraction of the shared resource occupied by resolver R_i in a stable state. As mentioned in remark 2, α_i is unchanged in the stable state. Thus the gap between the local (distributed algorithm) and the global (centralized algorithm) solutions is:

$$gap = \frac{X_i^*}{X_i + \alpha X'}$$

where X_i^* is the optimal view of resolver R_i where it can find a global optimal solution. As observed in Fig. 8, when the visibility set size $N \geq 20\%$, $(X_i + \alpha X')$ is large enough, and the gap between the distributed and the centralized algorithms is small. This gap is also small when $N = 5 - 10\%$ although many requests are blocked due to insufficient resources (Fig. 6). Note that the performance of the distributed algorithm (MMSC) is worse in term of utility score when $N > 10\%$ (Fig. 8). This is due to the effect of load balancing, as explained in section VI-C.1.

Acknowledgment

The authors would like to thank their colleagues from FUSION project for their help and support.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. MCC*, 2012, pp. 13–16.

[2] I. Poese *et al.*, "Improving content delivery with PaDIS," *IEEE Internet Comput.*, vol. 16, no. 3, pp. 46–52, May/Jun. 2012.

[3] H. Chan, P. Fan, and Z. Cao, "A utility-based network selection scheme for multiple services in heterogeneous networks," in *Proc. Int. Conf. Wireless Netw., Commun. Mobile Comput.*, Jun. 2005, pp. 1175–1180.

[4] X. Duan, Z. Niu, and J. Zheng, "Utility optimization and fairness guarantees for multimedia traffic in the downlink of DS-CDMA systems," in *Proc. IEEE GlobeCom*, Dec. 2003, pp. 940–944.

[5] R. J. La and V. Anantharam, "Utility-based rate control in the Internet for elastic traffic," *IEEE/ACM Trans. Netw.*, vol. 10, no. 2, pp. 272–286, Apr. 2002.

[6] M. Xiao, N. B. Shroff, and E. K. P. Chong, "A utility-based power-control scheme in wireless cellular systems," *IEEE/ACM Trans. Netw.*, vol. 11, no. 2, pp. 210–221, Apr. 2003.

[7] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford, "DONAR: Decentralized server selection for cloud services," in *Proc. SIGCOMM*, 2010, pp. 1–12.

[8] Z. Zhang *et al.*, "Optimizing cost and performance in online service provider networks," in *Proc. NSDI*, 2010, pp. 33–48.

[9] A. Sharma *et al.*, "A global name service for a highly mobile internetwork," in *Proc. SIGCOMM*, 2014, pp. 1–12.

[10] J. Bentham, *An Introduction to the Principles of Moral and Legislation*. Oxford, U.K.: Clarendon, 1789.

[11] M. A. Stone and B. C. Moore, "Tolerable hearing aid delays. I. Estimation of limits imposed by the auditory path alone using simulated hearing losses," *Ear Hearing*, vol. 20, no. 3, pp. 182–192, 1999.

[12] J. Nielsen, *Usability Engineering*. Cambridge, MA, USA: Academic, 1994.

[13] H. Xu and B. Li, "Joint request mapping and response routing for geo-distributed cloud services," in *Proc. INFOCOM*, Apr. 2013, pp. 854–862.

[14] M. A. Khan and U. Toseef, "User utility function as quality of experience(QoE)," in *Proc. ICN*, 2011, pp. 99–104.

[15] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Utility-based placement of dynamic Web applications with fairness goals," in *Proc. NOMS*, Apr. 2008, pp. 9–16.

[16] *Methods for Subjective Determination of Transmission Quality*. Accessed: Dec. 13, 2017. [Online]. Available: http://www.itu.int/rec/T-REC-P.800-199608-I/en

[17] T. K. Phan, E. Maini, D. Griffin, and M. Rio, "Utility-maximizing server selection," in *Proc. IFIP Netw.*, May 2016, pp. 413–421.

[18] S. Boyd and A. Mutapcic, "Subgradient methods," Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, Appl. Notes EE364b, 2006.

[19] *ILOG CPLEX Optimization*. Accessed: Dec. 13, 2017. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio

[20] *Data Center Map*. Accessed: Dec. 13, 2017. [Online]. Available: http://www.datacentermap.com/

[21] *Video Distribution Modelling*. Accessed: Dec. 13, 2017. [Online]. Available: https://github.com/richardclegg/multiuservideostream

[22] G. Van Brummelen, *Heavenly Mathematics—The Forgotten Art of Spherical Trigonometry*. Princeton, NJ, USA: Princeton Univ. Press, 2013.

[23] S. Gangam, J. Chandrashekar, Í. Cunha, and J. Kurose, "Estimating TCP latency approximately with passive measurements," in *Proc. PAM*, 2013, pp. 83–93.

[24] *FUSION Deliverable D4.3*. Accessed: Dec. 13, 2017. [Online]. Available: http://www.fusion-project.eu/deliverables/fusion-d4.3-public-final.pdf

[25] *FUSION Deliverable D5.3*. Accessed: Dec. 13, 2017 [Online]. Available: http://www.fusion-project.eu/deliverables/deliverable-d5.3-final.pdf

[26] Z. Gao, A. Venkataramani, J. F. Kurose, and S. Heimlicher, "Towards a quantitative comparison of location-independent network architectures," in *Proc. SIGCOMM*, 2014, pp. 259–270.

[27] E. Nordström *et al.*, "Serval: An end-host stack for service-centric networking," in *Proc. NSDI*, 2012, pp. 1–14.

[28] V. Jacobson *et al.*, "Networking named content," in *Proc. CoNEXT*, 2009, pp. 1–12.

[29] M. Caesar *et al.*, "ROFL: Routing on flat labels," in *Proc. SIGCOMM*, 2006, pp. 363–374.

[30] M. Gritter and D. R. Cheriton, "An architecture for content routing support in the Internet," in *Proc. USITS*, 2001, pp. 1–4.

[31] D. Naylor *et al.*, "XIA: Architecting a more trustworthy and evolvable Internet," *ACM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 50–57, 2014.

[32] M. Handley and A. Greenhalgh, "The case for pushing DNS," in *Proc. HotNets*, 2005, pp. 1–6.

Truong Khoa Phan received the B.Sc. degree from the HCMC University of Technology, Vietnam, in 2007, and the M.Sc. and Ph.D. degrees from INRIA/I3S, Sophia, France, in 2011 and 2014, respectively. He is currently a Research Associate with the Department of Electronic and Electrical Engineering, University College London, U.K. His current research interests include network optimization, cloud computing, multicast, and P2P.

David Griffin received the B.Sc. degree in electronic and electrical engineering from Loughborough University and the Ph.D. degree in electronic and electrical engineering from University College London. He is currently a Principal Research Associate with the Department of Electronic and Electrical Engineering, University College London. His current research interests include planning, management and dynamic control for providing QoS in multiservice networks, and novel routing paradigms for the future Internet.

Elisa Maini received the Ph.D. degree in computer and automation engineering from the University of Naples Federico II. She is currently an SDN/NFV Product Architect with Vodafone, U.K. Her current research interests include network optimization and modeling, software-defined networking, and network function visualization.

Miguel Rio is currently a Professor with the Department of Electronic and Electrical Engineering, University College London, where he researches and lectures on Internet technologies. His current research interests include real-time overlay streaming, network support for interactive applications, and quality of service routing and network monitoring and measurement.