2021 IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS) | 978-1-6654-4935-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/MASS52906.2021.00038

# Online Resource Allocation in Edge Computing Using Distributed Bidding Approaches

Caroline Rublein
Pennsylvania State University, USA
Email: clr292@psu.edu

Fidan Mehmeti
Pennsylvania State University, USA
Email: fidan.mehmeti@psu.edu

Mark Towers
University of Southampton, UK
Email: mt6g17@soton.ac.uk

Sebastian Stein
University of Southampton, UK
Email: ss2@ecs.soton.ac.uk

Thomas F. La Porta
Pennsylvania State University, USA
Email: tlp@psu.edu

***Abstract*—Edge computing has become a very popular service that enables mobile devices to run complex tasks with the help of network-based computing resources. However, edge clouds are often resource-constrained, which makes resource allocation a challenging issue. We focus on a distributed resource allocation method in which servers operate independently and do not communicate with each other, but interact with clients (tasks) to make allocation decisions. This provides robustness and does not require service providers to share information about their configurations or workloads. We propose a two-round bidding approach of assigning tasks to edge cloud servers, while taking into account various processing requirements and server constraints. We consider cases in which all jobs have equal utility, cases where jobs have different utilities but users do not disclose these utilities to servers, and cases where users disclose the utility of their jobs to servers. We evaluate the performance using extensive realistic simulations. Results show that our approach is very close to an optimal assignment, with discrepancy not exceeding $5\%$.**

***Index Terms*—Edge cloud computing, Optimization, Bidding.**

## I. Introduction

Edge computing [1] is an important new paradigm that enables mobile devices to request complex processing services. Edge clouds may be used for emergency services or tactical situations [2] where an access network with processing resources is deployed in the field, or in cases in which access service providers provide some computing resources to subscribers. Edge clouds are not as well-resourced as backbone cloud networks, so resource allocation in these edge clouds is an important problem.

In this paper we consider a *distributed system* in which the computing resources in the edge cloud do not communicate or collaborate with each other, but interact with clients to make allocation decisions. In this way, there is no need for centralized management, and the system is more robust. By not requiring centralization, we allow edge cloud resources provided by different entities to be shared without requiring the service providers to share details of their deployments. Our goal is to maximize the utility of the jobs served.

There are other possible architectures for such a system [3]. A *centralized system* can be built where all requests are submitted to a server that allocates jobs to computing resources in the edge cloud [4]. This has the advantage of having complete knowledge of submitted and running jobs while making allocation decisions. However, this is not ideal because the server performing resource allocation must have synchronized reports from all processing resources which may not be desirable from the service provider perspective.

There has been much work on auctions for resource allocation [5]. Many of these approaches provide guarantees on performance and are strategy-proof, meaning that the resource allocation is robust against dishonest clients. However, they often require a central server or many rounds of negotiation to converge to a solution [4]. Given that we are striving for a simple and robust solution, we use a simple bidding protocol limited to two communication rounds between clients and servers.

We consider a system in which users may not want to disclose the utility of their job to the edge cloud provider. We develop and evaluate algorithms for three different cases to account for this: (i) all users have equal utility; (ii) jobs have different utility, but users do not disclose the utility to servers; and (iii) users disclose the utility of their jobs to servers. Case (ii) is motivated by the situation in which partners share processing resources but may not want to share full information about their actions. Cases (ii) and (iii) raise certain challenges. For case (ii), the servers must set prices that are on sclae with job utilities so that users can evaluate whether it is profitable for them to execute a job. In case (iii), there is an incentive for users to over-value their jobs to improve their chances of being accepted. We design a method to cap this advantage while keeping the system simple.

We propose a two-round bidding method for delivering remote processing services to users. The users submit job requests that include their processing requirements (memory, network capacity, processing capacity and deadline), and may or may not include their utility, as described above. Servers set a price for each job depending on their current state and which jobs they would like to serve based on different criteria we describe below.

We consider two processing disciplines - pipeline and batch. In pipeline processing, data elements that are uploaded from a client are processed as they arrive. In this case, the bandwidth,

978-1-6654-4935-9/21/$31.00 ©2021 IEEE
DOI 10.1109/MASS52906.2021.00038

processing, and memory are all used simultaneously. In batch processing, all data is uploaded before processing takes place. In this case, the bandwidth is used at the start and the end of a job, but not while the job is executing.

We make the following contributions in this paper:

- We formally define an optimization problem for the online version of both the pipeline and batch processing for our system and solve them using a commercially available solver.
- We design the framework as a simple two-round bidding system for resource allocation.
- We develop algorithms for setting server prices depending on the problem setting: all jobs are equal, users do not disclose utility, and users disclose utility.
- We consider two basic pricing mechanisms and compare them to an optimal solution. We find that a *best-fit* pricing mechanism based on a knapsack problem is better than a *congestion-based* pricing scheme, and that our algorithm is within approximately $5\%$ of the optimal.

## II. System Overview

Our system has two sets of procedures that run in parallel - the *bidding* procedure and the *processing* procedure. Our work focuses on the bidding procedure, but is impacted on the discipline of the processing procedure. The bidding system runs in epochs and consists of two rounds, $R1$ and $R2$. In each epoch, the jobs that have arrived in the previous epoch participate in the bidding for server resources.

Clients submit requirements of jobs to a set of servers to request resources. The servers, based on the requirements and their current state, set a price for each job and reply to the clients with this price. The clients then choose which server to select for their job and notify that server. The server will then either accept the job, or if it receives too many positive responses, reject it in which case the client is not served.

As soon as the bidding phase is complete, accepted jobs start to execute as described below, and the next bidding epoch begins considering the jobs that arrived during the just completed bidding epoch. Fig. 1 shows the system operation at one server. Consider bid epoch 2. In this epoch, the request for jobs that were submitted to the server in the previous bid epoch (bid epoch 1, job set 2) are processed at the server and the interaction between the server and users take place. The server considers the state of all the jobs it is currently processing when setting prices. At the end of bid epoch 2, a new set of jobs are accepted and start to execute. At this time, all the job requests that were received during bid epoch 2 (job set 3) are processed.

In the processing phase, we consider both *pipeline* and *batch* processing for the processing procedure. These differ in the way that resources are used, so they impact the resource allocation. In the pipeline system, once a job is accepted, the user starts to upload materials to be processed, processing begins and processing memory is occupied. Thus all three resources (storage, computation, and bandwidth) we consider in our bidding phase are used simultaneously. In the batch system, once a job is accepted, the client uploads all the material to the server, but processing does not begin until the full upload is complete. Results are returned to the user after processing is completed. Thus bandwidth is used before and after processing resources and processing memory, but not at the same time.

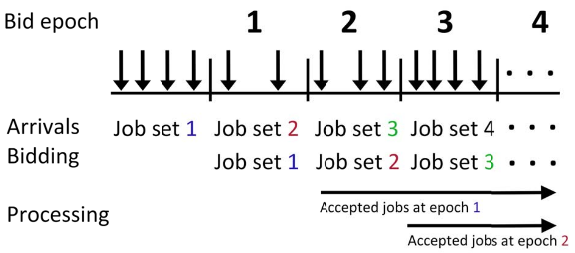

Fig. 1. The arrival of jobs, along with the bidding and processing procedures.

The main challenge in the system is how to set the prices for jobs. We consider two approaches to price setting. In the first, servers set a price based solely on their current usage, which we call *congestion pricing*. In the second, servers set a price based on *best-fit*, that is, a server favors jobs that have requirements that best fit its unused resources.

Both methods face a similar challenge: if prices are set too high, then too few jobs may respond in the second round, leading to under-utilization. However, if prices are set too low, then too many jobs may respond in the second round, causing servers to reject jobs which are then not served.

In the next sections we present our optimal bidding formulations and heuristics for the pipeline and batch processing cases. Although the bidding structure is the same, the formulation and heuristics for setting prices and accepting jobs is different in the two cases because of the way in which the processing resources are used.

## III. Pipeline Processing Optimization Formulation

In this section, we provide the optimization formulation for resource allocation for an online system in which jobs arrive over time and are processed in a pipeline fashion. Recall that in pipeline processing, data items are processed as they are uploaded by the users. For example, if a user is submitting stored images for object classification or labeling, the images are processed as they are received.

There are $|\mathcal{I}|$ servers, and a total of $|\mathcal{J}|$ tasks submitted over time. The system is time slotted, and we consider a time horizon of $|\mathcal{N}|$ slots. The arrival time of task $j$ is $a_j$; its deadline, by which the task must be served after arrival, is $d_j$. The utility achieved if a task is served is $U_j$. Task $j$ has a storage requirement of $s_j$, and a total computational requirement of $K_j$.[1] The total storage capacity of server $i$ is $S_i$, and its total computational capacity is $C_i$ units per slot.

Note that we do not include the bandwidth constraint in this formulation because it would make the problem infeasible to solve even with a solver. This will lead to our optimization formulation yielding higher performance than if we were able

[1] We assume that in this case the data storage for a task is loaded instantly.

to model the bandwidth constraint, but this is acceptable because in our heuristic we account for the bandwidth constraint so the comparison to the optimal is conservative.

The decision variables are: (i) $x_{i,j}$, whose value is 1 if task $j$ is assigned to server $i$, and 0 otherwise, and (ii) $\kappa_j(n)$, which denotes the amount of computing resources allocated to task $j$ in slot $n$.

Once a job is allocated to a server, the data needed to run the task are stored in the server and will occupy the memory until the task is executed completely. On the other hand, the computation power has an elastic nature, meaning that a task can receive a different amount of computing resources in different slots. E.g., if at a given slot all the other tasks are finished, and in the next slot only one task remains active while no other tasks have been assigned, the server can decide to increase the assigned computing power in order to complete the task more quickly to reclaim resources for jobs that may arrive in the future. Alternatively, if there are too many jobs assigned at a given slot, for a job whose deadline is not close, the server can decide not to allocate any processing power to that job in a slot, in order to optimize performance.

To capture the fact that while the task data is stored on a server it will remain there until the task is run completely, we use an indicator variable, which for slot $n$ is defined as

$$\theta_j(n) = \begin{cases} 1, & \text{if} \min\{a_j + l_j | \kappa_j(a_j + l_j) > 0\} \leq n \\ & \qquad \leq \max\{a_j + l_j | \kappa_j(a_j + l_j) > 0\} \\ 0, & \text{otherwise.} \end{cases}$$

where $l_j$ is a timestep[2] during which the job will run to completion ($0 \leq l_j \leq d_j - 1$).

The optimization problem is as follows.

$$\max \sum_{i=1}^{|\mathcal{I}|} \sum_{j=1}^{|\mathcal{J}|} U_j x_{i,j} \tag{1}$$

$$\text{s.t.} \sum_{l_j=0}^{d_j-1} \kappa_j(a_j + l_j) = K_j x_{i,j}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{2}$$

$$\sum_{j=1}^{|\mathcal{J}|} s_j x_{i,j} \theta_j(n) \leq S_i, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{3}$$

$$\sum_{j=1}^{|\mathcal{J}|} \kappa_j(n) x_{i,j} \leq C_i, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{4}$$

$$\sum_{i=1}^{|\mathcal{I}|} x_{i,j} \leq 1, \quad \forall j \in \mathcal{J}, \tag{5}$$

$$x_{i,j} \in \{0, 1\}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{6}$$

$$\kappa_j(n) = 0, \quad \forall j \in \mathcal{J}, n = \{1, \ldots, a_{j-1}, a_j + d_j, \ldots, N\}, \tag{7}$$

$$\kappa_j(n) \geq 0, \quad \forall j \in \mathcal{J}, n = \{a_j, \ldots, a_j + d_j - 1\}. \tag{8}$$

The objective (1) is to maximize the total utility value over all tasks in all the servers. The left-hand side of Constraint (2) is the sum of allocated processing cycles over all slots that the task is active, whereas the right-hand side is either the total computing power required to finish the task (if the task is allocated), or 0 in the case when the task does not receive service. Note that when the latter occurs, it enforces all the $\kappa_j$'s to be 0. The finite storage capacity of a server is described by (3), whereas its finite computing power per slot by (4). Constraint (5) imposes the limitation that a task can be served by at most one server, while (6) depicts the fact that a task is either assigned or not. Finally, the last two constraints capture the fact that before the arrival and after the departure of a task, no resource allocation is needed, and while the task is active the amount of resources in a slot will either be strictly positive or 0, as explained above.

This optimization formulation corresponds to an optimal case in which there is a centralized entity with a complete knowledge of all the parameters for all the servers and tasks, and even of the future arrivals, i.e., it acts like an oracle.

The solution of this problem for instances with a large number of users and time horizon is infeasible. First, it is unrealistic to assume that the centralized entity will know when a given task of given characteristics will arrive. Second, this in an NP-hard problem. Namely, even for a single slot $n$, this optimization problem is a 2-dimensional knapsack problem, which are known to be NP-hard [6].

Therefore, in the next section we propose a heuristic that, as will be seen in Section V, provides a result that is very close to that of the optimal solution (for not very large instances of the problem). It is a two-round bidding method of assigning tasks to the servers and allocating the resources to them, which gives both servers and users some agency over the outcome.

## IV. Pipeline Heuristics

In this section we present our algorithms for the pipeline processing for three different cases of what information the users disclose to the servers. In particular, the algorithms focus on how the prices are set, and we evaluate the different pricing mechanisms for each case in the subsequent section.

### A. Case 1: No Disclosure

In this first case, neither the users nor the servers use utility to make any decisions. So, the objective is to maximize the number of jobs completed by the system. We consider both congestion pricing and best-fit pricing to judge the burden a job will place on a server.

*1) Congestion Pricing:* In this method, each server's first-round price is directly proportional to the size of the job in each dimension and to the ratio of the job's size to the server's free space in each dimension (i.e., the currently unused server resources). Although we have assumed bandwidth to be infinite in the optimal case, we account for it in the heuristics. The parameters $\hat{S}_i$, $\hat{C}_i$, and $\hat{B}_i$ represent the remaining (non-allocated) storage, computation, and bandwidth resources on server $i$; and $s_j$, $\kappa_j(n)$, and $b_j$ represent the storage, computation per timestep, and bandwidth of job $j$. A scaling factor $f$ exists so that this pricing can still be useful in Case 2. This is described in the next subsection. The first round prices for timestep $n$ are set at

[2] We will use notions *slot* and *timestep* interchangeably throughout the paper.

$$\left(\frac{s_j}{\hat{S}_i} + \frac{\kappa_j(n)}{\hat{C}_i} + \frac{b_j}{\hat{B}_i}\right) \cdot f, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}. \tag{9}$$

Each user receives a price from all servers, and then chooses the lowest price server to request service. When the servers receive the second round request, they run a knapsack algorithm to fit the most jobs. The jobs that do not fit are rejected. Allocated jobs may receive excess server resources (if they are available) to complete more quickly.

*2) Best-Fit Pricing:* The goal in this method is to fit jobs by taking into account the different resources. Thus a knapsack algorithm is used in both rounds. The first round consists of running a knapsack on all the jobs submitted to the server and assigning prices based on if jobs fit in the knapsack. As with Case 1 described above, the second round consists of running another knapsack, but only for the jobs that chose a particular server, since not all jobs may return to a particular server.

The first-round pricing now falls into two tiers for the two categories of jobs: those inside the knapsack and those outside. If a job is inside the knapsack, its price is 1 in order to most strongly attract that job to the server. If the job is outside the knapsack, its price must be determined some other way. If the price is set too high, then no other jobs other than those that fit in the knapsack will return in the second round.

One simple method is to set the price of jobs that do not fit in the knapsack in round one to a common high value. Since it is likely that some jobs will fit in the knapsack of one or more servers, not all jobs that fit in the knapsack in round one will return in the second round, thus leaving some space in servers in round two. Other jobs may not fit in any knapsack in round one, and if all the prices they receive are the same, they will not know which server has the best chance to serve them in round two. As an alternative, instead of setting the price of those jobs that do not fit in the knapsack in round one to a high common value, we increase the price of the jobs that do not fit to be a value above 1 by adding a factor related to how close they were to fitting. In this way, jobs that do not fit in any knapsack can pick the server they are most likely to be accepted by for the second round. We consider two ways to increase the price, according to congestion or violation, below.

1) Outside knapsack (KP) - Congestion:

$$1 + \left(\frac{s_j}{\hat{S}_i} + \frac{\kappa_j(n)}{\hat{C}_i} + \frac{b_j}{\hat{B}_i}\right) \cdot f, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}. \tag{10}$$

2) Outside KP - Violation:

$$1 + \Big(\frac{s_j + k_s(n)}{\hat{S}_i} + \frac{\kappa_j(n) + k_c(n)}{\hat{C}_i} + \frac{b_j + k_b(n)}{\hat{B}_i}\Big) \cdot f, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}. \tag{11}$$

The parameters $k_s(n) = \sum s_j\theta_j(n)$, $k_c(n) = \sum \kappa_j(n)$, and $k_b(n) = \sum b_j\theta_j(n)$ are the sums of the storage, computation, and bandwidth requirements, respectively, of all jobs in the knapsack in timestep $n$. Essentially, congestion represents what proportion of the server's currently available resources job $j$ would consume, whereas violation represents how easily job $j$ could have been added to the knapsack result.

Using these methods of setting the price, some jobs that are outside of the knapsack may still return in the second round, allowing the server to fill in gaps left by jobs that did not return.

### B. Case 2: Utility Considered by Users

In this case, each job has its own utility. The users know the utility of their jobs, but do not disclose the utility to the servers. The users consider their utility when deciding if they should request service after seeing prices from the servers. If the prices a job receives from the servers in the first round are all greater than its utility, then the job will remove itself from the system and place itself back into the pool for the next timestep.

Since jobs are now comparing their prices to their utility, the scaling factor shown in Case 1 is now important. The scaling factor must be set based on the distribution of utilities. Through simulation, our optimal scaling factor was found to be about $1.1$ standard deviations below the mean utility.

The algorithms that are run on the servers for this case are the same as for Case 1. The only difference is that users will not reply to a server that has a higher price than their job's utility.

### C. Case 3: Disclosure to Servers

In this case, the users disclose the utility of their jobs when they submit the requirements. Now, the servers maximize the utility of their remaining space in the R1 knapsack, rather than the number of jobs. Due to the best-price (violation variant) producing the best performance in Case 2's results, we choose to use only that variant for Case 3 testing.

Since jobs disclose their utilities, they can now lie to the servers about their utility. There are two underlying reasons for jobs' dishonesty: to have a better chance of getting in the knapsack (over-reporting utility), or to receive a cheaper price (under-reporting utility). Given that under-reporting utility will make it less likely for a job to be accepted, we focus on capping the damage that can be done by users over-reporting the utility of their jobs.

In R1, the server reduces their price by a factor that caps the impact of cheating by the user. If a job returns in R2 and is accepted, the server charges the actual price without a discount. Algorithm 1 shows how the price is computed in each round. The parameter $\alpha$ is the margin of cheating (percentage), and the parameter $\beta$ is any positive constant. The inverse of the violation is used so that jobs with a smaller violation are given a larger discount (and thus a lower price). The knapsack runs in O($n^g$) time, where $n$ is the number of active jobs in the slot, and $g$ is the number of generations (constant $\approx$50) used in the off-the-shelf genetic algorithm knapsack implementation. Pricing decisions are run in O($n$) time, so the overall complexity of the algorithm is O($n^g$).

The factor $\alpha$ works as follows. If a job bids a factor of $\beta$ above its real value and it fits in the knapsack in R1, it will receive a bid price of its real utility, $U + \beta U - \alpha U$. If $\beta$ is greater than $\alpha$, i.e., it over-reported by more than $\alpha$, the returned bid will be higher than the actual utility of the job, so the job will not accept the bid. In this case the server will

**Algorithm 1:** Case 3 pricing

```
Round 1 (R1)  if Accepted into knapsack then
    price = $U - \alpha U$;
else
    if Under threshold then
        price = $U - \min(\frac{1}{violation(Eq.11)}, \frac{\alpha}{2})U$;
    else
        price = $U + \beta U$;
    end
end
Round 2 (R2)  if Accepted into knapsack then
    price = $U - \frac{1}{violation(Eq.11)}U$;
else
    Go to pool;
end
```

accept another job in R2. If the job over-reported by less than $\alpha$, the returned bid would still allow the job to be profitable so it could be selected in R2, perhaps at the cost of an honest job that would achieve higher utility. However, the amount of over-reporting and lost profit is capped by $\alpha$. The value of $\alpha$ must be set carefully because if it is too high, then the system will attract too many jobs in R1 that it cannot fit in R2.

Consider a server that advertises a price that is discounted by 10% ($\alpha = 0.1$) from the actual price it will charge. A dishonest user submits a job that has a real utility of 92, but reports a utility of 105 (14% higher than its real value) to the server in hopes of being accepted because of its high utility. Now consider that the cost of this job to the server is 100. Given the discount, the server will advertise a price of 90 to the user. The user will accept the bid because 90 is below its real utility of 92. However, when the job is accepted the server will actually charge the user 100, its real price, and thus the user will pay more than its job is worth. The only way that the user can be guaranteed to not lose value is to cap its over-reporting at the percentage of the discount being given by the server. The server does not disclose the value of this discount, but it effectively caps the loss of utility.

## V. Results for Pipeline Processing

In this section we evaluate the performance of the pipeline processing system including different settings, workloads, and pricing methods.

Servers each have storage, computation, and bandwidth resources. Jobs also have particular storage, computation, and bandwidth requirements, as well as a deadline and utility. Each of these resources is normally distributed among the jobs in the workload. The distribution variables for both jobs and servers are shown in Table I, and are chosen to provide moderate congestion ($\approx 50\%$) when the arrival rate almost matches the number of servers. The chosen ratio of storage to bandwidth resembles the case where images are uploaded, and those with particular features are downloaded to the user. The servers are client-agnostic, i.e., each job is treated individually and clients are assumed to consider each job's profit independently. Therefore, client ownership of multiple jobs is not represented in the simulations.[3]

TABLE I
Normally distributed variables for servers and jobs

| Resource | $\mu$ | $\sigma$ |
|---|---|---|
| Storage $S_i$ (GB) | 3.6 | 0.3 |
| Computation $C_i$ (MB/s) | 72 | 30 |
| Bandwidth $B_i$ (MB/s) | 2000 | 400 |
| No. of images $s_j$ (MB) | 200 | 50 |
| Storage $s_j$ (MB) | 1240 | 50 |
| Computation $K_j$ (MB/s) | 47 | 15 |
| Bandwidth $b_j$ (MB/s) | 620 | 50 |
| Deadline $d_j$ (slots) | 8 | 2 |
| Utility $U_j$ | 40 | 10 |

We consider two workloads: "small" and "large". The small workload has an arrival rate of $\mu = 4$ jobs per timestep, and the large workload has an arrival rate of $\mu = 7$ (both are normally distributed, with $\sigma = 2$). Both workloads have 200 timesteps of job arrivals, and 20 empty timesteps to allow allocated jobs to finish running. Thus, the small workload has a total of $\approx 800$ jobs (a moderately loaded system), and the large workload a total of $\approx 1400$ jobs (a congested system). Both workloads are based on [7], which details the amount of time required to perform edge detection on $1980 \times 1020$ images using a core i5 processor (hence the MB/s units for $C_i$ and $K_j$). We assume the time required scales linearly with the number of images being processed. Each timestep of the heuristic takes about 5 seconds to run, the majority of which is spent computing the knapsack.

### A. Optimal Results

The optimization formulation was modeled using Gurobi. In a small scenario consisting of 4 servers and 22 jobs, our Case 3 heuristic (utility disclosure, no cheating) is within $5.2\%$ of the optimal achieved utility (i.e. the most optimal solution found by the Gurobi solver). For the optimization formulation, we assume bandwidth is infinite, and the heuristic simulation still performs well even when the optimal solution is given this advantage. In the remainder of the simulation results shown below, 8 servers were used. For each set of results, 10 instances of the problems were run and the average is shown.

### B. Case 1 Results

As shown in Figs. 2 and 3, the best-fit pricing clearly performs better than the congestion pricing. Fig. 2 shows the average percentage of jobs rejected out of the entire workload. Fig. 3 displays the average total utility completed per workload. The difference in performance is particularly apparent in the large workload, where the best-fit pricing shows a 30% average decrease in rejection, and an almost $200\%$ increase in utility completed.

[3] We have run simulations with other values of the parameters with similar conclusions drawn.

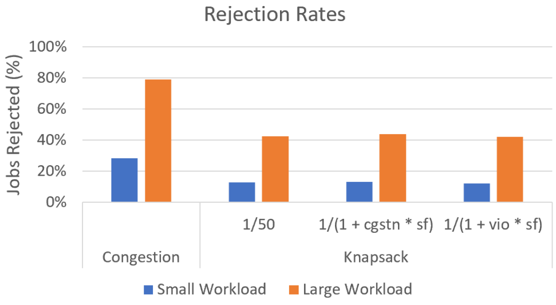

Fig. 2. Rejection rates of the small & large workloads under different pricing schemes (congestion, and the 3 best-fit variants) for Case 1.

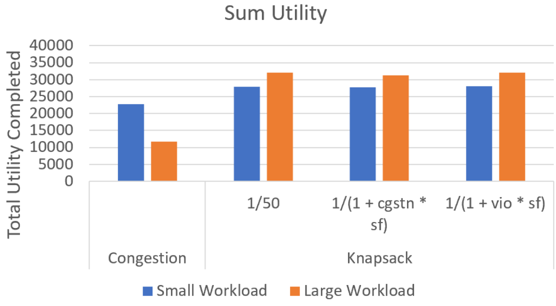

Fig. 3. Total utility completed for the small & large workloads under different pricing schemes (congestion, and the 3 best-fit variants) for Case 1.

Within the best-fit pricing, the violation variant resulted in a slightly greater utility completed than the congestion pricing and also provides higher utility than when a simple two-tier pricing system is used in which jobs that do not fit in the knapsack in R1 are given a price of 50.

### C. Case 2 Results

The total utility for Case 2 (users do not disclose job utility to the servers, but consider it in their choice of server) shown in Fig. 4 reveals that the various best-fit pricing schemes perform significantly better than congestion pricing. Here, however, the violation variant improves performance more significantly than in Case 1.

### D. Case 3 Results

For Case 3 (users disclose job utility) we tested various cheating profiles for the job, in addition to the all-honest case. In each profile, 20% of the users over-value their jobs by a certain amount (5, 10, 15, or 20%).

Our simulations demonstrate that the loss of utility due to cheating is always capped to $\alpha$%, as shown in Fig. 5. Fig. 5 also demonstrates that as jobs cheat by higher percentages, their chance of being accepted drops drastically, thereby discouraging cheating by high amounts. Fig. 6 shows the total utility completed under each profile. Although most of the cheating profiles do not have a clear maximum, the honest profile completes the most utility when $\alpha = 10\%$. Therefore, choosing $\alpha = 10\%$ will result in the most honest utility being completed, while cheating is capped at 10%.

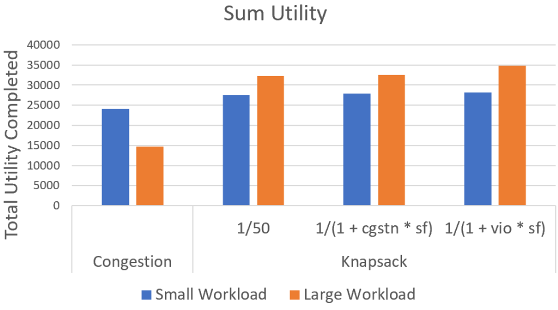

Fig. 4. Total utility completed for the small & large workloads under different pricing schemes (congestion, and the 3 Best-fit variants) for Case 2.

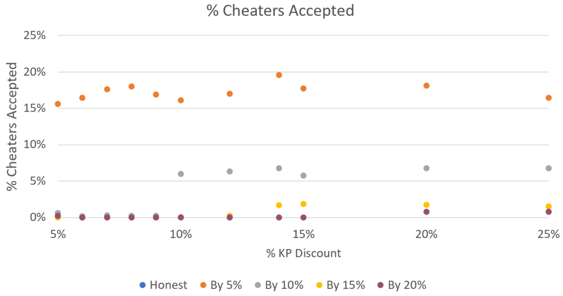

Fig. 5. The percentage of overvalued jobs that are accepted for different cheating profiles.

As shown in Fig. 7, using any best-fit pricing results in better performance, especially for a congested system. Revealing the utility of the jobs to the servers results in the greatest performance. However, simply having users consider the utility of their own jobs when making decisions without disclosing the utility to the servers still performs within 5% of full disclosure, which bodes well for the environments in which users decline to disclose their utility.

## VI. Batch Processing Optimization and Heuristics

In this section we consider the batch processing policy. In this case, once a job is accepted via the bidding process, all data must be uploaded before processing can begin. This might be the case, for example, if a full video is being uploaded for action analysis.

### A. Formulation

The process for batch jobs consists of three phases after the task has “arrived” to the system: loading the data to one of the servers, processing the task, and sending the resulting data to the user after the task is done. In this formulation we do include the bandwidth constraints. The time it takes to load the data of task $j$ is denoted by $d_{j,u}$, the processing time by $d_{j,p}$, and the time it takes to send back the results by $d_{j,d}$. These are expressed in the number of timesteps.

The amount of data sent per slot from the task to the server is $\sigma_j(n)$, whereas the amount of data of the results (server-user) sent during slot $n$ is $\sigma_j^{'}(n)$. The total amount of the results for the solved task $j$ is $s_j^{'}$. The parameter $B_{u,i}$ denotes

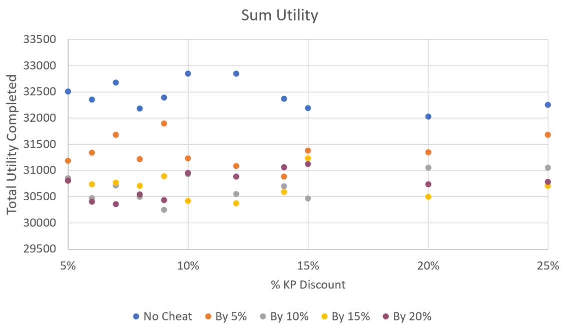

Fig. 6. The total utility completed for different cheating profiles.

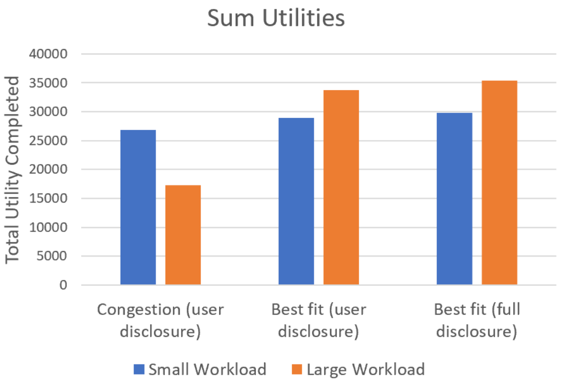

Fig. 7. Total utility for the small & large workloads for the 3 different cases.

the maximum amount of data that can be uploaded per slot to server $i$ (equivalent to uplink bandwidth), whereas $B_{d,i}$ denotes the maximum amount of results that can be sent per slot to the corresponding tasks by server $i$. The parameter $\theta_{p,j}$ in (16) is the same as $\theta_j(n)$, defined in Section III. The other parameters not specified here remain unchanged from Section III.

The optimization formulation related to this approach is given by the following. The decision variables are: $x_{i,j}$, $\sigma_j(n)$, $\kappa_j(n)$, $\sigma_j^{'}(n)$, $d_{j,u}$, $d_{j,p}$, and $d_{j,d}$.

$$\max \sum_{i=1}^{|\mathcal{I}|}\sum_{j=1}^{|\mathcal{J}|} U_j x_{i,j} \tag{12}$$

$$\text{s.t.} \sum_{l_j=1}^{N} \sigma_j(l_j) = s_j x_{i,j}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{13}$$

$$\sum_{l_j=1}^{N} \kappa_j(l_j) = K_j x_{i,j}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{14}$$

$$\sum_{l_j=1}^{N} \sigma_j^{'}(l_j) = s_j^{'} x_{i,j}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{15}$$

$$\sum_{j=1}^{|\mathcal{J}|} s_j x_{i,j} \theta_{p,j}(n) \leq S_i, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{16}$$

$$\sum_{j=1}^{|\mathcal{J}|} \kappa_j(n) x_{i,j} \leq C_i, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{17}$$

$$\sum_{j=1}^{|\mathcal{J}|} \sigma_j(n) x_{i,j} \leq B_{u,i}, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{18}$$

$$\sum_{j=1}^{|\mathcal{J}|} \sigma_j^{'}(n) x_{i,j} \leq B_{d,i}, \quad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}, \tag{19}$$

$$\sum_{i=1}^{|\mathcal{I}|} x_{i,j} \leq 1, \quad \forall j \in \mathcal{J}, \tag{20}$$

$$x_{i,j} \in \{0,1\}, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \tag{21}$$

$$\sigma_j(n) = 0, \quad \forall j \in \mathcal{J}, n = \{1, \ldots, a_{j-1}, a_j + d_{j,u}, \ldots, N\}, \tag{22}$$

$$\sigma_j(n) \geq 0, \quad \forall j \in \mathcal{J}, n = \{a_j, \ldots, a_j + d_{j,u} - 1\}, \tag{23}$$

$$\kappa_j(n) = 0, \quad \forall j \in \mathcal{J}, \quad n = \{1, ..., a_j + d_{j,u} - 1, a_j + d_{j,u} + d_{j,p},.., N\} \tag{24}$$

$$\kappa_j(n) \geq 0, \quad \forall j \in \mathcal{J}, \quad n = \{a_j + d_{j,u}, \ldots, a_j + d_{j,u} + d_{j,p} - 1\}, \tag{25}$$

$$\sigma_j^{'}(n) = 0, \quad \forall j \in \mathcal{J}, \quad n = \{1, ..., a_j + d_{j,u} + d_{j,p} - 1, a_j + d_{j,}.., N\}, \tag{26}$$

$$\sigma_j^{'}(n) \geq 0, \quad \forall j \in \mathcal{J}, \quad n = \{a_j + d_{j,u} + d_{j,p}, \ldots, a_j + d_j - 1\}, \tag{27}$$

$$d_{j,u} + d_{j,p} + d_{j,d} \leq d_j, \quad \forall j \in \mathcal{J}, \tag{28}$$

$$1 \leq d_{j,u} \leq d_j - 2, \quad \forall j \in \mathcal{J}, \tag{29}$$

$$1 \leq d_{j,p} \leq d_j - 2, \quad \forall j \in \mathcal{J}, \tag{30}$$

$$1 \leq d_{j,d} \leq d_j - 2, \quad \forall j \in \mathcal{J}. \tag{31}$$

Constraint (13) captures the amount of uploaded data to the server over time until the entire data is stored, if the task is served, or 0 otherwise. Similarly, constraints (14) and (15) correspond to the allocation of processing resources and bandwidth for sending the results back to the user, respectively. As there is a strict order of performing actions (processing cannot start until the entire data is stored on the server, and sending back the results cannot start before the task is processed completely), we use the constraints (22)-(27) to restrict the order of operations. The finite storage capacity of every server is captured by (16). Note that we use the same indicator variable $\theta$, as in the pipeline formulation, to denote the time instants when a part of the server's storage is taken by the task's data. The servers' finite computational capabilities are described by (17), whereas the finite bandwidth of the servers both in uplink and downlink are represented by (18) and (19), respectively. Inequality (20) constrains the assignment of every task to at most one server, while (21) expresses the fact that a task is either assigned or not. Finally, (29)-(31) impose the need for each of the three phases to take at least one slot.

*Note*: The unit of $B$ is in fact MB (megabytes). However, as the slot duration is fixed ($\Delta t$), we have $B = W \cdot \Delta t$, where $W$ is the bandwidth in Mbps. So, $B$ is proportional to $W$. Hence, we use $B$ to describe the bandwidth.

As in the pipeline optimization formulation, this optimization problem corresponds to the optimal case of a central entity that has a complete overview of the system. Since this is also an NP-hard problem, in Section VI-B we resort to tractable heuristics.

### B. Heuristic

For the batch processing, we define a heuristic for Case 3 (in which the clients disclose their utility to the servers). This provides a basis for the other cases' heuristics to be easily developed. The general process of job submission and the double bidding remain the same under the batch paradigm. However, some changes were made to the knapsack, pricing, and storage/memory usage.

Since each job must strictly progress through its three phases, the servers must guarantee that different resources will be available for each allocated job at different times. Therefore, just before the knapsack process is begun, the server will assign temporary intermediate deadlines ($d_{j,u}$, $d_{j,p}$, and $d_{j,d}$) to each job that keep $d_{j,u}$ and $d_{j,d}$ as long as possible while still fitting the job's computation onto the server. Then, the knapsack process involves checking multiple timesteps. A knapsack is run for each timestep from the current one until the maximum deadline in the pool. If a job will not be consuming a certain resource in a timestep (for example, processing resources are not consumed while upload bandwidth is being used), it will be zeroed out for that timestep.

Since the upload phase is now distinct from the processing phase, a job is allocated storage/memory space on the server only once it has completely finished uploading its data. This memory is kept throughout the entire processing phase, and released as soon as the results download begins. This emulates a server that stores data in storage device as it is received, and then moves it into processing memory, which is the constrained memory in our system, for processing. When the processing is complete, the results are moved into external memory to be downloaded to the user, thus freeing processing memory for other jobs.

For pricing, we are only considering Case 3, in which jobs' utility is disclosed to servers. In this system, we do not use the congestion or violation factor as the basis for the price discount. Instead, we based the discount on the number of times a job succeeds in the knapsack process since there will be a knapsack process corresponding to each timestep. The higher the number of timesteps in which a job is successful, the closer it is to fitting into the system. This maintains the spirit of the pipeline discounts since the score is also directly correlated to a job's size.

### C. Comparison to Optimal

We modeled the optimization formulation using Gurobi to emulate a centralized, omniscient solution. We tested the performance on smaller scenarios consisting of 8 servers and 24 jobs under three traffic scenarios using the same job and server dimensions and workloads as described in Table II: (i) the same amount of data is uploaded to the server at the start of processing as is downloaded as results to the user (50/50); (ii) 90% of the total data used is uploaded to the server to start the job, and 10% of the total data is downloaded to the user as results; and (iii) 10% of data is uploaded and 90% of data is downloaded. The memory the jobs require is equal to the total bandwidth consumed.

The optimal solution completed all jobs for the 50/50 case, but rejected $\frac{1}{24}$ jobs for the 10/90 and 90/10 cases. The heuristic rejected $\frac{4}{24}$ jobs for all 3 scenarios, thereby showing reasonably good performance, within about 20% of the optimal for the 50/50 case, and 15% for the 10/90 and 90/10 cases.

### D. Utility Achieved in Batch Workload

Here we consider the following scenario. Users have recorded and stored videos on their devices. They submit their videos to the network to have actions detected and marked. The videos are returned to the mobile devices after they have been processed. An equal amount of bandwidth is required in the uplink and downlink, and the memory required is equal to the size of the videos being processed. As far as memory usage is concerned, we make the same assumption as in the optimization formulation. As data is uploaded, it is stored in external memory until it is fully received. It is then moved into processing memory. Once the processing is complete, the results are moved into external memory to be downloaded.

The statistics for the jobs and servers in this scenario are shown in Table II (video statistics based on [8], [9]). Eight servers and 250 jobs were created based on these normal distributions. Total job bandwidth $b_j$ is not listed, as it equals the storage size of the job. For this workload, users upload videos with a length of 10 minutes on average and resolution of $320\times180$. The "small" workload has an arrival rate of $\mu = 4$ jobs per timestep, and the "large" workload has an arrival rate of $\mu = 7$ (both are normally distributed, with $\sigma = 2$).

In general, batch processing achieved ≈80% of the total utility while rejecting 20% or fewer jobs under the small workload, and achieved 65-72% utility and rejected 28-34% of jobs under the large workload. Within each workload, as the traffic split increasingly favored the upload bandwidth (e.g. 10/90 to 50/50), the rejection rate increased by ≈2%, causing the total utility achieved to drop by the same amount.

We also ran simulations to determine if the traffic split between upload and download impacted the performance of the resource allocation algorithm. We describe the results here briefly due to space constraints. We considered cases of equal upload and downloads, where 90% of the bandwidth used was for upload and 10% was for download, and where 10% of the bandwidth used was for upload and 90% was for download. The 90/10 case emulates, for example, a user uploading photos for object localization, and the server returning 10% of the photos that contained the object in a bounding box. The memory required for each job is the total bandwidth used.

Our results showed that the more upload bandwidth jobs require, the fewer jobs are allocated, leading to less utility completed. This is because jobs immediately consume some upload bandwidth when they are allocated to a server. Then, as jobs continue to arrive in the pool, a server may not have enough upload resources in the next few timesteps to fit new jobs in the knapsack. This does not occur in the 10/90 case because jobs are moved into processing faster and can make use of extra processing resources that are available.

TABLE II
VIDEO CASE STUDY STATISTICS

| Resource | $\mu$ | $\sigma$ |
|---|---|---|
| Total video length (min) | 10 | 2.5 |
| Storage $s_j$ (MB) | 18.9 | 3.5 |
| Computation $K_j$ (kB/s) | 360 | 60 |
| Deadline $d_j$ (slots) | 12 | 4 |
| Utility $U_j$ | 40 | 10 |
| Storage $S_i$ (MB) | 30 | 4 |
| Computation $C_i$ (kB/s) | 300 | 50 |
| Bandwidth $B_i$ (MB/s) | 30 | 4 |

## VII. RELATED WORK

Within the broad area of cloud computing, a wide range of resource allocation approaches exist, such as: application placement, resource scheduling, task offloading, load balancing, resource allocation, and resource provisioning [3], [10]. The concept of multi-dimensional bin packing has been proposed in [11], whereas basic auction-based resource allocation mechanisms in cloud computing, and auction variations, are described in [5]. Additional auction mechanisms that can be applied to cloud computing are presented in [4].

While in most of the proposed auction mechanisms truthfulness is a concept that is user-related, a truthful multi-unit double auction mechanism was proposed in [12] that enforces both users and servers to act truthfully.

Edge-MAP [13] proposes a client-to-cloud model for tasks with extremely short deadlines (lower than 100 ms). Using a Vickrey-English-Dutch (VED) auction, the system achieves a unique minimum competitive equilibrium price. Due to this property, the system is scalable and adjustable to network topologies that change.

An alternative market-based framework by Nguyen *et al.* allows for resources to be efficiently allocated by edge nodes that are dispersed [14], where the market equilibrium is found through an optimal resource allocation of bundles to services such that the task budget is not violated.

In [15], the authors consider the problem of joint service placement and request scheduling that maximizes the expected number of requests served per time slot, given various constraints in terms of the resources. To achieve this, the authors propose a two-time-scale framework, where the service placement is performed on a longer time scale (of frames), whereas the request scheduling is performed on the shorter time scale of slots. The proposed algorithm in [15] achieves 90% of the optimal performance. The main difference in our setup is that we try to maximize the total utility of served jobs, and we also consider the case where the users are not truthful.

## VIII. CONCLUSION

In this paper, we proposed a decentralized approach in assigning tasks to servers in edge clouds, in which we allow a flexible allocation of resources to tasks over time, so long as they meet the deadline requirements of the tasks. Two processing disciplines were considered: pipeline and batch. We proposed a two-round bidding approach of delivering processing tasks remotely to servers. Results show that our algorithms provide a performance that is close to the optimal, with the level of mismatch not larger than 5%, while introducing a significantly lower complexity.

## ACKNOWLEDGMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

## REFERENCES

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, 2018.

[2] F. Mehmeti, N. Felemban, Z. Lu, K. Wheatman, G. Cirincione, and T. F. La Porta, "Quality of information in gathering information via video analytics for military networks," *IEEE Communications Magazine*, vol. 59, no. 2, 2021.

[3] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian, "Resource management approaches in fog computing: A comprehensive review," *Journal of Grid Computing*, 2019.

[4] D. Kumar, G. Baranwal, Z. Raza, and D. P. Vidyarthi, "A systematic study of double auction mechanisms in cloud computing," *Journal of Systems and Software*, vol. 125, 2017.

[5] H. Wang, H. Tianfield, and Q. Mair, "Auction based resource allocation in cloud computing," *Multiagent and Grid Systems*, vol. 10, 2014.

[6] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer, 2004.

[7] N. Naz, A. H. Malik, A. B. Khurshid, F. Aziz, B. Alouffi, M. I. Uddin, and A. AlGhamdi, "Efficient processing of image processing applications on CPU/GPU," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–14, Oct. 2020.

[8] N. Felemban, Z. Lu, T. L. Porta, and K. Chan, "Video processing of complex activity detection in resource-constrained networks," in *Proc. of IEEE GlobalSIP*, 2016.

[9] N. Felemban, "On-demand video processing in wireless networks an application: Action recognition," Master's thesis, The Pennsylvania State University, 2016.

[10] X. Fang, Z. Cai, W. Tang, G. Luo, J. Luo, R. Bi, and H. Gao, "Job scheduling to minimize total completion time on multiple edge servers," *IEEE Tran. on Network Science and Engineering*, vol. 7, no. 4, 2020.

[11] L. Epstein and M. Levy, "Dynamic multi-dimensional bin-packing," *Journal of Discrete Algorithms*, vol. 8, no. 4, 2010.

[12] E. Segal-Halevi, A. Hassidim, and Y. Aumann, "MUDA: A truthful multi-unit double-auction mechanism," in *Proc. of AAAI*, 2018.

[13] A. G. Tasiopoulos, O. Ascigil, I. Psaras, and G. Pavlou, "Edge-map: Auction markets for edge resource provisioning," in *Proc. of IEEE WoWMoM*, 2018.

[14] D. T. Nguyen, L. B. Le, and V. Bhargava, "Price-based resource allocation for edge computing: A market equilibrium approach," *IEEE Transactions on Cloud Computing*, 2018.

[15] V. Farhadi, F. Mehmeti, T. He, T. L. Porta, H. Khamfroush, S. Wang, and K. S. Chan, "Service placement and request scheduling for data-intensive applications in edge clouds," in *Proc. of IEEE INFOCOM*, 2019.