APPENDIX A
LINEARIZATION OF THE CONSTRAINTS

The product $x_{sm}(t) \cdot x_{\sigma n}(t)$ appearing in the constraint (13) can be linearized through the variable $p^{s\sigma}_{mn}(t)$ according to a standard approach adopted in mathematical optimization (see e.g., [55]). To this end, we introduce the following linear constraints:

$$p^{s\sigma}_{mn}(t) \leq x_{sm}(t), \quad \forall s, \sigma \in S, \forall m, n \in M \tag{21}$$

$$p^{s\sigma}_{mn}(t) \leq x_{\sigma n}(t), \quad \forall s, \sigma \in S, \forall m, n \in M \tag{22}$$

$$p^{s\sigma}_{mn}(t) \geq x_{sm}(t)+x_{\sigma n}(t)-1, \quad \forall s, \sigma \in S, \forall m, n \in M \tag{23}$$

$$p^{s\sigma}_{mn}(t) \geq 0, \quad \forall s, \sigma \in S, \forall m, n \in M \tag{24}$$

In a similar way, the product $x_{sm}(t-1) \cdot x_{\sigma m}(t)$, which appears in the constraint (14), can be linearized through the variable $q^{s\sigma}_{mn}(t)$:

$$q^{s\sigma}_{mn}(t) \leq x_{sm}(t-1), \quad \forall s, \sigma \in S, \forall m, n \in M \tag{25}$$

$$q^{s\sigma}_{mn}(t) \leq x_{\sigma n}(t), \quad \forall s, \sigma \in S, \forall m, n \in M \tag{26}$$

$$q^{s\sigma}_{mn}(t) \geq x_{sm}(t-1) + x_{\sigma n}(t) - 1, \quad \forall s, \sigma \in S, \forall m, n \in M \tag{27}$$

$$q^{s\sigma}_{mn}(t) \geq 0, \quad \forall s, \sigma \in S, \forall m, n \in M \tag{28}$$

APPENDIX B
PROOF OF NP-HARDNESS

Proposition 2. *The* OMEC *problem is NP-Hard.*

Proof. The proof is based on showing that OMEC is a variant of the Bin Packing Problem, a well-known NP-Hard problem in combinatorial optimization (see e.g., [44]). More in depth, we show that OMEC is a generalization of the Bin Packing with Usage Cost (BPUC) problem, originally introduced in [56], which additionally includes 2 types of bin capacities and considers a multiperiod time horizon.

In the BPUC, we are given i) a set I of items and each item has a weight w_i with $i \in I$; ii) a set J of bins and each bin $j \in J$ has a capacity C_j. A bin is used when at least one item is put into it; when a bin j is used, we face a fixed non-negative cost f_j and a variable cost, namely a non-negative cost c_j for each unit of used capacity of j. The BPUC consists of assigning each item to exactly one bin so that the bin capacity is not exceeded and the total cost of using the bins is minimized. If we introduce 1) a binary variable $y_j \in \{0,1\}$ for every $j \in J$ that is equal to 1 if bin j is used and 0 otherwise, 2) a binary variable $x_{ij} \in \{0,1\}$ for every $i \in I$, $j \in J$ that is equal to 1 if item i is put in bin j and 0 otherwise, then the BPUC can be modelled as the following Binary Linear Program:

$$\min \sum_{j \in J} \left[f_j \cdot y_j + c_j \cdot \left(\sum_{i \in I} w_i \cdot x_{ij} \right) \right] \tag{29}$$

$$\sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I \tag{30}$$

$$\sum_{i \in I} w_i \cdot x_{ij} \leq C_j \cdot y_j \qquad \forall j \in J \tag{31}$$

$$y_j \in \{0,1\} \qquad \forall j \in J \tag{32}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in I, \forall j \in J \tag{33}$$

where the objective function (29) pursues the minimization of the total costs, the constraints (30) impose that each item is assigned exactly to one bin and the constraints (31) impose the capacity of used bins. We refer the reader to [56] for an exhaustive description of the Binary Linear Program used to model BPUC.

We establish a correspondence between the elements of OMEC and those of the generalized BPUC with multiple time periods and 2 type of bin capacities noticing that in OMEC:

1) the set of PSs S corresponds to the set of bins J of BPUC and each PS has a memory and a CPU capacity;
2) the set of VMs M corresponds to the set of items I and each VM requests an amount of memory and CPU that can be interpreted as two distinct types of weights of an item in BPUC;
3) each VM must be assigned to exactly one PS (this is like the single assignment of an item to a bin expressed by (30));
4) a PS must be powered-on to host a VM and the sum of memory and CPU requested by VMs assigned to the PS must not exceed the capacity of the PS - these two conditions correspond with having two distinct bin capacity constraints in (31), one for each type of capacity;
5) the objective function of BPUC contains a fixed cost part due to power-on PSs and a variable cost part that depend upon the capacity of the PSs that is consumed by the VMs (this is like the objective function of BPUC that include usage costs).

Given these correspondences, we can interpret BPUC as a special single-period and single-capacity type case of OMEC and since BPUC is NP-Hard also OMEC is NP-Hard. □

Algorithm 3 Pseudo-Code of the FFD algorithm

Input: $\gamma_m(t)$,$\mu_m(t)$
Output: $x_{sm}(t)$, flag_bin

```
1: VM_sorted=sort_VM(γm(t),'descend');
2: curr_s_index=1;
3: curr_VM_index=1;
4: curr_VM_assignment=init_array(|M|);
5: flag_bin=-1;
6: while flag_bin == -1 do
7:     curr_VM_assignment[VM_sorted[curr_VM_index]]=curr_s_index;
8:     if (check_CPU_mem(curr_VM_assignment,curr_s_index,
9: γm(t),μm(t))==true) then
10:        curr_VM_index++;
11:        if curr_VM_index==|M|+1 then
12:            flag_bin=1;
13:        end if
14:    else
15:        curr_s_index++;
16:        if (curr_s_index==|S|+1) then
17:            flag_bin=0;
18:        end if
19:    end if
20: end while
21: xsm(t)=write_conf(curr_VM_assignment);
```

APPENDIX C
FIRST-FIT DECREASING DESCRIPTION

We briefly describe the main steps of the First-Fit Decreasing (FFD) [44], an algorithm generally exploited to solve the Bin Packing Problem. The main goal of FFD is to pack as much items as possible in the bins, in order to reduce the number of bins that

are used. In our context, the items are the VMs and the bins are the PSs. As a result, the FFD aims to always limit the number of PSs powered on, and therefore to reduce the processing costs.

Alg. 3 reports the FFD pseudo code. The algorithm requires as input the CPU requirements ($\gamma_m(t)$) as well as the memory ones ($\mu_m(t)$), for the current TS. FFD then produces as output the VM to PS assignment ($x_{sm}(t)$), from which it is also possible to infer the number of PSs in AM. Moreover, a flag, denoted with `flag_bin`, is introduced. This flag is set to a value larger than zero if the algorithm is able to find an admissible VM to PS assignment. Otherwise, if the flag is set to zero or a value lower than zero, FFD has not been able to find a feasible solution.

The key idea of FFD is to analyze sequentially the CPU requests in each TS, and to allocate accordingly the VM to PSs in order to satisfy the CPU and the memory requirements. In particular, the VMs are initially sorted considering the amount of requested CPU in decreasing order (line 1). Then, the current PS index and the current VM index are initialized to 1 (line 2-3). In addition, the array storing the current VM to PS assignment is initialized to zero values (line 4). This array, which has size $|M|$, reports for each VM the index of the assigned PS. Moreover, the algorithm flag is initialized to a negative value (line 5).

The core of the algorithm (lines 6-20) is a while cycle, which is terminated if: i) all the VMs are assigned to the PSs, or ii) it is not possible to assign all the VMs to the PSs. In particular, the current VM (considering the VM index in the array of ordered VMs) is assigned to the current PS (line 7). Then, a check on the CPU and on the memory requested by the current set of VMs assigned to the current PS is performed (line 8). If the total amount of CPU and memory in each PS is below the maximum values, then the current VM is kept on the PS, and the following VM is analyzed (line 10). Clearly, if all the VMs have been analyzed and assigned (line 11-12), the flag is set to 1 and the while cycle is stopped. On the other hand, if the CPU and memory check fails, the algorithm considers the next PS (line 15) as candidate one to allocate the current VM. Note that, if all the PSs have been analyzed (without finding a feasible assignment), the flag is set to 0 (line 17) and the while cycle is ended. Finally, the algorithm copies the array of the current assignment in the matrix $x_{sm}(t), \forall s \in S, \forall m \in M$.

Focusing on the time complexity, the initial VM sorting (line 1) can be done in $\mathcal{O}(|M| \log |M|)$ time. Moreover, the initialization of the variables (lines 2-5) can be done in $\mathcal{O}(|M|)$. In addition, the check on CPU and memory is done in $\mathcal{O}(|M|)$ for the current PS. The external while cycle (lines 6-20) requires at most $\mathcal{O}(|M| \cdot |S|)$ iterations. Finally, the $x_{sm}(t)$ assignment can be done in $\mathcal{O}(|M| \cdot |S|)$ time. Overall, the FFD algorithm has a time complexity in the order of $\mathcal{O}(|M|^2 \cdot |S|)$.

Focusing on the space complexity, both the VM sorted (line 1) and the current VM assignment (line 7) are stored in two arrays of size $|M|$. Moreover, the VM to PS assignment (line 21) is stored in a matrix of size $|M| \cdot |S|$. Therefore, the overall space complexity is in the order of $\mathcal{O}(|S| \cdot |M|)$.

Appendix D
Next-Fit Decreasing Description

We then consider a modified version of the Next-Fit Decreasing (NFD) algorithm [44], with the goal of providing a solution able to distribute the CPU requests across the VMs. In contrast to FFD, in fact, the main goal of NFD is keep the PSs generally powered on, in order to reduce the CPU load on each PS.

Alg. 4 reports the NFD pseudo code. The input parameters of the algorithm are the CPU $\gamma_m(t)$ and the memory $\mu_m(t)$ requests by the VMs. The output is the VMs to PSs assignment $x_{sm}(t)$, as well a flag reporting the algorithm status. Initially, the VM are sorted, based on the decreasing amount of requested CPU (line 1). In addition, the total number of served VMs is set to zero (line 2), and the current PS and VM indexes are set to 1 (lines 3-4). Moreover, also the current VM to PS assignment is initialized to zero values (line 5). In addition, an array, used to store the number of PSs considered during the allocation of each VM, is also initialized (line 6). Finally, the algorithm flag is initially set to a negative value (line 7). Similarly to FFD, the core of the algorithm is a while cycle, which continuously checks the flag value (line 8-28). In particular, the current VM (taken from the ordered array of VMs) is assigned to the current PS (line 9). Then the CPU and memory requirements are checked for the current PS (line 10). If the total CPU and the total memory are higher than the maximum values, the current VM is deallocated from the current PS (line 12), and the number of PSs considered for the current VM is increased (line 13). On the other hand, if the current PS can host the current VM, the VM index is increased (line 15), as well as the total number of VM served (line 16). Clearly, if all the VMs have been successfully assigned, the flag is set to 1, and the while cycle ends (lines 17-19). In any case, the current PS index is increased (line 21). Differently from other classical implementations of the NFD algorithm, we reset the current number of PSs (lines 22-24) to check if the current VM can be assigned to PSs considered in the assignment of the previous VMs. In case there is at least one VM that cannot be assigned to any PS, its total number of considered PSs is equal to $|S| + 1$. We therefore perform a check on this condition, and eventually stop the algorithm (lines 25-27) if it is not possible to assign all the VMs to the PSs. Finally, the output matrix $x_{sm}(t) \forall s \in S, \forall m \in M$ is built (line 29).

In the following, we analyze the time complexity of the NFD algorithm. The sorting of the VMs (line 1) can be done in $\mathcal{O}(|M| \log(|M|))$ time. In addition, the initialization of the variables is performed in $\mathcal{O}(|M|)$ (lines 2-7). The check on the CPU and memory (line 10), as well as the check on the number of considered PSs for each VM (line 25), require at most $\mathcal{O}(|M|)$ iterations. In the worst case, the while cycle (line 8-28) is repeated for each VM and each PS, thus requiring $\mathcal{O}(|M| \cdot |S|)$ iterations. Finally, the construction of the output matrix $x_{sm}(t)$ is done in $\mathcal{O}(|M| \cdot |S|)$. Therefore, the overall complexity of NFD is in the order of $\mathcal{O}(|M|^2 \cdot |S|)$.

Finally, we analyze the space complexity of NFD. The sorting of the VMs (line 1), the current VM assignment (line 5), and the number of PSs for each VMs (line 6) require arrays of size $|M|$. Moreover, the VM to PS assignment (line 29) is stored in a matrix of size $|M| \cdot |S|$. Therefore, the overall space complexity is in the order of $\mathcal{O}(|S| \cdot |M|)$.

Appendix E
Lower Bound Description

The goal of the Lower Bound (LB) is to provide a methodology to easily compute the minimum values of processing and maintenance costs. Intuitively, the LB aims at maximizing the number of PSs in SM at each TS, in order to: i) reduce the processing costs, ii) reduce the maintenance costs (when the number of transitions is neglected). Moreover, we do not consider the impact of migrations costs, as well as the costs due to data transferring.

Algorithm 4 Pseudo-Code of the NFD algorithm

Input: $\gamma_m(t)$,$\mu_m(t)$
Output: $x_{sm}(t)$, flag_bin

```
1: VM_sorted=sort_VM(γm(t),'descend');
2: VM_served=0;
3: curr_s_index=1;
4: curr_VM_index=1;
5: curr_VM_assignment=init_array(|M|);
6: s_considered=init_array(|M|);
7: flag_bin=-1;
8: while flag_bin == -1 do
9:     curr_VM_assignment[VM_sorted[curr_VM_index]]=curr_s_index;
10:    if (check_CPU_mem(curr_VM_assignment,curr_s_index,
11: γm(t),μm(t))==false) then
12:        curr_VM_assignment[VM_sorted[curr_VM_index]]=0;
13:        s_considered(VM_sorted[curr_VM_index])++;
14:    else
15:        curr_VM_index++;
16:        VM_served++;
17:        if VM_served==|M| then
18:            flag_bin=1;
19:        end if
20:    end if
21:    curr_s_index++;
22:    if curr_s_index == |S| + 1 then
23:        curr_s_index=1;
24:    end if
25:    if (check_s_cons(s_considered,|S|) == false) then
26:        flag_bin=0;
27:    end if
28: end while
29: xsm(t)=write_conf(curr_VM_assignment);
```

More formally, we initially store the maximum value of available CPU from the PSs:

$$\hat{\gamma}^{MAX} = \max_{s \in S} \gamma_s^{MAX} \tag{34}$$

We then denote with $\hat{N}_{AM}(t)$ the minimum number of PSs to be powered on at a given TS t. $\hat{N}_{AM}(t)$ is defined as:

$$\hat{N}_{AM}(t) = \left\lceil \frac{\sum_{m \in M} \gamma_m(t)}{|S| \cdot \hat{\gamma}^{MAX}} \right\rceil \tag{35}$$

In the following, we compute the average PS utilization (denoted with $\hat{u}(t)$) as:

$$\hat{u}(t) = \frac{\sum_{m \in M} \gamma_m(t)}{\hat{N}_{AM}(t) \cdot \hat{\gamma}^{MAX}} \tag{36}$$

We then store the minimum value of P_s^{MAX} from the PSs:

$$\hat{P}^{MAX} = \min_{s \in S} P_s^{MAX} \tag{37}$$

Similarly, we store the minimum value of P_s^{IDLE} from the PSs:

$$\hat{P}^{IDLE} = \min_{s \in S} P_s^{IDLE} \tag{38}$$

We then introduce the variable $\hat{P}(t)$ to store the average power consumption. $\hat{P}(t)$ is computed as:

$$\hat{P}(t) = \left[\hat{u}(t)\left(\hat{P}^{MAX} - \hat{P}^{IDLE}\right) + \hat{P}^{IDLE}\right] \tag{39}$$

Finally, the LB for the processing costs is then computed by considering the contribution up to previous TS $(t-1)$ plus the contribution at current TS:

$$C_E^{PROC-LB}(t) = C_E^{PROC-LB}(t-1) + K_E \cdot \delta(t) \cdot \hat{N}_{AM}(t) \cdot \hat{P}(t) \tag{40}$$

Focusing then on the maintenance costs, we initially store in the variable $\hat{\phi}^{AM}$ the minimum PS FR:

$$\hat{\phi}^{AM} = \min_s \phi_s^{AM} \tag{41}$$

Similarly, we store in the variable $\hat{AF}^{SM}$ the minimum AF experienced by a PS always in SM:

$$\hat{AF}^{SM} = \min_s AF_s^{SM} \tag{42}$$

We then compute the number of PSs in SM as:

$$\hat{N}_{SM}(t) = |S| - \hat{N}_{AM}(t) \tag{43}$$

In the following, we compute the total AF as:

$$\hat{AF}^{TOT}(t) = 1 - \left(1 - \hat{AF}^{SM}\right) \cdot \frac{\sum_{t'=1}^{t} \hat{N}_{SM}(t') \cdot \delta(t')}{|S| \cdot \tau^{ALL}(t)} \tag{44}$$

The LB of the maintenance costs is then expressed with the following formula:

$$C_M^{TOT-LB}(t) = C_M^{TOT-LB}(t-1) + K_R \cdot \delta(t) \cdot |S| \cdot \hat{AF}^{TOT}(t) \cdot \hat{\phi}^{AM} \tag{45}$$

Finally, the LB of the total costs is computed as the summation of processing plus maintenance costs:

$$C^{TOT-LB}(t) = C_E^{PROC-LB}(t) + C_M^{TOT-LB}(t) \tag{46}$$

Three considerations hold for this LB. First, we assume that the most impacting effect on the PS consolidation is driven by CPU requirements, and not by memory ones.[7] Second, we compute the minimum number of PSs which could theoretically satisfy the CPU requirements. However, in a real environment, the actual values of the single CPU requests may impose to use a larger number of PSs in AM. Third, in the maintenance costs we consider only the impact of SM duration, which always introduces a positive effect. The number of transitions, which tends to notably increase the maintenance costs, is not taken into account in the LB computation.

Appendix F
Impact of VM Delay Constraints

Let us denote with $\theta^{(s1,s2)}$ the experienced delay of a single VM when it is migrated from PS $s1 \in S$ to PS $s2 \in S$. In addition, let us denote with θ_m^{MAX} the maximum delay that can be tolerated by VM $m \in M$, which we assume does not change across the set of TSs.

We first analyze the impact of delay on MECDC. To this end, we assume that a set of VMs, denoted with Υ, has a stringent delay constraint, i.e., $\theta^{(s1,s2)} > \theta_m^{MAX}, \forall m \in \Upsilon, \forall s1, s2 \in S, s1 \neq s2$. In particular, the VMs belonging to the Υ set, shall not be migrated across the PSs, in order to ensure the delay constraint. On the other hand, we assume that the set of $\overline{\Upsilon} = M \setminus \Upsilon$ VMs has a loose delay constraint, i.e., $\theta^{(s1,s2)} \leq \theta_m^{MAX}, \forall m \in \overline{\Upsilon}, \forall s1, s2 \in S, s1 \neq s2$. In other words, such VMs can be safely moved across the set of PSs, without violating their delay constraint.

In the following, we run MECDC by considering the Tot-CPU VM subset, 15 VMs, 4 PSs, and 5 years of lifetime. The remaining parameters are kept the same as in Sec. 8.1. Moreover, we consider

7. Even though the memory requirements are not considered in our computation, they could be potentially added as an additional term. We leave this aspect as future work.

TABLE 9
Impact of Delay on the MECDC algorithm.

$\|\Upsilon\|$	0	5	10	12	15
Energy [Wh]	22833	22833	22833	22833	22833
Tot. Migrations	2065	2065	2065	2065	2065
Migrations with Penalty	0	534	1437	1706	2065

different cardinalities of the Υ set, i.e., $|\Upsilon| = \{0, 5, 10, 12, 15\}$. In particular, when $|\Upsilon| = 0$ all the VMs can be safely migrated across the set of PSs. On the other hand, when $|\Upsilon| = 15$ all the VMs have stringent delay constraints. For the intermediate values, we instead randomly assign the VMs to Υ, until reaching the required cardinality $|\Upsilon|$.

We point out that, even in the case in which all the PSs are always powered on, there might be the need of migrating VMs across the set of PSs, due to the fact that: i) the CPU and memory requirements have to be satisfied in each TS, ii) the future requests in terms of CPU and memory are supposed to be not known in advance. Consequently, we keep track of: i) the total number of migrations, and ii) the number of migrations not satisfying the VM delay constraint, which we denote as migrations with penalty.

Tab. 9 reports the obtained results. Several consideration hold in this case. First, the migrations of VMs are a pretty rare event, i.e., a migration occurs on average every 13.25 $[days]$ for each VM. Second, the variation of $|\Upsilon|$ impacts the number of migrations with penalty. The larger is $|\Upsilon|$, the higher is the number of migrations with penalty. Third, the energy consumption does not vary, due to the fact that $|\Upsilon|$ is not explicitly considered by MECDC.

Overall, these numbers already prove that the impact on migration delay is rather limited, due to the fact that MECDC tends to keep a stable solution in terms of VM allocation and in terms of PSs powered on vs. time. However, we have designed a delay-aware version of the MECDC algorithm (called MECDC-DA), by introducing the following changes:

- in the Adaptive Bin Packing function, we check if the delay for moving the current VM from the current PS to the destination PS is lower than θ_m^{MAX} after line 12 of the function. If the constraint is not satisfied, the current VM is not assigned to the destination PS;
- lines 9-24 of phase 1 of the MECDC algorithm are repeated twice: in the first iteration, the algorithm considers only the VMs belonging to the Υ subset; in the second iteration, if the CPU and memory constraints are still not satisfied, all the VMs are considered as candidate ones to be migrated.

Tab. 10 reports the obtained results. Interestingly, MECDC-DA is able to reduce the number of migrations, as well as the migrations with penalty, for all the values of $|\Upsilon| > 0$. In the worst case ($|\Upsilon| = 15$), the number of migrations with penalty decrease from 2065 with MECDC to 1309 with MECDC-DA, thus reaching a 37% of reduction. Moreover, the results with lower values of $|\Upsilon|$ are even more promising. For example, when $|\Upsilon| = 10$, a total number of 7 migrations with penalty is experienced across the whole set of VMs, meaning that such event is extremely rare. Finally, we can also note that a minor impact on the energy consumption is experienced.

Summarizing, we have provided a demonstration of how the MECDC algorithm can consider the delay. In any case, the considered scenario assumes that live migrations can be performed for most of VMs without impacting their delay requirements (i.e., we assume that we are in the cases $|\Upsilon| \leq 10$). This can be achieved for example by reserving an amount of network bandwidth to perform the live migrations [22], [57]. However, we believe that future work can be done in the cases when $|\Upsilon| > 10$, which are representative e.g. when VMs are used for mission critical purposes.

TABLE 10
Impact of Delay on the MECDC-DA algorithm.

$\|\Upsilon\|$	0	5	10	12	15
Energy [Wh]	22833	22607	22623	22604	22605
Tot. Migrations	2065	1179	434	626	1309
Migrations with Penalty	0	3	7	266	1309

Algorithm 5 Pseudo-Code of the TS Variation Function

Input: $\gamma_m^{ORIG}(t)$,$\mu_m^{ORIG}(t)$,$D_{mn}^{ORIG}(t)$,κ
Output: $\gamma_m^{MOD}(t)$,$\mu_m^{MOD}(t)$,$D_{mn}^{ORIG}(t)$

```
1:  for m=1:|M| do
2:      curr_index=1;
3:      for t=1:|T| do
4:          if curr_index==κ then
5:              γ_m^MOD(t)=max_(t−κ+1:t) γ_m^ORIG(t);
6:              μ_m^MOD(t)=max_(t−κ+1:t) μ_m^ORIG(t);
7:              for n=1:|M| do
8:                  D_mn^MOD(t)=max_(t−κ+1:t) D_mn^ORIG(t);
9:              end for
10:             curr_index=1;
11:         else
12:             curr_index++;
13:         end if
14:     end for
15: end for
```

APPENDIX G
IMPACT OF TS DURATION

In the last part of our work, we have analyzed the impact of varying the TS duration on the obtained results. To this aim, we have considered the original input data from the Materna-3 trace, which we recall has a TS granularity of $\delta(t) = 5$ [minutes]. We have then applied the TS variation function reported in Alg. 5 to obtain the modified input data for each value of TS duration. In particular, the function requires as input the original CPU request $\gamma_m^{ORIG}(t)$, the original memory request $\mu_m^{ORIG}(t)$, the original amount of exchanged data between VMs $D_{mn}^{ORIG}(t)$ (which is computed with the procedure reported in Sec. 7), and the TS reduction factor, which is denoted by κ. The function is then able to produce as output the modified CPU request $\gamma_m^{MOD}(t)$, the modified memory request $\mu_m^{MOD}(t)$ and the modified amount of exchanged data between VMs $D_{mn}^{MOD}(t)$. Specifically, the new data is computed as the maximum value between the current TS and the last κ TSs (lines 2-13). In this way, we always ensure that the SLAs are met by the owner of the CDC.

We have then considered the Tot-CPU VM subset. Fig. 8 reports the variation of the total amount of requested CPU by the VMs vs. the TS index, for different TS duration $\delta(t)$. Clearly, the $\delta(t) = 15$ [minutes] and the $\delta(t) = 60$ [minutes] curves are obtained by setting $\kappa = 3$ and $\kappa = 12$, respectively. Interestingly, we can note that there is always a strong variability in the total requested CPU. In addition, as $\delta(t)$ increases, the total CPU also

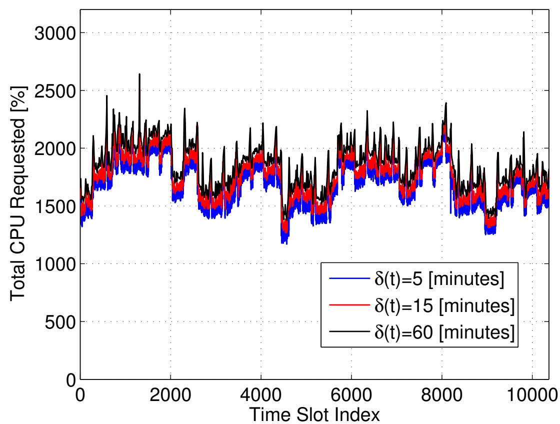

Fig. 8. Total Variation of CPU vs. the TS index for different TS duration $\delta(t)$(Tot-CPU subset).

TABLE 11
Total costs vs. the variation of the TS duration $\delta(t)$ for the different strategies (Tot-CPU subset).

	FFD	NFD	MECDC	LB
$\delta(t) = 5$ [minutes]	97230 [$]	27379 [$]	22833 [$]	14800 [$]
$\delta(t) = 15$ [minutes]	150560 [$]	24643 [$]	22910 [$]	15434 [$]
$\delta(t) = 60$ [minutes]	198560 [$]	24009 [$]	23532 [$]	16271 [$]

increases, as a result from the composition of the different terms coming from the single VMs.

We have then run the MECDC, NFD, FFD, and LB algorithms for the different values of TS duration $\delta(t)$, considering a total period of time equal to 5 [years]. Tab. 11 reports the obtained results in terms of total costs. As expected, the LB costs tend to increase when $\delta(t)$ is increased, due to the increase in the total request of CPU by the VMs (shown in Fig. 8). On the other hand, NFD is able to reduce the costs when the TS duration increases. By further investigating this issue, we have found that the main contribution to the reduction of the costs in this case is due to the migration costs, which tend to decrease. This is due to the fact that, by increasing the TS duration, the VMs tend to have requests that are more constant over time, resulting in a lower number of migrations. On the other hand, a large increase in the total costs is experienced by FFD. In this case, we have found that there are two servers which are frequently powered on / off, resulting then in large maintenance costs. Finally, the MECDC algorithm is able to achieve the lowest costs compared to NFD and FFD. Eventually, a slight increase is experienced when $\delta(t)$ is increased. Therefore, we can conclude that our solution is robust against the TS variation.