Order Matters for Accounting Idempotent Resources

Technical Report 0815-2018

Wenqiu Yu and Lin Zhong
Rice University, Houston, TX

Abstract

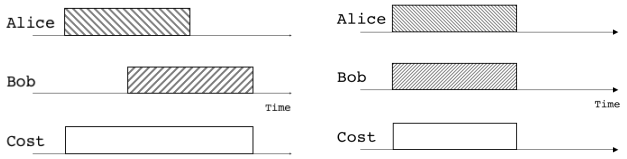
Modern operating systems track the resource usage by software principals to manage them and to inform developers and users. This work cares about an important type of resource that can be concurrently used by multiple principals and whose cost remains largely constant regardless of the number of concurrent users. We call such resources idempotent. Examples of idempotent resources include shared memory, wakelocks in Android, and I/O devices such as GPS receivers. Existing systems attribute the cost of an idempotent resource to software principals without considering the order in which the latter requested the resource. This work shows this practice is harmful and the order matters for an efficient system. We present an analytical framework with which both the resource accounting policy and a software principal's strategy can be formally analyzed. Using this framework, we reveal the flaws of existing accounting policies and show a simple policy that considers order can eliminate these flaws. We also briefly discuss how resource accounting in existing systems can be retrofitted to be order-aware and its broader implication on systems design.

1 Introduction

> *Alice enters a study room at 8pm and turns on the light. Bob enters the room at 10pm. Then Alice leaves at midnight. Finally Bob leaves at 2am and turns off the light since nobody is there.* How should Alice and Bob split the electricity bill of the light?

Modern operating systems (OSes) face similar questions in order to manage the system and to inform the developers and users. Alice and Bob are software principals, e.g., threads, processes, or applications. The light represents a system resource, either software or hardware. The problem is known as *resource accounting*.

In this work, we investigate the accounting of resources that are concurrently used by multiple principals. Specifically, we investigate *idempotent* resources, perfectly exemplified by the light in the opening example. A resource is idempotent if its cost does not increase when the number of its concurrent users increases beyond one. Many resources in modern computers are or can be approximated as idempotent. For example, shared memory is idempotent because it is the same physical memory no matter how many processes share it. GPS, as well as a Wi-Fi connection, has a significant idempotent component: keeping GPS on or the Wi-Fi connected costs significant amount of energy that is independent of how many principals need them to be on or connected.



(a) Unoptimized ***(b)*** Optimized

Figure 1. Properly attributing the cost has the potential to significantly reduce the cost by coalescing the resource uses.

The behaviors of principals are important for an economical system. The cost of resources can be significantly reduced if software principals align their uses properly. Figure 1 illustrates this: if the two apps, Alice and Bob, align their uses as in Figure 1b, the overall cost will be significantly lower as compared to the unoptimized case in Figure 1a. A good accounting policy should lead to optimized cost by incentivizing software principals to cooperate. In §2, we give a precise definition of idempotent resources.

Modern OSes attribute the cost of a resource based on its *state*, i.e., which principals are currently using it. They do not consider the order with which principals request the resource. To use the opening example again, they attribute the electricity bill to the users in the room without caring about the order in which they entered the room. While they achieve the secondary goal of holding principals responsible, they do not lead to optimized cost as we will show in this work. We survey the state of the art in §3.

This work argues that order matters for accounting idempotent resources. In support of this argument, we make the following contributions. First, we present a framework in which both accounting policies by the OS and the strategy employed by a software principal can be formally represented and analyzed in §4. Second, using this framework, we show that the commonly used order-unaware policies do not lead to optimized cost. Finally, we present two order-aware accounting policies in the simplest form in §5, and show that the *First-Only* policy guarantees an optimized system.

We further discuss the challenges and opportunities of order-aware policies in §6 before concluding with the limitations of this work and ways to overcome them in §7. Key results of this paper appear in an abbreviated form in [14].

2 Background

This work cares about resources that can be *concurrently used* by multiple principals. Often whether a resource can be concurrently used is a matter of abstraction and information hiding. For example, when cycle usage information is not available, a CPU core can be considered to be concurrently used by multiple threads. However, if cycle usage information is available, the core can be considered to be time-shared, instead, and each principal's usage can be precisely measured. Therefore, when we say *concurrently used by multiple principals*, we mean that there is no measurement about a principal's individual contribution to the usage except the fact that it is using the resource.

Idempotent resource: When the cost of a concurrently used resource does not increase as the number of users goes beyond one as shown in Figure 2 (Left), we say it is idempotent. Shared memory is idempotent because its cost, measured by memory used, does not increase at all with usage as measured by the number of processes sharing it. Many resources have an idempotent component because there is often a significant jump in cost when the usage becomes non-zero, as illustrated by Figure 2 (Right). For example, the power cost of CPU usage has an idempotent component because an idle CPU can enter a low-power mode. Once a software starts to use the CPU, it has to bring the CPU out of the low-power mode, incurring a constant power increase, which is the idempotent component, in addition to a power cost determined by the percentage utilization of the CPU by the software.

Resource Accounting: Resource accounting is about attributing the cost of a resource to its users. In a computer system, the OS attributes the cost to running software principals, i.e., threads, processes, or applications. It does so to hold principals responsible for their resource use, identify rogue principals, and encourage behaviors in principals that lead to a better system. A well designed policy should encourage behaviors by principals that lead to a reduced system cost.

Definition 2.1 (Economy)**.** When the cost of a resource is minimized, the system achieves *Economy* for this resource.

3 Related Work

Most practical systems and research prototypes account idempotent resources with a policy that attributes the cost at any time point equally to every concurrent principal, e.g. [3–6, 10]. We call this policy *Everybody Equally*. A few attribute idempotent cost to principals proportionally according to principals' non-idempotent costs, e.g., AppScope [13] attributing GPS's energy cost to an app based on the number

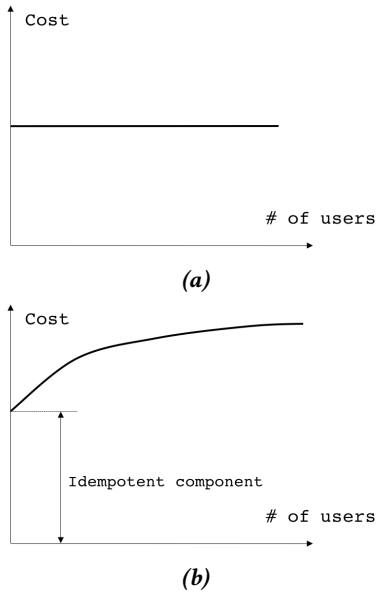


Figure 2. (a) Idempotent resource; ***(b)*** Resource with an idempotent component

of location updates the app requests. In several cases, the system does not bill principals for idempotent cost at all, and considers the consumption to be an inherent system cost, e.g., Joulemeter [7] and ECOSystem [15]. None considers the order of idempotent resource use by principals in accounting.

Given its popularity, we focus on the Everybody Equally policy, which has two variants. Everybody Equally (I) attributes the entire cost to every principal. As a result, each principal takes full responsibility of the cost, and has no incentive to align its use with others. Therefore, the system cannot achieve *economy* as we will elaborate in §5.1. On the other hand, Everybody Equally (II) divides the cost amongst the sharing principals. A key result we will present in §5.1 shows that a system with this variant cannot achieve economy either. Below we discuss several concrete examples.

Linux shared memory: In Linux, separate processes can get access to the same physical memory. Shared physical memory contains System V shared memory for communication, and shared libraries. It is an idempotent resource that can be attached by multiple processes concurrently. The kernel uses *resident set size* (RSS) to account the memory cost with Everybody Equally (I), which attributes the full cost of a shared memory to each sharing process. The kernel also uses *proportional set size* (PSS) to account the memory cost with Everybody Equally (II) [8].

Android Battery calculates the battery usage of every app using a linear model that relates the usage of various resources to energy consumption [2]. For idempotent resources, including `wakelocks`, Wi-Fi, GPS and other sensors, the accounting scheme employs Everybody Equally (I). For example, an app can acquire a `wakelock` to make sure the system

stays in the G0 state instead of entering a very low-power state. The energy cost of a `wakelock` is billed to all apps that hold it [3]. Apps can also acquire a lock for other resources like Wi-Fi to prevent it from sleeping. For GPS, an app is billed for the full energy cost no matter how many other apps are also using it.

Shapley value-based accounting: Dong et al. [4] consider an energy-consuming system of concurrent running processes to be a cooperative game. Their solution is based on the Shapley value, which attributes total surplus in a cooperative game to every player according to their contributions at any point in time. Since all principals as well as their contributions to an idempotent resource are equal by definition, Shapley value spreads the idempotent energy cost amongst all principals averagely, and is equivalent to Everybody Equally (II).

4 System model for idempotent resource

We next present a formal model for the relationship between a software principal and an idempotent resource it consumes. This model provides a unified framework for us to analyze accounting policies in §5.

The model consists of a server and clients. The server represents an idempotent resource; clients represent software principals. A client requests a service from the server and releases the latter via messages. When the server is serving at least one client, we say the sever is ON. Because the resource is idempotent, the cost of a service (per unit time) is constant, independent of how many clients are being served. When no client is being served, we say the server is OFF and for simplicity, assume the cost is zero.

4.1 Server

For simplicity, we assume the server has unlimited capacity so that a client always gets the requested service instantly. With this, the server's sole *objective* is to minimize its cost, i.e., to stay OFF as much as possible. The only mechanism the server can use toward its objective is the *accounting policy*: how its cost is attributed to the served clients. By properly billing the clients, the server encourages them to send requests in a way such that the server can be OFF as much as possible.

The server maintains a queue Q, called accounting queue, to record the served clients and the order of their requests. Once a client requests the service, it is added to the bottom of the queue if it is not already in the queue; when the client releases the service, it is removed from the queue. Since we only care about idempotent resources, after receiving a request from a client, the server ignores the latter's subsequent requests. In other words, the accounting queue will not have duplicate entries.

Server Knowledge: The accounting queue represents the server's knowledge. That is, the sever knows which clients it is serving and the order they first requested the service. The server does not have other knowledge. For example, it does not profile the clients. Its accounting policy is solely based on the accounting queue.

Policy Space: With this model, an accounting policy can be represented by a vector $\vec{B}$ with each element B_i representing the bill for the ith client in the accounting queue. Since we are interested in the properties of a given policy, we assume the server does not change its policy.

4.2 Client

A client's need for service is represented by *task*. When a task is generated, it has two parameters: the amount of time of service needed or *task length*, denoted by T, and the amount of time that it has to finish by or *task deadline*, whose length is denoted by duration D. We assume a client needs service when it generates a new task. Obviously, $D \geqslant T$ for a given task. The task must be finished D seconds after it is generated. To finish a task, the client can generate an arbitrary number of requests to the server, as long as the total service time equals the task length. We call the resulting requests a *request distribution*. This is illustrated by Figure 3, which shows two possible request distributions given the task deadline and length. A client can be in either of two states: when it is being served, it is ACTIVE; otherwise, it is IDLE.

Task state: A task also has three states: *urgent*, *normal*, and *done*. In order to finish the task before the deadline, the client must ensure the remaining time before the deadline (d) is no less than the remaining task time (x), i.e., $d \geqslant x$. When $d > x$, we say the task is *normal* because the client can still wait to request a service while meeting the deadline. Once $d = x$, the client must request a service immediately and remain ACTIVE until the deadline. In this case, we say the task is *urgent* because the client can not release the service. When $x = 0$, we say the task is *done*. Once a task is generated, its state is *normal* because $d = D \geqslant x = T$. The final state of every task is *done* at its deadline.

Because we assume a client will be served whenever it requests a service, its sole objective in the model is to minimize its *bill*, i.e., the cost attributed to it, under the constraint that all tasks get served before their deadlines. The only mechanism a client can use is the timing of its requests/releases, or the request distribution. It achieves the best case if its bill is zero. For example, if a server attributes the total cost to the first client who sends request to it, a client can avoid billing if it requests a service only when the server is already on and releases the service whenever it is about to be billed. This illustrates how the server's accounting policy can affect the behaviors of clients.

Client knowledge: First, we assume the server's accounting policy and cost rate are known to all clients. In practice, this is reasonable since the system designer is likely to disclose

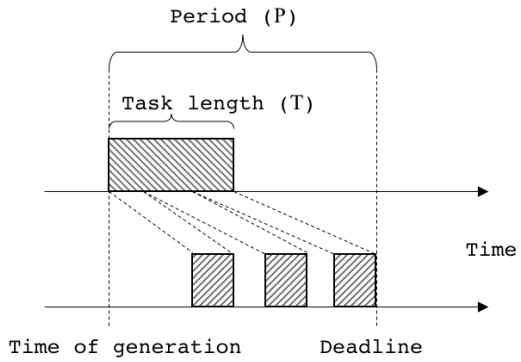


Figure 3. Two possible request distributions given the task deadline and length

this information to developers to encourage good behaviors in their software. Otherwise, it is often easy for a developer to infer such information by running multiple programs and inspecting their bills.

For clients to strategize to reduce their bills, they also need *runtime* information. At the extreme, the server could allow a client to read its accounting queue. Even if the server anonymizes the queue before sharing it with a client, malicious clients can construct a side channel using this knowledge, either to covertly communicate between themselves or to steal secrets about others. Therefore, we do not consider this level of knowledge as secure.

We primarily consider two levels of client knowledge. Let B denote the bill a client is receiving per unit time (if it is currently using the service) or would receive (if it is interested in using service). We note that the server can easily compute B for the "would" case by pretending the client is added to the end of the accounting queue. At the first level, the client only knows if B is zero or not. With this knowledge, a client is able to strategize so that it can use the service with a zero bill. This level of knowledge is also very secure because the client can infer very limited information about the accounting queue. The second level is one step further: the client knows the exact value of B. We note that for some policies, these two levels are the same.

Strategy Space: The client strategy can be represented by a state machine (S, δ, S_{task}, B). In this machine, the state set is $S = \{active, idle\}$. In this work, we limit ourselves to *memoryless* strategies such that the state transition function only considers current runtime information: its own state (S), the state of its task (S_{task}) and the bill (B). That is:

$$\delta : S \times S_{task} \times B \rightarrow S$$

Transitions happen when S_{task} or B changes. The state transition function of this state machine defines the client strategy. Based on the definition of task states, a client must be ACTIVE when its task is *urgent* and be IDLE when its task is *done*.

Definition 4.1 (Minimum Strategy)**.** We are interested in strategies that can minimize the bill of clients independent of the behaviors of other clients. We call such strategies *minimum* strategies.

4.3 System Design and Theorems

Given the client knowledge level, the designer of a system gets to decide the server's accounting policy. The goal is to identify the accounting policies that can achieve *economy*. Formally, we want to discover policies for which we can prove two theorems:

Existence of Minimum Strategy: there exists a Minimum strategy for clients.

Achievement of Economy: if every client assumes a minimum strategy, the server's cost will be minimized.

We note if the accounting policy is *balanced*, i.e., the sum of its bills to clients equals the cost of the server, the existence of a minimum strategy implies economy.

Theorem 4.2. *If there exists at least one minimum strategy for a balanced policy and every client adopts a minimum strategy, the sever's cost will be minimized and therefore achieves economy.*

Proof. Given that a minimum strategy exists and every client adopts a minimum strategy, each client has a minimum bill. Therefore, the sum of these bills is also minimum. Because the policy is balanced, the sum of bills equals to the cost of the server. Consequently, the sever's cost is minimum and it achieves economy. □

5 Accounting policies

We next use the model described above to analyze the common used Everybody Equally policy, which is order-unaware, and two simple, novel policies that consider the order. Our results show that the common used, order-unaware policy can lead to wasteful systems while one of the simple, order-aware policy *First-Only* can guarantee an economical, stable system.

5.1 Everybody Equally

We have already introduced the commonly used Everybody Equally policy at the beginning of §3. The problem with the first variant is obvious: a client will always receive the entire cost of the resource and as a result, has no incentive or information to strategize. This is true for both levels of client knowledge we consider in this work. The same is true for Everybody Equally (II) when a client has the first level of knowledge, i.e., knowing only whether its bill will be zero or not, because the client will always know that it will receive a non-zero bill.

Everybody Equally (II) with the second level of client knowledge is more interesting. In this case, a client knows how much its bill (B) is or would be. For this case, we prove the following negative result.

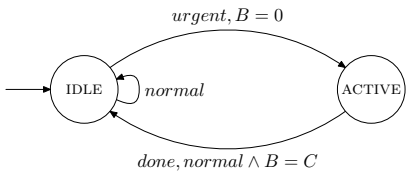


Figure 4. Minimum strategy represented by a two-state machine

Theorem 5.1. *There is no minimum strategy for clients if the clients are equally dividing the cost of an idempotent resource and each knows its own bill.*

Proof sketch: When bill B changes, a client who intends to starts a request does not know whether other clients will start simultaneously to lower the bills. Even if all clients with *normal* tasks agree to start a request when their bills would be of a certain value, they still cannot predict whether other clients whose tasks are *done* will generate new tasks in the future. If new tasks start before the current deadline of a client, that client may lower its bill by deferring its request. See Appendix for complete proof.

To consider order, the server only needs to bill clients based on their positions in the accounting queue. We next study two extremes of such order-aware policies: one that attributes the entire cost to the client at the top of the accounting queue; and the other to that at the bottom. Surprisingly, the First-Only policy leads to a stable and efficient system while the Last-Only does not. Because both policies only bill a single client at a time, a client will see its bill either as zero or the entire cost. As a result, the two levels of client knowledge are essentially the same. As a result, we no longer distinguish them below.

5.2 First Only

The first order-aware policy attributes all cost to the client on the top of the accounting queue, who requested the service before all other clients in the queue, henceforth the name *First-Only*. With this policy, a client will receive a zero bill until it rises up to the top of the accounting queue. Under such a policy, a clever client could only request a resource when it will receive a zero bill unless its task gets into *urgent*, which is illustrated by Figure 4.

We prove that the following positive result.

Theorem 5.2. *The strategy shown in Figure 4 is a minimum strategy under the First-Only accounting policy.*

Proof sketch: Why the strategy is minimum is quite intuitive. Before its task becomes *urgent*, a client requests a service when it receives a zero bill, and stops when the bill becomes non-zero. This way it maximizes its time using the service for free for any task. See Appendix for a complete proof.

Because the First-Only policy is balanced, by combining Theorem 5.2 and Theorem 4.2, we have:

Theorem 5.3. *When all clients adopt a minimum strategy such as the one in Figure 4, the server's cost will reach the minimum.*

Combined, the two theorems above say that if the server adopts the First-Only accounting policy, all clients will employ a greedy and selfish strategy as the one shown in Figure 4. And as a result, the server's cost will be minimized. Essentially, the First-Only policy aligns the interests of the server and the clients.

5.3 Last Only

We were surprised to find that the other simple, order-aware accounting policy is not as lucky. The Last-Only policy bills the entire cost to the last client that requests the resource, i.e., the last entry in the accounting queue. With this policy, once a client requests the service, it will be billed immediately for the entire cost of the server until it stops or another client (not yet in service) requests.

Theorem 5.4. *There is no minimum strategy for clients under the Last-Only accounting policy.*

Proof. Assume that Client X requests a service at t_1, and releases it at t_2. Suppose the next request from other clients starts at t_n. If $t_1 \leq t_n < t_2$, X will be billed from t_1 to t_n. Otherwise, it will be billed for the entire service time from t_1 to t_2. Its bill depends on the time t_n when another client requests, namely a future event that X cannot predict. To minimize its bill, X's strategy should be based on unpredicted future event, which is impossible for a real client because all physical principals behave as causal systems whose response depends on past and current states only. Therefore, there exists no current states based strategy for a client to minimize its bill under the Last-Only accounting policy. □

6 System Implications

We next discuss systems implications of implementing and using order-aware accounting policies, especially First-Only, in modern systems. We first survey how existing systems solve the above two problems in order to set the context for implementing the First-Only policy. We will end up discussing more philosophical implications of using resource accounting to encourage (indirect) cooperation amongst software principals in modern systems.

6.1 Existing systems

As existing systems mostly adopt Everybody Equally and there is no need for order information. For Variant I, because every sharing principal will be billed for the entire cost of the resource, the system does not even need to tally the number of sharing principals. For example, to account the energy cost of wakelocks. Each Android app simply tracks what wakelocks it has and the system calculates the energy cost based on

that. For Variant II, because the system has to divide the cost equally amongst all sharing principals, it often keeps a reference counter for the number of sharing principals. For example, shared memory structure in Linux uses a counter to keep track of the number of processes attached to it.

As shown in §5.1, under the Everybody Equally policy, clients cannot strategize to minimize their bills, even if they are aware of their own bills. As a result, although existing systems do provide mechanisms for principals to access their billing information, it is not intended for clients to strategize at runtime. Rather, billing information is usually intended for the use by human developers and users. Unix-like systems usually have billing information available to userspace principals via a virtual filesystem. For example, the memory usage per process in Linux is in `/proc/<pid>/statm`.

6.2 Implementing First-Only Policy

As the model in §4 suggests, an implementation of an accounting policy must be concerned with: (*i*) how the system keeps the server knowledge for each resource; (*ii*) how the system allows clients to access the client knowledge.

To implement any order-aware accounting policy, we must implement the accounting queue and its operation efficiently. For the First-Only Policy, we only need to care about three operations: (1) to identify the top of the queue, the first client; (2) to verify if a client is in the queue and append it to the end if it is not; (3) to remove a client from the queue. While a queue can be easily realized with a linked list, verifying if an entry exists in the list and removing an entry from the list require time proportional to the list length. To make both operation constant time, an idea is to add a resource pointer (or reference) in the task control block (TCB) of principal to represent the resource under question. The pointer is `NULL` when the principal is not in the queue. When adding the principal to the list, the pointer is set to the newly added list entry. To verify if the principal is in the queue, the OS only needs to check if the pointer is `NULL`. To remove it from the queue, the OS only needs to obtain its TCB, which is usually constant time in modern OSes, and remove the list entry pointed by the resource pointer.

The key for the First-Only policy to achieve the system Economy is a client's knowledge whether it receives or would receive a non-zero bill so that it can use the resource for free. This requires the system to communicate to all interested clients whether the resource is in use or not, i.e. whether the server is ON or OFF in our model. One way to realize this is to allow interested principals to register themselves for a resource and to use `signal` to notify them when the resource's status change.

Android sets timers for idempotent resources in every app structure. It can utilize these timers to identify the order of these apps, and a list in the operating system is not necessary. When doing accounting, Android traverses all apps to compute their battery usage separately. To achieve the First-Only policy, it just need to pick the app of the earliest timer when doing traversing, and add the battery uses to it.

6.3 Systems with gaming principals

We note that order-aware accounting policies open an interesting direction for OS research. Today's OSes do not intend principals to cooperate. Rather, the OS gather all the information and direct how principals should behave, e.g. via scheduling. This philosophy is apparent in recent works into reducing application activities for energy saving, e.g., timer coalescing [11] and others [9, 12]. There is a fundamental limit to this philosophy: the OS does not have all the semantics of principals, e.g., the deadline to acquire a resource. One way to overcome this limitation is to augment the API with which a principal requests a resource, to add information about deadline, e.g. [1]. This not only breaks existing APIs but also places all the burden (and authority) on the OS. We note the real-time OSes usually supply interfaces for principals (often called tasks) to inform the OS about their scheduling deadlines.

Order-aware accounting policies reflects a different philosophy: the OS provides necessary information (and incentive) to encourage cooperation amongst principals who share resources. This shifts much of the decision burden to the principals. Implementing these policies requires the OS to provide new APIs for principals to register their interests and to deliver corresponding information to registered applications. Such augmentation, however, would be largely orthogonal to existing APIs for requesting resources.

We further note that order also matters, more broadly, for resources whose costs increase non-linearly with the number of concurrent users. Idempotent resources are just a very special case for such *non-linear* resources. We suspect non-linear resources, in general, will require different accounting policies which we leave for future work.

7 Concluding Remarks and Future Work

This work presents a small, highly theoretical contribution to the design of operating systems, specifically how an OS should bill its principals for resource use, i.e., *resource accounting*. We study a special type of resource, idempotent resource, whose cost does not increase as the number of users increases. While existing systems use state-based, order-unaware policies to account their costs, we show, using an analytical model, this is suboptimal and a simple, order-aware policy can lead to optimal system resource use in theory.

The root cause of the failure of order-unaware policies is that they fail to incentivize principals to "cooperate", specially to coalesce their resource uses. The First-Only policy overcomes this exact problem by letting principals know whether they can use the resource for free or not and only billing the earliest one to use it. To use the opening example again, the system allows all users to know whether the light

is on or off in the room and bill the user who is in the room before others for the electricity. This incentivizes users to share the light as much as possible, which is good for the system; it also awards users who can wait to take a free ride.

Limitations of this work: There are two major limitations to this work. First, in order to make our analysis tractable, we had to make simplifying assumptions, especially in the model presented in §4. We limited ourselves to memoryless servers and clients; we did not consider the cost of state transition for either server or client, which could be nontrivial in some cases. Nevertheless, these issues are easy to address albeit they can complicate the math.

The second, more delicate limitation is our lack of empirical data. Toward showing the effectiveness of order-aware policies, we will not only realize a prototype OS, likely based on Android, as described in §6, but also revise existing apps or develop new ones to exploit the First-Only accounting policy, and subject them to real mobile users. We expect this to be significantly more challenging and time consuming.

Acknowledgements

This work is supported in part by NSF Award #1701374.

References

[1] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: Interfaces for better power management. In *Proc. ACM MobiSys*, 2004.

[2] Android BatteryStats. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/BatteryStats.java.

[3] Android PowerManager.Wakelock. https://developer.android.com/reference/android/os/PowerManager.WakeLock.

[4] M. Dong, T. Lan, and L. Zhong. Rethink energy accounting with cooperative game theory. In *Proc. ACM MobiCom*, September 2014.

[5] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *Proc. USENIX OSDI*, pages 323–338, 2008.

[6] M. Gordon, L. Zhang, B. Tiwana, R. Dick, Z. M. Mao, and L. Yang. PowerTutor, 2009. http://ziyang.eecs.umich.edu/projects/powertutor.

[7] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proc. ACM Symp. Cloud computing*, pages 39–50, 2010.

[8] Linux Proc Filesystem. http://man7.org/linux/man-pages/man5/proc.5.html.

[9] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *USENIX Annual Technical Conference*, pages 563–575, 2015.

[10] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *ACM SIGPLAN Notices*, pages 65–76, 2013.

[11] Timer Coalescing. https://en.wikipedia.org/wiki/Timer_coalescing.

[12] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *Proc. ACM MobiSys*, 2013.

[13] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *USENIX Annual Technical Conference*, pages 1–14, 2012.

[14] W. Yu and L. Zhong. Order matters for accounting idempotent resources. In *Proc. ACM APSys*, August 2018.

[15] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review*, 36(5):123–132, 2002.

Appendix: Complete Proofs

We provide the complete proofs of Theorem 5.1 and Theorem 5.2.

Proof for Everybody-Equally Policy

Theorem 5.1 says there is no minimum strategy for Everybody Equally (II). Our key insight towards proving it is that a client has to employ different strategies to minimize its bill in different contexts, i.e., task states and deadlines of other clients. However, a minimum strategy must be context-free, optimizing the bill in all contexts since a client has no knowledge regarding the context. Therefore, such a minimum strategy does not exist for Everybody Equally (II).

Proof. Assume a minimum strategy, called *Strategy X*, exists, and its transition function is δ. First, because a client must turn to ACTIVE when its task becomes *urgent* and turn to IDLE when its task is *done*. We have:

$$\delta : \text{IDLE} \times urgent \times * \rightarrow \text{ACTIVE} \tag{1}$$

o

$$\delta : \text{ACTIVE} \times done \times * \rightarrow \text{IDLE}. \tag{2}$$

where $*$ indicates the bill can be any value.

Next, assume there are three clients. Below we will construct three contexts to show that a client cannot minimize its bill by applying *Strategy X* in all three contexts. In *Strategy X*, transition happens only when task state or B changes, and B various with the behaviors of other clients. The first two contexts illustrate two simple scenarios to show the actions a client must take to minimize its bill when others start and stop their requests, respectively. We then show these actions will not lead to a minimum bill in the last context. Therefore, a context-free minimum strategy does not exist.

Context I: As in Figure 5a, $client_1$ and $client_2$ generate tasks at time t_0, and $client_3$ has no task. Assume that $D_1 > D_2$, which means the deadline of $client_1$'s task is later than $client_2$. Both clients have identical task length $T_1 = T_2 = R$. $R < D_1 - D_2$.

According to Equation 1, at time t_1, $client_2$'s task turns to *urgent* and it becomes ACTIVE. For $client_1$, B decreases to $C/2$ at t_1, and it will take a transition action. It has two choices for transition.

First, $client_1$ becomes ACTIVE at t_1 as in Figure 5a. According to Everybody Equally (II), its final bill is $R/2$.

Alternatively, $client_1$ can stay IDLE at t_1. As in Figure 5a, the next change of task state or B occurs at t_2 when $client_1$ ends its request, which is the next transition time for $client_1$. $client_1$ will start its request at or after t_2, and is the only ACTIVE client in the system then. It's billed time will be R.

Since $client_1$ has a lower bill in the first choice above, an IDLE client should turn to ACTIVE when $B = C/2$ in a minimum strategy. There must be

$$\delta : \text{IDLE} \times * \times (B = C/2) \rightarrow \text{ACTIVE}. \quad (3)$$

in *Strategy X*. An IDLE client turns to ACTIVE when another client is ACTIVE.

Context II: As in Figure 5b, both $client_1$ and $client_2$ generate tasks at t_0, while $client_3$ generates a task at t_2. The deadline of $client_1$ and $client_3$ is t_3, and that of $client_2$ is t_1. The task lengths $T_1 = 2R, T_2 = T_3 = R$ and $t_2 - t_1 > R$.

Assume that in *Strategy X*, a client stays ACTIVE when its task is *normal* and $B = C$, i.e. others release the server and it is the only ACTIVE client in the system.

Take account of Equation 1-3, a client turns to ACTIVE when its task is *urgent* or another client is ACTIVE, while staying ACTIVE if its task is *normal* and it is the only ACTIVE client. The request distribution is as in Figure 5b. The bill of $client_1$ is $3R/2$ according to Everybody Equally (II).

However, if $client_1$ releases at t_1, and synchronizes its next request with $client_3$ after t_2, its bill will be R, which means $3R/2$ is not minimum. Therefore, the assumption is false. In *Strategy X*, an ACTIVE client turns to IDLE if its task state is *normal* and $B = C$.

There must be

$$\delta : \text{ACTIVE} \times \textit{normal} \times (B = C) \rightarrow \text{IDLE} \quad (4)$$

in *Strategy X*. An ACTIVE client turns to IDLE if others release the server.

Context III: As in Figure 5c, assume that $client_3$ generates three tasks of identical length R at t_0, t_1 and t_2. $client_2$ generates two tasks of length R at t_0 and t_2, and $client_1$ generates a task of length R at t_0. Every task is generated immediately after the previous deadline. The last deadlines of $client_1$ and $client_3$ client are t_3, which is earlier than the task deadline of $client_2$.

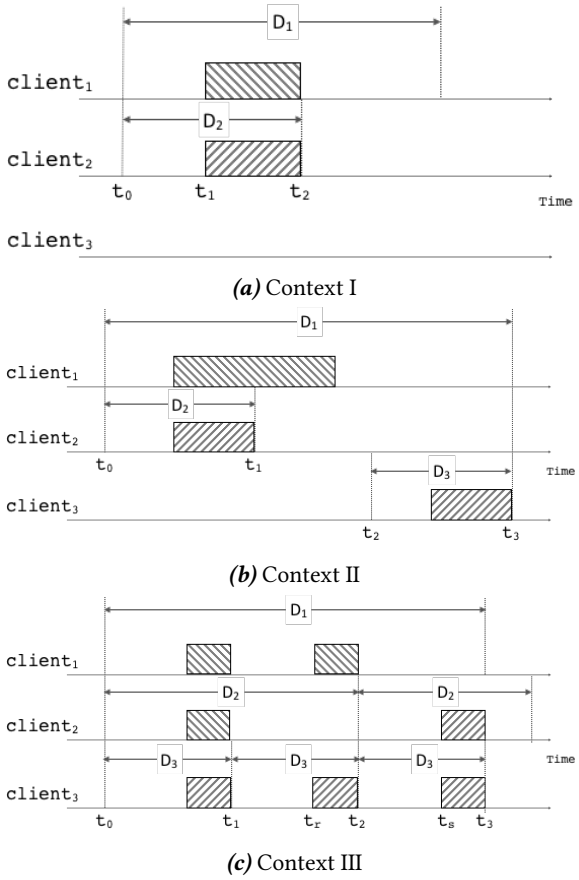
According to Equation 1-4, a client turns to ACTIVE when its task is *urgent* or another client is ACTIVE, while turning to IDLE if its task is *normal* and others are IDLE. the request distribution is as in Figure 5c. $client_1$ is billed for $5R/6$.

Whereas, if $client_1$ does not request at t_r, instead starts a request at t_s when $client_3$ starts a new request, its bill will be $2R/3$. Thus, $5R/6$ is not minimum. *Strategy X* whose transition function containing Equation 1-4 is not a minimum strategy for *Context III*.

Consequently, there is no minimum strategy that works for all contexts. Under the Everybody Equally Policy (II), a client has to employ different strategies to minimize its bill for different contexts. □

Proofs for First-Only Policy

We next prove Theorem 5.2 that there exists a minimum strategy under First-Only. Since the policy is balanced, if such a minimum strategy exists, the server achieves *Economy* based on Theorem 4.2. Below we prove that *Strategy Y*, illustrated in Figure 4, is a minimum strategy for First-Only.



(a) Context I

(b) Context II

(c) Context III

Figure 5. Client strategy states machine of Strategy X

Strategy Y

Let $D_{i,m}$ denote the duration between the time the m_{th} task of $client_i$ is generated and its deadline t_d. The task time in this duration is T_i. At any time point $t \in D_{i,m}$, the remaining task time is represented as $x_i(t)$. The common knowledge B indicates how much a client is or would be accounted if it becomes *active*. B has two possible values $\{0, C\}$.

The strategy below describes the principles of *Strategy Y*.

- In a duration, the initial client state is IDLE and the task state is *normal*. $x_i(t) = T_i$, i.e. the remaining task time is equal to the total task time.
- A task turns to *urgent* when $x_i(t) = t_d - t$, i.e. the remaining task time is equal to the remaining time before the deadline.
- A task turns to *done* when $x_i(t) = 0$, i.e. the task is finished and the remaining task time is zero.
- An URGENT/DONE task state lasts to t_d.
- At t_d, the task state of a client turns to *done* if it's *urgent*.

- An IDLE client sends a request and turns to ACTIVE when its task is *urgent* or $B = 0$.
- An ACTIVE client releases the server and turns to IDLE if its task is *normal* and $B = C$.
- An ACTIVE client also turns to IDLE if its task state becomes *done*.

When a client is ACTIVE, the server is ON. The task state of the client is *urgent* or it has a zero bill. Therefore, we have:

Proposition 7.1. *If a client has a non-zero bill, its task state must be urgent.*

Otherwise it will release the server.

Since only *normal* tasks can change their states, if the task state turns to *urgent* at time t_u, it must be *normal* before t_u and be *urgent* after t_u. According to the strategy, when a client task becomes *urgent*, it turns to ACTIVE. We have

Proposition 7.2. *In a duration, if the task state of a client turns to urgent at t_u, it is normal before t_u. After t_u, the client is* ACTIVE.

Proof for Theorem 5.2

Based on the propositions of *Strategy Y* above, we prove that *Strategy Y* is a minimum strategy. Since the bill is proportional to the time that a client has a non-zero bill with First-Only, *Strategy Y* is a minimum strategy if a client can minimize its non-zero billed time by employing this strategy.

Proof. Suppose $client_i$ adopts *Strategy Y* and generates a request distribution in duration $D_{i,m}$. We will prove that $client_i$ cannot reduce its bill by modifying its request distribution in this duration under the same condition.

First, according to Proposition 7.1, if $client_i$ has a non-zero bill in $D_{i,m}$, there is a moment its task state is *urgent*.

Next, suppose $client_i$ turns to *urgent* at time t_u. We will prove that it has a non-zero bill only after *urgent*, and it cannot reduce the billed time by shifting requests after *urgent*.

According to Proposition 7.2, the task state of $client_i$ is *normal* before t_u. According to Proposition 7.1, $client_i$ has a zero bill before t_u. It must have a non-zero bill after t_u.

Since $client_i$ has a non-zero bill only after t_u, it can only reduce the time that it has a non-zero bill after t_u. Dividing its request after t_u into multiple slices, in order to reduce the time, $client_i$ should shift at least one of these slices to where it is IDLE before the deadline.

Since $client_i$ is ACTIVE after t_u according to Proposition 7.2, the IDLE state only occurs before t_u. In order to reduce the billed time after t_u, $client_i$ must shift at least a request slice after t_u to where before t_u.

Since the task state of $client_i$ is *normal* before t_u, when $client_i$ is IDLE, the server is OFF. Otherwise the $client_i$ will start a request according to the strategy. Therefore, if $client_i$ shift a request slice to where it is IDLE, it would be the first one to send a request, and be billed as the top one in the accounting queue until the end of the request slice. Finally, if $client_i$ shifts a request slice after t_u to where before t_u, it would still be billed for the same request slice. The time it is billed does not decrease.

Consequently, $client_i$ cannot reduce its time of non-zero bill by modifying its request distribution generated based on *Strategy Y*. *Strategy Y* is a minimum strategy. □