The Burden of Time on a Large-Scale Data Management Service

Etienne Mauffret, Flavien Vernier, Sébastien Monnet

▶ **To cite this version:**

Etienne Mauffret, Flavien Vernier, Sébastien Monnet. The Burden of Time on a Large-Scale Data Management Service. Advanced Information Networking and Applications (AINA-2023), Mar 2023, Federal University of Juiz de Fora, Brazil, France. hal-04040554

HAL Id: hal-04040554
https://hal.science/hal-04040554

Submitted on 22 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Burden of Time on a Large-Scale Data Management Service

Etienne Mauffret[1 2][0000−0001−8444−6293], Flavien Vernier[2][0000−0001−7684−6502], and Sébastien Monnet[2][0000−0002−6036−3060]

[1] École Normale Supérieure de Lyon - LIP, 46 Allée d'Italie 69007 Lyon, France
{firstname.lastname}@ens-lyon.fr
[2] Université Savoie Mont Blanc - LISTIC, 5 chemin de bellevue, Annecy-le-vieux, 74 940 Annecy, France, {firstname.lastname}@univ-smb.fr

Abstract. Distributed data management services usually run on top of dynamic and heterogeneous systems. This remains true for most distributed services. At large scale, it becomes impossible to get an accurate global view of the system. To provide the best quality of service despite this highly dynamic environment, those services must continuously adapt. To do so, they monitor their environment and store events and states of the system in memory. In this paper, we propose a model to formalize this memory and three strategies to use it. We explain the theoretical difference between those strategies and conduct an experimental evaluation. We show that providing the ability to "forget" old events leads to better performance. However, using fading events (events that progressively disappear) rather than events that suddenly disappear leads to even better performance and is more adequate to detect habits and recurrent behaviors.

1 Introduction

With the advent of large scale services and applications comes the rising of large scale distributed data management systems. Those systems face an ever-changing and heterogeneous environment: new components come and go continuously. Such systems also face dynamic workloads: the number of users may vary, their activity (frequency, type, etc.) their location, etc. Many services are built on top of such volatile conditions and should be able to provide a good quality of service regarding the evolution of the system or the workloads induced by users. There are many works that take past into consideration to adapt to the future environment behavior. Every distributed system that aims to adapt has to grab and store information about its environment. For instance, in [1], the authors present a leader election algorithm in a dynamic network. They combine a wave algorithm with the temporally ordered routing algorithm to face the evolving network. In [2], the authors propose a way to build failures detectors in a Mobile Ad-Hoc Network. In the Linux kernel memory management mechanisms, there is a lot of work concerning this problem (access/dirty bits, double LRU lists, shadow page-cache, etc.). However, even if the weight given to store past events is very important, in the field of distributed systems, to our knowledge, there is no work focusing on studying various approaches and their impact on the system performance. For instance, in the context of reputation systems [3], great importance is given to the

weight that should be given to the past actions of other nodes according to their freshness, frequency, and regarding the other actions. But this is done in a particular context, where nodes monitor each other and try to attribute a reputation level to others.
In a previous work, we presented CAnDoR [4], a dynamic data placement algorithm taking into account the consistency protocol in use, the workload and an approximate view of system state. In such a service, the local representation of the global environment and the management of past events have a crucial impact on the quality that the service can provide. Indeed, giving the same importance to an "old" event and an event that just occurs may lead to bad adaptation choices and alter the service. This method can also be used to build learning algorithms like proposed by [5] or [6]. For example, let us consider a dynamic data placement service that only consider the workload induced by users. The service will place data such that user interested in those data can access them quickly. To determine which users are interested in a piece of data, the service could compare users accesses. If the service does not differentiate access that was just requested to that access that was requested some time before, it cannot precisely compute which users are interested in the piece of data now. Old requests would still have an important impact on the data placement choice, which will not be as good as it could be. However, only considering the most recent requests could equally lead to bad decisions.
In this paper, we propose a formal model to represent the local memory of a large scale service, and we use this model to compare the impact of three different strategies to use local memory of events to build an adaptable service. In the rest of this paper, we first describe a model to represent a knowledge memory (the set of stored past events and their associated weight) and 3 strategies to use this knowledge (Sect. 2). We then present an experimentation where we implement those strategies in a simulator to observe their impact on a distributed data placement service (Sect. 3). Lastly, we conclude and discuss limitations and future work (Sect. 4).

2 Memory Model

Let $\mathscr{S}$ be a dynamic service deployed on a large scale system. This service runs over a period of time $\mathbb{T} = [0; t_\infty]$. Traditionally, a large scale system can be viewed as different nodes that communicate by exchanging messages. Each node in the system participating in the execution has a local instance of this service. From now, only the point of view of a node is considered, thus when we refer to the service $\mathscr{S}$, we refer to the local execution of that service.
During the run, each instance of $\mathscr{S}$ builds a local memory that stores the history of events. The memory is composed of all events known by the node and some useful information attached to them. More precisely, an event e is defined by a tuple $e = \langle a_e; s_e; t_e \rangle$ with:

- a_e: the action performed,
- s_e: the source of the event,
- t_e: the time at which the event occurred.

According to the need of services, it may be possible to add components to this tuple, such as a recipient d_e for example. In this paper, we only consider the triplet $\langle a_e; s_e; t_e \rangle$. The set of every event known by a node is denoted $\mathbb{E}$. While

building a memory, it is interesting to consider the "importance" of each event. This "importance" can be represented by a weight that depends on the age of the event, its type, its index in the memory both any combination of factors. In order to determine the weight of each event in the memory, we define a weighting function noted $w_{\mathscr{S}}$, it provides a value between 0 and M that depends on the time t_k, an event e and the elapsed time since e occurred $\Delta_e^k = t_k - t_e$. For example, a service could want to forget or less consider the oldest events, or consider the impact of an event depending on its own history.
In this paper, the focus is on 3 strategies for using the memory. To model these strategies, let us define associated services. To explain the behavior of those services, we use a simple example, illustrated by Fig. 1. In this example, the current time is denoted t_k and past events are denoted from e_0 to e_{i+7}.

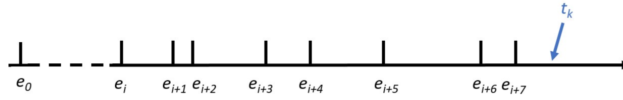

Fig. 1. Events occurrence at time t_k in a service

When an event occurs on a node, the local service adds this event to its memory with the timestamp t_e, the time of occurrence. When the service uses the event e at time step t_k, it computes the weight of this event using its weighting function: $w_{\mathscr{S}}(e, \Delta_e^k)$. This function can define the chosen strategy. We describe here ways of using memory using the weighting function. We consider that the memory always stores every known event with every piece of information. However, the strategy employed dictates how this knowledge can be used. It is important to notice that one service can rely on different strategies for different computations and decision processes.

2.1 Timeless Service

First, consider $\mathscr{T}$, a service that uses all the events the same way without taking their age into account. We call it a *timeless service*. Using the simple example, $\mathscr{T}$ will use every event from e_0 to e_{i+7} with the same weight. This service is illustrated by Fig. 2.

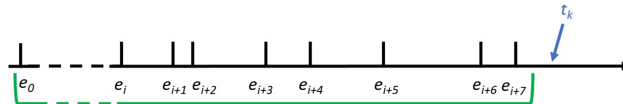

Fig. 2. Weight of event at time t_k with timeless service

An intuitive way to use the memory is to consider every event with an equivalent weight. For example, if a service needs to use the number of requests, it just needs

to count the number of events in the memory. This service does not consider the "age" of events and use a constant function as weighting function. An example of weighting function could be:

$$\forall(e, t_k) \in \mathbb{E} \times \mathbb{T}, w_{\mathscr{T}}(e, \Delta_e^k) = c.$$

One of the main limitation of this strategy is that it could induce inertia and prevent the service from reacting quickly enough. Indeed, by construction, the service does not make any difference between a recent and old event. Therefore, old events will have a non-negligible impact on the decision to make an adaptation or not. For example, let us consider $\mathscr{T}$ to be a data placement service try to place data near users interested in those data. If a user was previously interested in the data but is not active anymore, but a new user is interested in this data, $\mathscr{T}$ will give the same importance to each request. In this situation, illustrated by Fig. 3, $\mathscr{T}$ will keep putting the data closer to the user that was active at the beginning of the execution.

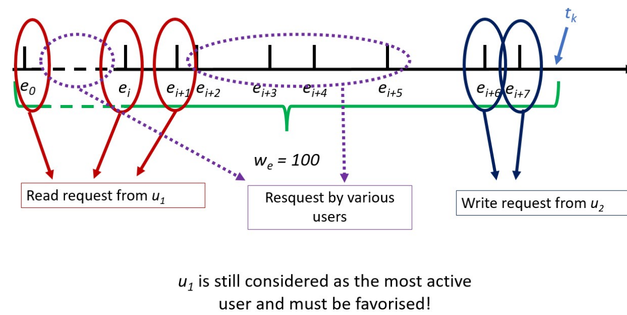

Fig. 3. Weight of event at time t_k with a timeless service

This strategy of memory usage is quite efficient if the sources of events behave roughly in a consistent way during the execution, i.e., if users send the same kind of request periodically during the execution, if the communication between nodes uses the same patterns, etc. If this behavior is expected from the participants, then the service will be able to adapt correctly without being disturbed by "noises" or small irregularity in habits. We expect such behavior in some specific services, such as a personal data service, where a user tends to access the same restricted data over time.

However, this strategy could also lead to an "average solution" for every user. If (a subset of) users have roughly similar habits and activities, the service will tend to consider those users as equivalent and propose a solution equally efficient for all of them. It could be possible to provide a better service for every user by trying to adapt to the users more frequently. Moreover, such a strategy does not take failures into account: every event is weighted equivalently even if one of the sources has failed, crashed or left the system.

2.2 Sliding Window Service

While the timeless strategy seems intuitive and can be efficient in some scenarios, it is usually more efficient to use a finer strategy. Indeed, as we previously saw, a timeless strategy leads to slower adaptations. In many services, it is preferable to provide a service that can adapt itself quickly. For example, a failure detector [7] needs to detect as soon as possible which nodes are still correct, an old event should not be considered. We then consider $\mathscr{W}$, a service that uses the concept of sliding window[8]. A period τ is determined and any event older than τ is ignored.

This service is illustrated by Fig. 4. In this example, the events $e_1, \ldots, e_{i+5}$ are older than τ. Thus, those events will not be used in adaptation computations. Events e_{i+6}, e_{i+7} occurred since less than τ time units and are therefore still in the time window: they will be used in adaptation. The service is thus able to

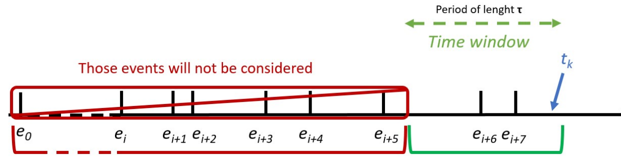

Fig. 4. Weight of event at time t_k with sliding window service

adapt quickly, as it only takes into account recent events. Any decision will be made according to recent behavior and any source that became inactive will lost its impact. This can be achieved by considering a step weighting function $w_{\mathscr{W}}$:

$$\forall (e,t_k) \in \mathbb{E} \times \mathbb{T}, w_{\mathscr{W}}(e, \Delta_e^k) = \begin{cases} c, \ \textbf{if } \Delta_e^k \leq \tau, \\ 0, \ \textbf{if } \Delta_e^k > \tau. \end{cases}$$

With such a function, the service will assign a weight of 0 the events older than τ and thus remove any impact they could have on the adaptation. This is illustrated by the example on Fig. 5. In this example, events that occurred before e_{i+6} are given a weight of 0 and thus are ignored by $\mathscr{W}$, while other events are assigned the weight $c = 100$.

When using a sliding window approach, it is important to correctly tune the size of the window, i.e., the period τ. This period can be either static or dynamically evolve through the execution, based on the number and kinds of events, for example. If the period is too short, too few events will be used, important events could be ignored, leading the service to take bad decisions. On the opposite, if the period is too long, too many events will be considered which could also lead to bad decisions, according to the service and its needs.

Some works use a double sliding windows approach. In this case, once an event is out of the first windows, a new weight is assigned to it until it leaves the second

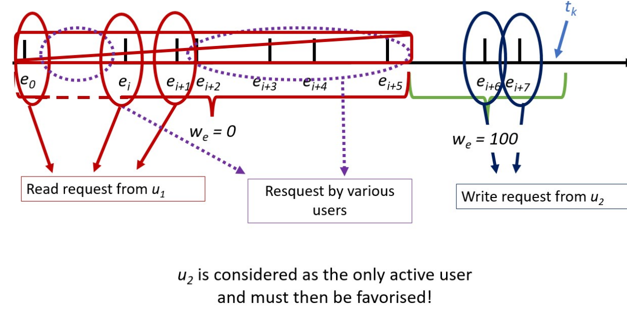

Fig. 5. Weight of event at time t_k with a timeless service

windows, with $c_1 > c_2, \tau_1 < \tau_2$ to represent those values:

$$\forall (e, t_k) \in \mathbb{E} \times \mathbb{T}, w_{\mathscr{W}}(e, \Delta_e^k) = \begin{cases} c_1, \textbf{ if } \Delta_e^k \leq \tau_1, \\ c_2, \textbf{ if } \tau_1 < \Delta_e^k \leq \tau_2, \\ 0, \textbf{ if } \Delta_e^k > \tau_2. \end{cases}$$

This approach allows keeping track of events that occurred before the current windows but are not recent enough to be fully considered in computations. This approach can be even more precise and generalized in fading event approach.

2.3 Fading Event Service

Many services aim for a quick adaptation but need to keep information for a longer period than just a period τ, or to give some nuances to events through time. Such service can use a fading event strategy. This strategy consists in making events less important with time. Many weighting functions can be used to implement this behavior, as long as it verifies the following properties:

1. Definition: $w_{\mathscr{F}}$ must be defined for any couple of event and age (e, Δ_e^k) (however, the function does not need to be continuous).
2. Evolution of memory: for a given event e, $w_{\mathscr{F}}$ is not a constant function over time.
3. Fading over time: for any event e, $w_{\mathscr{F}}$ is a pseudo-decreasing function over time, that is that $w_{\mathscr{F}}$ is decreasing over time except for a meager number of outliers.

This service is illustrated by Fig. 6. In this example, t_{i+6} and t_{i+7} will be fully considered, t_{i+4}, t_{i+5} slightly less, $t_{i+1}, t_{i+2}, e_{i+3}$ even less and so on.
There are many relevant functions that can be used to build a memory in a large-scale system. A way to represent such a function is the following:

$$\forall e \in \mathbb{E}, \exists\, t_{k_1}, t_{k_2} \in \mathbb{T}^2 \mid w_{\mathscr{F}}(e, \Delta_e^{k_1}) \neq w_{\mathscr{F}}(e, \Delta_e^{k_2}),$$
$$\forall e \in \mathbb{E}, \forall^* (t_{k_1}, t_{k_2}) \in \mathbb{T}^2 \mid t_{k_1} < t_{k_2} \Rightarrow w_{\mathscr{F}}(e, \Delta_e^{k_1}) < w_{\mathscr{F}}(e, \Delta_e^{k_2}),$$

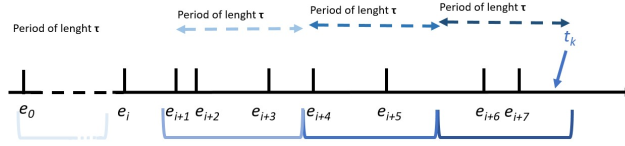

Fig. 6. Weight of event at time t_k with fading event service

where $\forall^*$ denotes "for all but a meager of elements". For the clarity of illustrations and explanations, we use a simple function that divides the weight of events by 2 every τ time units:

$$w_{\mathscr{F}}(e, \Delta_e^k) = \frac{100}{2^\delta},\ \delta = \lfloor \frac{\Delta_e^k}{\tau} \rfloor.$$

This function is a decreasing function over time and thus respects the needed properties while being easily implemented and does not imply a huge computation cost. The philosophy described hereafter still holds with most of the others functions that respects the 3 properties presented.
The impact of this function is illustrated by Fig. 7 where the most recent events have a weight of 100 which is periodically divided by 2. While the service theoretically never truly forget any event, at some point the weight became negligible and the event is almost forgotten. It is possible to provide some variation

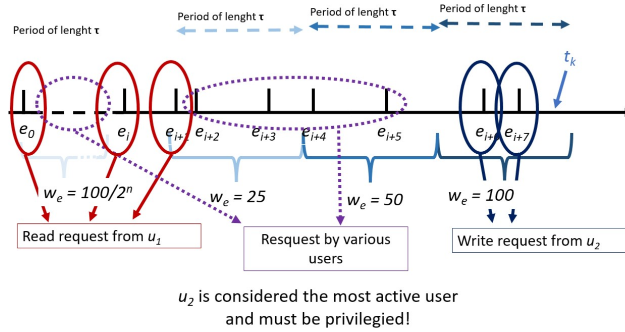

Fig. 7. Weight of event at time t_k with a timeless service

on purely fading events by allowing the function to occasionally increased the weight of an event, if this event became more relevant again for some reason (e.g., recurring events).
As for a sliding window strategy, it is important to tune correctly the value of τ. Preliminary studies suggest that a dynamic strategy that uses both the frequency and the kind of event allows reaching a well dimensioned period without prior knowledge of the system. This service gives more importance to the recent events, but without *forgetting* them. Instead, the weight is decreased over time.

Most of the time, services only need a decreasing function over time to run smoothly. However, it may be interesting to give more importance to old events if a pattern of events reoccurs. In this case, one may use a pseudo-decreasing function that will give a bigger weight to old events under some circumstances. Some mechanisms, such as the double LRUs in Linux, are based on this concept.

3 Experimentations

We propose here an evaluation of the three strategies presented in the previous section through simulation of a service that rely on its memory to take some adaptation decisions. CAnDoR is a distributed data placement service that consider both the consistency protocol and the user workload [4]. CAnDoR computes an efficient position for each replica of data requested by users, focusing on requests location of most acceded data in combination of the guaranties associated with the consistency model. In such a service, the computation directly uses the memory and the weighting functions can have a huge impact on the placement solutions found by the service. We consider here 4 different versions of CAnDoR:

- **The static version of CAnDoR $\mathscr{N}$:** which place the replica according to the guaranties but do not try to adapt to users behavior. Therefore, this service does not use any memory model. This service is used as a baseline to compare the efficiency of the memory strategies.
- **The timeless version of CAnDoR $\mathscr{T}$:** this service uses a timeless strategy to weight the requests made on data.
- **The sliding window version of CAnDoR $\mathscr{S}$:** this service uses a time window of length τ to determine which events must be considered for computation. The value of τ is based on the frequency of requests. The weighting function is such that $|t - t_i| < \tau \Rightarrow w(t,t_i) = 100$.
- **The fading event version of CAnDoR $\mathscr{F}$:** this service weights events according to the number of periods τ elapsed since the event occurred. More specifically, the weighting function described in Sec. 2 is used: $w_{\mathscr{F}}(t,t_i) = \frac{100}{2^\delta},\ \delta = \lfloor \frac{t-t_i}{\tau} \rfloor$.

Experiments presented here are made on CandorSim, a simulator based on peerSim [9]. CandorSim simulates 100 clusters, each of them is considered as a single node due to the fact that the communication inside the cluster is negligible. Users can request access to data to a cluster, while clusters can treat those requests and communicate with each other to periodically compute for better placement. The evaluated metric is the time needed for at least 95% users to get the requested data. The simulator has been calibrated with the result of real-world studies, such as [10], [11]: a message sent by a user needs 100 to 300 ms to reach a cluster while a cluster needs 30 to 150 ms to reach another cluster. As the memory strategy used can lead to different results according to the number and the kind of request, experiments have been made using 3 different behaviors for users.

Stable Behavior: In this experiment, some users send many requests during the whole execution and others only send occasional requests. This behavior is to be expected with application using personal data. Each data is acceded by a small group of users (doing regular requests) that can occasionally change locations (and be considered as new users by the service). Some occasional user may also perform some requests as well. We can observe that the strategy in use does not have a huge impact.

For all three services $\mathscr{T}, \mathscr{S}, \mathscr{F}$, the time needed to answer a request drops from 175 to 130 ms. This is due to the lack of adaptation in the system: the set of active users stays the same during the whole execution, and requests are uniformly made during the execution. Thus, considering whole requests or only the recent ones provide the same view of the system. Results of such execution are shown in Fig. 8.

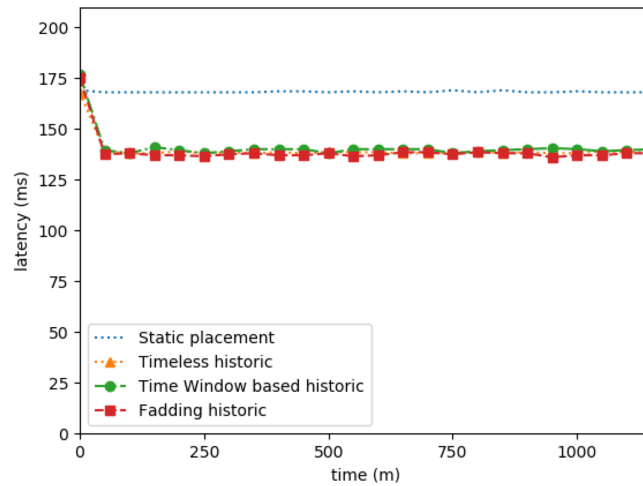

Fig. 8. Time needed to deliver requested data with stable behavior

Alternating Behavior: This time, users are divided into 3 groups. The first two groups send many requests but alternatively: while the first group is active, the second is not and reciprocally. The third group always sends occasional requests to add some noise to the computation. These behaviors represent applications where only a defined group of user access the data but regularly change location, or if two groups are sharing the data but at different times.
We can observe that with such a behavior, the timeless strategy provides worst performance. This can be explained by the impact of the inactive group: with a timeless strategy, the inactive group will still be considered with the same importance as the active one. We can thus see a huge latency peak, this peak corresponds to the change of activity. As the two groups alternate the activity, the second will always be less considered than the first one. The use of a sliding window or fading event strategy allows the service to quickly react to this change and while smaller latency peaks can be observed, the service reacts much quicker which provides a better global quality of service. The results of this scenario are shown in Fig. 9.
Unbalanced Alternating Behavior: This scenario is close to the previous one: two groups work consecutively while a third one is working sporadically to add some noise. However, the first two groups do not work during an equivalent pe-

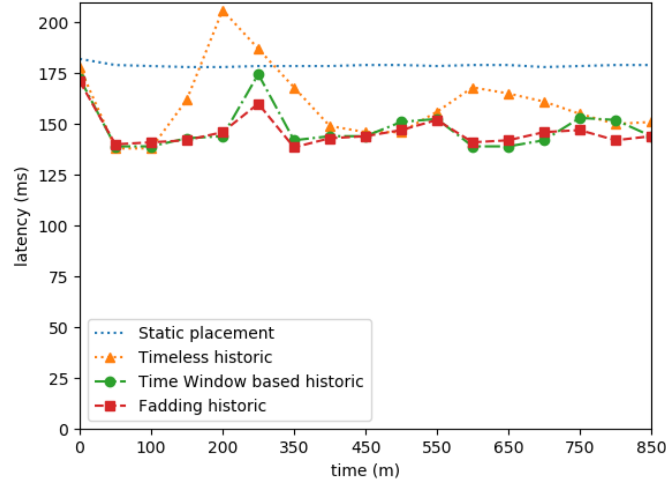

Fig. 9. Time needed to deliver requested data with alternating behavior

riod of time in this scenario: the first group is active for a longer period of time and is only temporarily replaced by the second group. The context of such application is similar to the previous one, but with unbalanced loads between the two active groups.

As a consequence, we can observe periodic latency peaks. Those peaks correspond to the period where the second group is active. As this group is less active, it will not be favored a lot by the service. This tendency is especially true with a timeless service, where the second group will never be favored over the first one, hence the periodical peaks. The use of a sliding window or a fading event strategy allows the service to quickly react to this change. Furthermore, the dynamic tuning of τ even allows the service to calibrate it to reduce the impact when the activity changes. The results of this scenario are shown in Fig. 10.

Those simulations show that the strategy of memory usage has a non-negligible impact on the system performance. In a service such as CAnDoR, that relies on its memory for computation, the use of a timeless strategy can lead to bad performance. In our experiments, we observed that a fading event based strategy and a sliding windows one lead to quick adaptation and thus better global quality of service. The simple version of fading events-based strategy seems to provide slightly better performance than sliding windows.

4 Conclusion and future works

Large scale services often run on top of dynamic systems and have to face continuously changing workloads. In order to provide a good quality of service and a smooth experience for users, such services must be able to adapt to their dynamic

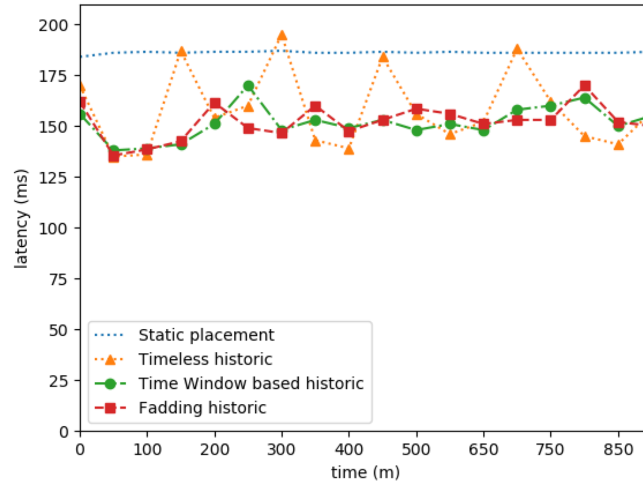

Fig. 10. Time needed to deliver requested data with unbalanced alternating behavior

environment. To do so, they usually have to monitor their environment, store past events in their memory, and take decisions to gracefully adapt.

In this paper, we presented a formal representation of the memory of individual nodes of a dynamic system and how they can manage their own memory (set of stored past events). We also propose a theoretical analysis of three strategies to use this memory when a node needs to make an adaptation decision. We used our model and weighting functions to influence the impact of some past events during the computation. More specifically, we proposed a set of restrictions to build *fading events* based memory: a memory that progressively diminish the impact of events with time. We then conduct experiments to evaluate the impact on the quality of service of a distributed data placement service according to the strategy in use. Those evaluations show that it is important to use a strategy that provides the ability to forget, completely to partially, past events. Fading event based strategy seems to provide better results, but not significantly compared to a sliding window strategy.

The strategies presented in this paper mostly rely on the time elapsed since an event occurs. We believe that finer (but more complex) strategies could be used by injecting other considerations, such as taking into account importance or rarity of events. Future works will address such strategies.

References

1. R. Ingram, P. Shields, J. E. Walter, and J. L. Welch, “An asynchronous leader election algorithm for dynamic networks,” in 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12, IEEE, 2009.

2. F. Greve, L. Arantes, and P. Sens, "What model and what conditions to implement unreliable failure detectors in dynamic networks?," in Proceedings of the 3rd International Workshop on Theoretical Aspects of Dynamic Distributed Systems, pp. 13–17, 2011.
3. A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision.," Decis. Support Syst., vol. 43, no. 2, pp. 618–644, 2007.
4. E. Mauffret, F. Vernier, and S. Monnet, "Candor: Consistency aware dynamic data replication," in 2019 IEEE 18th International Symposium on Network Computing and Applications (NCA), pp. 1–5, IEEE, 2019.
5. M. Iqbal, W. N. Browne, and M. Zhang, "Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems," IEEE Transactions on Evolutionary Computation, vol. 18, no. 4, pp. 465–480, 2013.
6. T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," IEEE Signal Processing Magazine, vol. 37, no. 3, pp. 50–60, 2020.
7. T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," Journal of the ACM (JACM), vol. 43, no. 2, pp. 225–267, 1996.
8. C.-H. Lee, C.-R. Lin, and M.-S. Chen, "Sliding-window filtering: an efficient algorithm for incremental mining," in Proceedings of the tenth international conference on Information and knowledge management, pp. 263–270, 2001.
9. A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in 2009 IEEE Ninth International Conference on Peer-to-Peer Computing, pp. 99–100, IEEE, 2009.
10. S. Agarwal, "Public cloud inter-region network latency as heat-maps," 2018.
11. D. A. Popescu, Latency-driven performance in data centres. PhD thesis, University of Cambridge, 2019.