# How to use the past to face the future?

Etienne Mauffret, Flavien Vernier, Sébastien Monnet

# How to use the past to face the future?

Etienne Mauffret
*LIP*
*Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342*
Lyon, France
Etienne.Mauffret@ens-lyon.fr

Flavien Vernier
*LISTIC*
*Savoie Mont Blanc University*
Annecy, France
Flavien.Vernier@univ-smb.fr

Sébastien Monnet
*LISTIC*
*Savoie Mont Blanc University*
Annecy, France
Sebastien.Monnet@univ-smb.fr

***Abstract*—Large-scale distributed systems are highly dynamic: nodes may crash, messages may be delayed or lost, new nodes may join, virtual machines may migrate... The workload may also vary a lot depending on the users or applications behavior. Thus, distributed systems and services must adapt continuously in order to remain efficient. To do so, distributed systems usually monitor their environment and maintain a knowledge of what has happened (e.g., heartbeats time-stamps for a failure detection mechanism, data access statistics for a distributed storage systems, etc.). Based on these observations, a distributed system can decide to adapt to better tackle future situations (migrate a virtual machine, create a new data replica, etc.). However, even if a system has a full event log, it is not a trivial task to decide how much and which part of the past must be taken into account. In this paper we propose and study the impact of multiple approaches, from "full-memory" to "no-memory" through "time-window" and "fading-memory" based strategies.**

***Index Terms*—Adaptative systems; distributed systems; monitoring**

## I. Introduction

With the advent of cloud platforms, more and more users rely on "on-line services". The arrival of 5G networks will certainly even increase the use of remote resources. Many applications, like social networks, on-line stores or on-line office suite, operate worldwide. Such services rely on a large pool of resources and handle requests emit by a huge number of geo-distributed users.

At such a large scale, and with so many users, the distributed on-line services have to deal with a heterogeneous and continuously changing environment. The underlying infrastructure is highly heterogeneous and dynamic: nodes may crash, network latencies can change, messages may get lost, new nodes may join, nodes performance may also change, etc. The workload is also heterogeneous and highly dynamic. There are night and days variations, weekly variations, some events/videos/publications may become popular suddenly, etc.

Therefore, while designing a distributed service, it is necessary to handle this highly dynamic environment. Fortunately, the cloud's economic "on-demand" model and recent technologies such as virtual machines, containers and orchestrators, give the distributed service designer access to virtually an infinite "on-demand" pool of resources. It is thus now possible to react and adapt if needed: by allocating more resources (physical nodes, virtual machines, bandwidth, storage space, ...), by migrating to other locations, by moving pieces of data... The possibilities offered by recent cloud platforms are numerous and affordable. Using new technologies, such as microservices, the distributed service adaptation can even be finely tuned. Nowadays, many distributed systems have been designed to adapt themselves to the environment changes [1], [2], [3], [4]...

However, to keep a distributed service well adapted to its continuously changing environment, it is necessary to build and maintain a knowledge about this environment. This concerns both the underlying infrastructure (through failure detection, network monitoring, server load monitoring, etc.) and the workload (e.g., by logging users' requests). This knowledge can then be used to forecast the future environment and slightly adapt the service to fit the forecasted future conditions.

As the knowledge is usually distributed among the nodes involved in the service (each node is usually responsible to log the events concerning itself and/or concerning resources it is responsible for) and as the adaptation has got to be performed quite frequently (to keep fitting the continuously changing environment) machine learning techniques may be hard to use there.

In this paper we formalize the representation of the acquired knowledge and study how it can be taken into account to build heuristics allowing a distributed system to adapt itself and to continue to offer good performance. We propose several approaches that can be observed in existing distributed systems and study their behavior under various scenarios. The goal is to study the weight one should give to past events according to: (i) their age (intuitively, recent events should weight more than old ones); (ii) their frequency (intuitively, frequent events should weight more than rare ones); or (iii) the associated user's priority (the system should give more weigh to events concerning users/pieces of data/...having a high priority).

The contribution of this paper are:

- A formal mean to represent the distributed event log, and a formal representation of various classical approaches: "full-memory", "no-memory", "time-window based memory", "fading-memory" and "frequency-aware fading-memory";
- A discussion about these approaches;
- An analyze of the behavior of the different approaches under various realistic senarii, using a previous work: a

ANR RainbowFS project (https://rainbowfs.lip6.fr/).

distributed data management system that needs to react to changing users behavior.

The following of this paper is organized as follows. Section II describes how we handle past events and presents the proposed approaches. Section III presents how the system can decide to adapt or not according to the strategy in use and the estimated benefits. Section IV give our evaluation results. Finally, Section V concludes the paper and gives some perspectives.

## II. Building and using event log

### A. Memory representation

Let $\mathcal{S}$ be a dynamic distributed service operating at large scale. This service runs for a given time $\mathcal{T} = [0; t_\infty]$. Each system node being part of the service runs a local instance. In the following we consider a node's point of view thus local executions of $\mathcal{S}$.

Each local instance of $\mathcal{S}$ must build a local event-log. This history contains both the events known by the node and their respective weight (note that this weight can be implicit, for instance, it can be deduced from an event timestamp).

Usually, an event $e$ is defined by a tuple composed of $a_e$, the source of the action $s_e$ and the timestamps of the action $t_e$. Depending on the service needs, more information can be added. For instance, a destination node $d$ for the action can be logged when events are messages. In the following, we consider events represented by the triplet presented here.

To set the weight of each event in the history $\mathcal{S}$ uses a weighting function $p_\mathcal{S}$. This function returns a value between 0 and 1 depending on the age of the event. This weight permits to define the strategy in use by $\mathcal{S}$ for its adaptation.

We mainly focus on 3 strategies. To illustrate lets consider 3 distributed services which differ only by their history usage strategy (i.e., the only difference is their weighting function:

1) $\mathcal{S}_{IT}$ : this service considers that all events have the same weight (whether they are recent or very old);
2) $\mathcal{S}_{FT}$ : this service uses a "time window", it considers only recent events (where "recent" depends on the window size);
3) $\mathcal{S}_{EE}$ : this service uses a fading memory model, the weight of an event decreases with time.

When an event $e$ occurs at time $t_e$ on a node, it is logged in the node's event log. While using the learned knowledge, at a time $t \in \mathcal{T}$, the importance of the event is weighted using $p_\mathcal{S}(\Delta t_e)$, whith $\Delta t_e = t - t_e$ being the elapsed time since $e$ occurred.

We compare these approaches under various simple scenarios. We consider that sources (nodes at the origin of events) may have two types of behavior: active or passive on a given period (i.e., generate events on this period or not). Depending on the distributed system, it can be useful to distinguish which resource(s) is impacted by events generated by an active node (i.e., to define that a node is active/passive at the granularity of a resource). For instance, a user can send requests for a piece of data $d_1$ but never for another piece of data $d_2$. In this case, it is stated that this user is active for $d_1$ and passive for $d_2$. This is used to describe the following scenarios:

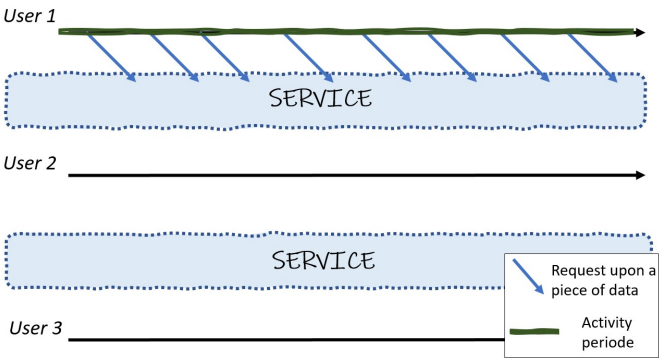



Fig. 1. Scenario 1

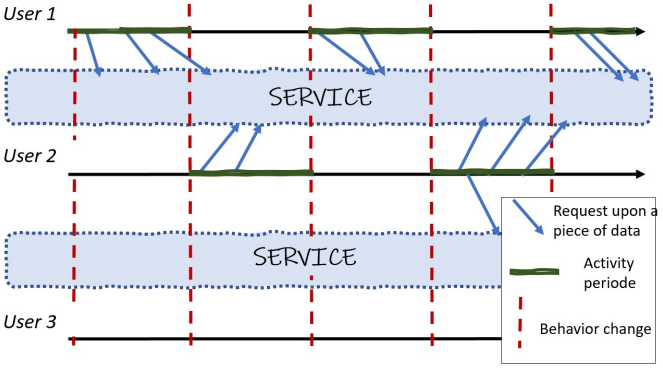



Fig. 2. Scenario 2

1) **Stable behavior**: sources (or user nodes) behave constantly during the whole execution: a source remains either active or inactive. This scenario models the use of a distributed service by static users (having a constant behavior). It is depicted by Figure 1 where user 1 is the only active source;
2) **Alternative behavior**: two sets of sources change their behavior regularly. Alternatively, the active group becomes inactive (for a given target) while the other becomes active. Several sources may remain inactive during the whole execution. This scenario is depicted by Figure 2 in which the users 1 and 2 switch from active to inactive periods alternatively (user 3 remains inactive);
3) **Unbalanced alternative behavior**: tow sets of sources switch from active to inactive alternatively independently. This scenario models, for instance, the activity of a group of users using a same service from two different locations, like from home and from work. It is depicted by Figure 3 in which users 1 and 2 alternate between active and passive periods while 3 remains inactive;
4) **Erratic behavior**: sources exhibit no particular habit, they behavior (active or inactive) can change at any time. This is depicted by Figure 4.

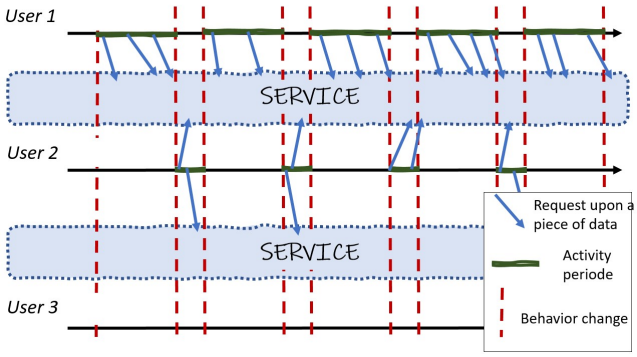



Fig. 3. Scenario 3

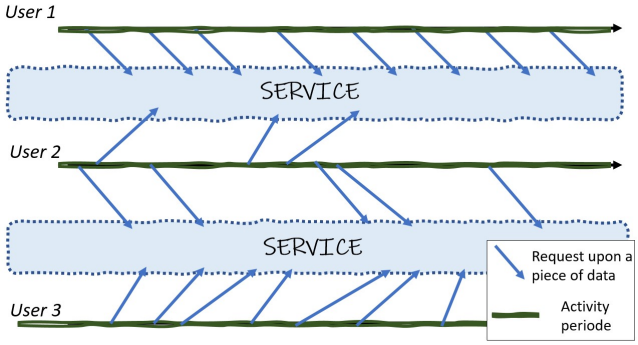



Fig. 4. Scenario 4

Figure 5 illustrates the events occurring along time, this figure will be used to explain the weight given to events between $e_i$ and $e_{i+7}$ depending on the strategy in use.

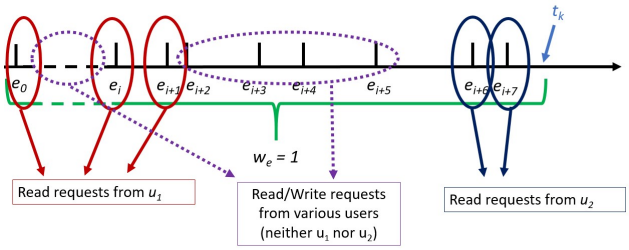
### B. Full memory strategy

*1) Concept:* Using this strategy, all the events have the same weight. Therefore, the age of an event is not taken into account. In other words, the weighting function is a constant: $\forall t_1, t_2 \in \mathcal{T}, p_{\mathcal{S}_{IT}}(t_1 - t_2) = 1$. This behavior is illustrated by Figure 6.

This simple strategy has a major drawback. As no particular weights are given, recent events have the same impact as very old event that occurred at the beginning of the execution. In case of configuration change, the adaptation will be very slow: the system will start to adapt only when the number of events that occurred in the new configuration is big enough to face the number of events in the old configuration.

*2) Points in favor:* If a system is rather stable, this strategy may be used. It can also be used in presence of erratic behavior: in this case, the system will converge to a mean configuration and will not try to adapt to better handle erratic behaviors, exceptions.

This is illustrated by Figure 7.




$u_1$ is considered as the most active user and computation should be avantaged by the service.

Fig. 7. Full memory strategy

*3) Full memory strategy limits:* In case of rapid changes, in the application behavior or in the infrastructure, the distributed service will take a very long time to adapt. In several scenarios the service should alternate between two (or more) configurations to better fit the current needs; using the full memory strategy would certainly lead to an average configuration offering poor performance for the various cases.

Another drawback is that such a strategy forces the nodes to keep a trace of all events since the beginning of the exécution. This may be costly in terms of storage. This naive strategy is thus not very adapted to a large-scale dynamic distributed service.

### C. Time-window based memory strategy

*1) Concept:* In order to adapt rapidly to changes, it is important to quickly take into account *recent* events. A common way to give a high importance to recent events is to forget old ones. The time-window based memory strategy consist of taking into account only recent events: the ones that fit in a time-window. The time window is a time frame of size $\tau$. Events that are older than $\tau$ are not taken into account anymore. $\tau$ can be either fixed or dynamic. Therefore, using this strategy, the weighting function $p_{\mathcal{S}_{FT}}$ returns 1 for recent events and 0 for the others. More formally:

$$\forall t_1, t_2 \in \mathcal{T}, p_{\mathcal{S}_{FT}}(t_1 - t_2) = \begin{cases} 1, & \textbf{if } (t_1 - t_2) \leq \tau \\ 0, & \textbf{if } (t_1 - t_2) > \tau \end{cases}$$

*2) Adaptability:* This strategy favors the present and remove all the impact of old events. The distributed service using this strategy can thus adapt quickly. For instance, if sources have recently changed their behavior, in the time window, most event will reflect this new behavior and the system will adapt to better face this new situation if it can. This strategy is illustrated by Figure 8.

However, this high adaptability presents some drawbacks: the distributed system using this strategy may be too sensitive to short term changes. Using such a strategy, it is possible

Fig. 5. Events during time $t_k$

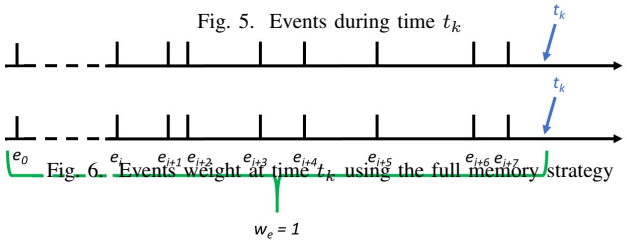



Fig. 6. Events weight at time $t_k$ using the full memory strategy

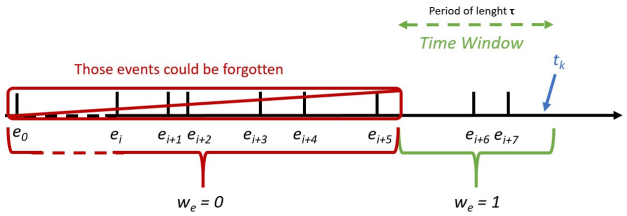



Fig. 8. Events weights at time $t_k$ using a time-window based strategy

that the system adapts too often. For instance, some important event sources may become inactive for a short period of time and the system may decide to reconfigure itself not taking them into account, while these sources will be active again soon as illustrated by Figure 9.

To tune the tradeoff between fast adaptation and stability, it is necessary to configure the time-window size.

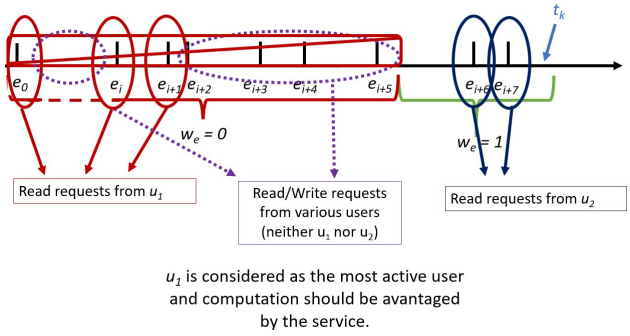



Fig. 9. Behavior while using a time-window based strategy

*3) Time-window size importance:* As stated above, the time-window size can be either fixed or dynamic. A too short time-window may lead to ignore important events and to a poor stability (i.e., a system that adapt too fast to short terms changes). A too long time-window will degrade the adaptation speed. At the extreme case, a very long time-window is equivalent to the full memory strategy. Fixing a time-window size requires a good knowledge *a priori*. In the Section III-B we study dynamic means to adapt the time-window size while the service is running. Having to configure this time-window size is the main limit of this strategy.

*4) Time-window advantages:* If the whole event-log is not needed for some other reasons (e.g., security, post-mortem analysis…), the system may only keep recent events: the ones that fit in the time-window. This provides the ability to periodically purge the event log and save a lot of memory.

Thus, if the time-window size is well configure, this strategy can give good results (fast adaptation, but not too fast), at a reasonable cost.

*5) Double time-window:* To lover the impact of the time-window size, and to permit the distributed system to adapt more gracefully, it is possible to use two time-windows. This is illustrated by Figure 10. This approach permits to forget events more gradually. Event leaving the first time-window will have their weight reduced, then after a second period of time (the second time-window, their weight becomes 0. Formally, for two periods $\tau_1, \tau_2$ such that $\tau_1 < \tau_2$ :

$$\forall t_1, t_2 \in \mathcal{T}, p_{\mathcal{S}_{FT}}(t_1 - t_2) = \begin{cases} 1, & \textbf{if } (t_1 - t_2) \leq \tau_1 \\ \frac{1}{2} & \textbf{if } \tau_1 < (t_1 - t_2) \leq \tau_2 \\ 0, & \textbf{if } (t_1 - t_2) > \tau_2 \end{cases}$$

This approach presents the same pros and cons than the single time-window approach: the size of the time-windows may be hard to set, the system will be too reactive if they are too small and not reactive enough if they are too long.

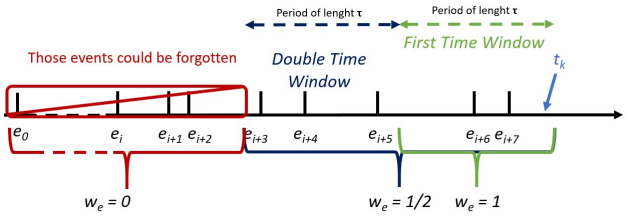



Fig. 10. Weight of events at time $t_k$ using a double time-window based strategy

### D. Fading memory strategy

*1) Concept:* Many distributed services have to adapt fast, but also need to take into account old past events. In particular to detect specific usage habits. The fading memory model is thus a good choice. This strategy is based on the use of a weighting function $p_{\mathcal{S}_{EE}}$ that gives a value decreasing with time. Many functions can be used, we have identified 3 main criteria to obtain a progressive vanishing of the history.

1) $p_{\mathcal{S}_{EE}}$ must be a piecewise-continuous function for all the execution time. $\forall t_1, t_2 \in \mathcal{T}, p_{\mathcal{S}_{EE}}(t_1 - t_2)$ is defined. It is not necessary that $p_{\mathcal{S}_{EE}}$ is a continuous function. That means a weight can be given for all the events.
2) $p_{\mathcal{S}_{EE}}$ must be decreasing. $\forall t_1, t_2, t_3 \in \mathcal{T}, t_2 \leq t_3 \Rightarrow p_{\mathcal{S}_{EE}}(t_1 - t_2) \leq p_{\mathcal{S}_{EE}}(t_1 - t_3)$. That means more an even is old, less is weight is important.
3) $p_{\mathcal{S}_{EE}}$ should not be constant. $\forall t_1, t_2 \in \mathcal{T}, \exists t_3 \in \mathcal{T} \mid p_{\mathcal{S}_{EE}}(t_1 - t_2) \neq p_{\mathcal{S}_{EE}}(t_1 - t_3)$. It can be constant *locally* it is the case for piecewise constant decreasing functions.

In many cases an event should never be completely forgotten. However, a very old single event should have a negligible weight (but not null). We thus add a fourth criterion:

4) $p_{\mathcal{S}_{EE}}$ should not have root.

The use of a continuous function permits to give a weight to each instant, but mas need many computation. In practice the use of a piecewise-continuous function is enough, like piecewise constant decreasing functions. This kind of function usually use a time period $\tau$ to define the timeframe during which the function is constant. This is very close to multi-time-window strategies, it can be seen as a generalization. We focus on the following function: $p_{\mathcal{S}_{EE}}(\delta t) = \frac{1}{2}^{\lfloor \frac{\delta t}{\tau} \rfloor}$, illustrated by Figure 11.

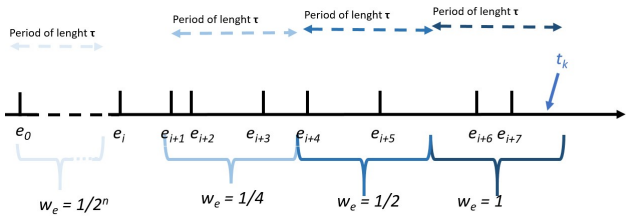



Fig. 11. Weight of events at time $t_k$ using a fading memory strategy

This function simply periodically divides by 2 the weight of past events. This low-cost operation provides the ability to decrease rapidly the weight of old events while still taking them into account, as shown by Figure 12. This strategy is used by the Berthier's failure detector to adapt to the changing latency [5].

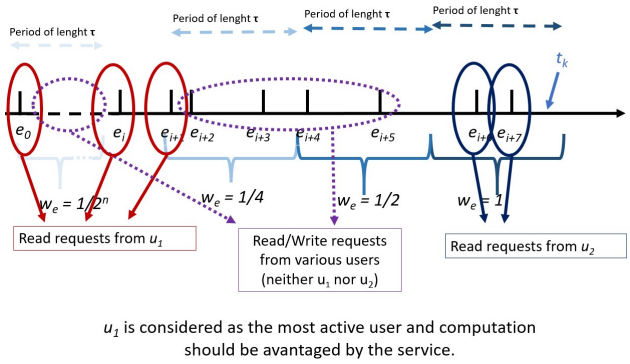



Fig. 12. Behavior while using a fading memory strategy

*2) History size:* Using the fading memory strategy usually implies to keep a trace of all the past events. This can induce a non negligeable cost. However, if the storage capacity of the nodes is limited, it is simple to not respect the last constraint 4. For instance, in the case of the function $p_{\mathcal{S}_{EE}}(\delta t) = \frac{1}{2}^{\lfloor \frac{\delta t}{\tau} \rfloor}$, it is possible to fix a maximum number of periods after which an event is forgotten.

Strategies based on fading memories can use well adapted weighting functions, at the cost of some computation.

*3) Taking frequency into account:* At last, the 2 constraint can be relaxed. This constraint imposes the use of a decreasing function. In the case where the service faces an event similar to an old one (e.g., access by a same user a same piece of data) it can increase the weight of the past event. This allows to take frequency into account.

## III. Service adaptation

### A. re-configuration

In order to decide if a distributed service should change its configuration, it needs to estimate the benefit of the change. This change is done through an adaptation $\alpha$. The service should compute the estimated cost of $\alpha$, $c(\alpha)$ and of the benefit $g(\alpha)$ ($g$ for gain).

The cost estimation is service dependent. In the case of a failure detector, it may consist in updating a local timer while in the case of a distributed data storage system, it may lead to data migration. Thus, we suppose that the service is able to estimate c($\alpha$).

The benefit $g(\alpha)$ is obtained by computing the difference between the current configuration $\sigma_{cur}$ and the configuration after the adaptation $\sigma_\alpha$. This value is estimated through a performance evaluation function $\mu$ which gives a value depending on an event $e$ and a configuration $\sigma$. This measure depends on the service. It can represent a detector accuracy, a data access latency, ...To estimate future performance, the service may rely on events present in the event log. We propose the following formula to compute the gain.

$$g(\alpha) = \sum_{e \in \mathcal{H}_S} (p_S(t - t_e)(\mu(\sigma_\alpha, e) - \mu(\sigma_{cur}, e)).$$

This approach consists in computing the performance difference for each event present in the log, weighted by its associated cost (depending on the strategy in use).

One the cost and the benefits have been estimated, the service can easily decide if it should adapt or not. in particular, if $g(\alpha) - c(\alpha) > 0$, then the service should estimate that it should adapt.

This example of method to adapt a distributed service highlight the importance to the weight given to the events in the event log. The service uses these weights to forecast the future.

### B. Estimating time-window size or fading period

Many strategies exploiting the knowledge in the event log are based on a period $\tau$. This period can be fixed or adaptable. A fixed period (time-window size/fading function) requires an a priori knowledge. This knowledge can be hard to acquire. It may be necessary to design an algorithm to adapt the period length itself. There again, the algorithm depends on the service.

*1) Case study: period estimation using CAnDoR:* To study the impact of the period size we have taken CAnDoR [6], a distributed data management service. CAnDoR has to adapt data replica placement in order to reduce the overall access latency, it takes into account users' location, other replicas

location and the consistency protocol in use. CAnDoR mechanisms are detailed in [6] and are outside the scope of this paper.

The event log is composed of read and write requests on pieces of data from users. Adaptations consist in moving pieces of data. The performance is simplified: we consider the median access time for all users.

In this evaluation users follow the "alternative" behavior detailed in Section II-A: at the beginning of the execution 2 groups of users are chosen uniformly at random. these 2 groups alternate their activity.

Figure13 compares the median access time of active users between a baseline (static placement, smart placement but without further adaptation) a dynamic placement using the fading memory strategy (left) and a dynamic placement using a time-window based strategy. The fixed period is set to $500$ seconds ($8,3$ minutes, purple solid line) then $1000$ seconds ($166$ minutes, green dashed line).

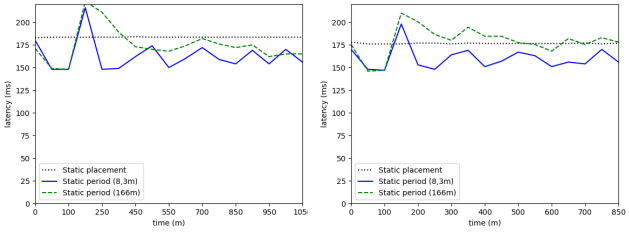



Fig. 13. Performance using fixed period. Fading memory (left) and time-window (right)

Time-window and fading memory present very similar behavior (when we compare executions using the same period value). While using fading memory and a long period (166 minutes), the period is to long and the service is close to static. .However, with a $8,3$ minutes period, the adaptation is fast and access times are improved.

*2) Over or under-adapting:* This first method to adapt the period length is based on the monitoring of the service adaptations. The service counts the number of re-configurations. It also counts the number of computations it did to decide whether or not it should adapt (i.e., reconfigure). If the ratio $nbreconfiguration/nbcomputation$ is close to $1$, the service over-adapts (each time it checks if it should adapt, it should according to its history and strategy). On the over hand, if this ratio is close to $0$ the service under-adapts. It almost never adapts.

This approach is simple to implement, but it is not accurate. How to make the difference between a well adapted service that remains well-adapted and a under-adapted service? However, it can be used asidealong with another method.

Figure 14 presents the performance while using this method (purple) compared to a fixed period. We can observe a little improvement.

*3)* shadow pages cache-*based method:* The second method the adapt the period length is inspired by a Linux Kernel

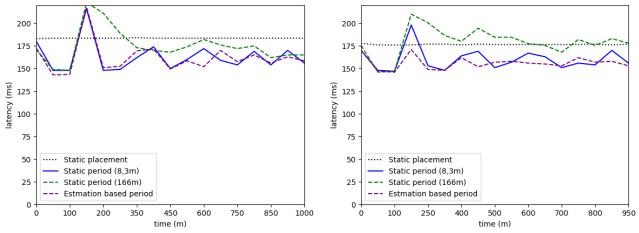



Fig. 14. Adapting the period length according to the number of re-configuration. Fading memory (left), time-window (right)

mechanism: the *shadow page cache* ( [7], [8]). The kernel keeps in memory reference of pages that were in the cache recently (a kind of list of recently evinced pages). If an accessed page is not in the cache but is in this list (the shadow page cache), the kernel can put it directly in the active pages (because it is almost like if it was a hit in the cache). In fact, the page would have been in the cache if it was slightly bigger.

In our system, we keep a view on the previous configuration, then at each request (or from time to time) we compute if the old configuration could have done better than the current one. This can help to detect too short periods. However, the cost induced by this method is not negligible.

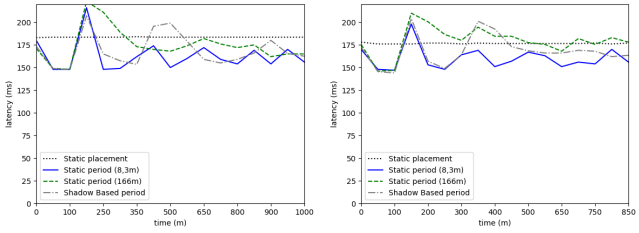



Fig. 15. Adapting the period length comparing with previous state. Fading memory (left) and time-window (right)

Figure 15 presents the performance obtained using this approach compared to static periods. There again, the gain is negligible. Some results are even worse than the ones obtain with static periods.

*4) Using the number of requests:* The third method is based on the number of received requests. First, we determine the number $k$ of events necessary for the service to perform a correct placement. At the end a period, the service count how many events occurred during the period : $n$ (number of events having a weight of 1). If $n$ is not "close" to $k$ then the service increase or decrease the period in order to allow future $n$ to get closer to $k$.

This approach allows the service to adapt the period depending on the activity.

Figure 16 shows the performance with this last approach (red) compared to the static one. The performance is there again slightly better but comparable.

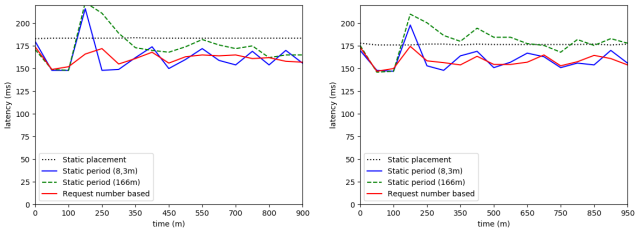



Fig. 16. Adapting the period length using the number of requests. Fading memory (left) and time-window (right)

## IV. Evaluating weighting strategies

The goal of this section is to study the impact of the proposed strategies to weight the past on the performance of our distributed data management service CAnDoR. We use 3 configurations of CAnDoR: $\mathcal{C}_{IT}$ a version of CAnDoR with full memory (i.e. all the events weight 1), $\mathcal{C}_{FT}$, using a time-window based strategy and $\mathcal{C}_{EE}$, based on the fading memory strategy. The weighting function used by $\mathcal{C}_{EE}$ is the function $p_{\mathcal{S}_{EE}}(\delta t) = \frac{1}{2}^{\lfloor \frac{\delta t}{\tau} \rfloor}$. $\mathcal{C}_{FT}$ and $\mathcal{C}_{EE}$ handle a dynamic period (adapted through the method base don the number of requests) as detailed in previous section.

The 3 configurations are evaluated with sources (i.e., users) following the scenarios described in Section II-A.

We use our CAnDoR simulator based on Peersim [9] to simulate 100 clusters able to store data copies; and groups of 10 users among 100 send requests to access a piece of data. We measure the median access time for these users, for each case. We use $\mathcal{P}$, a static data placement method taking consistency into account as a baseline (a smart placement at the start, but no adaptation).

*1) Stable behavior:* In this scenario, active users are designed at the beginning and remain active all the time. The placement service mainly focus on the ratio "requests sent by a given user"/"total number of requests" As the behavior is quite constant, this ratio does not change a lot and the use of various strategies has a very low impact here. We can observe on Figure 17 that the 3 strategies produce a very similar result. We can also see that the adaptation is efficient and brings a good benefit (from 175 to 130 $ms$) compared to the static data placement (that does not know which nodes will be active or not). $\mathcal{C}_{FT}, \mathcal{C}_{EE}$ induce small data movements when a user becomes punctually more active.

*2) Alternative behavior:* Under this scenario, sources are divided in 3 groups:

1) group A: active one period out of two, 10 users;
2) group B: active one période out of two, when group A is not active, 10 users;
3) group C: never active, the remaining 80 users.

The execution is divided in 250 minutes periods of activity/inactivity. Figure 18 presents the results obtained with this scenario. We can observe that after each behavior switch, $\mathcal{C}_{FT}, \mathcal{C}_{EE}$ have a quick latency peak (around 160 milliseconds) but the service adapts itself quickly to offer good performance

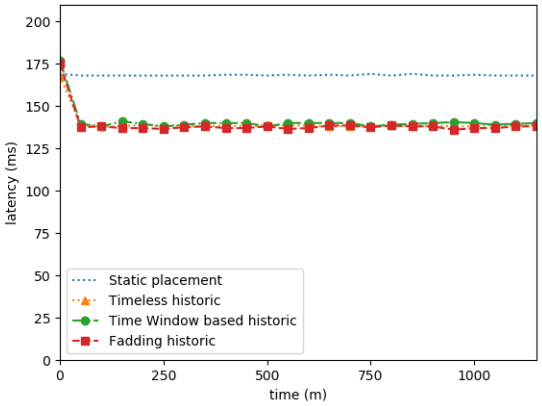



Fig. 17. Comparison of the weighting strategies under scenario 1

again. However, $\mathcal{C}_{IT}$ needs much more time to adapt and find efficient data placement (more than 100 minutes for the first switch). We can see that the latency peak decreases from switch to switch, but also becomes more and more long to lower.. This is because this configuration of the service gives the same importance to all the events, old or recent. After a long time, user switching will not affect the data placement, $\mathcal{C}_{IT}$ will then propose an average placement to the two groups without considering which one is active.

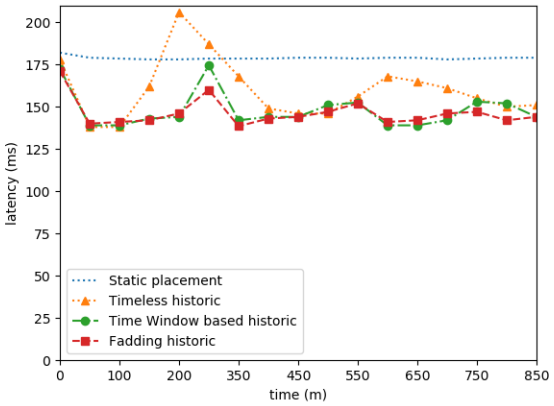



Fig. 18. Comparison of the weighting strategies under scenario 2

*3) Unbalanced alternative behavior:* In this scenario sources are divided in 3 groups:

1) group A: active one period out of two, 10 users;
2) group B: active one période out of two, 10 users;
3) group C: never active, the remaining 80 users.

The execution is divided in 250 minutes periods of activity/inactivity for group A and 50 minutes pour group B. The results of this experiment are presented in Figure 19. We can observe that $\mathcal{C}_{IT}$ exhibit good performance while group A is is active but poor performance while group B is active. Giving the same weight to all the request, $\mathcal{C}_{IT}$ gives a great advantage

to group A which is globally more active. $\mathcal{C}_{FT}$ and $\mathcal{C}_{EE}$ have good performance, however, at each switch, they need some time to adapt.

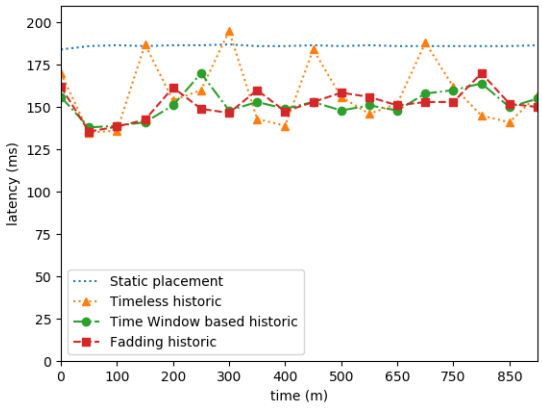



Fig. 19. Comparison of the weighting strategies under scenario 3

*4) Erratic behavior:* In this scenario, each user can send request (or not) periodically (or not). The results of this experiment is show by Figure 20. The figure shows that the dynamic methods cannot do better than the static one.

$\mathcal{C}_{FT}$ and $\mathcal{C}_{EE}$ try to adapt to event that will not reproduce, leading to poor performance. $\mathcal{C}_{IT}$ converges toward a placement equivalent for all users, close to a static one

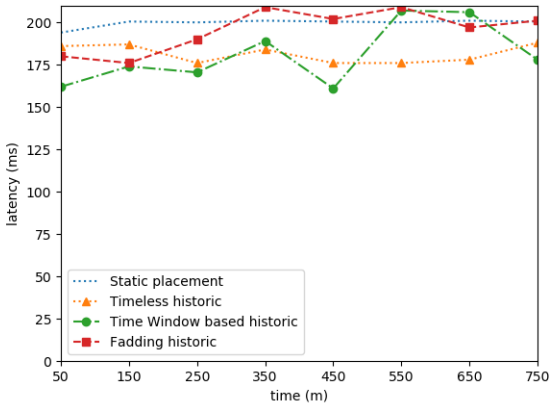



Fig. 20. Comparison of the weighting strategies under scenario 4

## V. Discussions & conclusions

Distributed systems need to gracefully adapt to their continuously changing environment. To do so, they need to monitor, detect changes and forecast the future. There are many means to use information gathered in the past by the monitoring system. In this paper we propose a simple formal representation of the history and we study different strategies. We propose the use of a weighting function over the past events in order to be able to give more or less importance depending on the age of the events.

We saw that it is obviously a good idea to give more importance to recent events. However, forgetting too quickly past events may lead to unstable systems changing their configuration too frequently. On the over hand, giving to much weight to old events may be harmful for the reactiveness. It is therefore necessary to finely tune the weighting function. In this paper we also suggest directions to automatically adapt the weighting function (i.e, adapt the adaptation mechanism itself).

We used CAnDoR, our distributed data management system designed to dynamically adapt the data layout according to the users' behavior, to experiment the various weighting functions and adaptation strategies.

We are currently working on the validation of our approaches with various scenarios. In a near future, we plan to compare the proposed approaches against machine learning techniques.

## References


[1] D. Yuan, Y. Yang, X. Liu, and J. Chen, "A data placement strategy in scientific cloud workflows," *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1200 – 1214, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X10000208

[2] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, 2001, pp. 170–179.

[3] R. O. Saber and R. M. Murray, "Consensus protocols for networks of dynamic agents," 2003.

[4] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," 2010.

[5] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 354–363.

[6] E. Mauffret, F. Vernier, and S. Monnet, "Candor: Consistency aware dynamic data replication," pp. 1–5, 2019.

[7] Johannes Weiner, "thrash detection-based file cache sizing," https://lwn.net/Articles/552327/, accessed: June 2020.

[8] Jonathan Corbet, "Better active/inactive list balancing," https://lwn.net/Articles/495543/, accessed: June 2020.

[9] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE, 2009, pp. 99–100.

[10] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE, 2002, pp. 404–409.

[11] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The/spl phi/accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE, 2004, pp. 66–78.

[12] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–34, 2016.

[13] S. Agarwal, "Public cloud inter-region network latency as heat-maps," 2018. [Online]. Available: https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[15] D. A. Popescu, "Latency-driven performance in data centres," Ph.D. dissertation, University of Cambridge, 2019.