Practical Uniform Peer Sampling under Churn

TECHNICAL REPORT 1/10 - MIDLAB 2010

Roberto Baldoni, Marco Platania, Leonardo Querzoni, Sirio Scipioni
Dipartimento di Informatica e Sistemistica "A. Ruberti"
Sapienza - Università di Roma
via Ariosto 25, 00185, Rome, Italy

***Abstract*—Providing independent uniform samples from a system population poses considerable problems in highly dynamic settings, like P2P systems, where the number of participants and their unpredictable behavior (e.g., churn, crashes etc.) may introduce relevant bias. Current implementations of the Peer Sampling Service are designed to provide uniform samples only in static settings and do not consider that biased samples can directly affect the correctness of algorithms relying on a uniformity property or be exploited by a malicious adversary to increase the effectiveness of its attacks to the system. In this paper we provide a practical solution to the biasing problem by deploying a fully distributed Peer Sampling Correction Module on top of a given, possibly biased, peer sampling service. Samples provided by the peer sampling service will be locally processed by this module, using computationally efficient hashing functions, before getting to the application. The effectiveness of our approach is evaluated through an extensive simulation-based study. Finally, we show the efficiency of the Peer Sampling Correction Module in a case study, namely the target selection attack prevention.**

***Keywords*-Large Scale Distributed systems, Peer Sampling, Bias Correction**

I. Introduction

Random Sampling is a common statistical practice concerned with the selection of individual observations intended to yield some knowledge about a population. Random sampling is a fundamental building block in the design of many different large scale distributed systems: selection of communication partners in gossip-based protocols [1], [2], clock synchronization [3], [4], slicing/ordering [5], data caching in unstructured peer-to-peer networks [6], distributed size estimation [7], overlay network management [8], [9], publish/subscribe [10], etc. This block is usually implemented by means of a peer sampling service (PSS) that is able to provide to running algorithms a continuous stream of samples. The correct behavior of the aforementioned algorithms is based on the ability of the PSS to provide a stream of independent uniform samples of the system membership despite churn, peer crashes and other events that can perturbate the system.

Guaranteeing uniform samples in a large scale distributed system is a difficult task: due to the sheer scale of such systems, maintaining on each node a complete view of the system membership from which random samples can be selected is unfeasible. Therefore, each node pertaining to the system can usually rely only on a limited membership view (*local view*) that can be updated at runtime [11]. The limited size of this view is potentially a source of bias: if a node is more represented than others in local views it could have a higher probability of being returned as a sample. This difficult task becomes a real challenge in a dynamic setting in which the system membership can change at any time. Computing a uniform random sample takes time and the decision taken at some moment is always based on a system membership that could be outdated; reducing the sampling computation time is not a solution if it is obtained at the expenses of a loss of sampling accuracy.

The problem of providing uniform samples in systems with small views has been studied in distributed hash table (DHT) [12] and in unstructured overlay networks maintained by gossip protocols [13]. Both results prove that uniform sampling can be achieved during periods of quiescence (i.e., when churn ceases). Even though studying the notion of uniform peer sampling in a continuously evolving population does not make so much sense from a theoretical viewpoint, nevertheless it has a strong practical impact as in real settings churn or failures cannot be avoided.

As a consequence, current PSS implementations are not able to provide uniform samples in practice. This bias may result in poor performance of applications, congestion in underlying networks, sub-optimal utilization of storage resources, etc. Moreover, the bias can also give raise to security concerns: an hypothetical adversary that has access to the peer sampling service could exploit the bias to identify among the system participants the best candidates for an attack and consequently amplify the effectiveness of its malicious actions (*target selection attack*). Disrupting or taking control of these members can, in fact, cause major damages to the application with relatively little effort.

In this paper we introduce a *Peer Sampling Correction Module* that, when applied to an existing PSS, is able to reduce the bias in the stream of random samples exploiting tractable hashing functions such as linear transformations, MD5 and SHA1. The Peer Sampling Correction Module is constituted by a set of *Sampler*s, similar to those introduced by Keidar et al. in [13], that returns near uniform samples out of a data stream in which elements recur with an unknown bias. The stream is generated by the underlying PSS and a set of interconnected samplers continuously analyzes this stream to output periodically a set of uniform random samples that can be then used by other algorithms or applications.

The effectiveness of the Peer Sampling Correction Module is evaluated through an extensive set of simulations that show both convergence speed (time to deliver a near uniform sample) and the bias reduction in static and dynamic environments. The evaluation points out an interesting trade-off between the bias reduction and the time taken to deliver two consecutive samples by the Peer Sampling Correction Module: the larger is the bias reduction, the heavier is the local computation. However our experimental results show that an effective bias reduction can be achieved by using computationally lightweight hashing functions. Moreover experiments show another interesting trade-off between the bias reduction and the number of inactive nodes in the stream of samples provided by the Peer Sampling Correction Module: higher reductions results in higher numbers of inactive nodes. However, the system can be configured to obtain the desired level of bias reduction and inactive nodes. Finally, we show the efficiency of the Peer Sampling Correction Module in a case study: to prevent the target selection attack.

The rest of the paper is organized as follows: Section II introduces the architecture of the Peer Sampling Correction Module, while Section III contains its evaluation. Section IV introduces the target selection attack and evaluates the PSCM in this case study. Finally, Section V discusses related works and Section VI concludes the paper.

II. Reducing bias in samples distribution

The problem of uniformly sampling processes in a dynamic setting has been tackled in [13] for the first time. The authors introduced a distributed algorithm, together with its supporting architecture, that can be used to build and maintain a connected overlay network despite churn or failures and even in presence of corrupted (byzantine) processes. One of the fundamental components of this architecture is the *Sampler* that, exploiting min-wise independent permutations, is able to deliver a uniform sample out of a stream of biased samples. This component was designed to converge after a certain amount of samples observed on the incoming stream towards a single unbiased sample. While this property can be useful, as shown in [13], to guarantee connectivity in presence of byzantine processes, algorithms depending on uniform peer samples periodically require new set of samples. Moreover, the practical applicability of the original *Sampler* poses non trivial problems due to the complexity of implementing efficiently min-wise independent permutations whose computational cost is exponential with respect to the size of the input [14].

In this section we introduce a *Peer Sampling Correction Module* (PSCM) that can be deployed on top of a PSS to reduce the bias possibly present in the distribution of samples it returns. Exactly as the PSS does, our module periodically delivers a set of samples that can be used by distributed applications to update their local views. Moreover the PSCM can be practically deployed employing lightweight functions (e.g. linear transformations) greatly reducing the computational cost of the bias reduction with respect to the min-wise independent permutations.

A. Node Architecture

We consider a system constituted by a set of uniquely identified nodes whose size is finite but unknown. Nodes can join and leave the system at any time; the failure of a node is considered equivalent to a leave.

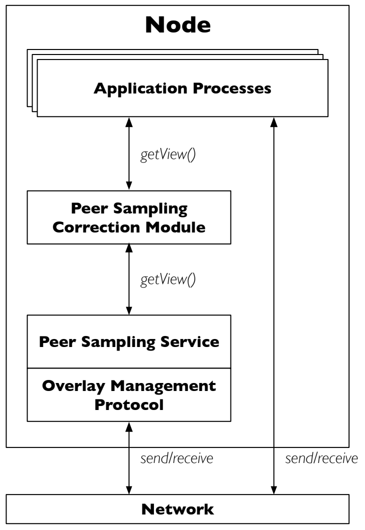


Figure 1. Node Architecture

Each node in the system implements the architecture depicted in Figure 1. Each application process running on a node only has a limited knowledge of the system represented by a *local view* containing the identifiers of a subset of size V of other nodes. Each application process executes some local computation and exchanges messages through a communication infrastructure only with other siblings running on nodes whose identifiers are contained in its local view. Each application process during its lifetime can update the content of its local view requesting a new view to a *Peer Sampling Service* (PSS) [11] through the *getView()* function. Node identifiers contained in views returned by the PSS are supposed to be random samples of the system population. Though the selection of random sample could be affected by bias, the PSS returns random sample of the nodes effectively into the distributed system, i.e. if a node crashes or leaves the system, the PSS does not include anymore the node in each view.

The PSS is able to periodically provide updated views, obtained by some computation executed on an overlay network connecting the entire system. The overlay network is maintained by an Overlay Management Protocol (OMP) that can guarantee connectivity of the system despite churn or failures. Note that we are not interested in specifying the protocols employed by either the PSS or the OMP as long as they satisfy the requirements expressed above; however, PSSs and OMPs

```
function void $Sampler.Reset()${
  1: $h \leftarrow randomFunc();\ minID \leftarrow \perp$
  2: }
function void $Sampler.nextView(view[])${
  1: for $i = 0$ to $view.length()$ do
  2:     if $minID = \perp \vee h(view[i]) < h(minID)$ then
  3:         $minID \leftarrow i$
  4:     end if
  5: end for
  6: }
function NodeID $Sampler.Sample()${
  1: return $minID$
  2: }
```

Figure 2. The Sampler's pseudo-code [13]

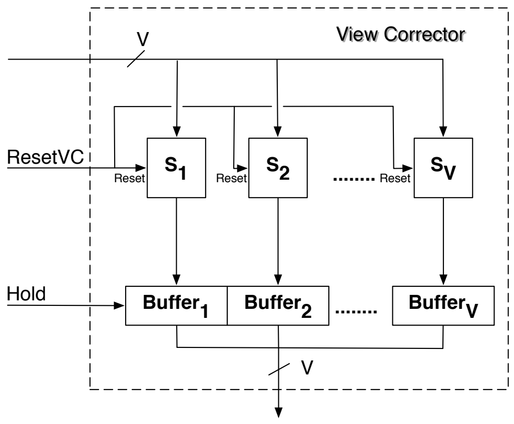


Figure 3. Architecture of the View Corrector

are often implemented as a single component [15], [16]. Here we assume that the PSS can deliver a new view every ΔT_{PSS} seconds.

In order to avoid problems introduced by bias in peer sample distributions we introduce, between the PSS and the application processes, a *Peer Sampling Correction Module* (PSCM). Application processes interact directly with this block that offers them the same interface that the PSS provides.

B. The Peer Sampling Correction Module

The fundamental component of the PSCM is the *Sampler* [13] whose behavior is described by the pseudo-code in Figure 2. Basically, the Sampler accepts as input views (i.e. biased sets of samples of size V) returned by the PSS and produces a single sample as output. The pseudo-code of the Sampler is composed by three parts: a *Reset()* function that initializes the Sampler resetting the $minID$ value and choosing at random a function h from a predefined set; a *nextView()* function that computes the sample $minID$ with the minimum image $h(minID)$ among samples observed since the last invocation of *Reset()*; finally the *Sample()* function the returns the current value of $minID$.

A relevant part of the pseudo-code is the invocation of $randomFunction()$ that randomly chooses a function h from a family. Possible families of functions are:

1) **linear transformations**: $h(x) = (ax + b) mod\ p$
2) **MD5**: $h(x) = MD5(a \circ x)$
3) **SHA1**: $h(x) = SHA1(a \circ x)$

More specifically, in 1) a and b are random values and p defines the size of the codomain of h. In particular p has to be as large as possible to minimize the probability of collisions among hash values. In 2) and 3) a is a random string concatenated with the identifier x. Consequently the initialization of a sampler lies in selecting some random values that will be used to compute h until a new $Reset()$ is invoked. For the sake of simplicity in the following we assume that all samplers use function chosen within a same family in order to describe and analyze separately the behavior of each family of functions.

Another component of PSCM is the *View Corrector* (VC) presented in Figure 3. The VC is composed by a set of V samplers, where V is the size of application processes' local views. All the samplers work concurrently: periodically receive the same input from the underlying PSS (i.e. sets of biased samples), compute the function h on each sample and maintain the current minimum as return value. The set of values maintained by the different samplers defines the set of samples returned by the VC itself. Note that, each sampler returns an independent value, as the choice of the function h is completely independent from other samplers.

The VC offers two functions: $ResetVC()$ that produces the simultaneous invocation of $Reset()$ on every sampler in the VC, and $Hold()$ that saves the current minimum of the different samplers in a buffer. The set of samples contained in the buffer represents the output of the whole VC, therefore an invocation to $Hold()$ causes an update of the output of the VC.

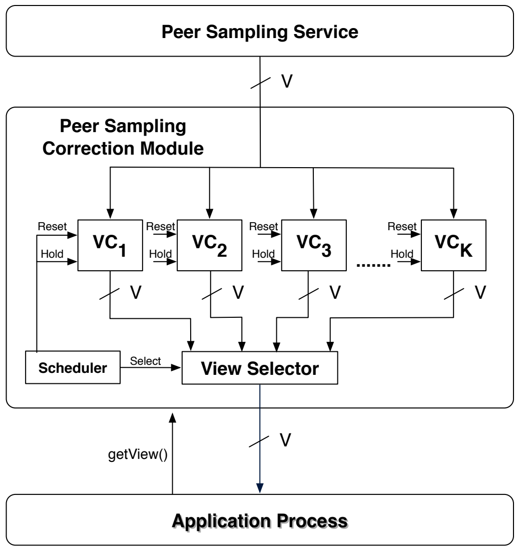


Figure 4. Architecture of the Peer Sampling Correction Module

Finally, the complete architecture of the PSCM is showed in Figure 4. It is composed by K VCs, each one able to return a set of samples with size V. A *view selector* is used to select the output of one VC at a time and use it as the output of the PSCM. The view selector is controlled by a *Scheduler* that periodically (with period ΔT):

1) holds the set returned by the i-th VC invoking on it the function $Hold()$);
2) resets the i-th VC invoking on it the function $ResetVC()$;
3) selects the value returned by the i-th VC as the return value of the PSCM;
4) increases i mod K.

Application processes can get an updated view using the getView() function that returns a set of V samples output by the VC currently selected by the Scheduler through the View Selector.

III. Evaluation

The aim of this section is to evaluate the behavior of the PSCM. More specifically, in the following we first analyze the ability of the VC component to provide uniform random samples and then we evaluate how much the PSCM can be configured to return updated view with a desired level of quality of bias reduction and with a desired period ΔT_{PSCM}. Finally we present an evaluation of computational cost of a real implementation of the PSCM.

Test details: All the simulations were realized using PeerSim [17] on top of which we implemented a PSS able to return peer samples with a predefined bias. Tests simulate two different kind of scenarios: a static scenario and a dynamic one. In the first case, the network is static (i.e. no nodes are added/removed during simulations) while in the second one, the system suffers a continuous addition /removal of nodes.

In both cases, tests have been conducted simulating 1000 nodes and varying the shape of the distribution of samples returned by the PSS in order to simulate the presence of bias. In static scenario, the sample distribution of our biased PSS has been shaped as a Pareto distribution with parameter α varying in the set $\{0.269, 0.349, 0.901\}$. These values have been chosen to recreate a scenario where 40%, 10% and 0.5% (respectively) of the nodes have probability 80% of being returned as samples by the PSS[1].

Evaluation in a dynamic scenario was conducted running the PSCM in a setting where some nodes constitute a stable core (i.e. they remain in the system for the whole duration of the test) and other nodes are continuously replaced with a rate that has been varied in the various tests. In this scenario we modeled the biased PSS discarding node identifiers from the stream of IDs provided by a uniform PSS with a probability inversely proportional to the node age (i.e. younger nodes are more likely to be discarded). More specifically, an identifier is discarded by the stream with a probability $P_i = e^{-\beta a_i}$, where β is a parameter that describes the level of bias and a_i represents the age of the node which identifier is i. This model is intended to simulate the behaviour of PSS based on view exchange techniques in a scenario characterized by strong churn. Reference values are represented by a scenario with perfectly uniform samples. Each single result plotted in the following graphs has been obtained by averaging 300 values collected during a simulation run.

The behaviour of the PSCM and its components has been evaluated by considering the following metrics:

1) **Frequency**: the frequency of a node at a certain time instant is the number of occurrences of its identifier in views of active processes. In an ideal setting all processes should have a same value of frequency; this implies that views returned by the PSCM contain, indeed, uniform random samples.
2) **Uniformity Error**: it represents the average standard deviation of the ratio between the frequency of each correct process and the frequency of the same process in presence of ideal sampling where each node has the same frequency F. In an ideal setting this value should be zero, i.e. the peer samples are perfectly uniform. Practically, obtaining this perfect uniform sampling is impossible and also pseudo-random uniform number generators show an Uniformity Error different from zero.
3) **Convergence Time**: it represents the number of distinct views provided by the PSS that a VC has to evaluate before being able to provide a view with a desired Uniformity Error. Note that, given the frequency at which the PSS is able to provide views, this figures gives a rough idea of the time needed to the VC to provide a set of unbiased samples.

The evaluation of the PSCM has been conducted in three phases. Firstly, in order to understand the behaviour of the VC submodule, we tested a PSCM containing a single VC ($K = 1$). In this scenario we evaluated how much the bias present in samples returned by the PSS is reduced with time and how much time it takes to the VC to reach a desired level of Uniformity Error using different function families for its internal samplers.

Secondly, we evaluated the impact of churn on the PSCM considering different node replacement rates. More specifically the aim of this phase was evaluating the distribution of samples provided by a VC in a dynamic setting where nodes are continuously added/removed. This study was conducted analyzing the number of inactive nodes in the view of active ones and the behavior of the bias in random samples produced by VCs.

Finally, we evaluated the PSCM as a whole, analyzing its configurability and computational cost. In order to do this, we implemented a PSCM leveraging standard Java SE Security APIs and then analyzed its behavior changing the number of VCs and their resetting frequency. Moreover, we evaluated the percentage of CPU time spent by the PSCM in

[1]The model is coherent with results proposed in [18]. Moreover the behaviour of the system does not change so much modeling the bias by means of different distributions. Consequently, for lack of space, we don't include similar experimental results in the paper

the time interval between two consecutive views provided by the underlying PSS.

All the tests, unless stated otherwise in the text, were conducted simulating a system with 1000 nodes and representing a biased PSS through views characterized by a Pareto distribution with shape 0.269.

A. *Evaluation of the View Corrector in a static setting*

In this first phase of our evaluation we want to investigate if a VC is able to reduce the Uniformity Error of biased samples produced by the PSS and, consequently, the obtainable level of the bias reduction that it is able to provide. This investigation is conducted in a static setting where the set of nodes participating to the system does not change.

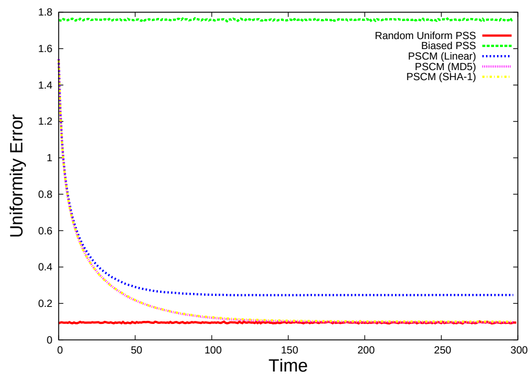


Figure 5. Uniformity Error evolution is shown over time; distinct PSCMs are compared with uniform and biased PSSs in a system with 1000 nodes and local views of size 100.

Figure 5 shows how the Uniformity Error changes as time passes. Note that the notion of time present in the graph is directly linked to frequency at which the PSS is able to deliver new views, i.e. each time tick on the graph is equal to ΔT_{PSS} seconds of real time. The curves depicted in the graph show how the MD5 and SHA-1 families of functions are able to provide a Uniformity Error that is close to the one characterizing an ideal PSS that employs pseudo-random functions to select samples among the set of currently active nodes. The curves perfectly overlaps after $Time = 150$. The linear transformation family, in the same scenario, delivers a remarkably small Uniformity Error, but it stabilizes at an asymptotic value that is larger than the one provided by MD5 and SHA-1.

These results are confirmed by the graph in Figure 6 where the Frequencies of nodes are shown; the results refer to a system that has reached a steady state, i.e. a system where the VC has converged to its final output (after the evaluation of 200 biased views). The curves show how the presence of the VC is able to drastically reduce the level of bias that becomes similar, for MD5 and SHA-1 families, to the one provided by an ideal uniform PSS where each process has the same frequency F.

Let us now analyze more in detail the dependency of the Uniformity Error from the level of bias present in samples

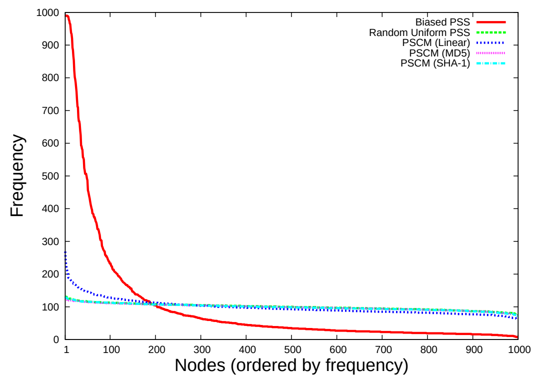
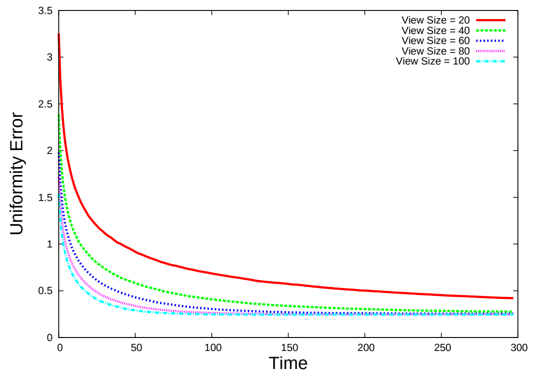
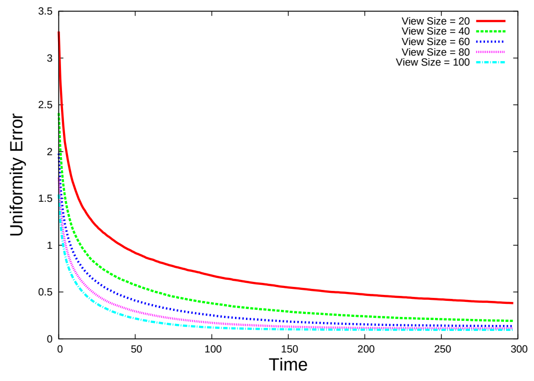


Figure 6. Frequencies of nodes after VC convergence; disctint PSCMs are compared with uniform and biased PSS in a system with 1000 nodes and local views of size 100.



(a) Linear Transformation



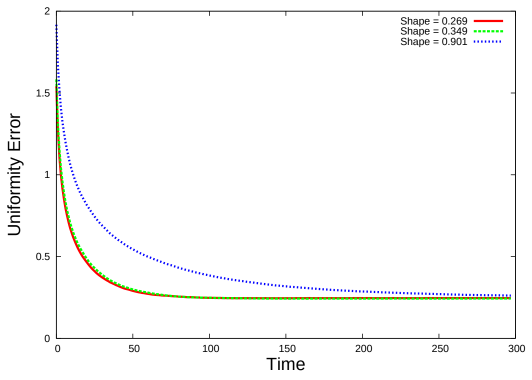
(b) MD5

Figure 7. Uniformity Error evolution is shown over time varying the size of local views for linear transformations and MD5 families of functions.

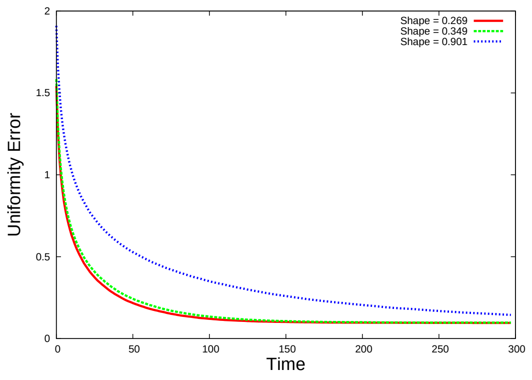
provided by the PSS and the size of local views. For the sake of simplicity we present only results related to linear transformations and MD5[2]. Figure 7 reports the Uniformity Error that can be obtained using linear transformations (Figure 7(a)) and MD5 (Figure 7(b)) varying the view size. The curves show that, in both cases, the Uniformity Error decreases more rapidly when the size of local views is increased and confirms that only MD5 is able to reach an Uniformity Error similar to one introduced by an ideal uniform PSS. This result outlines

[2]Results for SHA-1, not shown here, closely match those provided by MD5.

the important role that the local view size has on the time needed by VCs to converge to a desired level of Uniformity Error.



(a) Linear Transformation



(b) MD5

Figure 8. Uniformity Error evolution is shown over time varying the shape of the Pareto distribution for linear transformations and MD5 families of functions; the size of local views was 100.

Figure 8 reports the impact of different levels of bias (e.g. different shapes for the Pareto distribution used to simulate bias in samples) on the performance of the VC and on time it needs to converge to a desired level of Uniformity Error. The curves show that increasing the level of bias has a negative impact on the time needed to converge, but does not affect the final result, i.e. the asymptotical Uniformity Error. This result outlines the need to accurately characterize the bias present in samples in order to correctly infer the convergence time for the VCs. Indeed, this time value is a fundamental parameter to correctly configure the PSCM and, more specifically, the number K of VCs needed in the PSCM itself.

B. Evaluation of the View Corrector under Churn

This section describes the behaviour of the VC in a dynamic setting where churn is present. We considered a system composed by two sets of nodes: a stable core, constituted by 10% of the nodes in the system, and the set containing the remaining nodes that are continuously replaced at a fixed rate (i.e. the level of churn). This model simulates a scenario where the PSS is not able to react properly to dynamism and returns with high frequency old nodes, while younger ones are only rarely returned.

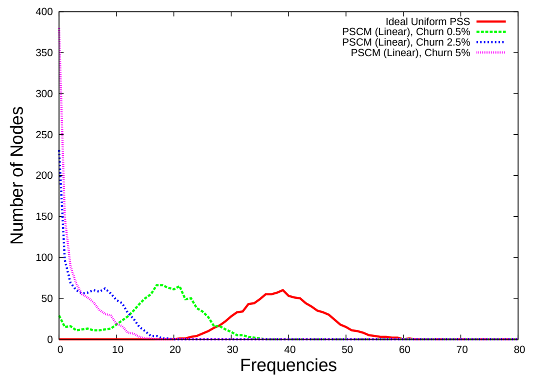


Figure 9. Distribution of Node Frequency under churn considering different levels of churn; the size of local views was 40.

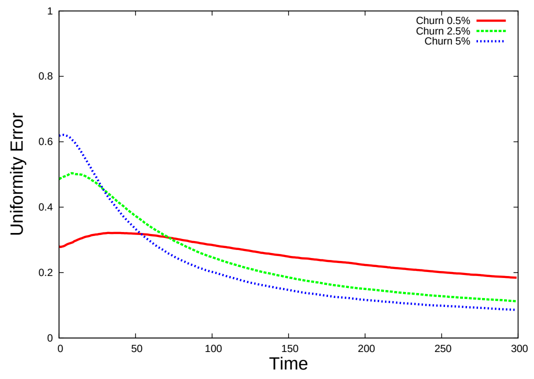


Figure 10. Impact of churn on the Uniformity Error for linear transformations family of functions; the size of local views was 40.

In this setting we expect views returned by the PSCM to contain a number of outdated samples caused by nodes that left the system. This number will increase as the churn level is raised. The presence of outdated samples causes a reduction of the average Frequency of nodes as shown in Figure 9. The graph reports the Frequency distribution after 300 time ticks considering different levels of churn (respectively 0.5%, 2.5% and 5% of nodes replaced per time unit) using linear transformations[3]. While the Frequencies for the ideal case are uniformly spread among a mean that is equal to the view size, as the churn level is increased Frequencies begins to shift toward the left side of the graph: this means that a larger number of nodes do not survive enough time in the system to appear in other node views.

Interestingly, this fact has a counterintuitive effect on the Uniformity Error. Figure 10 reports the evolution of Uniformity Error with time; the curves show how, after an initial transient state, the Uniformity Error decreases more rapidly with larger levels of churn. This behaviour is a consequence of the shift of Frequency toward values closer to 0. Clearly,

[3]Due to space restrictions we omit the results for the other two families of functions that showed a similar behaviour

SN	**Churn 0.5%**	**Churn 2.5%**	**Churn 5%**
50	0.146	0.468	0.636
100	0.282	0.664	0.791
200	0.462	0.810	0.890
300	0.572	0.875	0.931

Table I
PERCENTAGE OF OUTDATED SAMPLES CONTAINED IN VIEWS. THE TEST CONSIDERS A SYSTEM WITH VIEW SIZE 40 EMPLOYING LINEAR TRANSFORMATION.

	% of CPU Time
Linear	0.1
MD5	2.265
SHA1	3.525

Table II
THE PERCENTAGE OF CPU TIME SPENT IN CALCULATING FUNCTIONS IS SHOWN FOR DIFFERENT FAMILIES OF FUNCTIONS. THE TEST CONSIDER A SYSTEM WITH $SN = 150$ AND VIEW SIZE 100.

the curves reported in this graph show only part of the story: a larger level of churn reduces the Uniformity Error, but this is paid with larger amounts of outdated samples contained in the views returned by the PSCM. The PSCM gets rid of these samples periodically when VCs are reset, thus the period of reset can be tuned to reduce this issue. Table I reports the percentage of outdated samples present in views assuming that a VC is reset every SN time ticks. A correct estimation of the *sampling number* SN is thus fundamental to identify a good tradeoff between the Uniformity Error and the percentage of outdated samples reported by the PSCM in a setting characterized by the presence of churn.

C. Evaluation of the Peer Sampling Correction

In this section we evaluate the computational cost of the PSCM, with a number $K > 1$ of VCs, and we show how it is possible to configure this service to produce new views with a desired level of the Uniformity Error at a specific rate.

Firstly, starting from the results shown in previous section we know that it is possible, given a specific scenario (i.e. a precise characterization of the samples bias), calculate the number of biased views that a VC has to evaluate in order to converge to a target Uniformity Error. This number can be used to guide the realization of the whole PSCM. More specifically, the period ΔT_{PSCM} at which the PSCM can deliver updated unbiased views is defined as

$$\Delta T_{PSCM} \geq \frac{SN * \Delta T_{PSS}}{K} \qquad (1)$$

This formula directly follows from the architecture depicted in Section II-B where the scheduler resets the VCs using a round-robin procedure. In this way the time interval between two consecutive $ResetVC()$ invocations on the same VC is $K * \Delta T_{PSCM}$. Due to the fact that we want this time to be greater than $SN * \Delta T_{PSS}$, the previous formula follows.

Using this formula we can choose the number of VCs K that we need to obtain a desired update frequency of unbiased views. In particular, in order to obtain $\Delta T = \Delta T_{PSS}$ (i.e. obtain unbiased views at the same rate of views delivered by the underlying PSS), we must configure the PSCM composed with a number of VCs $K = SN$.

Finally, to evaluate the computational cost of this kind of configuration we used a 2GHz Intel Core Duo PC where we ran an implementation of PSCM realized using the Java SE Security API for the MD5 and SHA-1 functions. We configured the PSCM with $SN = 150$ and view size 100. This kind of configuration requires 15000 computations of the hashing functions and in table III-B we present a summary of the percentage of CPU Time spent by the PSCM when the underlying PSS provides a new view each time interval $\Delta T_{PSS} = 10s$. These results show how it is possible to compute a huge number of functions using very small computational time, particularly considering linear transformations. The trade-off between computational cost and efficiency in bias remotion justifies the employment of the linear transformation when we want to develop a light version of PSCM or when we have only machines with limited resources. Conversely, if more computational resources are available, more complex hashing functions like MD5 and SHA-1 can provide application with smaller levels of Uniformity Error.

IV. A Case Study: Prevention of Target Selection Attack

We evaluate the PSCM in a case study where we introduced an adversary able to corrupt nodes and we studied how much the PSCM can reduce the power of the adversary's attacks. In this scenario we evaluated the resilience of the whole system to adversary's attacks varying the percentage of nodes in the system that the adversary is able to corrupt.

More specifically, we assume the presence of an adversary that first selects a set of target nodes belonging to the system and then corrupts them in order to attack an application that runs on the top of a PSS. When a node is corrupted, the adversary can impose any behavior to the processes running on it through for example out of band communication (a non-corrupted node is said to be correct). We assume the adversary is able to access the PSS and collect the list of samples it returns (listening phase), but the adversary cannot influence in any way the behavior of the PSS.

The goal of the adversary is to maximize the presence of corrupted node identifiers in the local views of processes running on correct nodes, minimizing at the same time the total number of corrupted nodes.

In this attack the adversary can use the bias of the PSS and the small view size stored in each node to achieve its goal. The adversary observes the sequence of samples output by the PSS to estimate the current sample bias and to identify nodes returned with higher frequency selecting these nodes as targets. This behavior will increase the effectiveness of its attack as the identifiers of these processes will appear in a larger number of local views belonging to correct processes.

Let us remark that the target selection attack is different from the hub attack addressed in [13] and [19] because the latter addresses gossip-based implementation of a PSS while the target selection attack is done at the application level using the information output by the PSS (independently of its implementation design).

In order to quantify the effectiveness of this attack in presence of biased samples, we introduce another metric:

4 **Percentage of View Corruption**: it represents the average ratio between the number of corrupted node identifiers contained in the local views of correct ones and the size of local view. If samples are uniform this percentage should not exceed the percentage of corrupted nodes in the system.

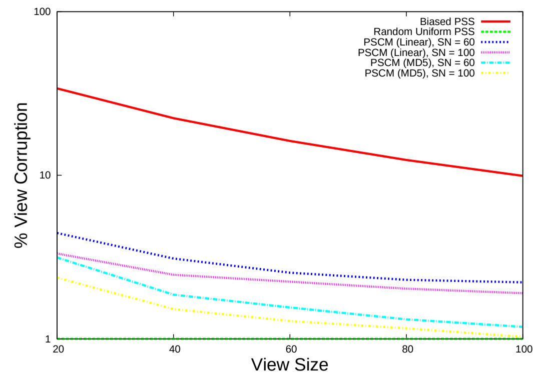


Figure 11. Impact of the Adversary is shown varying the size of local views for distinct PSCMs and different values of SN.

Figure 11 reports the percentage of view corruption varying the view size and the value of SN. We can see how, also for small view sizes and small values of SN, the PSCM is able to reduce this metric of an order of magnitude with respect to a biased PSS. For $SN \geq 100$ the percentage of view corruption reaches a level similar to the one showed by an ideal uniform PSS. This means that the adversary is not able to significantly increase its influence on the whole system if the PSCM is deployed and opportunely configured.

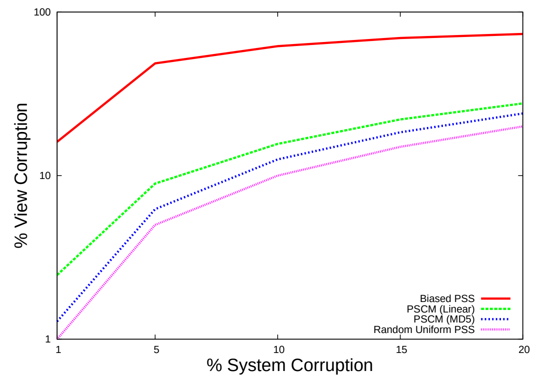


Figure 12. Impact of the Adversary is shown varying the percentage of corrupted nodes for distinct PSCMs and different values of SN.

Another important aspect is pointed out by Figure 12 where we analyze the behavior of the system with respect to the percentage of view corruption while increasing the percentage of nodes corrupted by the adversary. The graph shows how the PSCM is able to improve the overall resilience of the system even when the number of processes corrupted by the adversary grows. In particular, in presence of the PSCM the percentage of corrupted processes in local views grows linearly with the percentage of nodes corrupted. When the PSCM is not adopted, i.e. local views are updated with biased samples provided by the PSS, the figure grows exponentially up to 70%. This result shows how, thanks to the PSCM, application processes can obtain uniform random views also in presence of a huge number of corrupted processes.

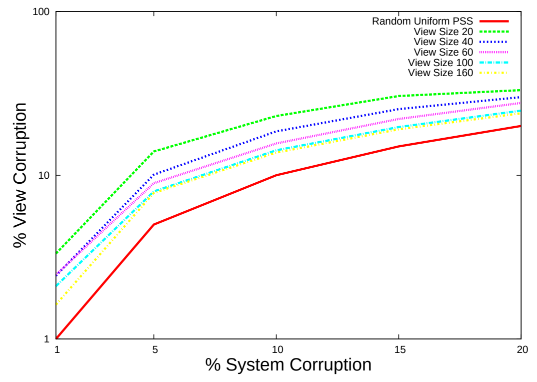


Figure 13. Impact of the Adversary is shown varying the percentage of corrupted nodes for a PSCM using linear transformations and different view sizes.

We conclude this evaluation by showing, in Figure 13, again the behaviour of the system with respect to the percentage of view corruption while increasing the percentage of nodes corrupted by the adversary. However, in this case the results refer only to a PSCM employing the family of linear transformations. The different curves depicted in the graph show this family of functions works at its best only when the VCs inside the PSCM can evaluate larger views. When the size of views returned by the PSS grows, the performances showed by the system equipped with the PSCM approach those obtainable in the ideal case.

V. Related Work

a) On the uniformity of samples in a network: Getting uniform sample in a network can be done through random walk [8], [20] and DHT [12] as well as gossiping. However the correctness of random walk-based sampling depends on the network topology. If actual topology is different from the assumed one, then the sample produced by the random walks may be far to be uniform [8], [18]. Let us remark that there are no study in the direction of making sampling based on random walk and on DHT resilient to byzantine attacks.

From the performance viewpoint, there are some papers in the literature that discuss unexpected performance behaviors exhibited when the PSS does not deliver anymore uniform samples [4], [10]. For example, in [4], the authors show how the bias introduced by the PSS can create a core of processes acting like a central cluster or an hub. The presence of this cluster could, in principle, create some benefits for the applications: the cluster is usually formed by peers that participate to the system computation for long periods of time, thus increasing the overall topology robustness. However, the presence of this cluster causes undesired behaviors as well: nodes within this cluster are usually the target of a large number of application messages, leading to load unbalance within the application and possibly to link congestions.

b) Peer Sampling Services resilient to byzantine attacks: The problem of designing peer sampling with small local views resilient to byzantine attack is a recent branch of research and it has been addressed only for peer sampling services based on gossiping [13], [19]. Many works on byzantine membership based on gossiping have either considered static settings where the full membership is known at all [21], [22] or focused on maintaining full local view [23], [24] rather than samples.

Keidar et al. in [13] propose a gossip-based protocol able to realize an uniform peer sampling service with small views in presence of byzantine processes. They formally prove that their solution is able to guarantee an uniform random sampling when churn into the system stops. Moreover they employ Samplers, such as we do in this paper, to be resilient to different kind of attacks by byzantine processes. Compared to their work, we take a practical approach to the realization of a peer sampling by working on the top of a possibly biased peer-sampling service with the aim of reducing such a bias on-the-fly and without stopping churn.

VI. Conclusion

An implementation of a uniform peer sampling service in a continuously evolving system produces a biased stream of network samples due to changing in network topologies and churn. Also this bias can be enhanced by attacks that target the internal peer sampling algorithm. Theoretical approaches have indeed formally proved that uniform peer sampling is possible only on quiescent systems [13]. Therefore providing a uniform peer sampling in a continuously evolving system is a practical challenge.

The paper has introduced a Peer Sampling Correction Module that is able to reduce the bias of sampling through a simple and fully decentralised solution without additional costs in terms of communication. The module is based on interconnection of samplers similar to the ones presented in [13] and on the usage of well-known hashing functions. The paper pointed out an interesting trade-off between the reduction of bias and the computational cost of this reduction at each node. Moreover proposed solution is also shown to be configurable to obtain an high level of bias reduction also under a continuous arrival and departure of nodes.

After the paper has analyzed how an application, working on the top of a biased peer sampling service, can be defeated easily by an adversary, executing a target selection attack, due to its capacity to select a small number of targets that can produce a great damage to the application. Finally we have shown how this peer sampling correction module can largely mitigate the effect of a target selection attack also using just a few samplers and lightweight hash function (e.g. linear transformation).

References

[1] P. T. Eugster, R. Guerraoui, A. Kermarrec, and L. Massouliè, "Epidemic information dissemination in distributed systems," *IEEE Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[2] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Transactions on Computer Systems*, vol. 23, no. 3, pp. 219–252, 2005.

[3] K. Iwanicki, M. van Steen, and S. Voulgaris, "Gossip-based synchronization for large scale decentralized systems," in *Proceedings of the Second IEEE International Workshop on Self-Managed Networks, Systems and Services*, 2006, pp. 28–42.

[4] R. Baldoni, A. Corsaro, L. Querzoni, S. Scipioni, and S. Tucci-Piergiovanni, "Coupling-based internal clock synchronization for large scale dynamic distributed systems," MIDLAB - University of Rome "La Sapienza". A preliminary version apeeared at DOA 2007. http://www.dis.uniroma1.it/~midlab/publications.php, Tech. Rep., 2008.

[5] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, 2006, pp. 117–124.

[6] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 84–95.

[7] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh, "Proceedings of the twenty-fifth annual acm symposium on principles of distributed computing," in *Peer counting and sampling in overlay networks: random walk methods*, 2006, pp. 123–132.

[8] C. Gkantsidis, M. Mihail, and A. Saberi, "Random walks in peer-to-peer networks," in *Proceedings of 23th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, 2004, pp. 130–140.

[9] C. Law and K. Siu, "Distributed construcion of random expander networks," in *Proceedings of 22th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, 2003, pp. 2133–2143.

[10] R. Baldoni, R. Beraldi, V. Quéma, L. Querzoni, and S. T. Piergiovanni, "Tera: topic-based event routing for peer-to-peer architectures," in *DEBS*, ser. ACM International Conference Proceeding Series, H.-A. Jacobsen, G. Mühl, and M. A. Jaeger, Eds., vol. 233. ACM, 2007, pp. 2–13.

[11] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: experimental evaluation of unstructured gossip-based implementations," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, 2004, pp. 79–98.

[12] V. King, S. Lewis, and J. Saia, "Choosing a random peer in chord," *Algorithmica*, vol. 49, no. 2, pp. 147–169, 2007.

[13] E. Bortinikov, M. Gurevich, I. Keidar, G. Kliot, and A. shaer, "Brahms: Byzantine resilient random membership sampling," in *Proceedings of 27th ACM Symposium on Principles of Distributed Computin*, 2008, pp. 145–154.

[14] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, 2000.

[15] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[16] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov, "Lightweight probabilistic broadcast," *ACM Transactions on Computer Systems*, vol. 21, no. 4, pp. 341–374, 2003.

[17] "Peersim: A peer-to-peer simulator," http://peersim.sourceforge.net. [Online]. Available: http://peersim.sourceforge.net

[18] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama, "Distributed uniform sampling in unstructured peer-to-peer networks," in *Proceedings of the 39th Hawaii International Conference on System Sciences*, 2006, pp. 223–233.

[19] G. P. Jesi, D. Hales, and M. van Steen, "Identifying malicious peers before it's too late: A decentralized secure peer sampling service," in *SASO*. IEEE Computer Society, 2007, pp. 237–246.

[20] L. Massouliè, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh, “Peer counting and sampling in overlay networks: Random walk methods,” in *Proceedings of the 25th annual ACM symposium on Principles of Distributed Computing*, 2006, pp. 123–132.

[21] D. Malkhi, Y. Mansour, and M. K. Reiter, “On diffusing updates in a byzantine environment,” in *Proceedings of the 18th Symposium on Reliable Distributed Systems*, 1999, pp. 134–143.

[22] Y. M. Minsky and F. B. Schneider, “Tolerating malicious gossip,” *Distributed Computing*, vol. 16, no. 1, pp. 49–68, 2003.

[23] H. Johansen, A. Allavena, and R. van Renesse, “Fireflies: scalable support for intrusion-tolerant network overlays,” in *Proceedings of the 2006 EuroSys Conference*, 2006.

[24] G. Badishi, I. Keidar, and A. Sasson, “Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004, pp. 201–210.