BuzzPSS: A Dependable and Adaptive Peer Sampling Service

Nuno Machado Francisco Maia Miguel Matos Rui Oliveira
HASLab
INESC TEC and U. Minho
Braga, Portugal
Email: {nuno.a.machado, francisco.a.maia, miguel.m.matos, rcmo}@inesctec.pt

***Abstract*—A distributed system is often built on top of an overlay network. Overlay networks enable network topology transparency while, at the same time, can be designed to provide efficient data dissemination, load balancing, and even fault tolerance. They are constructed by defining logical links between nodes creating a node graph. In practice, this is materialized by a Peer Sampling Service (PSS) that provides references to other nodes to communicate with. Depending on the configuration of the PSS, the characteristics of the overlay can be adjusted to cope with application requirements and performance concerns.**

Unfortunately, overlay efficiency comes at the expense of dependability. To overcome this, one often deploys an application overlay focused on efficiency, along with a safety-net overlay to ensure dependability. However, this approach results in significant resource waste since safety-net overlays are seldom used.

In this paper, we focus on safety-net overlay networks and propose an adaptable mechanism to minimize resource usage while maintaining dependability guarantees. In detail, we consider a random overlay network, known to be highly dependable, and propose BuzzPSS, a new Peer Sampling Service that is able to autonomously fine-tune its resource consumption usage according to the observed system stability. When the system is stable and connectivity is not at risk, BuzzPSS autonomously changes its behavior to save resources. Alongside, it is also able to detect system instability and act accordingly to guarantee that the overlay remains operational. Through an experimental evaluation, we show that BuzzPSS is able to autonomously adapt to the system stability levels, consuming up to 6x less resources than a static approach.

I. Introduction

Nowadays, almost every application or information service depends, at least indirectly, on a distributed system. A critical issue to address in a distributed system is *membership*. Membership is related with the ability to determine if a certain node is taking part or not in the distributed system and which are its peers, *i.e.,* the nodes that are also part of the system. In particular, from the point of view of a single node, this is actually translated into the ability to determine the nodes with which it can communicate with.

Naturally, the most immediate approach to implement a membership service in a distributed system is to maintain, at each node, a list of all other nodes in the system. This is called *full membership* and most of the classical distributed systems rely on such type of service. However, this is only achievable in small to medium sized systems. As the system grows in size, trying to maintain information about all the nodes in the system becomes unfeasible and impairs system scalability. To address these problems, membership services may provide each node with only a small list of other nodes in the system (*partial membership*). In this case, the node list is called *view* as it represents the point of view of such node with respect to the system configuration. Along this work, we focus on partial membership for large scale distributed systems and on the dependability of the membership service. In this context, dependability is given by the ability of a given correct node to reach every other correct node either directy or indirectly by traversing the graph induced by the partial views. As we will see below, the key is to maintain this graph connected in an efficient manner even in the presence of faults.

An important thing to notice is that different implementations of partial membership lead to different system configurations and behavior. In fact, if we consider the collection of partial views in the entire system, we can build a graph based on the logical links of those partial views. Naturally, depending on which type of graph emerges, the distributed system exhibits different properties. As an example, let us assume a distributed message dissemination application. If node 1 wants to send a message to node 7, it must search for node 7 in its view. If node 1 finds the correspondent link, then it can contact node 7 directly. Otherwise it must forward the message to other nodes. Depending on how the membership service populates the node's view, this process can be more or less efficient. For instance, the membership service could have populated the node's views in such a way that a logical ring or a binary tree between nodes would emerge. Here, the former would be less efficient than the latter, because disseminating a message in a logical ring executes in linear time, whereas the same task in a binary tree executes in logarithmic time.

The logical graph that emerges from the collection of the views is known as *overlay network*. Overlay networks support a variety of applications and can be optimized for content dissemination, load balance or fault tolerance. As expected though, in the design of the overlay, one must make a compromise between efficiency and dependability. Recalling the message dissemination example, in order to disseminate a message even when a fraction of the nodes may have failed, the system must resort to redundancy, which is costly.

Regarding dependability, it is known that random overlay networks, where views are composed of random node references, are particularly dependable due to their inherent redundancy. On the other hand, tree-based overlay networks are known to be very efficient but prone to dependability issues due to their frailty when exposed to node failure.

In the pursuit of a system design that is both dependable and efficient, one can resort to a mixed approach. Concretely, it is possible to deploy, simultaneously, an efficient overlay network and a more dependable one. The idea is to use the efficient overlay whenever it is available while, in case of failure, resort to the less efficient but dependable one. Alas, this approach comes with additional resource costs as two overlays must be maintained.

In this paper, we consider this mixed approach and focus on the dependable overlay used as a *safety net*. In particular, we propose a novel, dependable membership service that autonomously adapts to the deployment environment. The goal behind our approach is to provide an overlay that is continuously available, ensuring the system's dependability, with minimal resource usage cost. The intuition driving our design is that, typically, a distributed system instability is not constant. There are periods of system stability, where nodes do not fail, and periods of system instability, where nodes fail. Additionally, instability periods may be of different severity. Consequently, the dependable overlay network should be able to adjust its resource usage accordingly.

The main contribution of the paper is BUZZPSS, a dependable membership service that automatically adapts its resource usage according to the system actual stability degree. In a nutshell, BUZZPSS leverages online learning techniques to fine tune the rate at which nodes exchange messages with the peers in their view, such that this rate increases when nodes are leaving the system, and decreases otherwise.

Our experimental evaluation shows that BUZZPSS is able to significantly reduce the overall resource usage when compared with the traditional, static approach. Moreover, we show that BUZZPSS is able to do so completely autonomously, dynamically, and with minimal configuration effort.

The rest of the paper is organized as follows. In Section II, we provide some background and lay down some useful abstractions for the remainder of the paper. Section III describes the design of BUZZPSS and the intuition that drove it. The results experimental evaluation are presented in Section IV. In Section V we present a brief survey of related work and conclude the paper in Section VI.

II. BACKGROUND

For the purpose of clarity, in this paper, we consider the following abstraction. We consider a *membership service* that provides each node in the system with a list of peer references. These references abstract the actual physical network and represent a logical link between nodes and can be typified as an IP:port pair. It is assumed that these logical links can be mapped to physical links but how this is achieved is out of the scope of the present paper.

Additionally, we refer to the list of peer references maintained at each node as *node view*.

As described previously, we focus on partial membership where views are necessarily small with respect to the system size. Membership services are also known in the literature as Peer Sampling Services (PSSs) [1]. As the name suggests, a PSS implies a continuous sampling activity, which is necessary for the system to cope with node entrances and departures. In fact, node views need to be dynamically maintained in order to reflect such changes in the system configuration. The way this is achieved is implementation dependent. Different implementations can offer different properties with respect to the view characteristics. For instance, some implementations may ensure that nodes in the view are alive and reachable, while other may relax those guarantees and simply ensure that those nodes are alive and reachable with a certain probability.

In the context of the present paper, we are interested in peer sampling services that ensure connectivity between nodes at all times. Concretely, we are interested in providing a safety net membership service to which nodes can always resort in case other, more efficient, services are not available. Our main goal is thus to design and implement a dependable peer sampling service whose resulting overlay network is a connected graph. By definition, a connected graph is a graph where there is path between every pair of vertices. In our case, vertices are nodes and paths represent communication paths. Consequently, if there is a path between nodes it means that nodes can communicate with each other.

Different peer sampling services have been proposed in the literature [2], [3], [4], [5]. However, in this paper, we focus specifically on Cyclon [3]. The reason behind this decision is that Cyclon maintains a random overlay network between nodes. Random overlays are extremely resilient and offer desirable data dissemination guarantees as well [6], [7].

Cyclon works in two phases. First, it goes through a bootstrapping process, which can be centrally managed (for instance, resorting to well-known entry points from which new nodes can obtain references to other nodes) or rely on random walks [3]. In either case, the bootstrapping process provides each node with a list of f nodes randomly sampled from the entire system (which corresponds to the node's partial view). This first phase is only run once at system initialization. The second phase is run throughout the entire system lifetime. In this second phase, each node periodically exchanges its view with other nodes in the system and these periodic information exchanges allow the protocol to cope with system dynamism.

The PSS proposed in this paper follows the same bootstrapping phase as Cyclon but improves on the exchange phase. Thus, from this point on, we focus on the latter.

In Cyclon, each node is configured to run its exchange phase every T units of time. Each exchange works as follows. Node references in the view are tagged with an *age*. Such age is increased by one every T units of time. At each exchange, nodes choose the oldest node in their view and send to it their view, replacing the receiver node's reference with a self-reference of age 0. Upon the reception of an exchange message, nodes incorporate the received nodes into their view replacing the oldest reference with the one with age

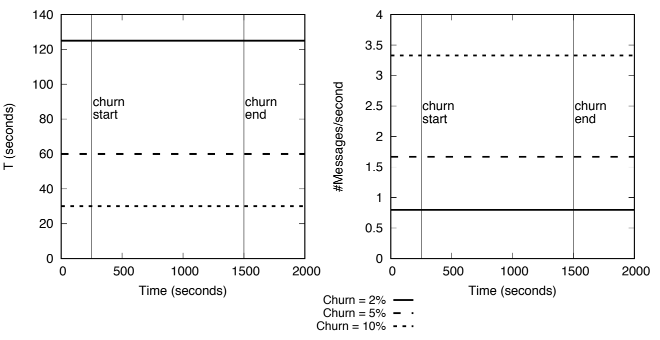

Fig. 1: Gossip period T and message rate that cause the overlay to break, for churn levels of 2%, 5%, and 10%.

0. Additionally, they reply to the sender with their own view. The result is a view exchange (or swap) where, from the point of view of the entire system, views are refreshed and updated. This way, nodes which are alive refresh their references in other node's views and nodes that are no longer active are eventually removed from every view.

Naturally, the T parameter is of critical importance as it defines the rate at which the protocol is able to incorporate system membership changes. A very low T means that nodes frequently exchange information and adapt quickly to node entrance and departure. In contrast, high values of T slow down the adaptation process (an excessively high T may even lead to graph partitions if the system is under high levels of node departures). Hence, one can say that peer sampling services have an inherent trade-off between dependability and efficiency (in terms of number of messages exchanged). Figure 1 supports this claim by plotting, for different levels of churn (namely 2%, 5%, and 10%), the gossip period T and the message rate that cause the overlay to break. The results where obtained from simulations of 2000 seconds with 100 nodes (each with a view size of 8), where churn was applied to the system during the interval $[250s, 1500s]$.

As the data shows, different levels of churn require different peer sampling rates. For example, with 2% of churn, the value of T can be safely set below 125s (which is the value that breaks the overlay), thus yielding a low message rate. Conversely, for a churn level of 10%, T must to be much lower (less than 30s) in order to maintain the overlay connected. Of course, for this latter case, the system will require more network bandwidth, as the message rate increases.

In the next section, we describe our technique to address the challenge of finding the value of T that allows the overlay to be both efficient and dependable, even in the presence of variable levels of churn.

III. Self-Tuning Peer Sampling

As shown in the previous section, configuring a peer sampling service (PSS) requires judiciously choosing the shuffling interval T. In fact, this choice corresponds to a non-trivial trade-off between dependability and efficiency. On one hand, a short T allows nodes to maintain fresh views (thus ensuring reliability when nodes leave the network), but implies a large number of messages being exchanged among the peers. On the other hand, a long T dramatically reduces the bandwidth occupation, albeit at the cost of the system's robustness.

In this section, we describe an adaptive solution, called BuzzPSS, to tune the shuffling period according to the system's status. The underlying intuition is simple: increase the value of T when the system is stable (*i.e.,* nodes are not leaving/joining the network), and reduce this value otherwise.

Designing such a self-tuning peer sampling service raises two main challenges though: *i)* how to assess the stability of the system (nodes have to decide on whether to change the shuffling interval based solely on local knowledge), and *ii)* how to define the magnitude of the adjustment of the gossip period (modifying T by a small amount will not incur significant efficiency improvements, whereas a coarse-grained adjustment might prevent the system from converging to a stable scenario).

We address the first challenge by observing nodes' *average view age* at the beginning of each shuffling period. Each entry in a node's view has an age indicating how many intervals have passed since the moment it was created by the neighbor node it points at [3]. Nodes that are alive communicate periodically with each other, sending their IDs with age 0. Hence, the age of a neighbor provides a rough estimation on whether that node is alive or not. Moreover, for a stable system, the average age of a node's view should remain consistent across time.

BuzzPSS relies on this observation to assess the stability of the system. Concretely, at the beginning of each shuffling period, BuzzPSS computes the current average view age and compares this value to that of the previous iteration. If the current average age is higher than the previous average age (within a given limit L), then BuzzPSS decreases T to force the node to refresh its view at a higher rate. Conversely, if the current average age is lower than the previous average age, then BuzzPSS increases T to save network bandwidth.

To address the second challenge, we leverage an online learning technique similar to *gradient descent* [8]. Online learning techniques are appealing for the context of this work, because they allow finding the shuffling interval that ensures stability even in the presence of irregular levels of churn. Moreover, unlike classic, offline machine-learning techniques [9], [10], online learning does not require any preliminary training phase nor suffers from overfitting.

In the following, we describe BuzzPSS in detail. We start by giving an overview of the gradient descent technique. Then, we show how BuzzPSS uses this method to automatically tune the length of the gossip period in order to improve bandwidth efficiency, without compromising reliability. Finally, we present an optimization to the base self-tuning algorithm, which leverages the stability of the system to further save network bandwidth.

A. Gradient Descent Overview

Gradient descent is a widely used algorithm for function approximation and is particularly well suited for exploration problems. In gradient descent methods, one wants to learn the vector of parameters θ that allow a given approximation function h_θ to better fit a data set. For each observed state of the system represented by s, a common strategy for the exploration of θ is to try to minimize the error between the approximate value given by $h_\theta(s)$ and the target value y.

Consider the following least-squares cost function that, for a given value of θ and state s, measures how close $h_\theta(s)$ is to the corresponding y:[1]

$$J(\theta) = \frac{1}{2}(y - h_\theta(s))^2 \tag{1}$$

The gradient descent algorithm thus consists in adjusting the estimation of θ after each state observation such that $J(\theta)$ becomes smaller and, hopefully, converges to the minimum. The update rule of θ at each iteration i is written as follows:

$$\theta_{i+1} = \theta_i - \alpha \nabla J_i(\theta) \tag{2}$$

where α is the so-called learning rate and $\nabla J_i(\theta)$ is the gradient of the cost function (*i.e.,* the vector of partial derivatives of J with respect to θ). This equation is also known as the Widrow-Hoff learning rule and allows the gradient descent algorithm to systematically take steps in the direction of the steepest decrease of J. Also, note that the magnitude of the update of θ is proportional to the error: for observations on which the prediction of $h_\theta(s)$ nearly matches the actual value of y, there is little need to change the parameters; in contrast, if the approximation value is far from the target value, θ will suffer a larger modification.

In the next section, we show how BUZZPSS applies the gradient descent method to dynamically tune the gossip period of a peer sampling service.

B. Using Gradient Descent Exploration

BUZZPSS employs gradient descent to adapt T such that the difference between the average ages of a node's view, observed in two consecutive periods, is minimized. Concretely, at iteration i, BUZZPSS adjusts the length of the gossip period T using the following update rule:

$$T_{i+1} = T_i - \alpha(\mathit{curAge} - \mathit{prevAge}) \tag{3}$$

where *curAge* and *prevAge* represent, respectively, the current average view age and the previous average view age of the node. Although other metrics could have been used, from our experiments the average view age offers good results as we will show further on. Notice that T is naturally decreased (increased) when the view's current average age is higher (lower) than the previous average age. This way, BUZZPSS is able to dynamically adapt the shuffling period to variations in the node population.

The magnitude of the adjustment is defined by the module of the error term $|\mathit{curAge} - \mathit{prevAge}|$, multiplied by the learning rate. Hence, for gradient descent to be effective, one must set the learning rate to an appropriate value, which can be challenging. Setting α too high may cause the algorithm to skip the global minimum of J. In turn, if α is too small, the algorithm will need many iterations to converge.

BUZZPSS addresses this issue by applying a technique called *bold driver*[11] to dynamically adapt the learning rate. Bold driver compares the error rate between consecutive iterations and, in case the error has reduced, the technique increases α by a small amount (typically 5%); otherwise, the technique decreases α drastically (typically by 50%). The rationale of bold driver is that the farther one is from the optimal value, the larger the step towards the solution should be (*i.e.,* the learning rate should be higher). On the other hand, the closer one is to the solution, the lower the learning rate should be, in order to avoid skipping the function minimum.

Algorithm 1 describes how BUZZPSS implements the aforementioned procedures. The algorithm starts by computing the current average view age (line 2) and uses Equation 3 to compute the new value of T (lines 3-4). BUZZPSS then checks whether the newly obtained period is higher than the minimum value allowed, and updates T accordingly (line 5).

Algorithm 1: BUZZPSS self-tuning configuration.

```
Input: learnRate  // initial learning rate for gradient descent
       T  // initial gossip period
       minT  // minimum value of T allowed
       view  // node's partial view

1  // adjust T using gradient descent
2  curAge ← getAverageAge(view)
3  error ← curAge − prevAge
4  tmpT ← T − learnRate × error
5  T ← max(minT, tmpT)

6  // apply bold driver to learning rate
7  if |error| ≤ prevError then
8      learnRate ← learnRate × 1.05  // increase learning rate by 5%
9  else
10     learnRate ← learnRate/2  // reduce learning rate by 50%
11 end
12 prevAge ← curAge
13 prevError ← error
```

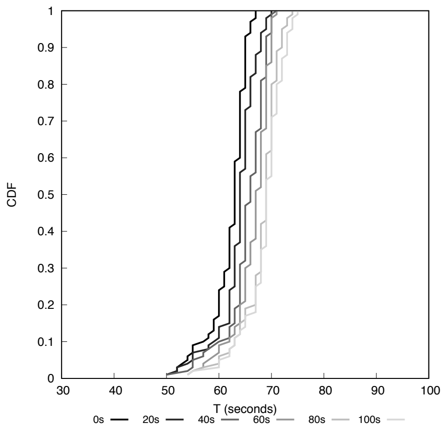

Fig. 2: Cumulative Distribution Function for T along a stable period of 100s and for 100 nodes.

[1]Since we are interested in performing online learning, our definition of the cost function $J(\theta)$ is only considering a single data point. This data point corresponds to the system state observed at each iteration.

Algorithm 2: BuzzPSS self-tuning configuration with stability reward.

```
   Input: learnRate // initial learning rate for gradient descent
          T // initial gossip period
          minT // minimum value of T allowed
          view // node's partial view
          L // error range within which the system is considered stable
          reward // reward value

1  // variable initialization
2  wpos ← 0
3  window[S] ← {True}

4  // in each iteration
5  curAge ← getAverageAge(view)
6  error ← curAge − prevAge
7  if |error| > L then
8      // the system is not stable: adjust T using gradient descent
9      window[wPos%S] ← False
10     tmpT ← T − learnRate × error
11     T ← max(minT, tmpT)

12     // apply bold driver to learning rate
13     if |error| ≤ prevError then
14         learnRate ← learnRate × 1.05 // increase learning rate by 5%
15     else
16         learnRate ← learnRate/2 // reduce learning rate by 50%
17     end
18     prevAge ← curAge
19     prevError ← error
20 else
21     // the system is stable
22     window[wPos%S] ← True
23     isStable ← True
24     for i ← 0; i < S; i++ do
25         isStable ← isStable ∧ window[i]
26     end
27     if isStable then
28         // reward stability by augmenting T
29         T ← T + reward
30     end
31 end
32 wPos ← wPos + 1
```

It is important to define a minimum interval length for T to prevent nodes for being constantly exchanging messages among themselves (which floods the network).

Afterwards, BuzzPSS uses the bold driver technique to adjust the learning rate (lines 7-11) and, finally, updates the values of *prevAge* and *prevError* for the subsequent iteration (lines 12-13). It should be noted that Algorithm 1 is performed by each node individually. Therefore, different nodes may exhibit different shuffling periods for the same iteration. However, in a stable system, all nodes are expected to eventually converge to similar values of T. This behavior can be observed in Figure 2, which depicts the CDF of T during a stable period of 100 seconds, for a system with 100 nodes. We used Algorithm 2 in the experiment and computed the value of T of each node at every 20-second interval. As expected, the results show that all nodes exhibit similar values of T, as the CDF contains short tails.

C. Improving Bandwidth Efficiency by Rewarding Stability

Algorithm 1 allows BuzzPSS to find the gossip period that yields a consistent average view age across iterations. However, the value of T to which gradient descent converges does not guarantee optimality in terms of bandwidth efficiency. In fact, for a fully stable system (where nodes never leave the network), T could in theory be set to infinity, as the connectivity would always be ensured. Naturally, in practice, this cannot be ensured and T needs a finite upper bound.

To further improve BuzzPSS's efficiency, we extend Algorithm 1 to leverage stability scenarios. The key idea is to augment T by a *reward* amount whenever the average view age does not exceed a given limit L for S consecutive iterations. Algorithm 2 describes the optimized version of BuzzPSS.

The algorithm resorts to a sliding window (implemented as a vector of size S) to check the stability of the system. All positions are initially set to *true* (lines 2-3) and, in each iteration, BuzzPSS updates the head of the window with a *true* or *false* value depending on whether the system is stable or not, respectively. BuzzPSS evaluates system's stability by checking whether the error between the previous and the current average view age falls within the error limit L (line 7). If the error is higher than L, then the system is considered unstable (line 9) and BuzzPSS should adjust T using gradient descent. In other words, BuzzPSS performs Algorithm 1 (lines 8-19). Otherwise, the system is considered stable for that iteration (line 22).

When BuzzPSS observes S consecutive stable iterations, it increases T by a *reward* amount, previously defined. Note that a single "unstable iteration" suffices for BuzzPSS to stop conceding rewards for, at least, the next S iterations. This design choice allows BuzzPSS to react quickly to changes in the view's age (typically caused by churn), thus ensuring robustness in the presence of catastrophic failures.

IV. Evaluation

Our evaluation of BuzzPSS focuses on answering the two following questions:

- **Dependability:** Does BuzzPSS maintain the overlay connected in the presence of variable levels of churn?
- **Efficiency:** Does BuzzPSS achieve significant resource usage savings?

We evaluated BuzzPSS by using the *Minha* framework [12]. Minha emulates various Java Virtual Machines on top of a single one, thus making it possible to run multiple hosts on a single machine. In our case, we conducted the experiments on an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (24 core) machine with 96GB of memory.

Simulations were performed for a population of 100 nodes with a view size of 8. Each experiment run took 2000 seconds, where churn was applied every 10 seconds during a given period of time.

Based on preliminary observations, the configuration of BuzzPSS is as follows: T was initially set to 5 seconds, *minT* to 2 seconds, *learnRate* to 1 second, *reward* to 5 seconds,[2]

[2]The *learnRate* was chosen in order to minimize convergence issues during the initial phase of the algorithm, whereas *reward* was chosen to rapidly improve efficiency in stable scenarios. We defer a more thorough sensitivity analysis of the initial parameters of the system to future work.

the error limit L to 2 (meaning that we consider the system to be stable if the variation in the average view age between iterations is less than 2), the sliding window size S to 3 (meaning that reward is given when the system is deemed stable for 3 consecutive iterations).

We evaluated BuzzPSS on the following scenarios.

- *Uniform Churn.* We performed three experiments, each corresponding to a different uniform level of churn, namely 2%, 5%, and 10%. For each experiment, churn was applied from second 250 to second 1500. Churn is implemented by removing a node and adding a fresh one. The added node is provided with an initial view of random nodes sampled from the system simulating a bootstrapping phase. After this bootstrap phase, each node tags its view nodes with an high age. By high we mean significantly higher than the expected system average. This implementation detail allows to accelerate the incorporation of these nodes in the system and to trigger the automated peer sampling service adaptation. Note that considering this high age for the initial view does not impact the original Cyclon protocol but allows BuzzPSS to quickly detect system instability.
- *Variable Churn.* We performed a single test where the churn level varies along time. In particular, during interval $[250s, 550s]$ we applied churn of 5%; for interval $[800s, 1100s[$ we applied churn of 10%; finally, for interval $[1100s, 1500s]$ we applied churn of 30%. Once again, churn is implemented by replacing a node for a fresh one. As a remark, notice that between the first two periods of churn, there was a hiatus of 250 seconds where the system remained stable. In contrast, between the latter two churn intervals, churn increased directly from 10% to 30%.
- *Catastrophic Failure.* We performed a single experiment where 50% of the system node crashes instantly. The goal of this scenario is to observe the behavior of BuzzPSS in the presence of a catastrophic failure, without other nodes joining the system.

For each scenario, we assessed the dependability and efficiency of BuzzPSS by, respectively, checking whether the overlay broke at the end of the experiment and measuring the length of the gossip period T at every 10-second interval. Additionally, we measured the message rate, which is computed as the mean number of messages exchanged per second in the entire network, and the average view size across all node views in the system.

In the following, we present the outcomes of our experiments and discuss the main findings.

A. Uniform Churn

Figure 3 depicts the gossip period and the message rate exhibited by BuzzPSS for the different scenarios of uniform churn (*i.e.,* 2%, 5%, and 10%). To assess the benefits of rewarding stability, we plot the results for both the baseline version of BuzzPSS, dubbed BuzzPSS$_{\text{GRAD}}$, which solely uses gradient descent to adapt the value of T (see Algorithm 1), and the extended version of BuzzPSS, dubbed BuzzPSS$_{\text{REW}}$, which increases T when the system is stable. In addition, we also illustrate in Figure 3 the gossip period for which the overlay breaks, denoted T_{Break}. The T_{Break} value was obtained through experiment and represents the minimum T for which we found the overlay would break. This means that static configurations with T higher than T_{Break}, despite incurring less bandwidth cost, will not ensure connectivity for the corresponding churn level.

Dependability. For all churn scenarios, both BuzzPSS$_{\text{GRAD}}$ and BuzzPSS$_{\text{REW}}$ were able to maintain the overlay connected during the experiment. Interestingly, Figure 3a shows that 2% churn did not have a significant negative impact on the network, as some nodes still considered the system to be stable. This caused BuzzPSS$_{\text{REW}}$ to reward stability for those cases (notice the slight increase of T for BuzzPSS$_{\text{REW}}$).

On the other hand, in Figure 3c, BuzzPSS$_{\text{REW}}$ even had a T above T_{Break} when the churn started, but our self-tuning mechanism was able to rapidly detect that some nodes left the network and adapted the shuffling period to accommodate for this fact. The average view age variation (Figures 3g-3h) reflects well BuzzPSS's self-tuning ability: while, for T_{Break}, the average age tends to increase when churn begins (which will eventually partition the overlay), both BuzzPSS schemes adapt to a state where the average age remains stable and the overlay robust. Note that BuzzPSS$_{\text{GRAD}}$ shows lower view age than BuzzPSS$_{\text{REW}}$ in Figure 3g because its T is shorter than that of the latter, which minimizes the impact of churn.

Another observation is that, as expected, the higher the churn, the quicker is the reduction of T (notice that the slope of the curve for BuzzPSS$_{\text{REW}}$ is steeper in Figure 3c than in Figure 3a). These results provide evidence to support our claim that BuzzPSS is effective in adapting T to cope with the churn level.

Efficiency. Analyzing the message rate of the two BuzzPSS algorithms, the data shows that BuzzPSS$_{\text{REW}}$ is clearly more efficient than BuzzPSS$_{\text{GRAD}}$. While the latter algorithm adjusts T until the average view age remains consistent across iterations, BuzzPSS$_{\text{REW}}$ further reduces the network bandwidth usage by attempting to increase the shuffling period when the system appears to be stable. For instance, at second 250, which corresponds to the end of initial period of stability, BuzzPSS$_{\text{REW}}$ presents a message rate 2.6x smaller than that of BuzzPSS$_{\text{GRAD}}$ (2.6 messages/s against 6.8 messages/s). At the end of the experiment, BuzzPSS$_{\text{REW}}$ achieves an improvement in efficiency with respect to BuzzPSS$_{\text{GRAD}}$ that is even more significant (up to 5.4x).

Finally, the comparison between BuzzPSS and T_{Break} shows that there is still some room for further reduction of network bandwidth, specially during the churn intervals. The reason is because BuzzPSS applies a conservative approach: the difference between the average view age between consecutive iterations must not exceed the allowed limit L, in order to prevent the overlay from breaking. This approach causes T to converge to a value that, while ensuring that the average age falls within the threshold, might not be the highest value that allows the overlay to remain connected. Finding such optimal

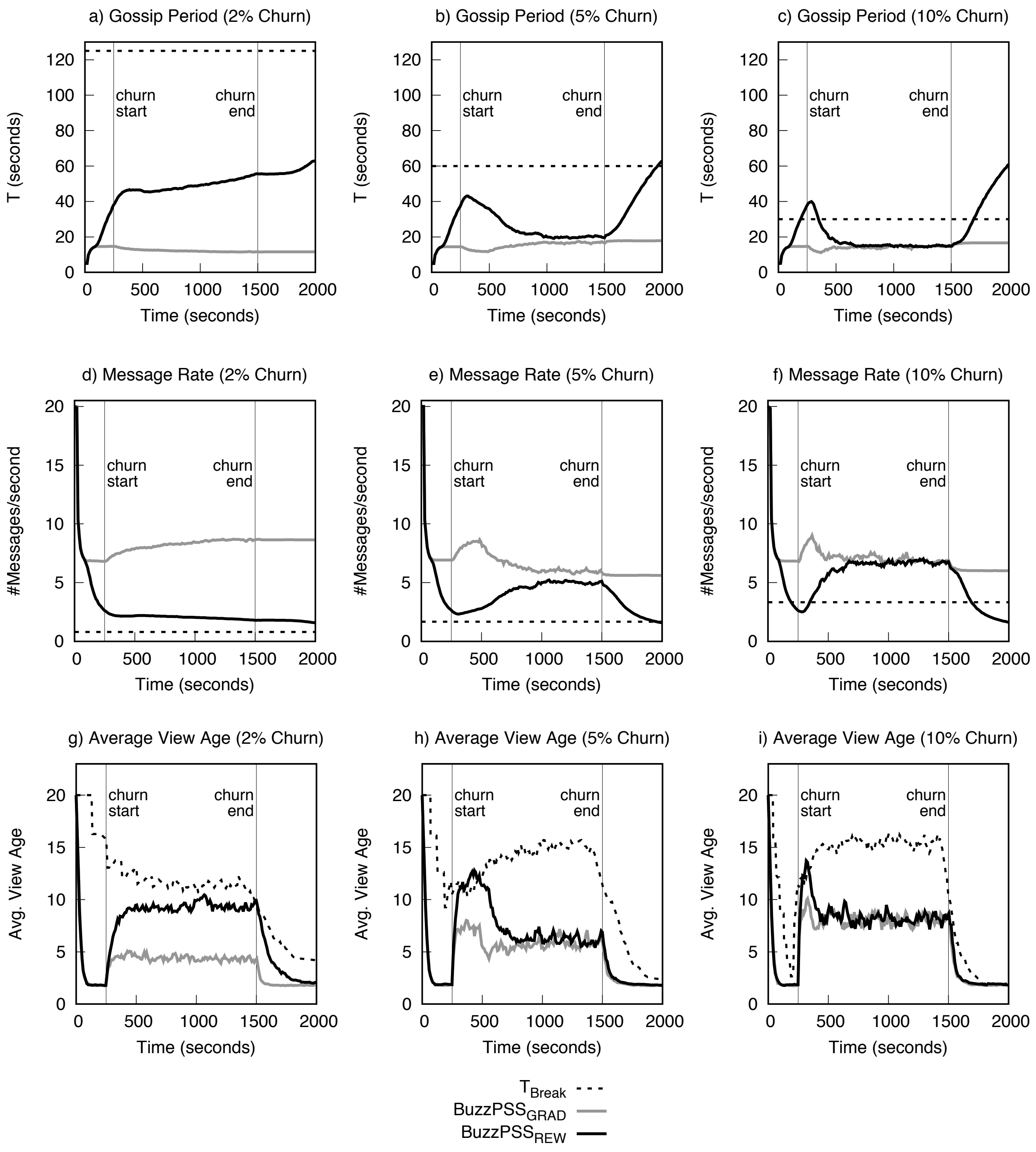

Fig. 3: *Uniform churn.* Gossip period (higher is better), message rate (lower is better), and average view age (lower is better) for three different uniform levels of churn: 2%, 5%, and 10%. BUZZPSS$_{\text{GRAD}}$ corresponds to Algorithm 1, BUZZPSS$_{\text{REW}}$ corresponds to Algorithm 2, and T_{Break} is the gossip period for which the overlay breaks. Both BUZZPSS versions yield a connected overlay for all scenarios.

value of T is not straightforward and we leave this research problem for future work.

B. Variable Churn

Figure 4 depicts the gossip period, the message rate, and the average view age exhibited by BUZZPSS for a scenario in which the churn level varies along time and interleaves with intervals of stability. Once again, we plot the results for BUZZPSS$_{\text{GRAD}}$, BUZZPSS$_{\text{REW}}$, and T_{Break}. In the following, we discuss the outcomes of the experiments in terms of dependability and efficiency.

Dependability. Both BUZZPSS$_{\text{GRAD}}$ and BUZZPSS$_{\text{REW}}$

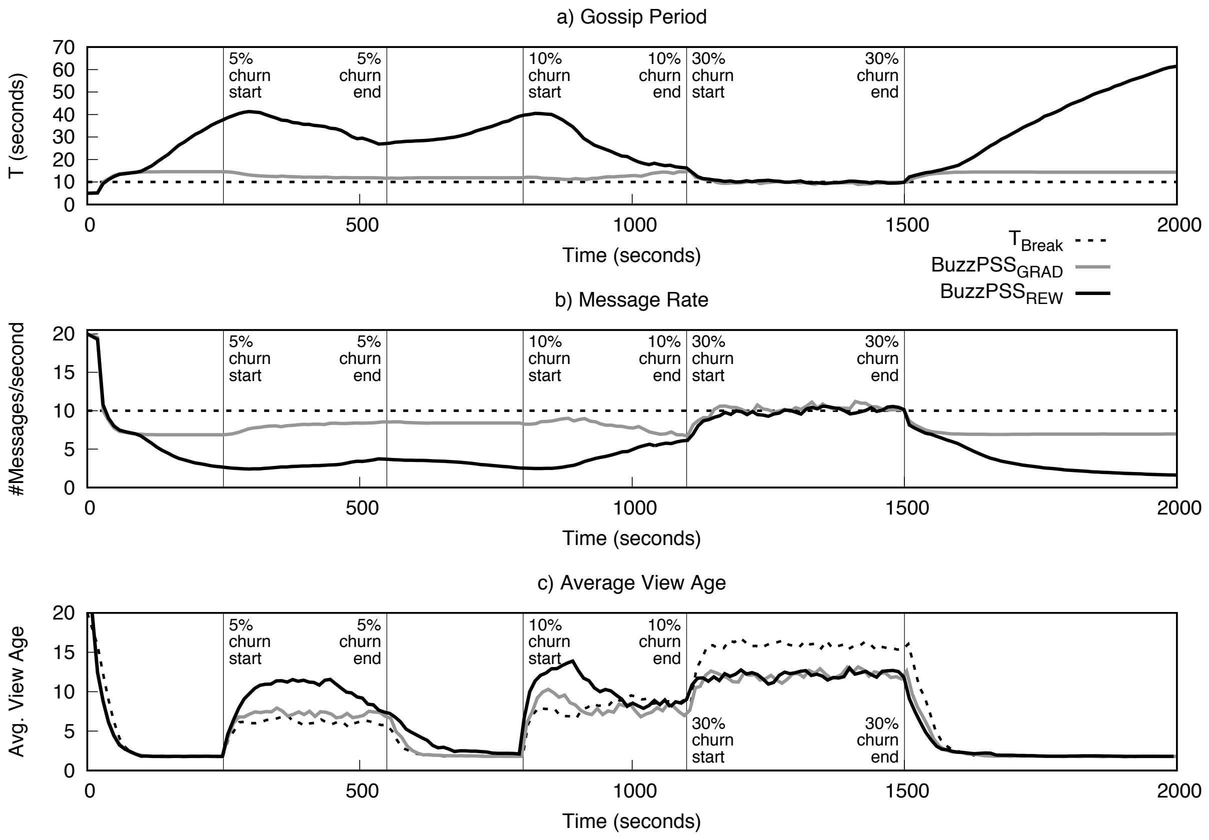

Fig. 4: *Variable churn.* Gossip period (higher is better), message rate (lower is better), and average view age (lower is better) for the scenario with variable churn. BUZZPSS$_{\text{GRAD}}$ corresponds to Algorithm 1, BUZZPSS$_{\text{REW}}$ corresponds to Algorithm 2, and T_{Break} is the gossip period for which the overlay breaks. Both BUZZPSS versions yield a connected overlay.

managed to effectively tune the gossip period and keep the overlay robust in the presence of the different levels of churn. Even when 30% of the nodes were replaced every 10 seconds, BUZZPSS provided a dependable peer sampling service. The average view age variation (plotted in Figure 4c) corroborates the effectiveness of the adjustment of the gossip period: the magnitude of the decrease of T during churn intervals always allowed the average view age to reach a value that suffices to cope with the node replacement rate.

Efficiency. As expected, Figure 4 shows once again that BUZZPSS$_{\text{REW}}$ consistently outperformed BUZZPSS$_{\text{GRAD}}$. However, the figure also shows a (somewhat) surprising outcome: using BUZZPSS$_{\text{REW}}$ resulted in much better network usage than T_{Break} (up to 6x less messages exchanged per iteration). The reason is because T_{Break} is set statically, hence, it must correspond to a value that accounts for the potential highest churn level supported (30% in our experiment). As a consequence, the system ends up exchanging much more messages than necessary during periods where the node population changes a smaller rates. In contrast, BUZZPSS$_{\text{REW}}$ is able to adjust the sampling period in real time according to the needs of the overlay, thus exchanging much less messages than static configurations.

C. Catastrophic Failure

The results for the catastrophic failure test are depicted in Figure 5. As it is observable, the massive departure of 50% of system nodes does not impact the behavior of the membership service. This is expected and desirable. In fact, even with 50% of nodes failing, the overlay maintains connectivity due to its randomness characteristics. Moreover, because nodes continue the view exchange process, failed nodes are quickly removed from node views not impacting the average view size (Figure 5c). BUZZPSS leverages such desirable properties and does not impact them negatively.

Dependability. With this result we show that BUZZPSS still guarantees dependability, *i.e.,* guarantees connectivity between all system nodes even when the system suffers from major failures. Naturally, like every other overlay network BUZZPSS cannot cope with a major failure that instantaneously breaks the overlay. If the overlay instantly breaks there is always the need for external intervention to repair it.

Efficiency. Finally, with respect to efficiency, BUZZPSS once again allows to save resources when using BUZZPSS$_{\text{REW}}$. The membership service detects that the system maintains its desirable characteristics and increases T leading to reduced message exchanges.

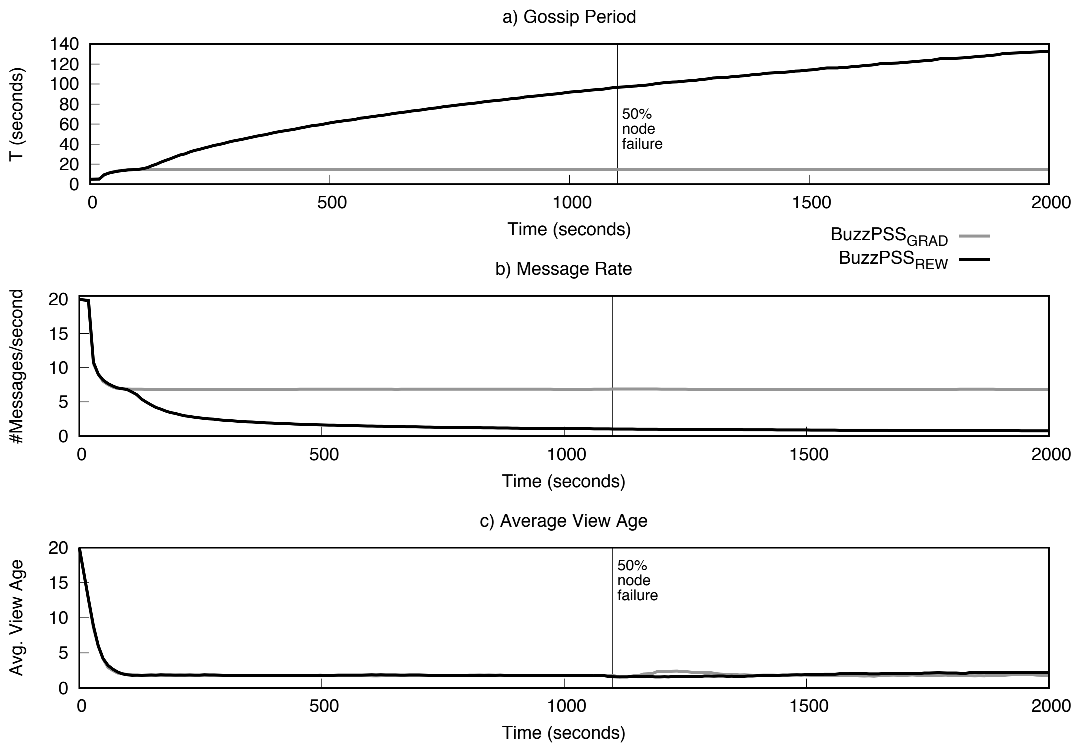

Fig. 5: *Catastrophic failure.* Gossip period (higher is better), message rate (lower is better), and average view age (lower is better) for the scenario with a catastrophic failure (*i.e.,* 50% of the nodes fail instantly). BUZZPSS$_{GRAD}$ corresponds to Algorithm 1 and BUZZPSS$_{REW}$ corresponds to Algorithm 2. Both BUZZPSS versions yield a connected overlay.

V. Related Work

There is an extensive body of work on PSS algorithms covering aspects such as topology-awareness [13], security [14] or slicing [15]. Several approaches explicitly maintain two overlays with different goals. For instance, HyParView [16] maintains an overlay with a very large view for robustness and a small one use exclusively for dissemination. Other approaches, such as Brisa [17], keep an unstructured overlay for robustness and build a sub-overlay organized as a tree. In this paper we focus exclusively on simpler techniques whose goal is to maintain the safety overlay connected even in adverse environments.

Generic overlays algorithms such as Cyclon [3] or NewsCast [18] are a good fit to act as safety overlays as their main design concern is robustness. For this class of algorithms, a wide range of trade-offs can be obtained simply by varying few parameters, as has been shown in [2]. For instance, within the same implementation one can tune the overlay to have a better degree distribution (*i.e.,* close to a uniform distribution) or to be very fast at removing references to failed nodes.

However, to the best of our knowledge, few proposals study the tuning of the period parameter as a mechanism to provide new trade-offs. One notable exception is [5], which tunes the period parameter of NewsCast. The period is adjusted in two cases: to overcome the effects of message loss and to provide better samples to an application that requires more samples per period than the ones available in the view. Another example is [19], which proposes an adaptive mechanism to tune the rate of message emission in heterogenous systems. The key idea of this approach is to disseminate and collect information about the resources available in the nodes of the view such that every node can adjust its emission rate according to the amount of resources available to the neighbor nodes and to the global level of congestion in the system. As we focus explicitly on a safety-net overlay, our goal is instead to reduce resource usage to the minimum while still ensuring connectivity.

Regarding online learning using gradient descent, this approach has been successfully applied in prior work to design adaptive systems for other purposes rather than the one presented in this work. For instance, Tuner [20] exploits gradient descent and reinforcement learning to adaptively determine the retry-on-abort policy for hardware transactional memory. In the context of data analysis, [21] uses gradient descent to automatically tune the parameters of spectral clustering. The work in [22] also resorts to this online learning technique to adjust the parameters of fuzzy logic controllers.

VI. Conclusion

In this paper we presented BUZZPSS, an adaptable and dependable peer sampling service. The main goal of BUZZPSS is to offer a reliable overlay network on which a distributed system can rely as a safety net. In detail, BUZZPSS is intended to maintain connectivity between system nodes at all times by ensuring that there is always a logical path between every pair of nodes. Additionally, and as the key contribution of the paper, BUZZPSS is intended to dynamically adapt its resource usage according to the system stability level.

BUZZPSS is built on top of Cyclon, a gossip-based peer sampling service. Cyclon is a proactive membership service that maintains a random overlay network across system nodes. The intuition behind this approach is twofold. First, random overlay networks are known to be highly resilient [7], [6]. Second, proactive approaches to fault tolerance are suitable for scenarios of high instability. This follows from the fact that proactive approaches take the initiative and continuously try to repair the system.

Unfortunately, these approaches are resource consuming and, during periods of system stability, even wasteful. Unlike traditional proactive approaches, BUZZPSS is fully autonomous and relies on online learning techniques to infer the stability of the system and adjust the proactive mechanism action rate accordingly.

We evaluate BUZZPSS on a number of different scenarios and levels of churn. The results of our experiments show that our approach is effective and can lead to significant resource usage savings.

Nevertheless, there are some open research challenges that we intend to address in the future. First, the current version of BUZZPSS needs to be parameterized with a maximum value for T. Without this upper bound, a long period of stability may lead BUZZPSS to increase T to a value that will impede the system to detect a sudden period of instability, thus compromising the dependability of the overlay. To address this issue, we plan to devise a mechanism to automate the configuration of the maximum value of T.

Finally, in this paper, we have considered scenarios of churn where nodes leave and are replaced by new ones, assuming negligible periods of bootstrapping. Experiments with a larger number of nodes, different bootstrapping periods, additional churn patterns, and variable network latency in a non-simulated setting can lead to additional challenges to BUZZPSS, which we plan to address in the future.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. The research leading to these results has received funding from different sources. Francisco Maia was partially supported by Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). The remainder authors received funding from ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project (POCI-01-0145-FEDER-006961), and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013.

REFERENCES

[1] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," *ACM/IFIP/USENIX International Conference on Middleware*, pp. 79–98, 2004.

[2] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

[3] S. Voulgaris, D. Gavidia, and M. van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[4] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," *International COST264 Workshop on Networked Group Communication*, pp. 44–55, 2001.

[5] N. Tölgyesi and M. Jelasity, "Adaptive peer sampling with newscast," in *International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Springer-Verlag, 2009, pp. 523–534.

[6] P. Erdős and A. Rényi, "On the evolution of random graphs," *Selected Papers of Alfréd Rényi, vol*, vol. 2, pp. 482–525, 1976.

[7] P. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulié, "From epidemics to distributed computing," *Computer*, 2004.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[9] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[10] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[11] R. Battiti, "Accelerated backpropagation learning: Two optimization methods," *Complex Systems*, vol. 3, no. 4, pp. 331–342, Aug. 1989.

[12] N. A. Carvalho, J. Bordalo, F. Campos, and J. Pereira, "Experimental evaluation of distributed middleware with a virtualized java environment," in *Workshop on Middleware for Service Oriented Computing*, ser. MW4SOC '11. ACM, 2011, pp. 3:1–3:7.

[13] A. Ganesh, A.-M. Kermarrec, and L. Massoulié, "HiScamp: self-organizing hierarchical membership protocol," in *ACM SIGOPS European Workshop*. ACM, 2002, pp. 133–139.

[14] G. P. Jesi, A. Montresor, and M. van Steen, "Secure peer sampling," *Comput. Netw.*, vol. 54, no. 12, pp. 2086–2098, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2010.03.020

[15] F. Maia, M. Matos, R. Oliveira, and E. Rivire, "Slicing as a distributed systems primitive," in *Sixth Latin-American Symposium on Dependable Computing*, ser. LADC '13, April 2013, pp. 124–133.

[16] J. Leitão, J. Pereira, and L. Rodrigues, "HyParView: A membership protocol for reliable gossip-based broadcast," in *International Conference on Dependable Systems and Networks*, ser. DSN '07. IEEE Computer Society, 2007, pp. 419–428.

[17] M. Matos, V. Schiavoni, P. Felber, R. Oliveira, and E. Rivière, "Lightweight, efficient, robust epidemic dissemination," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 987–999, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731513000269

[18] M. Jelasity, W. Kowalczyk, and M. V. Steen, "Newscast Computing," *Science*, vol. 266, no. IR-CS-006, pp. 1–24, 2003. [Online]. Available: http://www.cs.vu.nl/pub/papers/globe/IR-CS-006.03.pdf

[19] L. Rodrigues, J. Pereira, S. Handurukande, R. Guerraoui, and A. Kermarrec, "Adaptive gossip-based broadcast," in *International Conference on Dependable Systems and Networks*, ser. DSN '03. IEEE Computer Society, 2003, pp. 47–56.

[20] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *International Conference on Autonomic Computing*, ser. ICAC '14. USENIX Association, Jun. 2014, pp. 209–219.

[21] L. Zelnik-manor and P. Perona, "Self-tuning spectral clustering," in *Advances in Neural Information Processing Systems 17*. MIT Press, 2004, pp. 1601–1608.

[22] A. Habbi and M. Zelmat, "An improved self-tuning mechanism of fuzzy control by gradient descent method," in *European Simulation Multiconference*, ser. ESM '03. MIT Press, 2003, pp. 43–47.