Adaptive Peer Sampling with Newscast⋆

Norbert Tölgyesi[1] and Márk Jelasity[2]

[1] University of Szeged and Thot-Soft 2002 Kft., Hungary, `ntolgyesi@gmail.com`
[2] University of Szeged and Hungarian Academy of Sciences, Hungary, `jelasity@inf.u-szeged.hu`

Abstract. The peer sampling service is a middleware service that provides random samples from a large decentralized network to support gossip-based applications such as multicast, data aggregation and overlay topology management. Lightweight gossip-based implementations of the peer sampling service have been shown to provide good quality random sampling while also being extremely robust to many failure scenarios, including node churn and catastrophic failure. We identify two problems with these approaches. The first problem is related to message drop failures: if a node experiences a higher-than-average message drop rate then the probability of sampling this node in the network will decrease. The second problem is that the application layer at different nodes might request random samples at very different rates which can result in very poor random sampling especially at nodes with high request rates. We propose solutions for both problems. We focus on Newscast, a robust implementation of the peer sampling service. Our solution is based on simple extensions of the protocol and an adaptive self-control mechanism for its parameters, namely—without involving failure detectors—nodes passively monitor local protocol events using them as feedback for a local control loop for self-tuning the protocol parameters. The proposed solution is evaluated by simulation experiments.

1 Introduction

In large and dynamic networks many protocols and applications require that the participating nodes be able to obtain random samples from the entire network. Perhaps the best-known examples are gossip protocols [1], where nodes have to periodically exchange information with random peers. Other P2P protocols that also require random samples regularly include several approaches to aggregation [2, 3] and creating overlay networks [4–7], to name just a few.

One possibility for obtaining random samples is to maintain a complete membership list at each node and draw samples from that list. However, in dynamic networks this approach is not feasible. Several approaches have been proposed to implementing peer sampling without complete membership information, for example, based on random walks [8] or gossip [9–11].

Algorithm 1 Newscast

```
 1: loop                                      12: procedure ONUPDATE(m)
 2:    wait(Δ)                                 13:    buffer ← merge(view,{myDescriptor})
 3:    p ← getRandomPeer()                     14:    send updateResponse(buffer) to m.sender
 4:    buffer ← merge(view,{myDescriptor})     15:    buffer ← merge(view,m.buffer)
 5:    send update(buffer) to p                16:    view ← selectView(buffer)
 6: end loop                                   17: end procedure
 7:
 8: procedure ONUPDATERESPONSE(m)
 9:    buffer ← merge(view,m.buffer)
10:    view ← selectView(buffer)
11: end procedure
```

Gossip-based solutions are attractive due to their low overhead and extreme fault tolerance [12]. They tolerate severe failure scenarios such as partitioning, catastrophic node failures, churn, and so on. At the same time, they provide good quality random samples.

However, known gossip-based peer sampling protocols implicitly assume the *uniformity* of the environment, for example, message drop failure and the rate at which the application requests random samples are implicitly assumed to follow the same statistical model at each node. As we demonstrate in this paper, if these assumptions are violated, gossip based peer sampling can suffer serious performance degradation. Similar issues have been addressed in connection with aggregation [13].

Our contribution is that, besides drawing attention to these problems, we propose solutions that are based on the idea that system-wide adaptation can be implemented as an aggregate effect of simple adaptive behavior at the local level. The solutions for the two problems related to message drop failures and application load both involve a local control loop at the nodes. The local decisions are based on passively observing the local events and do *not* involve explicit failure detectors, or reliable measurements. This feature helps to preserve the key advantages of gossip protocols: simplicity and robustness.

2 Peer Sampling with Newscast

In this section we present a variant of gossip-based peer sampling called Newscast (see Algorithm 1). This protocol is an instance of the protocol scheme presented in [12], tuned for maximal self-healing capabilities in node failure scenarios. Here, for simplicity, we present the protocol without referring to the general scheme.

Our system model assumes a set of nodes that can send messages to each other. To send a message, a node only needs to know the address of the target node. Messages can be delayed by a limited amount of time or dropped. Each node has a partial view of the system (view for short) that contains a constant number of node descriptors. The maximal size of the view is denoted by c. A node descriptor contains a node address that can be used to send messages to the node, and a timestamp.

The basic idea is that all the nodes exchange their views periodically, and keep only the most up-to-date descriptors of the union of the two views locally. In addition, every time a node sends its view (update message) it also includes an up-to-date descriptor of itself.

Parameter Δ is the period of communication common to all nodes. Method getRandomPeer simply returns a random element from the current view. Method merge first merges the two lists it is given as parameters, keeping only the most up-to-date descriptor for each node. Method selectView selects the most up-to-date c descriptors.

Applications that run on a node can request random peer addresses from the entire network through an API that Newscast provides locally at that node. The key element of the API is method getRandomPeer. Though in a practical implementation this method is not necessarily identical to the method getRandomPeer that is used by Newscast internally (for example, a tabu list may be used), in this paper we assume that the application is simply given a random element from the current view.

Lastly, we note that although this simple version of Newscast assumes that the clocks of the nodes are synchronized, this requirement can easily be relaxed. For example, nodes could adjust timestamps based on exchanging current local times in the update messages, or using hop-count instead of timestamps (although the variant that uses hop-count is not completely identical to the timestamp-based variant).

3   Problem Statement

We identify two potential problems with Newscast noting that these problems are common to all gossip-based peer sampling implementations in [12]. The first problem is related to message drop failures, and the second is related to unbalanced application load.

3.1   Message Drop Failure

Gossip-based peer sampling protocols are designed to be implemented using lightweight UDP communication. They tolerate message drop well, as long as each node has the same failure model. This assumption, however, is unlikely to hold in general. For example, when a lot of failures occur due to overloaded routers that are close to a given node in the Internet topology, then UDP packet loss rate can be higher than average at the node. In this case, fewer copies of the descriptor of the node will be present in the views of other nodes in the network violating the uniform randomness requirement of peer sampling.

Figure 1 illustrates this problem in a network where there is no failure, only at a single node. This node experiences a varying drop rate, which is identical for both incoming and outgoing messages. The Figure shows the representation of the node in the network as a function of drop rate. Note the non-linearity of the relation. (For details on experimental methodology see Section 4.3.)
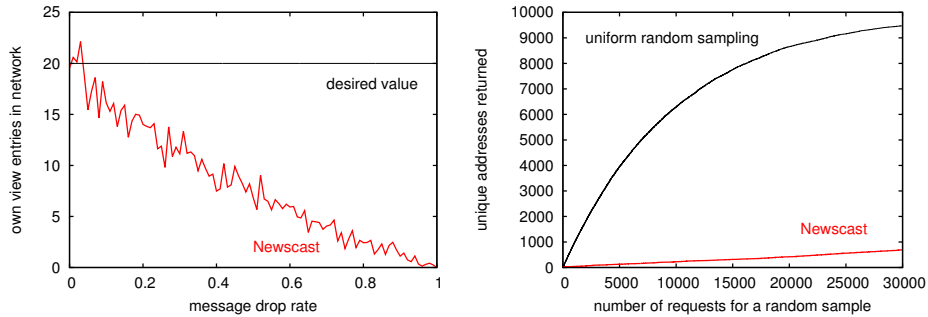
Fig. 1. Illustrations of two problems with gossip-based peer sampling. Parameters: $c = 20$, $N = 10{,}000$, the number of own entries in network is the average of 100 consecutive cycles.

3.2 Unbalanced Application Load

At different nodes, the application layer can request random samples at varying rates. Gossip-based peer sampling performs rather poorly if the required number of samples is much higher than the view size c over a time period of Δ as nodes participate in only two view exchanges on average during that time (one initiated and one passive). Figure 1 illustrates the problem. It shows the number of unique samples provided by Newscast as a function of time at a node where the application requests random samples at a uniform rate with a period of $\Delta/1000$.

4 Message Drop Failure

4.1 Algorithm

We would like to adapt the period Δ of Newscast (Algorithm 1) at each node in such a way that the representation of each node in the network is identical, irrespective of message drop rates. This means that ideally all the nodes should have c copies of their descriptor since the nodes hold Nc entries altogether, where N is the number of nodes.

We make the assumption that for each overlay link (i, j) there is a constant drop rate $\lambda_{i,j}$, and all the messages passing from i to j are dropped with a probability of $\lambda_{i,j}$. According to [14] this is a reasonable assumption. For the purpose of performing simulation experiments we will introduce a more specific structural model in Section 4.2.

The basic idea is that all the nodes passively monitor local messages and based on this information they decide whether they are under- or overrepresented in the network. Depending on the result, they slightly decrease or increase their period in each cycle, within the interval $[\Delta_{min}, \Delta_{max}]$.

Let us now elaborate on the details. All the nodes collect statistics about the incoming and outgoing message types in a moving time window. The length of this time window is Δ_{stat}. The statistics of interest (along with notations) are the following (note that we have omitted the node id from the notations for the sake of clarity): the number

of update messages sent (u_{out}), the number of update messages received (u_{in}), and the number of update response messages received (r_{in}).

From these statistics, a node approximates its representation in the network (n). Clearly, n is closely related to u_{in}. The exact relationship that holds for the expectation of u_{in} is

$$E(u_{in}) = \frac{1}{c} P_{in} n \Delta_{stat} \phi_{avg},\qquad(1)$$

where c is the view size, P_{in} is the probability that a message sent from a random node is received, and ϕ_{avg} is the average frequency of the nodes in the network at which they send update messages.

The values for this equation are known except P_{in} and ϕ_{avg}. Note however, that ϕ_{avg} is the same for all nodes. In addition, ϕ_{avg} is quite close to $1/\Delta_{max}$ if most of the nodes have close to zero drop rates, which is actually the case in real networks [14]. For these two reasons we shall assume from now on that $\phi_{avg} = 1/\Delta_{max}$. We validate this assumption by performing extensive experiments (see Section 4.3).

The only remaining value to approximate is P_{in}. Here we focus on the symmetric case, when $\lambda_{i,j} = \lambda_{j,i}$. It is easy to see that here $E(r_{in}) = P_{in}^2 u_{out}$, since all the links have the same drop rate in both directions, and to get a response, the update message first has to reach its destination and the response has to be delivered as well. This gives us the approximation $P_{in} \approx \sqrt{r_{in}/u_{out}}$.

Once we have an approximation for P_{in} and calculate n, each node can apply the following control rule after sending an update:

$$\Delta(t+1) = \begin{cases} \Delta(t) - \alpha + \beta & \text{if } n < c \\ \Delta(t) + \alpha + \beta & \text{if } n > 2c \\ \Delta(t) + \alpha(n/c - 1) + \beta & \text{otherwise,} \end{cases}\qquad(2)$$

where we also bound Δ by the interval $[\Delta_{min}, \Delta_{max}]$. Parameters α and β are positive constants; α controls the maximal amount of change that is allowed in one step towards achieving the desired representation c, and β is a term that is included for stability: it always pulls the period towards Δ_{max}. This stabilization is necessary because otherwise the dynamics of the system will be scale invariant: without β, for a setting of periods where nodes have equal representation, they would also have an equal representation if we multiplied each period by an arbitrary factor. It is required that $\beta \ll \alpha$.

Although we do not evaluate the asymmetric message drop scenario in this paper (when $\lambda_{i,j} \neq \lambda_{j,i}$), we briefly outline a possible solution for this general case. As in the symmetric case, what we need is a method to approximate P_{in}. To achieve this we need to introduce an additional trick into Newscast: let the nodes send R independent copies of each update response message ($R > 1$). Only the first copy needs to be a full message, the remaining ones could be simple ping-like messages. In addition, the copies need not be sent at the same time, they can be sent over a period of time, for example, over one cycle. Based on these ping messages we can use the approximation $P_{in} \approx r_{in}/(R n_{in})$, where n_{in} is the number of *different* nodes that sent an update response. In other words, we can directly approximate P_{in} by explicitly sampling the distribution of the incoming drop rate from random nodes.
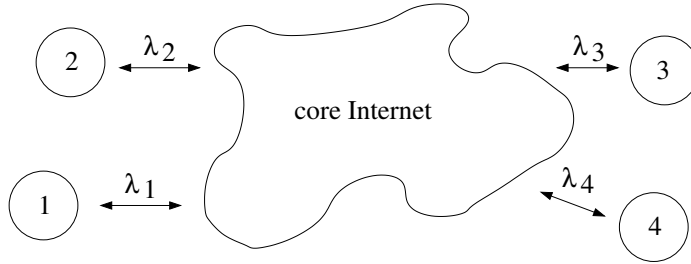
Fig. 2. The topological structure of the message drop failure model.

This solution increases the number of messages sent by a factor of R. However, recalling that the message complexity of gossip-based peer sampling is approximately one UDP packet per node during any time period Δ, where Δ is typically around 10 seconds, this is still negligible compared to most current networking applications.

4.2 Message Drop Failure Model

In our model we need to capture the possibility that nodes may have different message drop rates, as discussed in Section 3.1. The structure of our failure model is illustrated in Figure 2. The basic idea is that all the nodes have a link to the core Internet that captures their local environment. The core is assumed to have unbiased failure rates; that is, we assume that for any given two links, the path that crosses the core has an identical failure model. In other words, we simply assume that node-specific differences in failure rates are due to effects that are close to the node in the network. We define the drop rate of a link (i, j) by $\lambda_{i,j} = \lambda_i \lambda_j$.

We assume that each message is dropped with a probability independent of previous message drop events. This is a quite reasonable assumption as it is known that packet loss events have negligible autocorrelation if the time lag is over 1000 ms [14].

It should be mentioned that the algorithm we evaluate does *not* rely on this model for correctness. This model merely allows us (i) to control the failure rate at the node level instead of the link level and (ii) to work with a compact representation.

As for the actual distributions, we evaluate three different scenarios:

single-drop As a baseline, we consider the model where any node i has $\lambda_i = 0$, except a single node that has a non-zero drop-rate.
uniform Drop rate λ_i is drawn from the interval $[0, 0.5]$ uniformly at random.
exponential Based on data presented in [14] we approximate $100\lambda_i$ (the drop rate expressed as the percentage of dropped messages) by an exponential distribution with parameter $1/4$, that is, $P(100\lambda_i < x) = 1 - e^{-0.25x}$. The average drop rate is thus 4%, and $P(100\lambda_i > 20\%) \approx 0.007$.

4.3 Evaluation

In order to carry out simulation experiments, we implemented the algorithm over the event-based engine of PeerSim [15]. The parameter space we explored contains every
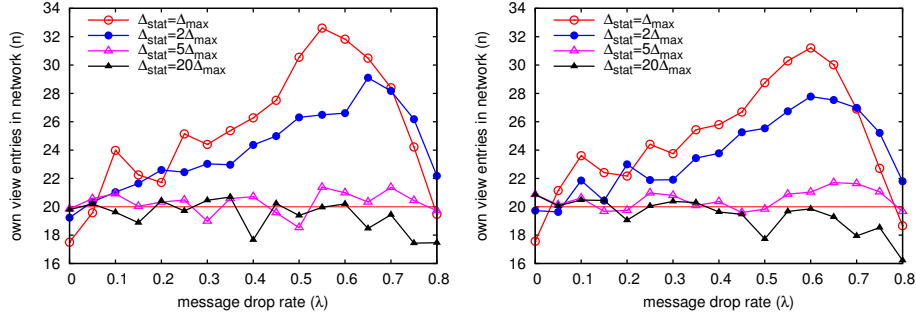
Fig. 3. The effect of Δ_{stat} on n. The fixed parameters are $N = 5,000$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution. Message delay is zero (left) or uniform random from $[0, \Delta_{max}/10]$ (right).

combination of the following settings: the drop rate distribution is single-drop, uniform, or exponential; α is $\Delta_{max}/1000$ or $\Delta_{max}/100$; β is 0, $\alpha/2$, or $\alpha/10$; and Δ_{stat} is Δ_{max}, $2\Delta_{max}$, $5\Delta_{max}$, or $20\Delta_{max}$. If not otherwise stated, we set a network size of $N = 5000$ and simulated no message delay. However, we explored the effects of message delay and network size over a subset of the parameter space, as we explain later. Finally, we set $c = 20$ and $\Delta_{min} = \Delta_{max}/10$ in all experiments.

The experiments were run for 5000 cycles (that is, for a period of $5000\Delta_{max}$ time units), with all the views initialized with random nodes. During a run, we observed the dynamic parameters of a subset of nodes with different failure rates. For a given failure rate, all the plots we present show average values at a single node of the given failure rate over the last 1500 cycles, except Figure 6 that illustrates the dynamics (convergence and variance) of these values.

Let us first consider the role of Δ_{stat} (see Figure 3). We see that small values introduce a bias towards nodes with high drop rates. The reason for this is that with a small window it often happens that no events are observed due to the high drop rate, which results in a maximal decrease in Δ in accordance with the control rule in (2). We fix the value of $20\Delta_{max}$ from now on.

In Figure 3 we also notice that the protocol tolerates delays very well, just like the original version of Newscast. For parameter settings that are not shown, delay has no noticeable effect either. This is due to the fact that we apply no failure detectors explicitly, but base our control rule just on passive observations of average event rates that are not affected by delay.

Figure 4 illustrates the effect of coefficients α and β. The strongest effect is that the highest value of β introduces a bias against nodes with high drop rates. This is because a high β strongly pushes the period towards its maximum value, while nodes with high drop rates need a short period to get enough representation.

In general, the smaller value for α is more stable than the higher value for all values of β, which is not surprising. However, setting a value that is too small slows down convergence considerably. Hence we will set $\alpha = \Delta_{max}/1000$ and $\beta = \alpha/10$.
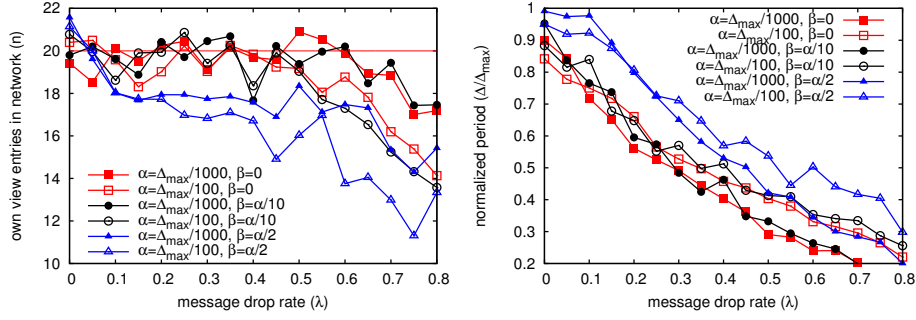
Fig. 4. The effect of α and β on n. The fixed parameters are $N = 5,000$, $\Delta_{stat} = 20\Delta_{max}$, exponential drop rate distribution.
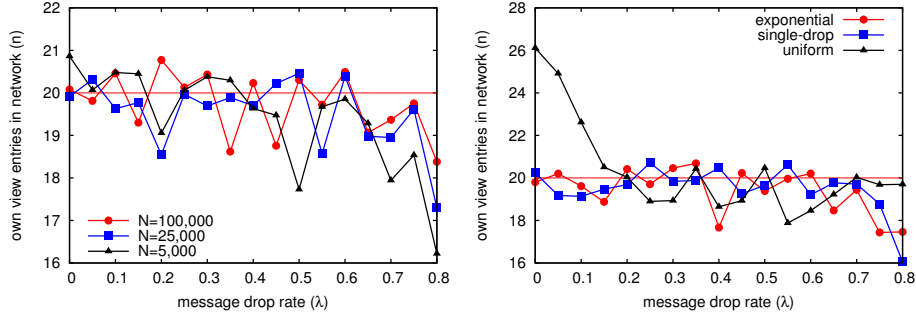

Fig. 5. The effect of network size and drop rate distribution on n. The fixed parameters are $N = 5,000$ (right), $\Delta_{stat} = 20\Delta_{max}$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution (left).

In Figure 5 we see that the algorithm with the recommended parameters produces a stable result irrespective of network size or message drop rate distribution. There is a notable exception: the uniform distribution, where nodes with very small drop rates get slightly overrepresented. To see why this happens, recall that in the algorithm we made the simplifying assumption that the average update frequency is $1/\Delta_{max}$. This assumption is violated in the uniform model, where the average drop rate is very high (0.25) which noticeably increases the average update frequency.

However, in practice, due to the skewed distribution, the average drop rate is small. Second, in special environments where the average rate is high, one can approximate the average rate using suitable techniques (for example, see [3]).

Finally, in Figure 6 we show the dynamics of Δ and n with the recommended parameter settings. We observe that convergence requires approximately 500 cycles after which the periods of the nodes fluctuate in a bounded interval. The Figure also shows n as a function of time. Here we see that the variance is large. However, most importantly, it is not larger than that of the original Newscast protocol where this level of variance is normal [12].
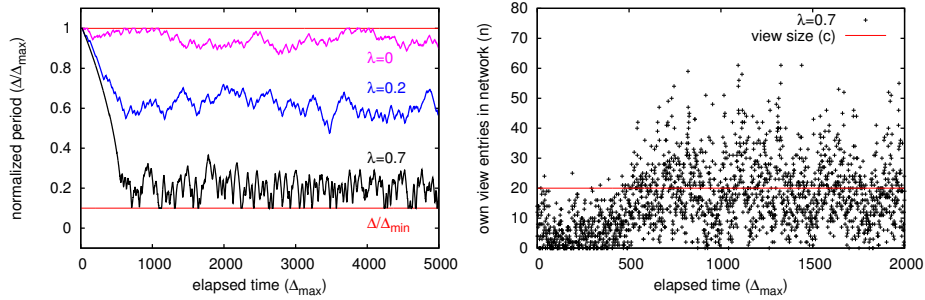
Fig. 6. The dynamics of Δ and n for $N = 5{,}000$, $\Delta_{stat} = 20\Delta_{max}$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution.

5 Unbalanced Application Load

5.1 Algorithm

If the application at a node requests many random samples, then the node should communicate faster to refresh its view more often. Nevertheless we should mention that it is *not* a good solution to simply speed up Algorithm 1 locally. This is because in this case a fast node would inject itself into the network more often, quickly getting a disproportionate representation in the network. To counter this, we need to keep the frequency of update messages unchanged and we need to introduce extra shuffle messages without injecting new information.

To further increase the diversity of the peers to be selected, we apply a tabu list as well. Algorithm 2 implements these ideas (we show only the differences from Algorithm 1).

Shuffle messages are induced by the application when it calls the API of the peer sampling service; that is, procedure getRandomPeer: the node sends a shuffle message after every S random peer requests. In a practical implementation one might want to set a minimal waiting time between sending two shuffle messages. In this case, if the application requests random peers too often, then it will experience a lower quality of

---

Algorithm 2 Extending Newscast with on-demand shuffling

1: **procedure** GETRANDOMPEER	11: **procedure** ONSHUFFLEUPDATERESPONSE(m)
2: $p \leftarrow$ findFreshPeer()	12: buffer $\leftarrow m$.buffer
3: tabuList.add(p)	13: counter $\leftarrow 0$
4: counter $\leftarrow$ counter+1	14: **end procedure**
5: **if** counter=S **then**	15:
6: counter $\leftarrow 0$	16: **procedure** ONSHUFFLEUPDATE(m)
7: send shuffleUpdate(view) to p	17: (buffer$_1$,buffer$_2$) $\leftarrow$ shuffle(view,m.buffer)
8: **end if**	18: send shuffleUpdateResp(buffer$_1$) to m.sender
9: **return** p	19: buffer $\leftarrow$ buffer$_2$
10: **end procedure**	20: **end procedure**

service (that is, a lower degree of randomness) if we decide to simply not send the shuffle message; or a delayed service if we decide to delay the shuffle message.

We should add that the idea of shuffling is not new, the Cyclon protocol for peer sampling is based entirely on shuffling [10]. However, Cyclon itself shares the same problem concerning non-uniform application load; here the emphasis is on adaptively applying *extra* shuffle messages where the sender does not advertise itself.

The tabu list is a FIFO list of fixed maximal size. Procedure findFreshPeer first attempts to pick a node address from the view that is not in the tabu list. If each node in the view is in the tabu list, a random member of the view is returned.

Note that the counter is reset when an incoming shuffle message arrives. This is done so as to avoid sending shuffle requests if the view has been refreshed during the waiting period of S sample requests.

Finally, procedure shuffle takes the two views and for each position it randomly decides whether to exchange the elements in that position; that is, no elements are removed and no copies are created.

5.2 Evaluation

Like in Section 4.3, we ran event-driven simulations over PeerSim. Messages were delayed using a uniform random delay between 0 and a given maximal delay.

In all the experiments, we worked with a scenario where peer sampling requests are maximally unbalanced: we assumed that the application requests samples at a high rate on one node, and no samples are requested on the other nodes. This is our worst case scenario, because there is only one node that is actively initiating shuffle requests. The other nodes are passive and therefore can be expected to refresh their own view less often.

The experimental results are shown in Figure 7. The parameters we examine are S, the size of the tabu list, network size (N) and the maximal delay. The values of the maximal delay go up to 0.3Δ, which is already an unrealistically long delay if we consider that practical values of Δ are high, around 10 seconds or more. In fact, in the present adaptive version Δ can be much higher since randomness is provided by the shuffling mechanism.

First of all, we notice that for many parameter settings we can get a sample diversity that is almost indistinguishable from true random sampling, especially when S and the maximal delay are relatively small. For $S = 2$ and $S = 4$ the returned samples are still fairly diverse, which permits one to reduce the number of extra messages by a factor of 2 or even 4. The tabu list is "free" in terms of network load, so we can set high values, although beyond a certain point having higher values appears to make no difference.

The algorithm is somewhat sensitive to extreme delay, especially during the initial sample requests. This effect is due to the increased *variance* of message arrival times, since the number of messages is unchanged. Due to variance, there may be large intervals when no shuffle responses arrive. This effect could be alleviated via queuing the incoming shuffle responses and applying them in equal intervals or when the application requests a sample.
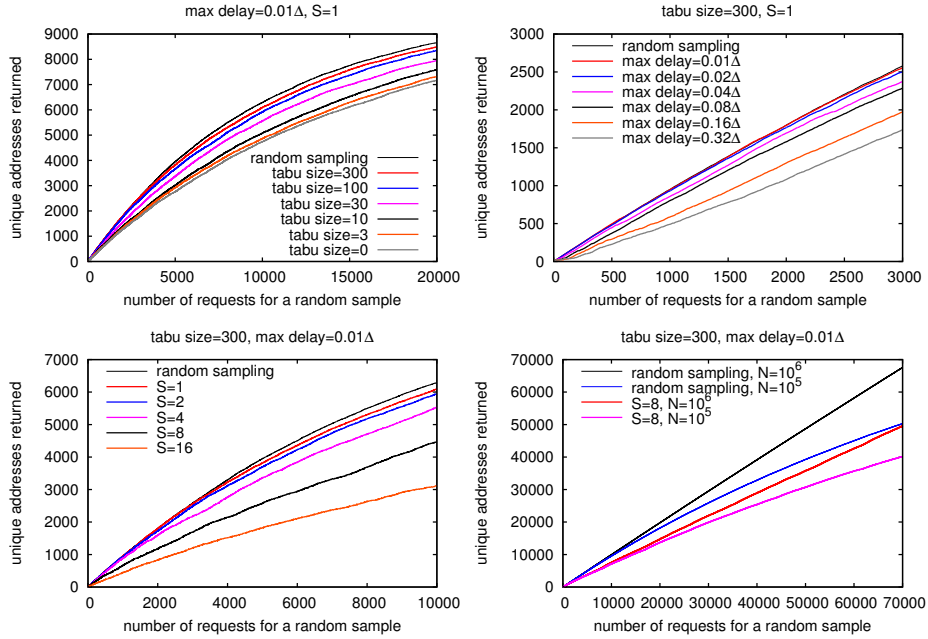
Fig. 7. Experimental results with adaptive shuffling for $N = 10^4$ if not otherwise indicated. In B/W print, lines are shown in the same line-type: the order of keys matches the order of curves from top to bottom.

Since large networks are very expensive to simulate, we will use just one parameter setting for $N = 10^5$ and $N = 10^6$. In this case we observe that for large networks randomness is in fact slightly better, so the method scales well.

6 Conclusions

In this paper we identified two cases where the non-uniformity of the environment can result in a serious performance degradation of gossip-based peer sampling protocols, namely non-uniform message drop rates and an unbalanced application load.

Concentrating on Newscast we offered solutions to these problems based on a statistical approach, as opposed to relatively heavy-weight reliable measurements, reliable transport, or failure detectors. Nodes simply observe the local events and based on that they modify the local parameters. As a result, the system converges to a state that can handle non-uniformity.

The solutions are cheap: in the case of symmetric message drop rates we require no extra control messages at all. In the case of application load, only the local node has to initiate extra exchanges proportional to the application request rate; but for any sampling protocol that maintains only a constant-size state this is a minimal requirement.

References

1. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), Vancouver, British Columbia, Canada, ACM Press (August 1987) 1–12

2. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03), IEEE Computer Society (2003) 482–491

3. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems **23**(3) (August 2005) 219–252

4. Bonnet, F., Kermarrec, A.M., Raynal, M.: Small-world networks: From theoretical bounds to practical systems. In: Principles of Distributed Systems. Volume 4878., Springer (2007) 372–385

5. Patel, J.A., Gupta, I., Contractor, N.: JetStream: Achieving predictable gossip dissemination by leveraging social network principles. In: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications (NCA 2006), Cambridge, MA, USA (July 2006) 32–39

6. Voulgaris, S., van Steen, M.: Epidemic-style management of semantic overlays for content-based searching. In Cunha, J.C., Medeiros, P.D., eds.: Proceedings of Euro-Par. Number 3648 in Lecture Notes in Computer Science, Springer (2005) 1143–1152

7. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F., eds.: Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers. Volume 3910 of Lecture Notes in Computer Science., Springer-Verlag (2006) 1–15

8. Zhong, M., Shen, K., Seiferas, J.: Non-uniform random membership management in peer-to-peer networks. In: Proc. of the IEEE INFOCOM, Miami, FL (2005)

9. Allavena, A., Demers, A., Hopcroft, J.E.: Correctness of a gossip based membership protocol. In: Proceedings of the 24th annual ACM symposium on principles of distributed computing (PODC'05), Las Vegas, Nevada, USA, ACM Press (2005)

10. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Journal of Network and Systems Management **13**(2) (June 2005) 197–217

11. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kermarrec, A.M., Kouznetsov, P.: Lightweight probabilistic broadcast. ACM Transactions on Computer Systems **21**(4) (2003) 341–374

12. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. ACM Transactions on Computer Systems **25**(3) (August 2007) 8

13. Yalagandula, P., Dahlin, M.: Shruti: A self-tuning hierarchical aggregation system. In: IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO), Los Alamitos, CA, USA, IEEE Computer Society (2007) 141–150

14. Zhang, Y., Duffield, N.: On the constancy of Internet path properties. In: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW '01), New York, NY, USA, ACM (2001) 197–211

15. PeerSim: http://peersim.sourceforge.net/