Peer-to-Peer Netw Appl (2009) 2:24–36
DOI 10.1007/s12083-008-0017-3

# Peer sampling with improved accuracy

**Elth Ogston · Stephen A. Jarvis**

Received: 14 April 2008 / Accepted: 8 October 2008 / Published online: 6 November 2008
© Springer Science + Business Media, LLC 2008

**Abstract** Node sampling services provide peers in a peer-to-peer system with a source of randomly chosen addresses of other nodes. Ideally, samples should be independent and uniform. The restrictions of a distributed environment, however, introduce various dependancies between samples. We review gossip-based sampling protocols proposed in previous work, and identify sources of inaccuracy. These include replicating the items from which samples are drawn, and imprecise management of the process of refreshing items. Based on this analysis, we propose a new protocol, Eddy, which aims to minimize temporal and spatial dependancies between samples. We demonstrate, through extensive simulation experiments, that these changes lead to an improved sampling service. Eddy maintains a balanced distribution of items representing active system nodes, even in the face of realistic levels of message loss and node churn. As a result, it behaves more like a centralized random number generator than previous protocols. We demonstrate this by showing that using Eddy improves the accuracy of a simple algorithm that uses random samples to estimate the size of a peer-to-peer network.

**Keywords** Node sampling · Peer-to-peer · Gossiping · Performance

E. Ogston (✉) · S. A. Jarvis
Department of Computer Science,
University of Warwick, Coventry, CV4 7AL, UK
e-mail: elth@dcs.warwick.ac.uk

S. A. Jarvis
e-mail: Stephen.Jarvis@dcs.warwick.ac.uk

## 1 Introduction

In distributed systems, directory services provide components with a means of locating each other. In peer-to-peer systems, which aim to decentralize operations, maintaining a complete directory of nodes is undesirably costly. A *node sampling service* is a degenerate form of directory, which instead of giving the address of a requested node, simply returns the address of a randomly selected node. While minimal, sampling services are central to many fundamental peer-to-peer operations. Statistical estimates of the value of global variables, or their distributions, can be made using sampling [11]. This provides a cheap method of performing operations like counting the number of nodes in the system [2, 13]. Sampling can also be used to define overlay networks that approximate random graphs. Such graphs are well connected and have a low diameter, making them an ideal basis for low-maintenance information dissemination algorithms [10, 12]. Finally, sampling can be used to create more structured overlays, by first examining a few random configurations to build a quick approximation of the desired structure, then improving the structure over time by continuing to test variations [16, 18].

Many of the operations built on top of node sampling services assume that samples are independent of each other and are drawn uniformly at random from the set of all system nodes [4, 9]. These properties are fairly easy to maintain in a centralized sampling service. In distributed versions, however, limitations on communication between locations create a tendency for samples at a given location and time to favor certain nodes. This problem arises because samples are drawn from a set of items representing the nodes, and this item set

must be distributed between locations. The problem is compounded by the presence of node and message failures. Failures are a source of non-determinism, which complicates the task of maintaining the item set.

In this paper we examine the problem of creating uniform samples of nodes in a peer-to-peer system using a distributed process. We review previous node sampling protocols [1, 3–5, 8, 9, 17, 19] and identify sources of non-uniformity and dependance between items drawn. We argue that these protocols can be improved if a stronger focus is placed on avoiding temporal and spatial dependancies between items and between items and their locations. We present a new protocol, Eddy, based on these observations. We demonstrate through simulations that Eddy does a better job than previous protocols of ensuring that all nodes are represented equally in the global item-set, and that it improves the accuracy of a simple application, estimating the network size, built on top of the sampling protocol. Understanding and removing sources of inaccuracy in node sampling is a first step towards improving the general performance characteristics of peer-to-peer sampling protocols.

In the following sections we outline the basic principles of gossip-based node sampling (Section 2) then discuss sources of error in sampling accuracy present in many protocols (Section 3). In Section 4 we present details of a new protocol, Eddy, which removes the avoidable sources of error identified in Section 3. Section 5 gives experimental results showing that Eddy improves sampling accuracy over previous protocols. These results extend work reported in [15], providing extensive comparisons to previous protocols under realistic operating conditions.

## 2 Fundamentals of peer-to-peer sampling services

In its most basic form, a node sampling service for a peer-to-peer system maintains a pool of *items*, representing each of the nodes. Samples are drawn from this pool. For a generic random sampling process there are thus two main factors that affect the distribution of the items returned: the relative number of items for each node in the pool, and the method of choosing items out of the pool. In a distributed sampling service, the pool of items is divided among a number of locations. Samples for a given node are taken from the local portion of the pool. This adds a third factor affecting sample distributions, the distribution of items among the locations.

The effect of location can be removed by placing one copy of each item in each location [3]. In large systems, however, such an arrangement is impractical, especially in the fully decentralized case where each node is a separate location. Instead, locations can be given a small number of items, and a mixing process can be used to move items between locations [5]. Ideally, each item should follow an independent random walk among the locations, so that at a given point in time, each item is in a random location. If mixing is uniform and faster than the sampling rate, a sampler's location will become invisible. In practice perfect independent uniform mixing is difficult to achieve, but a close approximation is usually sufficient.

Decentralizing the sampling process, however, also complicates the problem of maintaining the item pool so that all nodes are represented equally. When a node joins the system, it can create and insert the correct number of items for itself. When a node is removed, however, its items, which could be at any location, must also be removed. Further, the presence of failures means that items can be lost. When a node fails, the items in its local portion of the pool are removed from the system. When a message transmitting items between nodes fails, those items are also lost.

Distributed sampling protocols therefore contain two essential elements, a mechanism to randomly mix items, and a mechanism that rebalances the distribution of items in the item pool after nodes are removed or failures occur.

## 3 Sources of inaccuracy in gossip-based sampling protocols

Decentralized node sampling services are realized by means of gossiping protocols. A wide variety of such protocols have been proposed, including: lpbcast [5], Clearinghouse [1, 3], PROOFS [17], Newscast [8], Cyclon [19], AARG [4], and the Peer Sampling Framework [9]. In this section we discuss sources of inaccuracy in these protocols. Section 4.2 discusses how protocols can be improved to increase sampling accuracy.

Figure 1 gives pseudo-code for the central gossip procedure in AARG, an example of a minimal gossip protocol. Nodes each maintain a “local view” cache that contains a small set of $C$ items. Each item stores the address of a node participating in the protocol. To implement the mixing process, each node periodically “gossips” with a neighbor, chosen from among its items (line 3). In each gossip a node sends some of its items to a neighbor, which responds by sending some items back again. In order to handle failures or changes to the node set, items are refreshed as part of the gossip exchange.

```
1  void doGossip( ) {
2      //select neighbor
3      neighbor = cache.selectRandomItem();
4
5      //select items to send
6      sendItems = cache.selectRandomItems(GOSSIP_SIZE);
7      sendItems.add(new Item(self));
8
9      //do request, wait for reply
10     req = sendRequest(neighbor,sendItems);
11     replyItems = receiveReply(req);
12
13     //update cache
14     cache.add(replyItems);
15     while(cache.size() > C) {
16         cache.removeRandomItem();
17     }
18 }
19
20 Item[] handleGossipRequest(Item[] sendItems){
21     //select items to send back
22     replyItems = cache.selectRandomItems(GOSSIP_SIZE);
23     replyItems.add(new Item(self));
24
25     //update cache
26     cache.add(sendItems);
27     while(cache.size() > C) {
28         cache.removeRandomItem();
29     }
30
31     return replyItems;
32 }
```

**Fig. 1** A minimal gossip-based sampling protocol

Each time it gossips, a node adds a new item for itself (lines 7 and 23). These new items replace existing items, which are removed at the end of each gossip (lines 16 and 28).

At first glance, the exact details of the implementation of gossip-based sampling make little difference to overall system behavior. From the point of view of a single node, most protocols are good random number generators [9]. From the global system perspective, however, small differences in protocol operations can have significant effects on temporal and spatial dependancies between samples [4, 8, 9, 19].

Temporal dependancies between samples occur when the global item pool contains different numbers of items for each node at a particular point in time. Such differences are usually masked by the fact that which nodes are over or under represented varies. Nonetheless, an uneven representation of nodes can lead to less than optimal load balancing in applications that depend on the randomness of samples to assign simultaneous tasks. For instance, in the sampling protocol itself, it can cause nodes to be unevenly chosen as gossip partners. Protocols introduce temporal dependancies in two main ways: by replicating items and by not deleting a node's items at the same rate as they are refreshed.

Some sampling protocols introduce temporal dependancies by deliberately replicating items, causing others to be overwritten. For instance, in the exchange process instead of making an exact exchange of items, both nodes can choose to keep the newest items [8, 19] or random items [1, 4] or select random items, favoring newer ones [9]. Local optimizations, such as not allowing two items for the same node, or a node's own items in a cache [8, 9, 17, 19] similarly cause items to be deleted. The further restriction that caches must be kept full then causes others to be replicated to take their place. Finally, if items are left in the cache of a node ($p_1$) while waiting for an exchange reply from another node ($p_2$), an incoming request from a third node ($p_3$) can replicate items that are sent by $p_1$ to both $p_2$ and $p_3$ [1, 8, 9, 17, 19].

The process of refreshing items is another common source of temporal dependancies between items. In most protocols, new items are inserted at a constant rate. For instance, a new item for a node is added in each gossip request [4, 8, 9, 17, 19]. Item deletion, on the other hand, is only loosely managed. Often, the exact time an item is removed is left up to chance, again as part of the exchange process. When nodes keep exchanged items at random, older items are likely to be deleted eventually [1, 4, 17]. Recording how long an item has been in the system allows this process to be sped up by favoring the deletion of older items [8, 9, 19]. While these mechanisms delete items from all nodes at a constant rate, the time it takes to delete an item from a particular node can vary widely.

Spatial dependancies between items occur when protocols introduce a coupling between two items, or between an item and a particular location. Such dependancies are undesirable because they increase the chance that the virtual network defined by the protocol will partition. Should this happen, samples will not mix through the entire network. At any point in time, a sampling protocol can be viewed as defining a graph connecting nodes through directed links which go from the peer holding an item to the peer represented by the item. In general, the random placement of items means that protocols maintain highly-connected graphs [1, 6]. Spatial dependancies, however, can cause portions of the graph to become disconnected [4, 8]. While such partitions are highly unlikely, they are also highly undesirable because they cannot be recovered from without outside intervention, and they are often not detectable to the individual nodes.

Protocols introduce spatial dependancies in two main ways: by replicating items, and by inserting new

items in nodes that are neighbors of the nodes the items represent. When items are replicated in any of the ways discussed above, the two replicas are placed in neighboring nodes. Over time the mixing process causes them to move apart. However, when replication rates are high compared to the mixing rate, the fact that replicas are overwriting other items reduces the variety of items available in the local area. A similar effect occurs when new items for a node are inserted in its immediate neighbors. When a node gives an item representing itself to a neighbor during a gossip exchange [1, 4, 8, 9, 17, 19] it greatly increases the possibility of that neighbor choosing it as one of its next gossip partners. This increase the chance that two nodes will gossip with each other twice in a row, and thus the chance that an item will visit the same node twice in quick succession.

The inaccuracies discussed above can be minimized, Section 4 shows how. There are a number of further sources of dependancies between samples, however, which are more difficult to avoid. These include failures, the effect of location on sampling with replacement, and limited network connectivity. In the case of failures, when a node does not receive a reply to a gossip it can either assume that the request was not received, or that the reply message failed [4]. If the request was lost, the node should place the items it sent back into its cache to avoid them being deleted. If the reply was lost, however, doing this results in those items being replicated. The two options of leaving items in the local cache after they have been selected as samples or removing them to prevent them being selected twice also both bias the sampling process. Globally, sampling is done assuming replacement, but the limited size and turnover of the local cache means that replaced items have a good chance of being drawn twice in succession. Limited connectivity biases the random mixing process by limiting the degrees of freedom an item has at each step in its random walk. A further bias is introduced by nodes exchanging more than one item at a time, thus coupling items. Sections 4.2 and 4.3 discuss how protocols can be designed or protocol parameters can be set to reduce the consequences of these effects.

## 4 Eddy: an accurate sampling service

Eddy is a gossip-based peer sampling protocol that takes into account the factors that result in sampling inaccuracy, discussed in Section 3. The principle difference from previous protocols is that a focuses is placed on maintaining the invariant that all nodes are represented by an equal number of items, $C$. To do this, the process of refreshing items is kept independent from the item mixing process. In this section we present the Eddy protocol (Section 4.1), then follow with discussions of the measures taken to improve sampling accuracy (Section 4.2) and the factors influencing settings for protocol parameters (Section 4.3).

### 4.1 The Eddy protocol

*Adding nodes* Figure 2 gives pseudo-code for the procedure for adding new nodes. To join an existing sampling service, a node only needs to know the address of some other currently participating node. During a join a node adds $C$ items representing itself, and obtains $C$ existing items to fill its local cache. A node, $p_i$, joins the system by contacting a participating node, $p_j$ (line 3). $p_j$ provides $p_i$ with the current contents of its entire local view and the global parameter $C$ (line 19). $p_i$ sends join requests to each of the $C$ nodes represented by items in this view (line 11). These nodes pass this request on to a random neighbor (line 26). The receiving nodes respond to the join request by adding a new item, representing $p_i$, to their view and sending a random item from their view back to $p_i$ to be added to $p_i$'s view

```
1  void initialize(Item member){
2      //ask participating node for its knowledge about the system
3      req = sendInitializationRequest(member);
4      info = receiveReply(req);
5
6      C = info.getC();
7      cache = new Cache();
8
9      //insert own items, get random items
10     for(Item i : info.allItems()){
11         req = sendJoinRequest(i, new Item(self), 1);
12         replyItem = receiveReply(req);
13         cache.add(replyItem);
14     }
15 }
16
17 Info handleInitializationRequest() {
18     //return cache contents and system parameters
19     return new Info(cache.allItems(), C);
20 }
21
22 Item handleJoinRequest(Item sendItem, int hopCount){
23     if(hopCount > 0){
24         //forward request
25         neighbor = cache.selectRandomItem();
26         forwardJoinRequest(neighbor, sendItem, hopCount− 1);
27         return null;
28     }else{
29         //keep item, send another in reply
30         replyItem = cache.removeRandomItem();
31         cache.add(sendItem);
32         return replyItem;
33     }
34 }
```

**Fig. 2** Adding a new node

```
1  void doGossip( ) {
2      //select neighbor
3      neighbor = cache.selectNewRandomItem();
4
5      //select items to send (excluding neighbor)
6      sendItems = cache.removeRandomItemsExcluding(GOSSIP_SIZE,
               neighbor);
7
8      //do request, wait for reply
9      req = sendRequest(neighbor, sendItems);
10     replyItems = receiveReply(req);
11
12     //update cache
13     cache.add(replyItems);
14 }
15
16 Item[] handleGossipRequest(Item[] sendItems) {
17     //select items to send back
18     replyItems = cache.removeRandomItems(sendItems.size());
19
20     //if the cache did not have enough items, return some of those sent
21     replyItems.add(sendItems.removeRandomItems(sendItems.size() −
               replyItems.size()));
22
23     //update cache
24     cache.add(sendItems);
25
26     return replyItems;
27 }
```

**Fig. 3** Gossiping

(lines 30–32). Given that the caches of nodes already in the system contain random items, $p_i$ therefore inserts its items at random locations, and fills its own cache with random items[1]. To initialize a new sampling service, the initial node sets the value of $C$ and fills its cache with its own items.

*Gossiping* Figure 3 gives pseudo-code for the basic gossiping procedure. Nodes that are participating in the sampling protocol occasionally update their local view by exchanging some items with another node. Periodically, with interval $t$, a node, $p_i$ will send a gossip request to another node, $p_j$. $p_j$ is chosen by drawing an item uniformly at random from those in $p_i$'s local view, with the exception that any items that have already been drawn as gossip partners since their arrival in the cache will not be chosen a second time (line 3). The gossip request contains $g$ items which are removed from $p_i$'s local view (line 6) and added to $p_j$'s view (line 24). In return, $p_j$ sends a gossip reply containing $g$ items removed from its view (line 18). If $p_j$ does not have enough items in its cache, it returns some of those sent by $p_i$ (line 21). The returned items are added to $p_i$'s view (line 13). Items to exchange are chosen uniformly at random out of the nodes' local views, with the exception that $p_i$ will exclude the selected item representing $p_j$ from its choice (line 6).

*Refreshing items* Rather than attempting to remove items when nodes leave the system, or replace items that are lost due to failures, items are given a limited lifetime, $l$, after which they are removed and replaced with fresh items. Figure 4 gives pseudo-code for this item refresh procedure. When a node inserts an item it tags it with an expiry time (line 3). Nodes remove any expired items from their caches (line 9). At the time the item expires, the owner node inserts a new item. A node, $p_1$, inserts a new item at a random location by sending it to a neighbor, $p_2$, chosen randomly from its item cache, with the exception that any items that have already been drawn as insertion targets since their arrival in the cache will not be chosen a second time (line 15). $p_2$ forwards the item to a neighbor chosen randomly out of its own cache, $p_3$ (line 23). $p_3$ adds the item to its cache (line 26). This procedure assumes that nodes' clocks are roughly synchronized, for instance using the method given in Iwanicki et al. [7].

*Balancing cache sizes* The item refresh process does not insert new items at the same location where expired items are removed. As a result, the number of items in a node's cache will vary from the average size, $C$. Since the location of insertions and deletions is random, over time a node will have an average cache size of $C$. For practical reasons, however, it is desirable to

```
1  Item(Address owner) {
2      address = owner;
3      expiryTime = currentTime() + ITEM_LIFESPAN;
4  }
5
6  void expireItems(){
7      for(Item i : cache){
8          if(i.expiryTime > currentTime()){
9              cache.remove(i);
10         }
11     }
12 }
13
14 void insertItem(){
15     neighbor = cache.selectNewRandomItem();
16     sendInsertRequest(neighbor, new Item(self), 1);
17 }
18
19 void handleInsertRequest(Item sendItem, int hopCount){
20     if(hopCount > 0){
21         //forward request
22         neighbor = cache.selectRandomItem();
23         forwardInsertRequest(neighbor, sendItem, hopCount−1);
24     }else {
25         //keep item
26         cache.add(sendItem);
27     }
28 }
```

**Fig. 4** Refreshing items

[1]To simplify the pseudo-code the assumption is made that $p_j$ has exactly $C$ items in its cache (line 10). If the cache is larger, only $C$ join requests should be sent. If the cache is smaller, $p_i$ can fill its remaining cache slots with its own items

```
1   void doGossip( ) {
2       //select neighbor
3       neighbor = cache.selectNewRandomItem();
4
5       //select items to send (excluding neighbor)
6       sendItems = cache.removeRandomItemsExcluding(
                GOSSIP_SIZE, neighbor);
7
8       //do request, wait for reply
9       req = sendRequest(neighbor, sendItems, cache.size());
10      replyItems = receiveReply(req);
11
12      //update cache
13      cache.add(replyItems);
14  }
15
16  Item[] handleGossipRequest(Item[] sendItems, int
            neighborCacheSize) {
17      //determine number of items to send back
18      replySize = sendItems.size();
19      diff = neighborCacheSize − cache.size();
20      if( diff ≥ DELTA ){
21          replySize−−;
22      }else if(diff − −DELTA){
23          replySize++;
24      }
25
26      //select items to send back
27      replyItems = cache.removeRandomItems(replySize);
28
29      //if the cache did not have enough items, return some of those sent
30      replyItems.add(sendItems.removeRandomItems(replySize −
                replyItems.size()));
31
32      //update cache
33      cache.add(sendItems);
34
35      return replyItems;
36  }
```

**Fig. 5** Gossiping with cache balancing

maintain tighter bounds on the number items in the cache. This can be done by adding an item redistribution mechanism. We use a simple approach that moves items when nodes have cache sizes that vary by more than a given bound, $\delta$. Figure 5 gives pseudo-code for balancing caches as part of the gossiping procedure. Each node, $p_i$, includes its current cache size, $c_i$, in its gossip requests (line 9).When the receiving node, $p_j$, has a much smaller cache, $(c_j + \delta) \leq c_i$, it returns one less item in the gossip reply (line 21). Similarity, if $(c_i + \delta) \leq c_j$, $p_j$ returns one extra item (line 23).

*Message failures* Figure 6 gives pseudo-code for timing out gossip and join requests. Requests time out if a reply is not received within the gossip interval, $t$. If a node does not receive a reply message, it will assume that either the request message, the reply message, or the receiving node has failed. Items that were sent are put back into its cache. If a reply is delayed and received later, it is ignored. This can result in extra copies of items being created, and other items being lost. These mistakes will eventually be corrected by the refresh procedure.

```
1   void timeoutGossipRequest(Item[] sendItems) {
2       cache.add(sendItems);
3   }
4
5   void timeoutJoinRequest(Item sendItem) {
6       cache.add(sendItem);
7   }
```

**Fig. 6** Timeout of gossip and join requests

*Node failures* When a node fails the items that represent it become invalid and the items it holds in its cache are lost. The refresh procedure will eventually remove the failed node's items from the system, and replace valid items that were lost.

*Node removal* Figure 7 gives pseudo-code for removing a node. If a node anticipates that it will be removed from the system it can adjust the expiry times of its items accordingly (line 5), and use the insertion mechanism from the refresh procedure to place items in its cache back into the system before exiting (line 12).

### 4.2 Measures taken to improve sampling accuracy

The Eddy protocol includes a number of measures to avoid the sources of sampling inaccuracy discussed in Section 3. The protocol attempts to maintain the invariant that all nodes are represented by an equal number of items, $C$. This minimizes temporal dependancies, in which some nodes are more likely than others to be drawn at a given point in time. The main change required over previous protocols is that the process of refreshing items is managed using precise expiry times (Fig. 4). This decouples the item refresh process from the gossiping process. The correct number of items for a node are created when the node joins the system (Fig. 2). The gossiping process only exchanges items between nodes, it does not delete or replicate items (Fig. 3) as is often possible in other protocols (Fig. 1). Additionally, when a node exits the system an effort is

```
1   Item(Address owner) {
2       address = owner;
3       expiryTime = currentTime() + ITEM_LIFESPAN;
4       if(exitTime() < expiryTime){
5           expiryTime = exitTime();
6       }
7   }
8
9   void exit() {
10      for(Item i: cache.allItems()){
11          neighbor = cache.selectRandomItem();
12          sendInsertRequest(neighbor, i, 0);
13      }
14  }
```

**Fig. 7** Node removal

made to remove its items and keep the items it holds in its cache (Fig. 7).

Spatial dependancies occur when an item is coupled with a particular location, or two items are coupled with each other. Items are naturally coupled with the node they represent. Extra hops are added when inserting items when new nodes join (Fig. 2 line 26) or when items are refreshed (Fig. 4 line 23) to reduce this effect. Additionally, the item chosen as the gossiping partner cannot be sent as part of the gossip (Fig. 3 line 6). If a node chooses the same gossip partner twice in quick succession, it can have some of the items it sent returned. Choosing gossip and insertion partners without replacement from the cache reduces this possibility (Fig. 3 line 3 and Fig. 4 line 15).

Replicating items introduces both temporal and spatial dependancies. The only times item replication occurs in the Eddy protocol is after a failed reply message causes a gossip request or insertion request to time out (Fig. 6). A request will time out if the request message fails, the receiving node fails, or the reply message fails. If the request message or receiving node fail, the timeout procedures that replace items given in Fig. 6 should be used. If the reply message fails, nothing should be done. Without employing extra mechanisms, however, there is no way for a node to tell the difference between these cases. In this work we assume that all messages are equally likely to fail. This makes the case when items should be replaced more prevalent. If items are replicated incorrectly, the item refresh procedure will eventually correct the mistake.

### 4.3 Parameter selection

The Eddy protocol has several parameters: the gossip interval, the gossip size, the number of items per node ($C$), the target variation between cache sizes ($\delta$) and the item lifespan. The optimal values for these parameters depend on tradeoffs between a number of performance factors.

The gossip interval is determined by rate at which new samples are required. From the point of view of sampling accuracy the ideal gossip size is one. As the gossip size increases, the chance that two items will follow the same random walk among the nodes also increases. This consideration however needs to be balanced against the fact that a larger gossip size reduces the number of gossip messages required to maintain a given sampling rate.

Nodes should have large caches for a number of reasons, although these needs must be regulated by the cost of storing the cache and the cost of refreshing items. The larger the cache size, the more accurate the sampling will be. A large cache reduces the chance that two items will be selected together in successive gossips, thus compensating for larger gossip sizes. Since cache size is primarily determined by the number of items representing each node, $C$, larger caches also reduce the effect of message failures since each failure involves a smaller percentage of the items. The cache also acts as a buffer, thus a node with a larger cache can have larger temporary variations between its sampling rate and its gossip rate, can participate in more gossips simultaneously, and has less need to manage cache size variations (determined by $\delta$).

The optimal item lifespan is determined primarily by the rate at which errors occur, which is itself dictated by the churn rate and the message failure rate. Increasing the item lifespan reduces the number of messages needed to refresh items. This consideration must be balanced against the loss of accuracy resulting from the fact that an increased lifespan means errors take longer to fix.

The choice of protocol parameters depends on application requirements and system conditions. To optimize costs parameters could be varied over time. As discussed, the accuracy of a protocol depends on the fact that $C$ is a global constant. Other parameters could be varied between nodes. In systems in which connectivity restrictions exist, variations in the gossip rate and gossip size could be used to improve the information flow between poorly connected parts of the network.

## 5 Comparison experiments

In this section we present results of experiments which test the effectiveness of the measures taken by Eddy to improve the accuracy of gossip-based sampling. We examine two key properties: (1) Eddy's ability to maintain a global item set in which all active nodes are represented equally, and (2) the end effect of improvements in the uniformity and independence of local samples on a simple application.

As discussed in Section 3, temporal dependancies between samples occur when the global item set contains unequal numbers of items for each node. One of the main goals of the Eddy protocol is to avoid this situation. By running a simulation and periodically compiling a snapshot of the global item set we can test how well a protocol manages the item set, and how long it takes to repair the item set after errors occur. Spatial dependancies, where a sample at a particular location is more likely to represent one node than another, are more difficult to measure directly because

of the large number of possible relationships between samples and locations. However, the overall accuracy of a protocol can be tested by examining how well it supports an application that depends on the uniformity and independence of samples.

In the following experiments we consider two main scenarios. First, we study a stable population of nodes, in which we can isolate and examine inaccuracies introduced by normal protocol operations and the effects of failed messages (Section 5.1). Second, we consider the impact of node failures on accuracy, both in an isolated incident, and in a more realistic churn scenario where nodes are continuously failing and being replaced (Section 5.2).

We compare Eddy to the Cyclon protocol as given in Voulgaris et al. [19], and to the protocol given in Allavena et al. [1], which improves on the original Clearinghouse protocol [3]. For convenience, we call this second protocol "Clearinghouse+". The Cyclon protocol is a more complex version of the protocol given in Fig. 1, which emphasizes fast item replacement after node failures, using a hop-count age to identify and replace older items. The gossiping procedure mostly exchanges rather than replicates items. Jelasity et al. [9] show that these characteristics make Cyclon one of the more accurate of a class of protocols which also includes lpbcast [5], Newscast [8], and AARG [4]. The Clearinghouse+ protocol emphasizes fast mixing. When gossiping nodes exchange information on their full cache contents. During each round each node replaces its entire cache contents with selected addresses of nodes with which it has gossiped and a random selection of items from their caches. Item replacement is left to chance.

The problem of choosing optimal parameter values falls outside the scope of this paper. We consider the setting, used in most work on gossiping protocols, in which samples should be provided at a predetermined constant rate to all nodes, the maximum expected error rate is known, and there are no restrictions on connectivity between nodes. We use the following parameters: network size $N = 1000$, cache size $C = 25$, and a gossip frequency of once per second. With these parameter values each node knows of no more than 2.5% of the other nodes in the system. Work on the scalability of sampling protocols has shown that this is a small enough percentage to show representative behavior of very large systems [9, 19]. Eddy and Cyclon nodes exchange $g = 5$ items in each gossip, that is a gossip replaces 20% of the average number of items in the cache. Clearinghouse+ nodes use a fan-out, $f$, of 3, meaning that nodes gather cache information from three other nodes each round. The Clearinghouse+ refresh parameter $w$ is set to infinity, so that nodes store new items for all nodes that send them gossip requests during a round.

We run message-based simulations which take into account the parallel nature of the protocols, such as allowing a node to receive incoming gossip requests while waiting for a reply to a request of its own. Nodes have synchronized clocks. A setup phase in which initial nodes are added to the system is run before measurements for the main experiments are begun.

## 5.1 Stable node populations

We first compare the accuracy of protocols in the straightforward case in which the node population does not change. This allows us to isolate inaccuracies that are inherent in a protocol's operations from those caused by errors such as message and node failures. In the experiments in this section Eddy nodes refresh items at a constant rate of one item per 10 s, giving items a lifetime of $l = 250$ s.

### *5.1.1 Item population and cache sizes*

One of the aims of the Eddy protocol is to keep the number of items representing each node constant. Figure 8 gives the distribution of the number of items for each node found by halting the protocols and taking a system snapshot. For Eddy there are 25 copies of each item, as desired. In comparison, the number of copies of each item in the Cyclon protocol varies between 18 and 38. These results are consistent with those given in Voulgaris et al. [19]. Further experiments by Jelasity et al. [9] show that other protocol variations exhibit

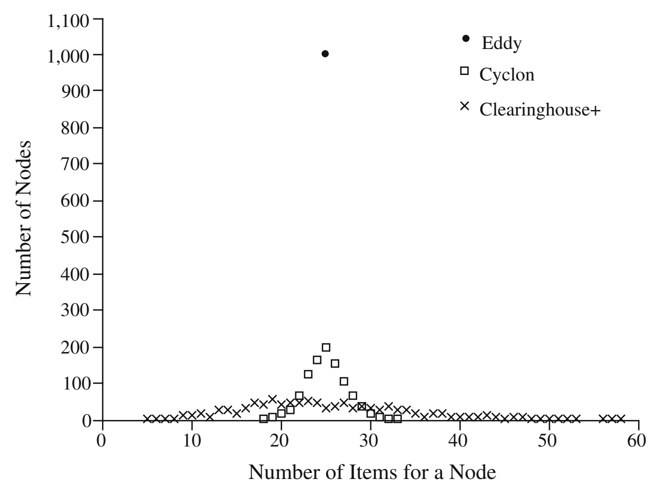


**Fig. 8** Distribution of the number of items representing each node

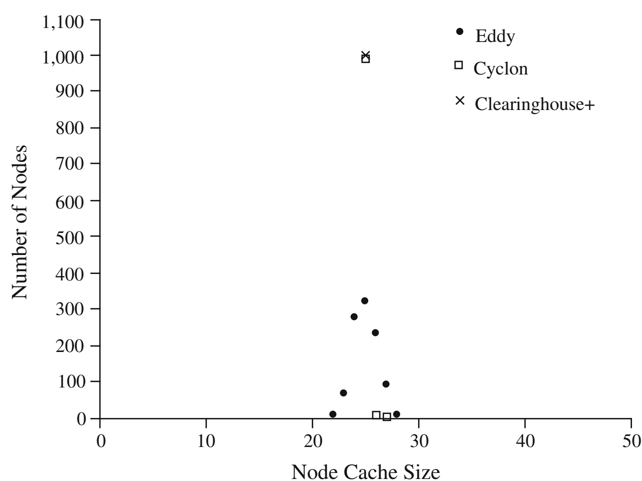


**Fig. 9** Distribution of cache sizes

an even wider distribution, especially those that copy rather than exchange items when gossiping. This effect can be seen in the data for Clearinghouse+ in which the number of items for each node varies widely, between 5 and 58.

One of the reasons Eddy can maintain equal numbers of each item is that it does not restrict new items to being added in the same locations as old items are removed. Since items expire and are inserted in random caches, cache sizes vary over time. The re-adjustment procedure corrects imbalances, but mostly at a slower rate than they are created. Figure 9 shows the distribution of the number of items in the cache of each node in a system snapshot. For an adjustment bound of $\delta = 3$, the cache sizes in Eddy vary between 22 and 28 items. In comparison, Cyclon maintains an approximately[2] constant cache size and Clearinghouse+ nodes all have $C$ items in their cache.

### *5.1.2 Network size estimation*

Estimating the size of a peer-to-peer network is a common application that makes use of node sampling [13]. For instance, the inverted birthday-paradox estimate, as described in Bawa et al. [2], can be used. Let $x$ be the total number of items seen before two items for the same node are detected. The estimate of $N$ is then $x^2/2$. We have the nodes each make repeated estimates of the network size by watching the stream of incoming items produced by the gossiping protocols. Table 1 gives the average and standard deviation of the values found by all nodes during a 16 min experiment. The table shows that the protocol used makes a difference to the size estimates. This method of estimating system size is very inaccurate, as shown by the large standard deviations. Nonetheless, the distribution of size estimates centers on the correct average value when a good random number generator is used. The size estimation algorithm using Cyclon estimates the network to be on average 10% larger than it would using a random generator, while using Eddy reduces this overestimate to being only 1.5% larger. The lack of item-set management in Clearinghouse+ results in an average size estimate that is only 40% of the actual size. Table 1 includes a run for Eddy in which item replacement is disabled, which shows that even in the simple case with a constant item set, size estimates based on gossiping differ slightly from those based on a random number generator.

**Table 1** Average and standard deviation of network size estimates

| | Average | Standard deviation |
|---|---|---|
| Ideal | 1000 | 0 |
| Random number generator | 1021 | 991 |
| Eddy | 1036 | 1005 |
| Cyclon | 1122 | 1041 |
| Clearinghouse+ | 403 | 485 |
| Eddy without replacement | 1031 | 997 |

### *5.1.3 Network unreliability*

Gossip protocols are robust to low levels of random message failure [4]. As discussed in Section 4, however, when a gossip exchange fails, and a node cannot detect if it was the request or the reply message that failed, items can be replicated or lost. Messages lost in the item insertion procedure also remove valid items. In the presence of such failures, the item set cannot be managed perfectly. Table 2 shows the end effect on network size estimates when the experiment from Section 5.1.2 is run on an unreliable network. For the Eddy and Clearinghouse+ protocols, losing up to about one percent of messages has little effect on the accuracy of size estimates. Low levels of message loss have a slightly larger effect on Cyclon. Higher rates of message failure disrupt the ability of all of the protocols to

**Table 2** Average size estimates with message loss

| | Eddy | Cyclon | Clearinghouse+ |
|---|---|---|---|
| 0% message loss | 1036 | 1122 | 403 |
| 0.1% message loss | 1033 | 1116 | 403 |
| 1% message loss | 1013 | 1042 | 405 |
| 10% message loss | 819 | 814 | 430 |

[2]The cache size is occasionally larger than $C$ due to gossip exchanges being interrupted by requests from other nodes.

maintain balanced item sets. We discuss this further in Section 5.2.

### 5.2 Dynamic node populations

The main source of complexity in node sampling protocols comes from the need to change the item set to reflect changes in the node set. Adding or removing nodes in a controlled manner is fairly straightforward since the item set can be managed accordingly. Node failures, however, are most easily handled by continually refreshing the item set. In this section we test the effectiveness of the item refresh process.

#### *5.2.1 Node failures*

Node failures have two effects on sampling: first, failed nodes leave behind items which are no longer valid, and second, when a node fails it removes valid items from the system. Sampling protocols thus need a refresh procedure to replace items. The rate at which the refresh is done, and the rate at which old items are removed, determines how quickly a protocol responds to a node failure.

To test the speed of the response to node failures, we run the 1000 node experiment scenario from Section 5.1, but cause 100 nodes to fail after 250 s. For comparison to Cyclon, in which each node inserts one new item each round, we increase the Eddy refresh rate to one item per node per second, giving items a lifetime of $l = 25$ s. The Clearinghouse+ protocol's parameters of $f = 3$, $w = \infty$ result in three new items being inserted for each node each round.

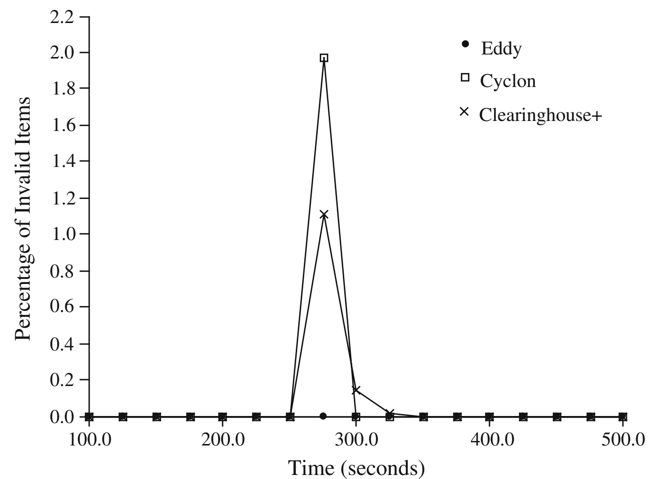


**Fig. 11** Percentage of invalid items for the experiment in Fig. 10

Figure 10 gives average network size estimates, measured by recording the estimates found by all nodes in 25 s intervals, and plotting the average at the end of each interval. The figure shows that both Eddy and Cyclon recover quickly, within 50 to 75 s, when nodes fail. Clearinghouse+ is not precise enough to record a noticeable difference. The time it takes size estimates to adjust is due to two factors. First, the item set needs to be refreshed to reflect the new composition of the node set. Second, the network size estimation algorithm takes some time to make each estimates. Given that most estimates are within $2N$ (Table 1), for $N = 1000$ repeated nodes are mostly detected within 64 samples. For the gossip size of $g = 5$, and an average of two gossips a second, most estimates will be completed within 6.5 s. The observed 50 to 75 s adjustment interval

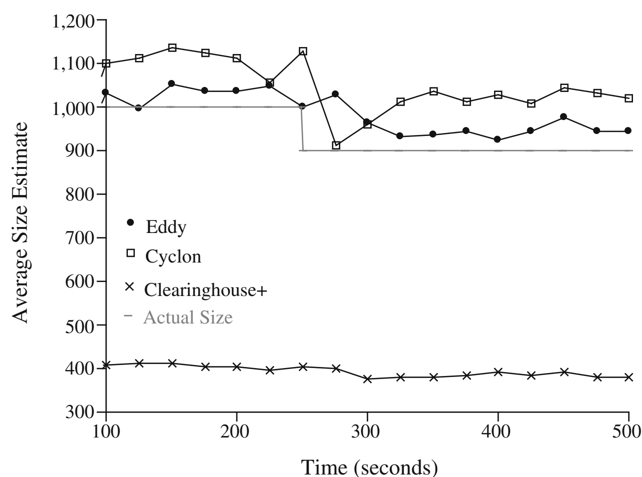


**Fig. 10** Average network size estimate per period: $N = 1000$, 10% of nodes fail at 250 s

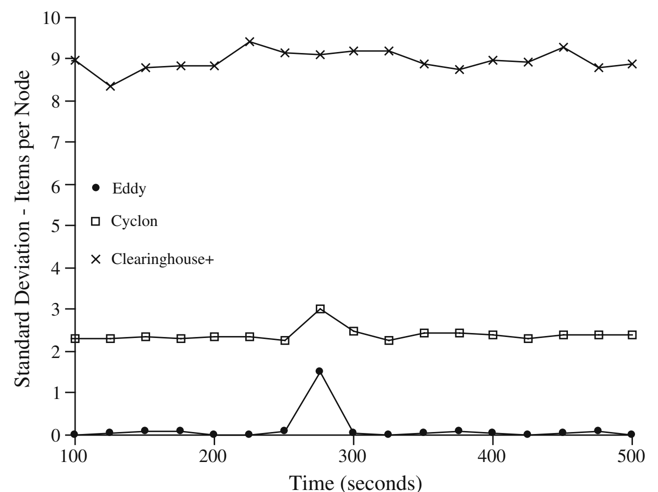


**Fig. 12** Standard deviation of the number of items per active node for the experiment in Fig. 10

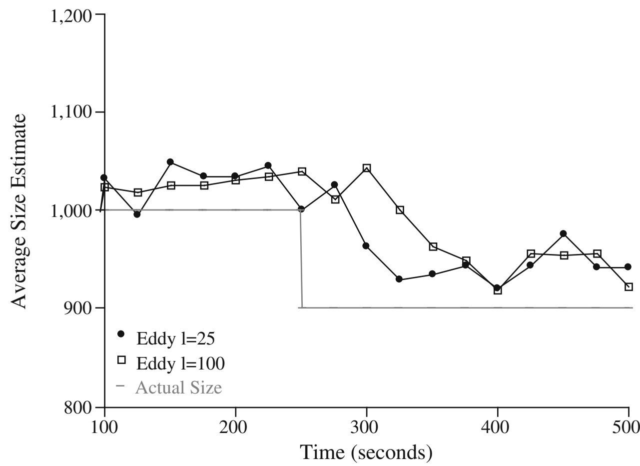


**Fig. 13** Average network size estimate per period: 10% of nodes fail at 250 s

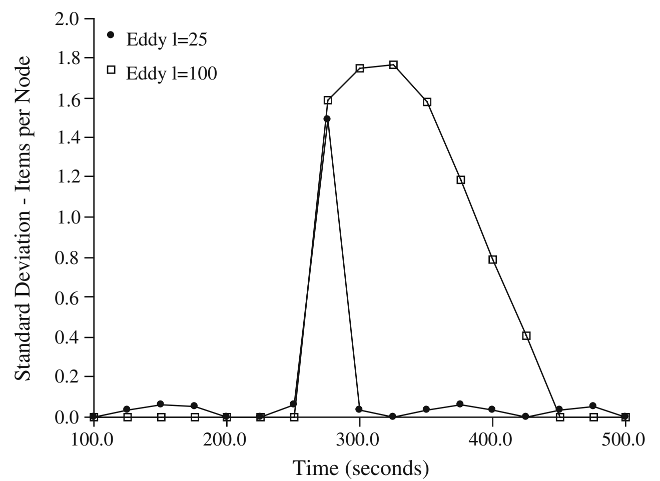


**Fig. 15** Standard deviation of the number of items per active node for the experiment in Fig. 13

is thus mostly due to the time it takes to rebalance the item set.

Figure 11 shows the percentage of invalid items remaining in the systems at the end of each 25 s period. This shows that Eddy removes invalid items within 25 s, and Cyclon removes them within 50 s, and Clearinghouse+ within 75 s. Figure 12 gives the standard deviation of the number of items per active node at the end of each 25-s period. Both Eddy and Cyclon take approximately 50 s to return to their normal distribution of items per node.

### 5.2.2 Item refresh rate

In the experiment in Section 5.2.1 the Eddy and Cyclon nodes each insert new items at a rate of once per second. Unlike previous protocols, Eddy separates the item refresh procedure from the mixing procedure, allowing item lifetimes to be adjusted to balance the cost of refreshing items against the rate of recovery from failures. To examine the effect of lowering the refresh rate we re-run the experiment in Section 5.2.1 for Eddy using an item lifetime of $l = 100$ s, in place of $l = 25$. Figure 13 shows that this has little effect on size estimates, although it now takes 100 s to remove invalid items (Fig. 14), and 200 s to even out item distributions (Fig. 15). The recovery time shown in Fig. 15 is longer than the 100 s it takes to refresh all items, because the presence of invalid items results in some item insertions failing. Thus it takes two times the item lifetime to fully recover from failures. This could be improved if insertions were acknowledged and failed insertions were retried.

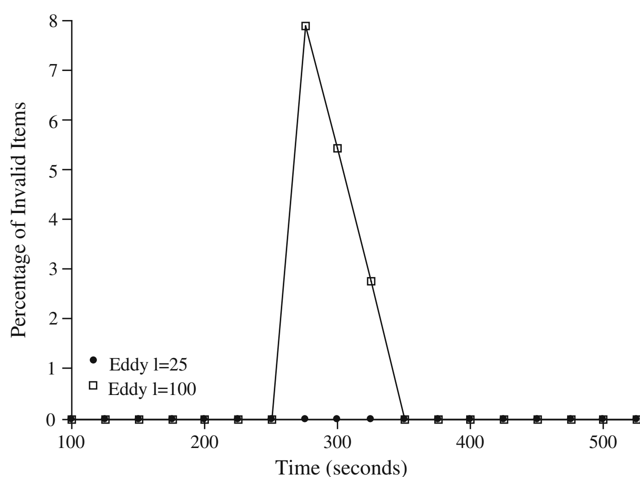


**Fig. 14** Percentage of invalid items for the experiment in Fig. 13

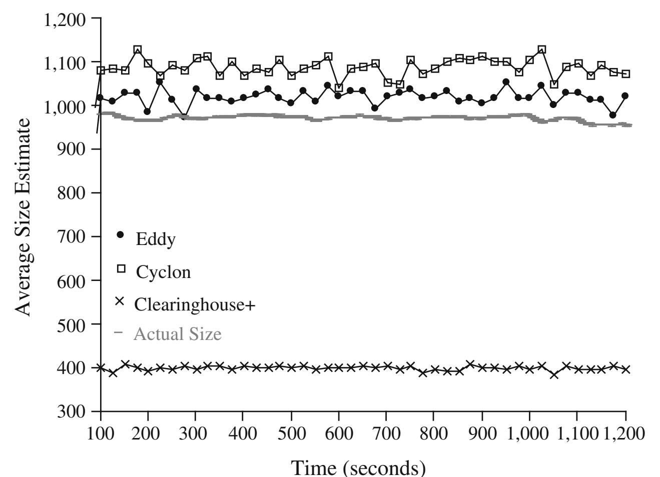


**Fig. 16** Average network size estimate per period with node churn

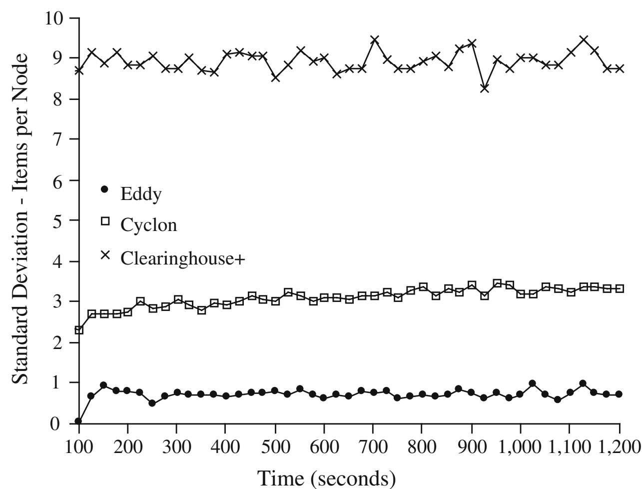


**Fig. 17** Standard deviation of the number of items per active node for the experiment in Fig. 16

*5.2.3 Node churn*

During the normal operation of a peer-to-peer system, nodes are continually added and removed. Leonard et al. [14] give a model for node churn based on observations of node lifetimes in actual p2p systems. They observe that node lifetimes follow a shifted Pareto distribution, $F(x) = 1 - (1 + x/\beta)^{-\alpha}, x > 0, \alpha > 1,$ with the parameters $\alpha = 3$ and $\beta = 1$ h. This gives nodes an expected lifetime of 30 min. In a system with 1000 nodes in the steady state, a node would be removed and another added on average every 1.8 s. This is much shorter than the time it taken by the protocols to fully recover from failures in the experiments in Section 5.2.1.

Figure 16 shows the average network size estimates made by each of the algorithms in a system where new nodes are added at a steady rate of one every 1.8 s, and nodes fail according to the lifetime distribution given above. Again, we plot the average estimate made by all nodes in the preceding 25 s. This level of churn does not have a large effect on the size estimates made by the algorithms. Figure 17 shows that churn also does not greatly affect the distribution of items maintained by the algorithms. This shows that Eddy maintains the improvements it makes in sampling accuracy in the presence of a realistic rate of node churn.

## 6 Conclusion

The Eddy protocol demonstrates that distributed sampling protocols can be improved by paying close attention to sources of temporal and spatial dependancies between samples. Temporal dependancies can be minimized by avoiding copying items, and using precise item expiry times, rather than relying partly on chance when refreshing items. Spatial dependancies can be avoided by not inserting new items in the direct vicinity of the node they represent. We demonstrate that making these changes results in an equal representation of nodes in the item set from which samples are drawn, and makes a measurable improvement in network-size estimates based on sampling both in a static network and under realistic network conditions, including network unreliability and node churn. Future work includes examining how protocol parameters can be set to match the requirements of the system in which the protocol is running, and how parameters can be automatically adjusted as these requirements change.

A Java implementation of the Eddy protocol can be downloaded from our website: www.dcs.warwick.ac.uk/research/hpsg/.

**Acknowledgements** This research is funded in part by the Engineering and Physical Sciences Research Council (EPSRC) UK grant number EP/F000936/1.

## References

1. Allavena A, Demers A, Hopcroft JE (2005) Correctness of a gossip based membership protocol. In: PODC '05: proceedings of the 24th annual ACM symposium on principles of distributed computing. ACM, New York, pp 292–301
2. Bawa M, Garcia-Molina H, Gionis A, Motwani R (2003) Estimating aggregates on a peer-to-peer network. Technical report, Stanford University
3. Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D (1987) Epidemic algorithms for replicated database maintenance. In: PODC '87: proceedings of the 6th annual ACM symposium on principles of distributed computing. ACM, New York, pp 1–12
4. Drost N, Ogston E, van Nieuwpoort RV, Bal HE (2007) Arrg: real-world gossiping. In: HPDC '07: proceedings of the 16th international symposium on high performance distributed computing. ACM, New York, pp 147–158
5. Eugster P, Guerraoui R, Handurukande S, Kouznetsov P, Kermarrec A-M (2003) Lightweight probabilistic broadcast. ACM Trans Comput Syst 21(4):341–374
6. Ganesh AJ, Kermarrec A-M, Massoulie L (2003) Peer-to-peer membership management for gossip-based protocols. IEEE Trans Comput 52(2):139–149
7. Iwanicki K, van Steen M, Voulgaris S (2006) Gossip-based clock synchronization for large decentralized systems. In: SelfMan '06: proceedings of the second IEEE international workshop on self-managed networks, systems and services, Dublin, June 2006, pp 28–42
8. Jelasity M, Kowalczyk W, van Steen M (2003) Newscast computing. Technical report, Vrije Universiteit Amsterdam, Department of Computer Science
9. Jelasity M, Voulgaris S, Guerraoui R, Kermarrec A-M, van Steen M (2007) Gossip-based peer sampling. ACM Trans Comput Syst 25(3):8

10. Karp R, Schindelhauer C, Shenker S, Vocking B (2000) Randomized rumor spreading. In: FOCS ’00: proceedings of the 41st annual IEEE symposium on foundations of computer science. IEEE Computer Society, Los Alamitos, pp 565–574
11. Kempe D, Dobra A, Gehrke J (2003) Gossip-based computation of aggregate information. In: FOCS ’03: proceedings of the 44th annual IEEE symposium on foundations of computer science. IEEE Computer Society, Washington, DC, p 482
12. Kermarrec A-M, Massoulie L, Ganesh AJ (2003) Probabilistic reliable dissemination in large-scale systems. IEEE Trans Parallel Distrib Syst 14(3):248–258
13. Kostoulas D, Psaltoulis D, Gupta I, Birman K, Demers A (2005) Decentralized schemes for size estimation in large and dynamic groups. In: NCA ’05: proceedings of the fourth IEEE international symposium on network computing and applications. IEEE Computer Society, Washington, DC, pp 41–48
14. Leonard D, Yao Z, Rai V, Loguinov D (2007) On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. IEEE/ACM Trans Netw 15(3): 644–656
15. Ogston E, Jarvis SA (2008) Improving the accuracy of peer-to-peer sampling services. In: ComP2P ’08: proceedings of the first international workshop on computational P2P networks: theory and practice. IEEE Computer Society, Athens
16. Ogston E, Overeinder B, van Steen M, Brazier F (2003) A method for decentralized clustering in large multi-agent systems. In: AAMAS ’03: proceedings of the second international joint conference on autonomous agent and multi agent systems, Melbourne, July 2003, pp 798–796
17. Stavrou A, Rubenstein D, Sahu S (2002) A lightweight, robust p2p system to handle flash crowds. In: ICNP ’02: proceedings of the 10th IEEE international conference on network protocols. IEEE Computer Society, Washington, DC, pp 226–235
18. Tan G, Jarvis SA (2007) Improving the fault resilience of overlay multicast for media streaming. IEEE Trans Parallel Distrib Syst 18(6):721–734
19. Voulgaris S, Gavidia D, van Steen M (2005) Cyclon: inexpensive membership management for unstructured p2p overlays. J Netw Syst Manag 13(2):197–217, June

**Elth Ogston** is a Post-doctoral Research Fellow the High Performance Systems Group at the University of Warwick. She obtained her Bachelors/Masters degree from the Massachusetts Institute of Technology in 1996 and subsequently joined HP Labs in Bristol. She completed her Ph.D. at the Vrije Universiteit Amsterdam in 2005.

**Stephen A. Jarvis** is an Associate Professor (Reader) in the Department of Computer Science at the University of Warwick. He is head of the High Performance Systems Group and also the Department’s Director of Research. Dr Jarvis has authored more than 125 refereed publications (including three books) on software and performance evaluation. While previously at the Oxford University Computing Laboratory, he worked on the development of performance tools with Oxford Parallel, Sychron Ltd and Microsoft Research in Cambridge. He has considerable experience in the field of peer-to-peer systems, with particular reference to overlay construction and performance optimization(including publications in ICDCS, INFOCOM, DSN and MASCOTS). His recent papers on this topic have received best paper awards, and he has published several IEEE Transactions Parallel and Distributed Systems articles in this area. He is also guest editor of a special issue of the International Journal of Parallel, Emergent and Distributed Systems dedicated to the performance analysis of P2P systems. Dr Jarvis has been a member of more than thirty international programme committees for high-performance and distributed computing. He is an external advisor for the Netherlands Organization for Scientific Research; co-organiser for one of the UK’s High End Scientific Computing Training Centres; Manager of the Midlands e-Science Technical Forum on Grid Technologies, and elected member of the EPSRC Review College.