ARRG: Real-World Gossiping

Niels Drost, Elth Ogston, Rob V. van Nieuwpoort and Henri E. Bal
Department of Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
niels, elth, rob, bal @cs.vu.nl
www.cs.vu.nl/ibis

ABSTRACT

Gossiping is an effective way of disseminating information in large dynamic systems. Until now, most gossiping algorithms have been designed and evaluated using simulations. However, these algorithms often cannot cope with several real-world problems that tend to be overlooked in simulations, such as node failures, message loss, non-atomicity of information exchange, and firewalls.

We explore the problems in designing and applying gossiping algorithms in real systems. Next to identifying the most prominent real-world problems and their current solutions, we introduce *Actualized Robust Random Gossiping* (ARRG), an algorithm specifically designed to take all of these real-world problems into account *simultaneously.* To address network connectivity problems such as firewalls we introduce a novel technique, the *Fallback Cache.* This cache can be applied to existing gossiping algorithms to improve their resilience against connectivity problems.

We introduce a new metric, *Perceived Network Size* to measure a gossiping algorithm's effectiveness. In contrast to existing metrics, our new metric does not require global knowledge. Evaluation of ARRG and the Fallback Cache in a number of realistic scenarios shows that the proposed techniques significantly improve the performance of gossiping algorithms in networks with limited connectivity. Even in pathological situations, with 50% message loss and with 80% of the nodes behind a NAT, ARRG continues to work well. Existing algorithms fail in these circumstances.

Categories and Subject Descriptors:

C.2.1 [Computer-Communication Networks]: [Network Architecture]: Distributed Networks

C.4 [Performance of Systems]: Fault Tolerance, Measurement techniques

General Terms: Algorithms, Design, Measurements, Performance, Reliability.

Keywords: Peer-to-Peer, Gossiping, Robust, Real-World.

1. INTRODUCTION

Information dissemination in distributed systems is usually achieved through broadcasting. Commonly, broadcasting is done by building a broadcast tree, along which messages are sent. This approach can also be taken in peer-to-peer systems [15]. However, maintaining such a tree structure in a large dynamic network is difficult, and can be prohibitively expensive. The broadcast tree may have to be rebuilt frequently due to changing network conditions or machines, or *nodes*, joining and leaving.

For peer-to-peer systems, an alternative to building broadcast trees is to use flooding [19]. Unlike broadcasting, flooding does not use any specific network structure to control the flow of messages between nodes. Upon receiving a new message, a node simply sends it to *all* its neighbors. With flooding, nodes can receive messages multiple times from different neighbors. Especially in situations where many nodes need to simultaneously and frequently disseminate information, both tree-based broadcasting and flooding are ineffective, because the number of messages passing through each node quickly exceeds its network capacity, processing capacity, or both.

Gossiping algorithms offer an alternative [6] to broadcasting and flooding when efficiency, fault tolerance and simplicity are important. The aim of gossiping algorithms is to severely limit the resources used at each node at any point in time. These algorithms are usually based on a small cache of messages stored on each node. Periodically, nodes exchange, or *gossip*, these messages with each other, thus updating their caches. This results in each node receiving a constantly changing set of messages. Over time, each node is likely, but not guaranteed, to see each message in the system. Thus, with gossiping, the resource usage of a node is bounded in exchange for a slower rate of information dissemination. Also, gossiping does not guarantee messages to be received in the same order they were sent, and messages might be lost or delivered multiple times.

Gossiping techniques are used in many applications, such as replicated name services [3], content replication [14, 8, 17], self configuration and monitoring [2] and failure detection [13]. One example application is our Zorilla [4] peer-to-peer middleware system, where we use gossiping to manage the overlay network used for resource discovery.

Since gossiping algorithms are a relatively new research topic, most research on gossiping has relied on theory and simulations for evaluation of algorithms. However, because of inherent limitations, simulations cannot take into account all aspects of a real system.

This paper focuses on the design, deployment and evaluation of gossiping algorithms in real-world situations. We focus on a gossiping-based membership algorithm capable of providing a *uniform random set of members* at each node, a problem commonly solved with gossiping techniques. Membership algorithms are an important building block for gossiping algorithms in general. In a membership algorithm, the messages which are gossiped actually contain identifiers of nodes, or *members*, of the system. Since most applications for gossiping algorithms rely on disseminating information uniformly across a certain system, the random members produced by the algorithms described in this paper are ideally suited as targets for gossip messages.

The contributions of this paper are as follows.

- We give an overview of all known and some new difficulties encountered when moving from simulation to application. These difficulties include race conditions due to non-atomic gossips, failing nodes, unreliable networks, and the inability to reach nodes because of firewalls and Network Address Translation (NAT).

- Although most gossiping algorithms are able to cope with *some* of the problems in real-world systems, there currently are no gossiping algorithms specifically designed to cope with *all known problems simultaneously*. We therefore introduce a simple, robust gossiping algorithm named *Actualized Robust Random Gossiping* (ARRG). It is able to handle all aforementioned problems.

- To address the connectivity problems encountered, we introduce a novel technique for ensuring the proper functioning of a gossiping algorithm, a *Fallback Cache*. By adding this extra data structure to an existing gossiping algorithm, the algorithm becomes robust against connectivity problems such as firewalls. The Fallback Cache is not limited to ARRG, but can be used with any existing gossiping algorithm. We have, for instance, also applied it to Cyclon [17]. Maintaining a Fallback Cache does not incur any communication overhead and does not alter the properties of the existing algorithm, except for making it more robust.

- To compare gossiping algorithms and techniques, we introduce a new performance measurement: *Perceived Network Size*. This novel metric has the advantage that it can be measured locally on a single node. Traditional metrics require global knowledge, which is impractical to obtain in real systems. In addition, the Perceived Network Size metric can be used to determine behaviors of gossiping algorithms previously evaluated using multiple separate metrics. Also, our metric is able to clearly show differences in both efficiency and correctness between gossiping algorithms.

- We evaluate ARRG in *simulations* and on a *real system*, using several realistic scenarios with NAT systems, firewalls and packet loss. In these conditions, ARRG significantly outperforms existing algorithms. We also apply ARRG's Fallback Cache technique to an existing gossiping algorithm (Cyclon), and compare its performance with ARRG. We show that, in systems with limited connectivity, algorithms with the Fallback Cache significantly outperform gossiping algorithms without it. Even under pathological conditions with a message loss rate of 50% and with 80% of the machines behind a NAT, ARRG still performs virtually the same as under perfect conditions. Existing gossiping algorithms fail under these circumstances.

The rest of this paper is organized as follows. In Section 2 we focus on the problems present in real-world systems as opposed to simulations and theory. We investigate to what extent current gossiping algorithms are able to handle these problems. In Section 3 we introduce the ARRG algorithm, followed by the introduction of the Fallback Cache technique in Section 4. We introduce the Perceived Network Size metric in Section 5. In Section 6 we evaluate the different algorithms and techniques in a number of use cases. Finally, in Section 7, we draw some conclusions and list future work.

2. REAL-WORLD PROBLEMS

In this section we give an overview of all known problems that must be addressed when implementing an abstract gossiping algorithm in a real system, and identify some problems not currently considered. We also examine related work, focusing on the solutions for these problems used by current algorithms. Table 1 gives an overview of the capability of different gossiping algorithms to address these problems. Each algorithm is rated according to its ability to overcome each problem. *NC*, sometimes listed alongside a rating, indicates that the literature available for this protocol does not consider this problem. In these cases we analyzed the algorithm itself to determine the rating. We also list ARRG, a simple and robust gossiping algorithm introduced in Sections 3 and 4.

2.1 Node failures

The first problem is that of node failures. A node may leave gracefully, when the machine hosting it is shut down, or it might fail, for instance when the machine crashes. In this context, the joining and leaving of nodes during the operation of the system is often called *churn*. Most, if not all existing gossiping algorithms take node failures into account.

A common method to handle node failures is *refreshing*, where new identifiers of nodes are constantly inserted in the network, replacing old entries. Combined with redundant entries for each node, this ensures that invalid entries are eventually removed from the network, and an entry for each valid node exists. All protocols listed in Table 1 exhibit this behavior, except the SCAMP [7] algorithm.

As an optimization, failures can also be detected. For example, the Cyclon [17] algorithm removes nodes from its cache when a gossip attempt fails. This technique requires the algorithm to be able to detect failures. If an algorithm only sends out messages, without expecting a reply, a failure cannot be detected without extra effort.

2.2 Network unreliability

The second problem we face is network unreliability. Most simulations assume that all messages arrive eventually, and in the same order as they were sent. In real networks, however, messages may get lost or arrive out of order. This may have an influence on the proper functioning of the algorithm, especially when the message loss rate increases. If non-reliable protocols such as UDP are used, message loss

	Node Failures	Network Unreliability	Non-Atomicity	Limited Connectivity
SCAMP [7]	+/-	+/-	NA	- NC
lpbcast [5]	+ NC	+/- NC	NA	- NC
ClearingHouse [3, 1]	+	+ NC	+	- NC
PROOFS [14]	+	+	+/-	- NC
Newscast [8, 18]	+	+/- NC	+/- NC	- NC
Cyclon [17]	+	+/-	+/-	- NC
ARRG without Fallback	+	+	+	+/-
ARRG	+	+	+	+

- = unaddressed
+/- = partially addressed
+ = fully addressed

NA = Not Applicable

NC = Not Considered in literature covered by this paper

Table 1: Robustness of gossiping algorithms

can be caused by network congestion, limited buffer space, and firewalls dropping UDP traffic.

A partial solution for this problem is to use a network protocol capable of guaranteeing message delivery and ordering, such as TCP. However, whereas UDP is connectionless, TCP connections have to be initiated, and consume more local and network resources than the more lightweight UDP protocol. Finally, using TCP still does not make communication completely reliable, as TCP connections themselves can still fail due to network problems, timeouts or crashes.

To completely overcome the network unreliability problem a gossip algorithm needs to expect, and be robust against, message failures and out of order delivery. In the ClearingHouse [1] algorithm this is, for instance, handled by sending out multiple requests, while only a portion of the replies are required for a proper functioning of the protocol.

A gossiping algorithm is, by definition, unable to completely overcome network unreliability if it tries to maintain an invariant between communication steps. For instance, the Cyclon [17] algorithm swaps entries in each gossip exchange, where some entries from one node are traded for entries from another. Because this swap mechanism does not create or destroy entries, the number of replicas of each node identifier in the entire network remains constant. However, if a reply message is lost, this invariant does not hold, as the entries in the message have already been removed from the sender, but are not added to the cache of the node to which the message is sent.

2.3 Non-atomic operations

The third problem that is encountered when implementing gossiping algorithms on a real system is the non-atomicity of operations that involve communication. For example, the *exchange* of data between the caches of two nodes is often considered to be an atomic operation during the design and simulation of gossiping protocols. In reality, the exchange consists of a request and a reply message, separated in time by the latency of the network. Therefore, after a node has initiated a gossip, it can receive one or more gossip *requests* before its own gossip has finished (i.e., the gossip reply has not arrived yet). With some existing protocols, this can lead to data corruption of the gossiping cache.

In simulations, message exchanges are often instantaneous, and network latency is implicitly assumed to be zero. In reality however, latency can of significance, as the window for the aforementioned race condition increases as the network latency increases. Simply delaying incoming gossip requests until the initiated gossip has finished, leads to a deadlock if there happens to be a cycle in the gossiping chain. Another possible solution, ignoring concurrent gossips, as the PROOFS [14] algorithm does, leads to a high failure rate, as shown in [16].

We argue that, when a gossiping algorithm is designed, exchanging information between nodes cannot be considered an atomic operation. Care must be taken that the state of the cache remains consistent, even when a request must be handled while a gossip attempt is underway.

2.4 Limited connectivity

The fourth and last problem that must be faced in real-world networks is limited connectivity between nodes in the network. Most private networks use firewalls that block incoming connections. This effectively makes machines unreachable from the outside. Systems can usually still make connections to the outside, though sometimes even these are restricted.

Another type of device which limits connectivity between computers is a *Network Address Translation* system. These NAT devices make it possible for multiple computers in a network to share a single external IP-Address. The drawback is that, in general, the machines behind this NAT are not reachable from the outside.

Though methods exist to make connections between machines despite firewalls and NATs [11], these techniques cannot successfully be applied in all circumstances, and some connectivity problems remain. Moreover, mechanisms to circumvent firewalls and NATs often require considerable configuration effort, and typically need information about the network topology.

Most current gossiping algorithms are designed with the explicit [16] assumption that any node can send messages to any other node. Therefore these algorithms are not resilient to network connectivity problems.

There are systems that try to overcome limited connectivity, such as Smartsockets [11], Astrolabe [12, 2], Directional Gossip [9, 10] and Failure Detection [13]. However, these systems use an explicit structure on top of a traditional gossiping algorithm. Usually, a hierarchical network is built up manually to route traffic. To overcome the connectivity problem without requiring manual configuration and explicit knowledge of the network, new techniques are needed. We will introduce a novel solution, the Fallback Cache, in Section 4.

3. ACTUALIZED ROBUST RANDOM GOSSIPING

To address the problems mentioned in Section 2, we introduce a simple gossiping algorithm, named *Actualized Robust Random Gossiping* (ARRG). This gossiping algorithm is an

```
1 void selectTarget() {
2     return cache.selectRandomEntry();
3 }
4
5 void doGossip(Entry target) {
6     //select entries to send
7     sendEntries = cache.
8         selectRandomEntries(SEND_SIZE);
9     sendEntries.add(self);
10
11    //do request, wait for reply
12    sendRequest(target, sendEntries);
13    replyEntries = receiveReply();
14
15    //update cache
16    cache.add(replyEntries);
17    while(cache.size() > CACHE_SIZE) {
18        cache.removeRandomEntry();
19    }
20 }
21
22 Entry[] handleGossipMessage(
23                     Entry[] sendEntries) {
24    //select entry to send back
25    replyEntries = cache.
26        selectRandomEntries(SEND_SIZE);
27    replyEntries.add(self);
28
29    //update cache
30    cache.add(sendEntries);
31    while(cache.size() > CACHE_SIZE) {
32        cache.removeRandomEntry();
33    }
34
35    return replyEntries;
36 }
```

Figure 1: Pseudo code for ARRG

example of a gossiping algorithm specifically designed for robustness and reliability. ARRG is able to address node failures, network unreliability and does not assume that operations involving communication are atomic. ARRG uses randomness as a basis for making all decisions. This is done to to make the algorithm robust against failures and to reduce complexity.

The pseudo code for ARRG is shown in Figure 1. Every time a node instantiates a gossip, it selects a random target node from its cache, and sends it a random set of cache entries, containing a number of elements from its own cache. The exact number is a parameter of the algorithm, denoted by SEND_SIZE. The node also adds an entry representing itself to the list sent (lines 6–12).

Upon receipt of the message, the target sends back a random set of cache entries, again including itself (lines 24–27). It also adds the received entries to its cache, ignoring entries already present in its cache. Since the cache has a fixed maximum size, the target may need to remove some entries if the maximum size is exceeded. The purged entries are selected at random (lines 29–33).

When the initiator of the gossip receives the reply from the target, the received entries are added to the local cache, ignoring duplicates. If this action increased the size of the cache beyond its maximum, random entries are removed until the number of entries becomes equal to the size of the cache, denoted by CACHE_SIZE (lines 16–19).

ARRG was explicitly designed to address the problems listed in Section 2 in a simple and robust manner. The main difference between other algorithms and ARRG is the explicit design choice to use the simplest solution available for each functionality required, while taking the problems listed in Section 2 into account. Moreover, any decision must be as *unbiased* as possible. An example of a bias is the removal of nodes after a failed gossip exchange with that node. This creates a bias against nodes which are not directly reachable because of a firewall or a NAT, making the content of a node's cache less random.

Table 1 lists the capabilities of ARRG in comparison to other algorithms. An algorithm which can also overcome many problems in real-world systems is ClearingHouse [1]. However, ClearingHouse is significantly more complex than ARRG. This may create a bias against some nodes, hindering robustness. For instance, ClearingHouse combines caches from other nodes with the identifiers of nodes which requested its own cache. Therefore, nodes with better connectivity have a higher chance of appearing in caches. Another difference between ClearingHouse and ARRG is the bandwidth usage. ClearingHouse requests the complete cache from multiple nodes in each round, but disregards a large part of this information, waisting bandwidth. ARRG in contrast only transfers a small fraction of a single cache.

Node failures are handled in ARRG by the constant refreshing of entries. New entries are constantly added to the system, as old ones are purged. ARRG depends on random chance to purge old entries. Some protocols [17] also take into account the age of entries, but as this adds both complexity and a bias against old entries, ARRG simply replaces randomly selected entries each time a gossip exchange is done.

To address both the non-atomicity and the unreliability issue a gossip request and its reply in ARRG can be seen as two separate gossips; there is no data dependency between the two. If either the request or the reply gets lost, both the node initiating the gossip and the receiver of the request are in a consistent state, and no information is lost. Because of this decoupling between a gossip request and reply, non-atomicity issues such as race conditions do not occur. ARRG explicitly does not assume the exchange to be an atomic operation.

To make ARRG more robust against connectivity problems nodes are *not* removed from the cache when a gossip attempt to this node fails. This is done because the difference between a node which is unreachable and a node which is defective is undetectable. So, to guarantee that nodes behind a firewall don't get removed from all caches, failed gossip attempts are simply ignored. The random replacement policy will eventually remove invalid entries.

Ignoring failed gossip attempts gives nodes behind a NAT or firewall a better chance of remaining in the cache of other nodes, and eventually getting to the cache of a node which *is* able to reach it. There is still a chance that a node may loose all cache entries to nodes it is able to reach, effectively leaving it with no nodes to contact and possibly removing itself from the network. ARRG contains an additional mechanism to keep this from occurring, our novel *Fallback Cache*. This technique is described in the next section.

Another reason ARRG is more robust against connectivity problems, is the *push-pull* mechanism used to perform a gossip exchange. When a node sends a gossip request con-

```
 1 // extra initialization
 2 fallbackCache = new Set();
 3
 4 // gossip function
 5 void doGossipWithFallback() {
 6     // call existing algorithm functions
 7     target = selectTarget();
 8     doGossip(target); // could fail because
 9                       // of a timeout or a
10                       // connection problem
11
12     if(successful) {
13         // remember this target
14         fallbackCache.add(target);
15         if(fallbackCache.size()>CACHE_SIZE) {
16             fallbackCache.removeRandomEntry();
17         }
18     } else {
19         // retry with Fallback entry
20         target = fallbackCache.
21                       selectRandomEntry();
22         doGossip(target); // if this fails,
23                           // just ignore it
24     }
25 }
```

Figure 2: Fallback Cache pseudo code

taining some entries of its cache, the target of this message replies with some of its own entries. When a message is sent out by a machine behind a NAT or firewall, a reply is usually expected by this firewall or NAT. This makes it possible for a node to receive entries from the target node it sent a request to, even though the requesting node normally is unable to receive messages.

4. THE FALLBACK CACHE

As described in Sections 2 and 3, current solutions for the network connectivity problem do not always suffice. As a possible solution we introduce the *Fallback Cache*. This technique adds robustness to an existing gossiping algorithm, without changing in any way the functioning of the algorithm itself. The Fallback Cache acts as a backup for the normal membership cache present in the gossiping algorithm. Each time a successful gossip exchange is done, the target of this gossip is added to the Fallback Cache, thus filling it over time with peers which are reachable *by this node*. Whenever a gossip attempt fails, the Fallback Cache is used to select an entry to gossip with instead of the one selected by the original algorithm. Since this Fallback entry has already been successfully contacted once, there is a high probability that it can be reached again.

Figure 2 shows the necessary extra (pseudo) code to add a Fallback Cache to an existing algorithm. Line 2 shows the initialization of the Fallback Cache. The cache itself is a set of node identifiers. We assume the original gossiping algorithm performs a gossip once every T seconds by first selecting a target in a *selectTarget* function, followed by an attempt to contact this node to do the actual gossiping by a *doGossip* method. To use the Fallback mechanism, the *doGossipWithFallback* function starting at line 5 has to be called instead.

The *doGossipWithFallback* function mimics the original algorithm, by initially calling *selectTarget* and *doGossip* in lines 7 and 8. If the gossip attempt is successful, this node is added to the Fallback Cache in line 14 after which the function returns. The cache has a maximum size. If a new entry is added while it has already reached the maximum size, a random entry is removed from the cache after the new entry is added (lines 15–17).

If the attempt is unsuccessful, an entry is selected from the Fallback Cache in line 20, and another gossip attempt is done (line 22) with this new target. This retry is only done once. When it also fails (or when the Fallback Cache is empty) the algorithm gives up, and the next gossip attempt will be done whenever the function is called again (usually after some fixed delay).

Although the original gossiping algorithm may remove entries when a gossip attempt fails, the Fallback Cache will never remove invalid entries. This is done to make the Fallback Cache more robust. A that is currently not reachable because of network problems, or because it is overloaded with requests, may become reachable again later. If invalid entries are removed from the Fallback Cache it might become empty, leaving the node with no nodes to contact. Invalid entries will eventually be overwritten by new entries when successful gossip exchanges are performed, thanks to the random replacement policy.

The Fallback mechanism does not interfere with the normal operation of a gossiping algorithm. If no errors occur, the Fallback Cache is never used. Therefore, all the properties present in the original algorithm are retained. This makes it possible to add robustness against connectivity problems without redesigning the rest of the algorithm.

When errors *do* occur, the Fallback mechanism guards against the cache of the node containing only invalid entries. Since the Fallback Cache holds only entries which were valid at the time and location they were added, this provides a much more reliable source of valid entries than the original cache, greatly reducing the risk that a node gets disconnected from the network, and reducing the risk that the network partitions.

The Fallback Cache in itself is not a good source for uniformly random nodes. The update rate is slow at one entry per gossip round, and the mechanism relies on the original gossip cache to provide new entries. The combination of the original and the Fallback Cache is therefore needed to provide a valid and robust gossiping algorithm. The original cache will contain a random subset of all the nodes in the system, while the Fallback Cache contains a random set of all the *reachable* nodes.

5. PERCEIVED NETWORK SIZE

Ideally, a gossiping algorithm should populate node caches such that the chance of finding an item in a particular cache is both independent of the other items in the cache, and independent of the items in neighboring nodes' caches. We define a gossiping algorithm to be *viable* if it is unlikely to result in a partitioning of the network. The closer an algorithm is to the ideal random case, the faster information is likely to move between any two chosen nodes, and thus the more efficient it is.

Previous analytical studies have mostly measured the degree of randomness in communication of gossiping algorithms by considering properties of the overlay network formed by neighborhood caches. For instance, the eventual occurrence of network partitions [5, 1], the presence of small-worlds like

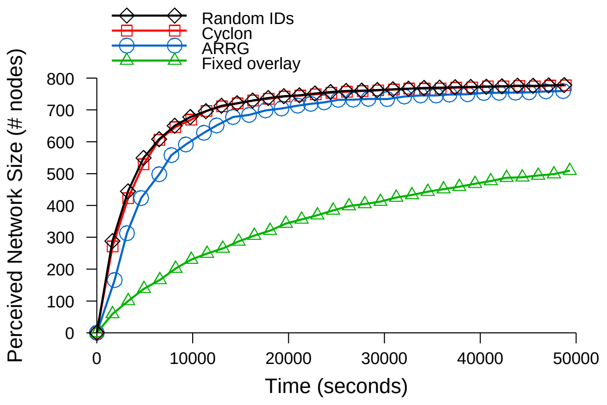

Figure 3: Perceived Network Size for four protocols

clustering [8, 18], or a widely varying node in-degree [1, 17] all indicate a certain amount of relatedness between cache contents. Such measures, however, require taking a snapshot of the state of all nodes, something which is non trivial in a real network. Voulgaris [16] also examines the randomness of the stream of items received by a single node using the Diehard Benchmark of statistical tests for pseudo-random number generators. These tests, however, only allow protocols to be labeled as random-enough, or not-random-enough. Moreover, running such a benchmark is costly, making it impractical to use in most systems.

For this study we introduce a novel measure of gossiping randomness, the Perceived Network Size, that can be applied at a single node, and that can distinguish differences in the degree of randomness of two algorithms, or in the degree of randomness at different locations in a network.

Consider the inter-arrival times, R, of items from a given node at another node in the network. Given that in a gossiping protocol all nodes enter the same number of items into the system, for a fully random protocol each item received should have a probability of $1/N$ of being from the selected node, where N is the number of nodes in the network. The distribution of the random variable R thus gives an indication of the randomness of the gossiping process. For instance, the average value of R, for a large enough sample size, will be N. Further, for a uniformly random process, a smaller sample size will be needed to get the correct average, than for a process where the distribution of items is skewed. Based on this, we define the *Perceived Network Size*, as the average inter-arrival time of a specific node identifier in a node's cache. In practice, we consider identifiers for all nodes rather than a single node, to increase accuracy and to speed up data collection. By examining the rate at which Perceived Network Size reaches the correct value as a node receives more information over time, we can compare the relative randomness of two gossiping processes, or of the same process at different nodes.

Figure 3 compares perceived network size versus time for ARRG, Cyclon, a protocol like ARRG but in which nodes' neighbors are fixed (Fixed Overlay), and for a stream of node IDs generated using a random number generator (Random IDs). All of these protocols can be considered to be *viable* in this setup as they all eventually reach the correct perceived network size of 800 nodes. The rate at which this value is reached, however, shows that the protocols in which the neighborhood overlay changes are significantly more *efficient* than the Fixed Overlay protocol. It can also be seen that ARRG is slightly less random than Cyclon, indicating the effect of the more precise age-based item removal used in Cyclon. Cyclon is almost indistinguishable from the "true" random process.

Besides allowing us to accurately compare viable protocols, the Perceived Network Size can also be used to detect the various behaviors considered in earlier studies. Protocols that result in small-worlds like clustering or exhibit widely varying node in-degrees are less efficient, and thus find the correct Perceived Network size more slowly. Network partitions result in a Perceived Network Size that is smaller than expected, as seen later in Figure 9. Slowly partitioning networks result in a decreasing Perceived Network Size (Figure 12).

6. EXPERIMENTS

To test the effectiveness of ARRG and its Fallback Cache technique we examine a number of different use cases. For the experiments we use the *Distributed ASCI Supercomputer (DAS-3)*[1], which consists of 5 compute clusters located across the Netherlands. As an example application we use Zorilla [4], a peer-to-peer middleware system. Zorilla uses the gossiping mechanism for resource discovery purposes. We evaluate the techniques described in this paper both with a *real* application on a *real* system, and using simulations.

We implemented both ARRG and Cyclon using the TCP protocol, and a Fallback Cache for both algorithms. Connections are set up using the SmartSockets library [11]. Both the normal and the Fallback Cache (if present) hold 10 entries. Each gossiping algorithm initiates a gossip exchange with a node from its cache every 10 seconds. During this exchange 3 entries are sent by each node.

To make a fair comparison between the algorithms, we implemented the possibility to run multiple gossiping algorithms with different implementations and settings concurrently. Each running algorithm has its own private data structures, and does not interact with other instances in any way.

When a gossiping algorithm starts, it needs some initial entries to gossip with. To start, or *bootstrap* the gossiping algorithm in Zorilla, or in this case multiple algorithms running concurrently, each node is given one or more other nodes when it starts running. This is usually a single node started before the others. In practice, the node used to bootstrap the system will probably not run permanently. So, to make the experiment as realistic as possible, the bootstrap entry is only available to each gossiping algorithm for a limited time, in this case 100 seconds. If a gossiping algorithm could use this bootstrap indefinitely, it would always be possible to restart an algorithm, making the difference between a failing gossiping algorithm and one performing poorly less distinguishable.

Parameters

We examined the parameters needed for gossiping protocols in various systems, in order to choose the parameters

[1]See http://www.cs.vu.nl/das3.

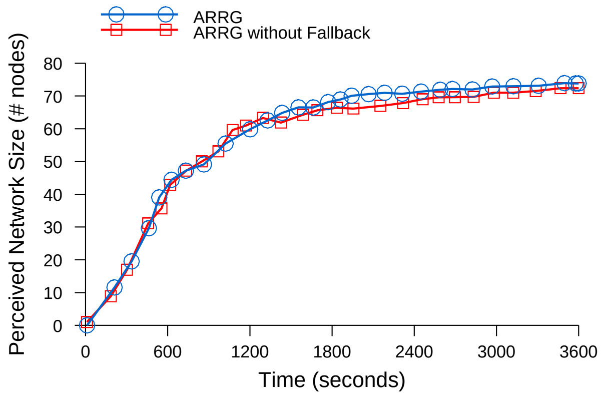

Figure 4: ARRG in a fully connected network

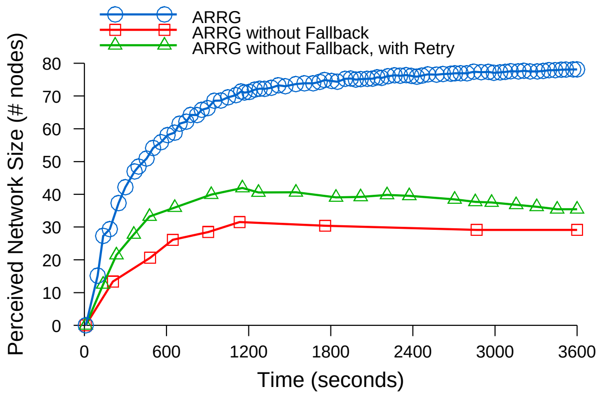

Figure 5: ARRG in the X@Home scenario

used in this section. In theory, increasing the gossip size of a protocol should proportionally increase the information dissemination rate of a completely random protocol. Also, increasing the number of nodes in the system should proportionally increase the amount of time it takes information to disseminate to a given fraction of the nodes.

We verified the theory in simulations, and found it to be approximately correct for the protocols and networks we study. Thus, for the experiments on a real system with 80 nodes we use small cache and gossip sizes of 10 and 3 items respectively. For simulations of 8000 nodes, we maintain the same cache to gossip ratio, but speed up the system by using larger values of 100 and 30 items. We further found that creating a larger cache to gossip ratio makes little difference to the average dissemination time, though it does decrease the amount of variation in the speed of information dissemination. The size of the caches, and the number of items exchanged each round are relatively small. We use these low values to show that gossiping algorithms can function with very limited resources.

6.1 The Fully Connected Scenario

For reference, and to determine the overhead of the Fallback Cache technique, we first measure on a network without any connectivity problems. We did this both in a simulated environment on 8000 nodes and in our test system using 80 nodes. We compare ARRG to ARRG without a Fallback Cache.

Figure 4 shows the results for the test system. There is no significant difference between ARRG with and without a Fallback Cache. As a Fallback Cache is only used when connectivity problems occur, this is expected. From this graph we can conclude the Fallback Cache causes no overhead. The simulated results are not shown here, as they are identical to the results from the experiment. Also note the Perceived Network Size of ARRG is converging to 80, the actual network size, showing ARRG is functioning properly in this setting.

6.2 The X@Home Scenario

To test the effectiveness of a Fallback Cache in more demanding situations, we continue with some experiments in a *X@Home* setting. In this scenario, a number of home users join together to form one big peer-to-peer network. Examples of such systems are the SETI@Home[2] project and file sharing applications.

The main reason for home users to have connectivity problems are NAT systems. As explained in Section 2.4, these are used to share a single connection between multiple machines, leaving these machines unreachable from the outside. Not all home users have a NAT system, as they may only have a single machine and do not need to share the connection. Also, some methods [11] exist to allow a machine behind a NAT to be reachable from the outside. In this scenario the machines connected to the peer-to-peer network can therefore be divided into two types. One type which is reachable from other nodes in the system, and one which is not.

As running an experiment with a large number of home machines is impractical, we use a simple technique to achieve the same result on our DAS-3 system. Each node which is in the group of *home* machines does not accept any incoming connections, and is only able to make outgoing connections. This is done at the socket level, so neither the application nor the gossiping algorithm itself are aware of their inability to receive gossip requests.

As a reference, we also include a *retry* version of ARRG. If an attempt to gossip fails, this version, like the Fallback version, attempts a new gossip with a different target. However, this new target is not selected from a special cache, but uses the normal selection mechanism instead. In the case of ARRG, this is simply a random entry from the cache.

This experiment consisted of 64 unreachable home machines and 16 *global* machines which are reachable normally, for a total of 80 nodes. Figure 5 shows the performance of ARRG on a global node. We compare ARRG with a version without a Fallback Cache and with a version that uses retries. The graph shows ARRG without the Fallback Cache is not a viable algorithm. As expected, it is unable to overcome the connectivity problems present in this scenario. The Perceived Network Size does not reach the actual network size, but only reaches 40, indicating that the gossiping network has partitioned. It also shows that employing a retry strategy, where another target is picked at random when a

[2]See http://setiathome.ssl.berkeley.edu.

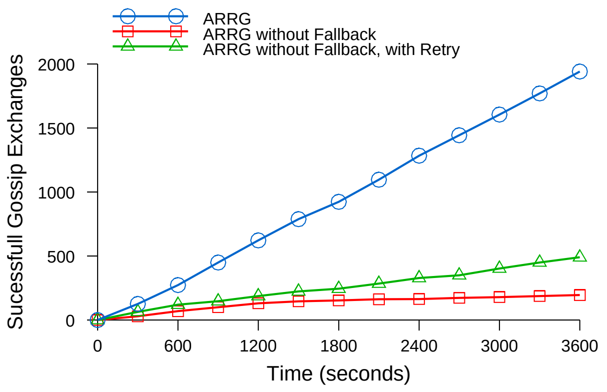

Figure 6: Gossip exchanges for ARRG in the X@Home scenario

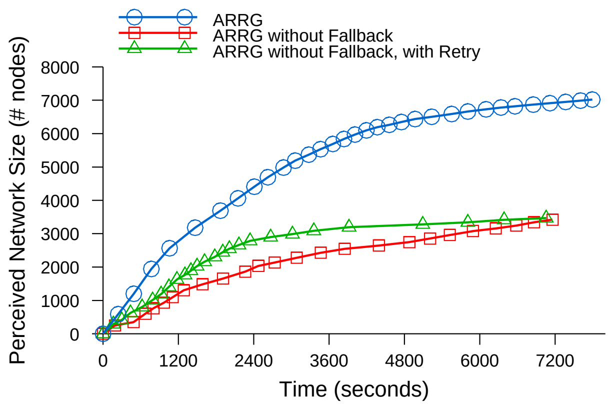

Figure 7: ARRG in a 8000 node simulation of the X@Home scenario

gossip attempt fails, also does not result in a viable protocol. However, when a Fallback cache is used, ARRG is able to perceive the entire network, showing almost identical performance to the fully connected reference case shown in Figure 4.

To clarify the underlying reasons for this difference in performance between the various versions of ARRG, Figure 6 shows the number of successful gossip exchanges for each version. These include both gossip exchanges initiated by the node and incoming requests from other nodes. The graph clearly shows that the Fallback Cache version of the algorithm performs almost an order of magnitude more successful gossip exchanges than the other versions. A large percentage of entries in the cache of each version are home nodes, which causes a gossip attempt to fail when selected. With the Fallback Cache in place, the algorithm will always choose a valid node in the second attempt. In the version without a Fallback Cache and with the retry version, there is no such guarantee, causing gossips to fail frequently.

To test the validity of our findings in a larger environment, we also ran a simulation of the X@Home scenario. The simulation considers 8000 nodes, consisting of 6400 home nodes

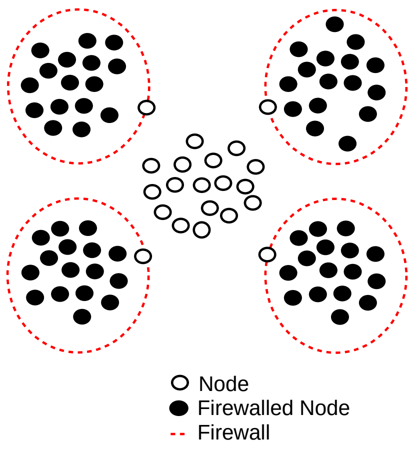

Figure 8: Grid use case Network Model

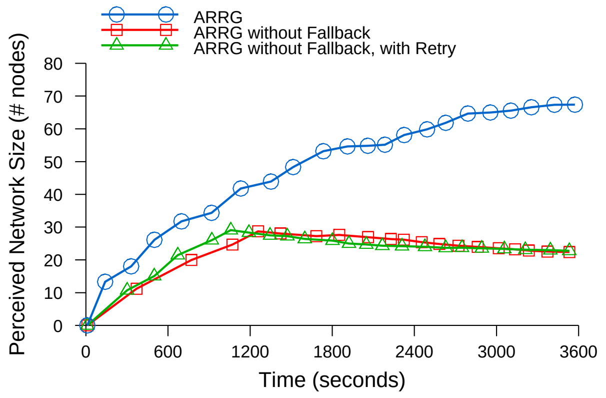

Figure 9: ARRG in the Grid scenario

and 1600 global nodes. To produce a information dissemination rate similar to that in Figure 5 we changed several parameters of the algorithm, as discussed in the introduction of this section. The cache size of each node was 100, and 30 entries were exchanged in each gossip. The size of the Fallback Cache was not increased, and remained at 10. Figure 7 shows the results of a simulated run of 2 hours. It shows that the performance of ARRG and the Fallback Cache technique are comparable to our real measurements.

6.3 The Grid Scenario

The last scenario where we test the Fallback Cache mechanism is the *Grid* scenario, depicted in Figure 8. This system consists of multiple separate clusters of machines which are protected by a firewall. This firewall will deny incoming connections from outside the cluster. However, connections between nodes inside a single cluster and outgoing connections are still possible. Each cluster has a single node which resides on the edge of the cluster, a so called *head node*. This node is able to receive incoming connections from the outside, as well as connect to the nodes inside the cluster. We implemented this setup using the SmartSockets library [11], which is able to selectively deny connections if requested using a configuration file. Again, nodes are not aware of these

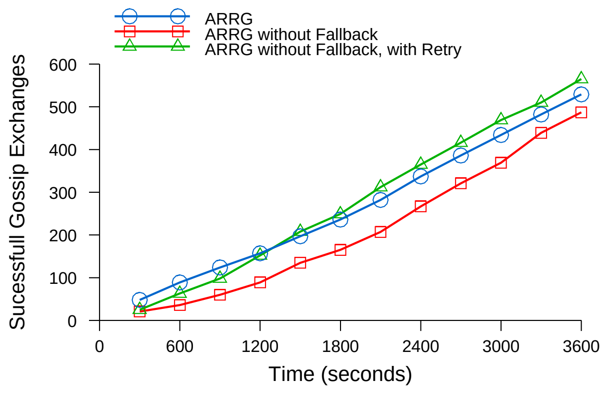

Figure 10: Gossip exchanges for ARRG in the Grid scenario

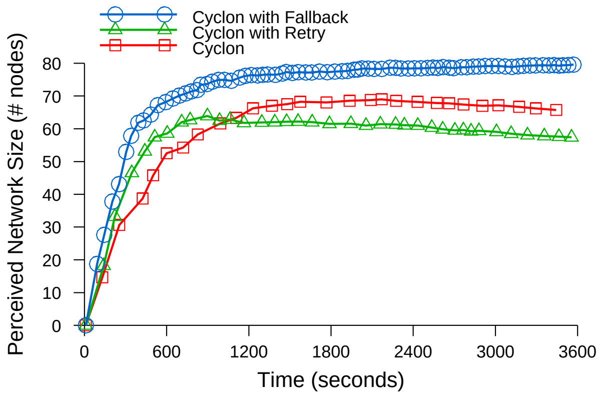

Figure 11: Cyclon in the X@Home scenario

restraints as they are enforced at the socket level. Our setup consisted of 4 clusters with 16 machines and one head node. The system also contained 17 *global* nodes, which were not protected by a firewall. The total number of machines in this system is therefore 85.

Figure 9 show the results for this experiment, measured at a global node. As expected, both the retry version of ARRG and the version without the Fallback Cache are not viable in this setting. From 1200 seconds onward, both versions show a decline of the Perceived Network Size. This shows that the gossiping network has partitioned, and nodes are only gossiping with a subset of the network. Analysis of the contents of the gossip caches shows that each node is in fact only communicating with nodes inside its own cluster. The Fallback mechanism, again, is able to compensate for the connectivity problems. The Fallback Cache contains a random subset of all *reachable* nodes, in this case both nodes within a node's own cluster and the global nodes in the system.

Figure 10 shows the number of successful gossip exchanges done by each algorithm. Unlike the previous experiment, there is only a small difference between them. In the previous experiment, failure of the protocol was caused by a *lack of reachable nodes*. In the grid case, being able to reach nodes is not sufficient. These nodes also need to be distributed correctly. The retry version of ARRG and the version without the Fallback Cache are only gossiping with nodes inside their own cluster, causing the network to partition.

Notice the performance of the Fallback Cache in the grid scenario is slightly less than its performance in the X@Home and fully connected scenarios. This is due to the fact that the cache of the algorithm is not able to determine the difference between nodes inside its own cluster and global nodes. With this information, the number of gossips within its own cluster could be limited. This is important because nodes in the local cluster usually have a tendency to contain a less random set of the world than global nodes. Cluster information is available in the gossiping algorithms that use structure such as Astrolabe [12], but this information needs to be supplied manually. We leave automatically determining and exploiting network structure as future work.

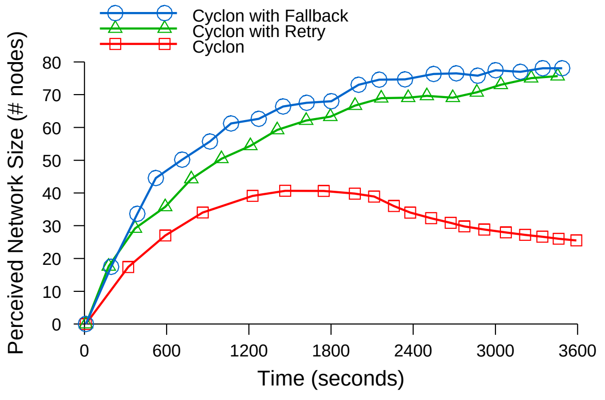

Figure 12: Cyclon in the Grid scenario

6.4 Cyclon

To determine the effectiveness of our Fallback mechanism in other algorithms than ARRG, we also tested it with the Cyclon [17] gossiping algorithm. We compared three different versions of Cyclon, the regular Cyclon algorithm, a version which retries once in case of a failure, and a version including our novel Fallback cache. Figure 11 shows the result for the X@Home scenario (on a global node). In this case the normal version of Cyclon is not viable. Although it almost reaches the entire network size, the Perceived Network Size quickly starts declining. Over time, the Cyclon network partitions. The same was observed for the retry version. With the addition of the Fallback Cache, Cyclon performs very well in this scenario. The Fallback cache completely compensates for the connectivity problems, and Cyclon is as fast as it was in the ideal situation (See Figure 3). Thus, with Fallback Cyclon is viable in the X@Home scenario.

Figure 12 shows Cyclon in the grid scenario. Normal Cyclon again fails, though much faster this time. It clearly creates a partition of the network, as the Perceived Network Size is converging slowly to the size of a single cluster. Cyclon with retry manages to overcome the network limi-

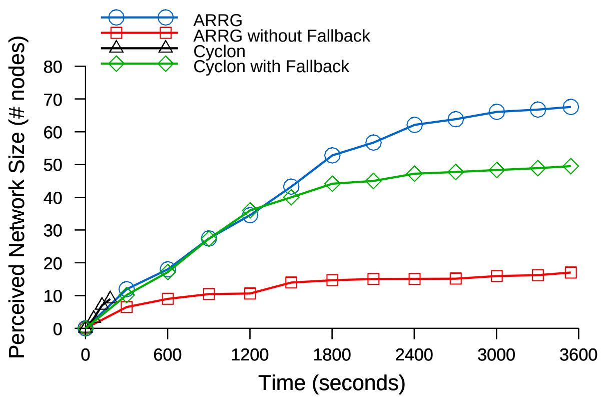

Figure 13: ARRG and Cyclon in the X@Home scenario, with 50% message loss

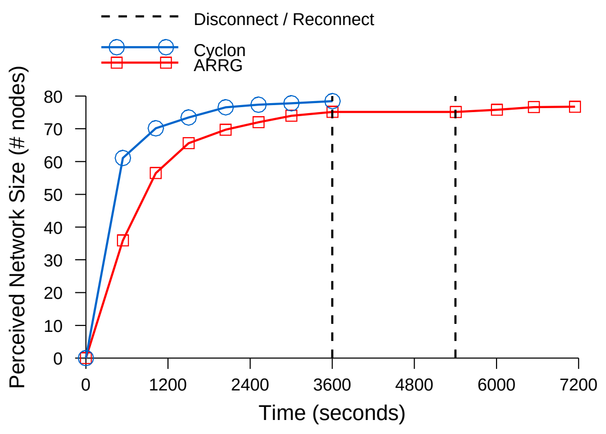

Figure 14: ARRG and Cyclon in the Disconnect scenario

tations. However, this is most likely due to the aggressive purging by Cyclon. In a slightly modified setting, for instance with fewer global nodes, the retry version will probably fail as well. The Fallback version of Cyclon again is viable, converging to the full network size.

6.5 Pathological Situations

By design, ARRG and its Fallback cache are robust against transient network problems such as congestion or link failure. This is achieved by retaining as much information as possible. Both ARRG's normal and Fallback caches do not remove invalid entries, but only replace entries with new valid entries as they become available. Reduced network performance therefore does not result in the loss of entries. To test this robustness we performed experiments in two pathological cases.

The first system setup is identical to the X@Home case, but in addition, we introduced 50% messages loss. This performance degredation could for instance occur in an environment where the network is congested, or where a poor wireless connection is present. We implemented the message loss at the socket layer. Figure 13 shows the results for both ARRG and Cyclon. Several conclusions can be drawn from this graph. First, the ARRG algorithm is able to overcome even this pathological case, without any noticeable performance degradation. Second, the Cyclon algorithm is not viable in this scenario. Even with the addition of a Fallback Cache, Cyclon does not perform well, and is not viable.

The reason for Cyclon's low performance can be found in the manner Cyclon handles failures. When a gossip attempt to a node fails, Cyclon removes its entry. For a gossip exchange to succeed both the request and the reply need to be delivered successfully. Since half of the messages are lost, the failure rate in this scenario is 75%. This causes Cyclon to quickly run out of entries in its cache. Adding a Fallback Cache only partially fixes this problem, as Cyclon is unable to do enough successful exchanges to fill its cache with valid entries, and relies almost solely on the Fallback cache.

In the second test case nodes become disconnected from the network for a time. The results of this test are shown in Figure 14. The setup is identical to the Fully connected scenario of Section 6.1. After an hour, 16 of the 80 nodes are disconnected from the network, and are neither able to reach the remaining 64 nodes nor the other disconnected nodes. After another half an hour the nodes are reconnected.

The graph shows ARRG recovers from the network problems, and Cyclon fails to resume gossiping after the network is restored. ARGG is able to resume as the normal and the Fallback cache still contain valid entries, as ARRG does not remove any entries during the disconnect. Cyclon does remove invalid entries, diminishing the number of entries in the Cyclon cache during the time the network is disconnected. At the time the network is restored, Cyclon does not have any entries in its cache left, and is thus not able to resume gossiping. We found that adding an additional Fallback cache makes Cyclon robust against this failure, as the Fallback cache still contains valid entries (not shown).

6.6 Summary

Table 2 provides a summary of all experiments performed in this section. It shows that ARRG with a Fallback cache is able to overcome all problems presented. It also shows that the Fallback cache is an invaluable part of ARRG, as without it ARRG does not function properly in all cases. Adding the Fallback cache to Cyclon significantly increases its robustness, making it robust against all problems, though in the scenario where we introduce message loss Cyclon's performance is reduced.

7. CONCLUSIONS AND FUTURE WORK

We studied the design and implementation of gossiping algorithms in real-world situations. We addressed the problems with gossiping algorithms in real systems, including connectivity problems, network and node failures, and non-atomicity. We introduced *ARRG*, a new simple and robust gossiping algorithm. The ARRG gossiping algorithm is able to handle all problems we identified by systematically using the simplest, most robust solution available for all required functionality. The *Fallback Cache* technique used in ARRG can also be applied to any existing gossiping protocol, making it robust against problems such as NATs and firewalls.

We introduced a new metric for the evaluation of gossiping algorithms: *Perceived Network Size*. It is able to clearly

		Fully Connected	X@Home	Grid	Message Loss	Disconnect
Cyclon		+	-	-	-	-
Cyclon	Retry	+	-	+/-	-	-
Cyclon	With Fallback	+	+	+	+/-	+
ARRG	No Fallback	+	-	-	-	+
ARRG	Retry	+	-	-	-	+
ARRG		+	+	+	+	+

+	=	pass
-	=	fail
+/-	=	reduced performance

Table 2: Results of experiments for different scenarios and algorithms

depict the performance of an algorithm, without requiring information from all nodes in the network. We evaluated ARRG, in several real-world scenarios. We showed ARRG performs well in general, and better than existing algorithms in situations with limited connectivity. In a pathological scenario with a high loss rate and 80% of the nodes behind a NAT system, ARRG still performs well, while traditional gossiping techniques fail.

Future work includes automatically discovering and exploiting the structure of a network. Also, we want to compare the usage of gossiping techniques to alternatives such as broadcast trees, by using ARRG in a number of different applications, including a distributed registry, and grid programming models. These applications could be evaluated by deployment on large scale systems.

8. ACKNOWLEDGMENTS

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment). We kindly thank Jason Maassen, Kees van Reeuwijk, Ceriel Jacobs, Kees Verstoep and Frank Seinstra for all their help. We also like to thank the anonymous reviewers for their insightful and constructive comments.

9. REFERENCES

[1] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *PODC '05: Proc. of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, New York, NY, USA, 2005. ACM Press.

[2] K. P. Birman, R. v. Renesse, and W. Vogels. Navigating in the Storm: Using Astrolabe to Adaptively Configure Web Services and Their Clients. *Cluster Computing*, 9(2):127–139, 2006.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proc. of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[4] N. Drost, R. V. van NieuwPoort, and H. E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proc. of GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing*, Singapore, may 2006.

[5] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[6] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From epidemics to distributed computing. *IEEE Computer*, 37(5):60–67, 2004.

[7] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[8] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast computing. Technical report, Vrije Universiteit Amsterdam, Department of Computer Science, 2003.

[9] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *EDCC-3: Proc. of the Third European Dependable Computing Conference on Dependable Computing*, pages 364–379, London, UK, 1999. Springer-Verlag.

[10] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *DISC '00: Proc. of the 14th International Conference on Distributed Computing*, pages 253–267, London, UK, 2000. Springer-Verlag.

[11] J. Maassen and H. E. Bal. Solving the Connectivity Problems in Grid Computing. In *Proc. of The 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Monterey, CA, USA, June 2007. Accepted for Publication.

[12] R. v. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[13] R. v. Renesse, Y. Minsky, and M. Hayden. A gossip-based failure detection service. In *Middleware'98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, England, September 1998.

[14] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flash crowds. In *ICNP '02: Proc. of the 10th IEEE International Conference on Network Protocols*, pages 226–235, Washington, DC, USA, 2002. IEEE Computer Society.

[15] G. Tan, S. A. Jarvis, X. Chen, and D. P. Spooner. Performance analysis and improvement of overlay construction for peer-to-peer live streaming. *Simulation*, 82(2):93–106, 2006.

[16] S. Voulgaris. *Epidemic-Based Self-Organization in Peer-to-Peer Systems.* PhD thesis, Vrije University Amsterdam, 2006.

[17] S. Voulgaris, D. Gavidia, and M. Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[18] S. Voulgaris, M. Jelasity, and M. van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Proc. of the 2nd International Workshop on Agents and Peer-to-Peer Computing (AP2PC03)*, Melbourne, Australia, July 2003.

[19] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of the International Conference on Distributed Computing Systems*, pages 5–14, Vienna, Austria, July 2002.