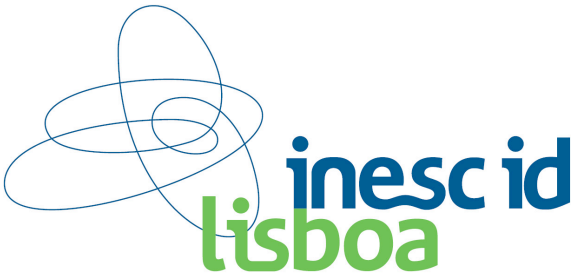


Technical Report RT/36/2009

Overnesia: a Robust Overlay Network for Virtual Super-Peers

João Leitão
INESC-ID/IST
jleitao@gsd.inesc-id.pt

Luis Rodrigues
INESC-ID/IST
ler@ist.utl.pt

Jul 2009

Abstract

Unstructured P2P networks have been widely used to implement resource location systems that support complex queries semantics. Unfortunately these systems usually rely on search algorithms based on some variant of flooding, which generate a significant amount of duplicate messages. An effective way to minimize the cost of query flooding in unstructured P2P networks is the use of super-peers.

On the other hand, super-peers may become overloaded or may fail, and have a negative impact on the performance and connectivity of the overlay. These risks can be circumvented by replicating super-peers. Replication serves the dual purpose of supporting load distribution and fault-tolerance purposes. This paper proposes a novel algorithm to construct an overlay network connecting replicated super-peers. We have called the resulting overlay, Overnesia. The paper also proposes techniques to perform query routing that leverage on the unique properties of Overnesia to effectively distribute the query processing load among replicas.

This technical report replaces the following document: INESC-ID Tec. Rep. 56/2008, Dec 2008.

Overnesia: a Robust Overlay Network for Virtual Super-Peers

João Leitão
INESC-ID / IST
jleitao@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID / IST
ler@ist.utl.pt

***Abstract*—Unstructured P2P networks have been widely used to implement resource location systems that support complex queries semantics. Unfortunately these systems usually rely on search algorithms based on some variant of flooding, which generate a significant amount of duplicate messages. An effective way to minimize the cost of query flooding in unstructured P2P networks is the use of super-peers.**

On the other hand, super-peers may become overloaded or may fail, and have a negative impact on the performance and connectivity of the overlay. These risks can be circumvented by replicating super-peers. Replication serves the dual purpose of supporting load distribution and fault-tolerance purposes. This paper proposes a novel algorithm to construct an overlay network connecting replicated super-peers. We have called the resulting overlay, Overnesia. The paper also proposes techniques to perform query routing that leverage on the unique properties of Overnesia to effectively distribute the query processing load among replicas.

I. Introduction

There are two main approaches to build peer-to-peer (P2P) overlays: structured and unstructured approaches. Structured approaches such as DHTs [13], [19] are very efficient to support exact queries but may exhibit poor performance in face of high dynamics in the number of participants, due to the concurrent entry and departure/failure of multiple nodes (a phenomena often known as churn [12], [16]). Additionally, they do not bring significant advantages for very complex queries [3], [6]. Unstructured approaches have the advantages of having a simpler design due to the lack of constraints on node location in the overlay topology. They also have the potential to be more resilient, both to churn and node failures. Usually, queries in unstructured P2P overlays are implemented using search algorithms that rely on some sort of (blind or informed) flooding [7], [11].

An efficient way to minimize the cost of query flooding in unstructured P2P networks is the use of super-peers [6], [18]. An unstructured P2P network using super-peers has a two-tier hierarchical structure: at the higher level, super-peers organize themselves in an unstructured overlay; at the lower level, regular peers connect to one or more super-peers. Typically, each super-peer maintains a consolidated index for all the regular peers that are attached to it. Therefore, queries only need to be flooded in the super-peer overlay.

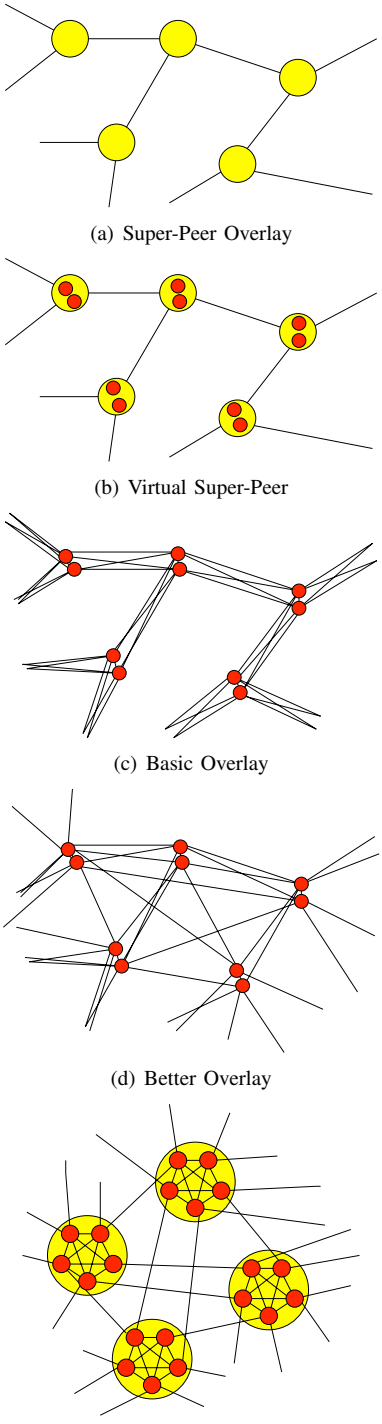
Unfortunately, the use of super-peers makes the overlay less robust, as the recovery from a super-peer failure may have a non-negligible cost: regular nodes must find new suitable super-peers and, then, consolidate index(es) need to be be rebuilt. Moreover, super-peer based overlays are less robust to informed attacks, as it is enough to attack the super-peers to disrupt the overlay operation. Finally, super-peers may become bottlenecks in the system, as they have to process a large number of queries.

One way to solve the problems mentioned above is to replicate super-peers. If done appropriately, such replication may bring several advantages. To start with, the overlay may become more robust and harder to attack, as one needs to disrupt all the replicas of a super-peer to cause that peer's consolidated index to be rebuilt from scratch. Secondly, query processing may be shared among the different replicas of the super-peers, increasing the capacity of the super-peer overlay.

A naive approach to construct an overlay of replicated super-peers would be to depart from an overlay of non-replicated super-peers (Figure 1(a)) and substitute each node by a *virtual super-peer* (Figure 1(b)) constructed by replicating the original node and the links to its neighbors (Figure 1(c)). However, such naive approach is far from fully exploiting the potential benefits of replication, as the extra redundancy in terms of nodes is not leveraged to increase the connectivity among virtual super-peers. If each replica is connected to a *different* virtual super-peer, the resulting overlay becomes more connected, as a result of the existence of additional diverse paths among virtual nodes, which can help to increase resilience, decrease the network diameter, and offer better load-distribution during query processing. Such richer overlay is depicted in Figure 1(d).

Although, for simplicity, in the previous figures we have used a replication degree of 2, we can generalize the idea for larger replication degrees. The resulting overlay would then look like the overlay illustrated in Figure 1(e): an overlay consisting of islands of fully connected nodes (that constitute virtual super-peers) and where islands are connect among each other. We have called this overlay: *Overnesia*.

This paper has two main contributions. First, it presents an algorithm to construct Overnesia. The algorithm is fully decentralized and executed by each super-peer. It ensures that the super-peers auto-organize to build a network of well connected virtual super-peers. Moreover, the protocol achieves a controlled distribution of super-peers in clusters, such that each virtual super-peer has approximately the same number of replicas. *Overnesia* can handle dynamic systems where nodes can leave, join or fail at any moment. Second, it proposes and analyzes different strategies to perform query routing over



(a) Super-Peer Overlay

(b) Virtual Super-Peer

(c) Basic Overlay

(d) Better Overlay

(e) Overnesia

Fig. 1. An overlay of virtual super-peers

the resulting *Overnesia*. These strategies aim at ensuring that queries cover all virtual super-peers, as a flood in a non-replicated super-peer overlay would, but in such a way that the query load is distributed among the super-peers that constitute each virtual super-peer and among the links that connect them.

The remaining of the paper is organized as follows. The algorithm to construct Overnesia is described in Section II and the query routing strategies introduced in Section III. The performance of all these algorithms is analyzed in Section IV and later compared with related work in Section V. Finally, Section VI concludes the paper and gives some pointers for future work.

II. Overnesia

A. Rationale

The main goal of Overnesia is to build an overlay network (of super-peers) with the following characteristics: i) each peer belongs to a single cluster called a *Nesos*[1] (each *Nesos* has an identifier named *cID*); ii) each peer knows the identity of all other members of its *Nesos* (denoted by *nView*) and maintains a link to each of these neighbors (thus, *Nesoi* are fully connected); iii) each peer maintains links to other *Nesoi*. The desired target size of each *Nesos* is a protocol parameter; this allows our protocol to be used in a wide range of applications with different fault tolerance requirements.

A key aspect of our design is that Overnesia does not attempt to ensure that each *Nesos* has exactly the desired target size. Such goal would be very hard to achieve in large scale dynamic environments, where multiple joins, leaves, and failures may happen concurrently. In fact, the behavior of Overnesia is controlled by the following parameters:

- *Target* Nesos *Size* ($\texttt{NS}^T$)*:* The target desirable number of *Nesos* neighbors kept by each node. Members of a *Nesos* attempt to prevent the further growth of the *Nesos* as soon as it reaches a size of $\texttt{NS}^T + 1$.
- Nesos *Max Size* ($\texttt{NS}^{MAX}$)*:* If the size of a *Nesos* becomes larger than threshold $\texttt{NS}^{MAX} + 1$, *Nesos* members coordinate to split it into two distinct (smaller) *Nesoi*.
- Nesos *Minimum Size* ($\texttt{NS}^{MIN}$)*:* If the size of a *Nesos* becomes smaller than threshold $\texttt{NS}^{MIN} + 1$, *Nesos* members gradually try to abandon the *Nesos* to join a (larger) *Nesos*, causing the graceful fading of the smaller *Nesos*.

Besides controlling the size of *Nesos*, Overnesia strives to promote the creation of multiple, distinct, inter-*Nesos* connections. The existence of these diverse connections has the following advantages: makes the global overlay robust to node and link failures, reduces the clustering among different *Nesoi*, and lowers the diameter of the overlay (allowing queries to be disseminated more efficiently). For this purpose, each peer maintains an external view with identifiers of nodes located in other existing *Nesoi* (called the *eView*, whose size θ is also a protocol parameter). As it will become clear later in the text, the fact that each *Nesos* has a (probabilistic) unique identifier

[1] *Nesos* is the Greek word for island, whereas *Nesoi* is the plural (islands).

eases the task of balancing inter-*Nesos* neighboring relations among nodes of a given *Nesos*.

Each node maintains a link to all its internal (*nView*) and external neighbors (*eView*). These links are maintained using a TCP connection, that is used for all communication between the two connecting peers. The use of TCP is motived by two reasons: *i)* it allows the communication between peers to be network friendly as we leverage in TCP flow control mechanisms. Moreover, we can model the system without considering message losses between peers; *ii)* TCP is used as an unreliable failure detector, has it has been previously proposed by the authors of [10]. The failure detector is used, for instance, to improve the accuracy of queries, by allowing to remove from the overlay the indexes of failed nodes in a timely manner, and also to adapt the overlay in face of node failures.

Additionally, each node owns an additional view, named *pView*, containing additional random peer identifiers. This view is used as a backup list to cope with node failures.

B. Algorithm

Algorithm 1: Overnesia Protocol Overview

```
1:  upon event Init do
2:      trigger Join Procedure

3:  every ΔT1 + random(π) do
4:      if #nView > NS^MAX then
5:          trigger Divide Procedure
6:      else if #nView ≤ NS^MIN
7:          trigger Collapse Procedure

8:  every ΔT2 do
9:      if # external neighbors < θ
10:         trigger External Neighboring Procedure

11: every ΔT3 with a probability ρ do
12:     trigger Anti-entropy Procedure
```

Overview We start by providing a macroscopic perspective on the operation of the algorithm. The protocol to build Overnesia can be explained by the combination of five complementary sub-protocols: a *Join Procedure* used to join nodes to the overlay; a *Divide Procedure* used to prevent the size of *Nesos* to exceed the NS^{MAX} threshold; a *Collapse Procedure* used to gracefully eliminate *Nesoi* of very small size; an *External Neighboring Procedure* in charge of promoting the creation of good inter-*Nesos* links; and, finally, a Nesos *Anti-entropy Procedure* used to maintain the consistency of the intra-*Nesos* information. Algorithm 1 shows when these components are invoked.

The reader should note that there are some dependencies among these sub-protocols. To reduce the cost, and increase the robustness and parallelism of the join procedure, the *Nesos* size is allowed to temporarily exceed its upper threshold, in face of multiple concurrent join requests. This will trigger (at some random time, to avoid global synchronization) the execution of the divide procedure. On the other hand, node failures may bring a *Nesos* size below the lower threshold, triggering the *Nesos* collapse procedure. Failures also affect the number of active external neighbors maintained by each node. In this case, the external neighboring procedure is used to locate new external neighbors, such that each node reaches the target *external degree* (θ). Finally, due to expected high concurrency, nodes may have inconsistent views of their current *Nesos* filiation. To address such scenarios, and also to help *Nesos* members to balance their external neighboring relations, periodically with a given probability ρ, every node executes an anti-entropy procedure where it exchanges information with a random *Nesos* neighbor.

In the following paragraphs we will describe in more detail each component of the Overnesia protocol, illustrating its operation with simplified pseudo-code, when adequate.

Join Procedure In order to join the overlay, a new node a sends to a node that already belongs to the overlay (called the *contact*[2] node) a JOIN request. This request is forwarded in the overlay using a limited length random walk, using preferably links to distinct *Nesoi*. The random walk terminates when a *Nesos* with size smaller than NS^{T} is found. If no such *Nesos* is found before the random walk time to live expires, the new element is added to the *Nesos* where the random walk ends, regardless of its size. When a is accepted into a *Nesos*, a JOINREPLY message is sent to a, that uses the information on the message to update its *cID* identifier and to establish neighboring relations with all remaining members of the *Nesos* by sending NEIGHBORINGREQUEST messages. This procedure is depicted in Algorithm 2.

Divide Procedure This procedure is executed when the size of a *Nesos* exceeds the threshold NS^{MAX} and its purpose is to split a *Nesos* into two smaller *Nesoi*. The intuition behind the division of *Nesos* as a mechanism to generate new *Nesoi* is that it avoids the creation of a large number of small (potentially unitary) *Nesoi*. By dividing a large *Nesos* in two, similar to what living cells do, one can generate two stable and independent *Nesoi*, with a low impact on the overlay connectivity.

The algorithm, depicted in Algorithm 3, is initiated by the *Nesos* member with smaller identifier when, after a periodic test, it detects that the size of its local nView is equal or above the parameter NS^{MAX}. The initiator generates two new random *Nesos* identifiers (namely ID_a and ID_b) and divides the current *Nesos* membership in two ordered sets a and b, sending this information to the remaining *Nesos* members in a NESOSDIVISION message.

When a NESOSDIVISION message is received it is put in *quarantine*, for a period of time greater than twice the maximum RTT. The message is stored in a set named *pending division vector*, or simply *pdv*. The *quarantine* period aims at avoiding that multiple concurrent *Nesos* divisions are initiated when the *Nesos* view is not fully consistent (and more than one node believes to have the lower identifier). At the end of the quarantine period, the NESOSDIVISION message is accepted if

[2]The first node to join the overlay is an obvious exception: it only has to generate a random *Nesos* identifier.

Algorithm 2: Join Procedure

```
1:  upon event Init do
2:      cID ⟵ ⊥
3:      trigger Send(JOIN, contact)

4:  upon event Receive(JOIN, newNode) do
5:      if #nView < NS^T then
6:          trigger Send(JOINREPLY, sender, cID, nView )
7:          nView ⟵ nView ∪ sender
8:      else
9:          n ⟵ n ∈ eView or n ∈ nView
10:         trigger Send(FORWARDJOIN, n, sender, TTL)

11: upon event Receive(FORWARDJOIN, sender, newNode, ttl) do
12:     ttl ⟵ ttl −1
13:     if #nView < NS^T or ttl = 0 then
14:         trigger Send(JOINREPLY, newNode, cID, nView )
15:         nView ⟵ nView ∪ newNode
16:     else
17:         n ⟵ (n in eView or n in nView) and n ≠ sender
18:         trigger Send(FORWARDJOIN, n, newNode, TTL)

19: upon event Receive(JOINREPLY, sender, id, view) do
20:     cID ⟵ id
21:     ∀ n ∈ view do
22:         trigger Send(NEIGHBORINGREQUEST, n, cID)
23:         nView ⟵ nView ∪ n

24: upon event Receive(NEIGHBORINGREQUEST, sender, id) do
25:     if cID = id then
26:         nView ⟵ nView ∪ sender
27:     else
28:         trigger Send(DISCONNECTREQUEST, sender)

29: upon event Receive(DISCONNECTREQUEST, sender) do
30:     if sender ∈ nView then
31:         nView ⟵ nView \ sender
32:     else if sender ∈ eView then
33:         eView ⟵ eView \ sender
34:     if sender ∉ pView then
35:         pView ⟵ pView ∪ sender
```

Algorithm 3: Divide Procedure

```
1:  upon event CHECKNESOSSIZE TIMER do
2:      if cID ≠ ⊥ and pdv = ∅ then
3:          if #nView ≥ NS^MAX and ∄ n: n ∈ nView: n.nID < nID then
4:              ID_a ⟵ get new unique id
5:              ID_b ⟵ get new unique id
6:              a ⟵ {myself} ∪ SelectHalf(nView )
7:              b ⟵ nView \ a
8:              pdv ⟵ NESOSDIVISION (myself, cID, ID_a, ID_b, a, b)
9:              ∀ n ∈ nView do
10:                 trigger Send(NESOSDIVISION, n, cID, ID_a, ID_b, a, b)
11:             setup timer (EXECUTENESOSDIVISION TIMER, RTT ∗2)

12: upon event EXECUTENESOSDIVISION TIMER do
13:     if pdv ≠ ∅ then
14:         s ⟵ s ∈ pdv → ∄ x: x ∈ pdv ∧ x.sender.nID < s.sender.nID
15:         if myself ∈ s.a then
16:             ∀ n ∈ nView do
17:                 if n ∈ s.a then
18:                     trigger Send(NESOSUPDATE, n, cID, s.ID_a, true)
19:                 else if position(myself,s.a) = position(n, s.b) then
20:                     trigger Send(NESOSUPDATE, n, cID, s.ID_a, false)
21:                     nView ⟵ nView \ n
22:                     eView ⟵ eView ∪ n
23:                 else
24:                     trigger Send(DISCONNECTREQUEST, n)
25:                     nView ⟵ nView \ n
26:             cID ⟵ s.ID_a
27:         else if myself ∈ s.b then
28:             ∀ n ∈ nView do
29:                 if n ∈ s.b then
30:                     trigger Send(NESOSUPDATE, n, cID, s.ID_b, true)
31:                 else if position(myself,s.b) = position(n, s.a) then
32:                     trigger Send(NESOSUPDATE, ( n), cID, s.ID_b, false)
33:                     nView ⟵ nView \ n
34:                     eView ⟵ eView ∪ n
35:                 else
36:                     trigger Send(DISCONNECTREQUEST, n)
37:                     nView ⟵ nView \ n
38:             cID ⟵ s.ID_b
39:         pdv ⟵ ∅

40: upon event Receive(NESOSUPDATE, sender, ID_old, ID_new, isNesos) do
41:     if isNesos = true then
42:         if cID ≠ ID_new then
43:             if pdv = ∅ or cID ≠ ID_old then
44:                 trigger Send(DISCONNECTREQUEST, sender)
45:                 nView ⟵ nView \ sender
46:     else
47:         if sender ∈ eView then
48:             if sender.cID ≠ ID_new then
49:                 if ID_new = cID then
50:                     eView ⟵ eView \ sender
51:                     nView ⟵ nView ∪ sender
52:                 else
53:                     update local information on cID of sender
54:         if pdv = ∅ or sender ∉ nView then
55:             trigger Send(DISCONNECTREQUEST, sender)
56:             nView ⟵ nView \ sender
57:     if pdv ≠ ∅ then
58:         trigger EXECUTENESOSDIVISION TIMER

59: upon event Receive(NESOSDIVISION, sender, ID, IDa, IDb, a, b ) do
60:     if cID = ID then
61:         if pdv = ∅ then
62:             setup timer (EXECUTENESOSDIVISION TIMER, RTT ∗2)
63:         pdv ⟵ pdv ∪ NESOSDIVISION (sender, cID, IDa, IDb, a, b)
```

no other NESOSDIVISION message has been received from a node with smaller identifier. Notice that the quarantine period might be hard to calculate in highly dynamic environments. However in situations were this mechanism fails, it only results in the temporary disconnection of a very small number of nodes. These nodes can rejoin the overlay, for instance relying in the pView (which we describe later).

When a NESOSDIVISION message is accepted by a node d, it adopts the *Nesos* division proposal included in the message. Thus, it updates its local cluster identifier. Moreover, d sends NESOSUPDATE messages to all nodes of its new *Nesos* to speed the convergence of the algorithm (in case they have not adopted yet the originating NESOSDIVISION message). Finally, node d sends a DISCONNECTREQUEST to all nodes that do not belong to its new *Nesos*, except to the node d' that occupies the same position in the complementary *Nesos* filiation set in the NESOSDIVISION message. To node d', d sends a request to establish a inter-cluster link; this ensures that the two new *Nesoi* remain well connected to the rest of the overlay.

Collapse Procedure This procedure is used to gracefully disband a *Nesos* whose size has fallen below the threshold parameter $\mathrm{NS}^{MIN} + 1$, by migrating its members to other, more suitable, *Nesoi*. This procedure is decentralized. Each node makes a periodic test and if it notices that its cluster size is too small it takes the initiative to relocate itself to another *Nesos*, resulting in the collapse of its older *Nesos*. To avoid abrupt collapse of a *Nesos*, nodes only decide to initiate the procedure with a given probability p, which increases as the size of the *Nesos* decreases. Algorithm 4 depicts this procedure. A RELOCATEREQUEST message is propagated in

a manner similar to the join procedure described before. Notice however that, if the local *Nesos* size has become stable meanwhile, the source of the relocation request cancels its relocation by issuing a DISCONNECTREQUEST message to the node that replies with a RELOCATEREPLY message.

Algorithm 4: Collapse Procedure

```
1:  upon event CHECKNESOSSIZE TIMER do
2:      if #nView < NS^MIN then
3:          with a probability of: (1 − #nView/NS^MIN) do
4:              n ⟵ n ∈ eView or n ∈ pView or n ∈ nView
5:              trigger Send(RELOCATEREQUEST, n, myself, cID, TTL)

6:  upon event Receive(RELOCATEREQUEST, sender, node, ID, ttl) do
7:      ttl ⟵ ttl −1
8:      if cID ≠ ID and #nView ≤ NS^T then
9:          nView ⟵ nView ∪ node
10:         if ttl > 0 then
11:             trigger Send(RELOCATEREPLY, node, cID, nView )
12:     else if ttl > 0 then
13:         n ⟵ n ∈ eView or n ∈ nView
14:         trigger Send(RELOCATEREQUEST, n, node, ID, ttl)

15: upon event Receive(RELOCATEREPLY, sender, id, reloc_view) do
16:     if #nView < NS^MIN then
17:         ∀ n ∈ nView do
18:             trigger Send(DISCONNECTREQUEST, n)
19:             nView ⟵ nView \ n
20:         ∀ n: n ∈ eView ∧ n.cID = id do
21:             trigger Send(NESOSUPDATE, n, cID, id, false)
22:             eView ⟵ eView \ n
23:             nView ⟵ nView ∪ n
24:         cID ⟵ id
25:         ∀ n: n ∈ reloc_view ∧ n ∉ nView do
26:             trigger Send(NEIGHBORINGREQUEST, n, cID)
27:     else
28:         trigger Send(DISCONNECTREQUEST, sender)
```

External Neighboring Procedure To ensure that Overnesia remains connected, the protocol attempts to maintain, at each peer, a pre-defined number θ of external neighbors, *i.e.*, links to other *Nesoi*. Thus, a node that has less than θ external neighbors actively tries to establish external links by sending an EXTERNALREQUEST message to the overlay. This message is propagated using a fixed-length random walk, that tries to find a suitable neighbor in another *Nesos*. An external neighbor is considered suitable if it also has less than θ external neighbors and does not belong to the *Nesos* of the sender, nor to other *Nesoi* to which the sender is already connected.

In the particular case when the source of the EXTERNALREQUEST message has no external neighbor, a special flag (named *empty*) is set to true in the random walk. In this case, if the random walk terminates before a suitable neighbor is found, the last visited node becomes a neighbor of the source, even if it already has θ external neighbors (and needs to disconnect from a random external neighbor in order to maintain a number of external neighbors equal to θ).

Anti-entropy Procedure In face of concurrent joins and crashes, the nView maintained by different nodes in the same *Nesos* may diverge. To increase the intra-*Nesos* consistency, a simple gossip-based anti-entropy procedure is executed inside the *Nesos*. Periodically, with a given probability ρ, every node n selects another peer p in the *Nesos* and sends to it a message

Algorithm 5: External Neighboring Procedure

```
1:  upon event CHECKEXTERNALCONNECTIVITY TIMER do
2:  if #eView < θ then
3:      k ⟵ ∅
4:      ∀ n ∈ eView do
5:          k ⟵ k ∪ n.cID
6:      d ⟵ d ∈ eView or d ∈ nView or d ∈ pView
7:      if eView = ∅ then
8:          trigger Send(EXTERNALREQUEST, d, myself, cID, k, true, TTL)
9:      else
10:         trigger Send(EXTERNALREQUEST, d, myself, cID, k, false, TTL)

11: upon event Receive(EXTERNALREQUEST, sender, node, ID, k, empty, ttl) do
12:     ttl ⟵ ttl −1
13:     if cID≠ID and cID ∉ k and #eView < θ and ∄ n ∈ eView: n.cID=ID do
14:         eView ⟵ eView ∪ node
15:         trigger Send(EXTERNALREPLY, node, cID, ID)
16:     else if ttl > 0
17:         d ⟵ d ∈ eView or d ∈ nView
18:         trigger Send(EXTERNALREQUEST, d, node, ID, k, empty, ttl)
19:     else if empty = true
20:         n ⟵ n ∈ eView
21:         trigger Send(DISCONNECTREQUEST, n)
22:         eView ⟵ eView \ n
23:         eView ⟵ eView ∪ node
24:         trigger Send(EXTERNALREPLY, node, cID, ID)

25: upon event Receive(EXTERNALREPLY, sender, ID, ID_k) do
26:     if #eView = θ then
27:         trigger Send(DISCONNECTREQUEST, sender)
28:     else
29:         eView ⟵ eView ∪ sender
30:         if ID_k ≠ cID then
31:             trigger Send(NESOSUPDATE, sender, ID_k, cID, false)
```

containing its own view of the *Nesos* current filiation[3]. This allows p to detect missing peers in its nView. Moreover if p detects some missing nodes in n's nView it replies to n with a similar message.

Additionally, anti-entropy is also used to balance the external neighbors, by lowering the number of nodes in a *Nesos* that hold external connections to a same remote *Nesos*. When sending the gossip message, n also sends the list of the *Nesos* identifiers of his external neighbors. If a node receives two consecutive gossip messages that refer to a *Nesos* of one of its external neighbors, it simply disconnects from that external neighbor, using a DISCONNECTREQUEST message. The reception of two gossips is required to promote some stability in the overlay network topology. The anti-entropy mechanism is only executed by nodes with an empty *pdv* set. This prevents the mechanism from being activated on a *Nesos* that is about to execute a divide procedure.

C. Increasing the Fault-Tolerance

To increase the fault-tolerance of Overnesia, we use an approach similar to the one described in [10], *i.e.*, we augment the state of each peer with a random, unbiased, partial view of the entire overlay. This view, called *pView* is maintained using a low cost background protocol, based on the exchange of shuffle messages. Moreover, whenever a node has to remove a correct peer from its nView or eView, or when it receives a

[3] ρ can be small; in our experiments we determined that a ρ value of 0.1 is adequate.

request sent by a peer which is not in one of those sets, that peer identifier can be added to the pView.

D. Replication of Consolidated Indexes

For simplicity we omitted from the description of Overnesia the mechanisms to replicate the consolidated indexes among peers that belong to the same *Nesos*. Such mechanisms are orthogonal to the main contributions of our paper and can be trivially implemented using either push or anti-entropy mechanisms in a layer on top of our overlay. That layer should be notified whenever a node is added or removed from Overnesia nView set. It can then maintain a copy of the consolidated index for each element of the virtual super-peer.

III. Query Routing Strategies

In this section we propose a number of query routing strategies that take into consideration the unique characteristics of Overnesia. Our purpose is not to thoroughly cover all possible query strategies; this is a very rich research topic on its own[3], [9], [17] that will be addressed in future work. Instead, we just aim at showing that one can indeed leverage on Overnesia properties to implement query strategies that outperform traditional blind search algorithms based on flooding in (regular) unstructured overlay networks.

The baseline strategy that is considered in this paper is query flooding in the overlay network of unreplicated super-peers. This strategy ensures that every super-peer is visited by the query procedure and that results are fully accurate (*i.e.*, if the search item exists in the overlay, then a match is guaranteed to be found). A limitation of this approach is that super-peers can be easily overloaded, given that they need to route and process each and every query. On the other hand, the replication of super-peers in Overnesia, provides not only fault-tolerance but also the opportunity for query processing load distribution among replicas of a *Nesos*. For that purpose, we need to devise query routing strategies that allow queries to visit each and every *Nesos* in the overlay while minimizing the number of members of the same cluster that are involved in the processing of each query (ideally, only one node from each *Nesos*).

***Nesos*-aware Flooding** *Nesos*-aware flooding operates by flooding the Overnesia overlay. Thus, if this strategy is used, when a node receives a query is forwards it to all its neighbors, except to the neighbor from which it received the query. However, a node is only required to process the query if it is received from a different *Nesos*, since all nodes in a *Nesos* share the same consolidated index. This prevents some amount of redundant processing of queries. Assuming that queries are issued to nodes in the super-peer overlay uniformly at random, this strategy should, intuitively, distribute the load of query processing uniformly among nodes. Additionally, when a query is received from a *Nesos* neighbor, the node avoids to forward it to the remaining nodes inside its *Nesos*, lowering the number of redundant messages transmitted during the dissemination.

Similarly to most query flooding protocols, *Nesos*-aware flooding uses a time-to-live (TTL) parameter to prevent the query to loop forever in the network. This value is decreased even if the message is forwarded inside a *Nesos*. This ensures that this strategy is comparable, in terms of performance, with regular flooding protocols.

***Nesos*-aware Gossip** *Nesos*-aware gossip operates by having each node to gossip a query to a limited number of *external* neighbors. This number is called the gossip fanout, f. To avoid forwarding a query to a *Nesos* that has already been visited, each query message carries the identifiers of the *Nesos* that have already been involved in the query processing. In detail, this query routing algorithm operates as follows. In the first gossip step, the query is propagated to *all* nodes of the *Nesos*. Then, in the second step, each member of the *Nesos* propagates the query to *all external* neighbors. This allows to expand the breadth of the query in the first gossip steps (see for instance, [2] for the advantages of such approach). Further gossip steps attempt to forward the query to f external neighbors from *Nesoi* that have not been visited yet; if a node has less than f external neighbors that meet this condition, the query is relayed to an internal neighbor.

The intuition behind this strategy is to expand the use of the abstraction of virtual super-peer, and have each virtual super-peer gossip with a given fanout of other existing virtual super-peers. Similar to other gossip protocols (for instance [4]) the protocol can be parameterized with a time-to-live and a fanout parameters. The fanout identifies the (maximum) amount of distinct *Nesoi* to which each *Nesos* should forward each received query. Similarly to the *Nesos*-aware flooding strategy above, a node which receives a query from a *Nesos* neighbor does not process it. Furthermore, the TTL value is not decremented when a query is relayed inside the *Nesos*.

Improved *Nesos*-aware Gossip The improved *Nesos*-aware gossip strategy is a variant of the strategy described above. The strategy combines gossip with random walks. In the first rounds, it uses gossip as described above. After a number G of gossip rounds, query propagation uses biased random walk (i.e., the fanout is reduced to 1 after G gossip rounds). Random walks are forwarded for an additional number of rounds, an additional parameter named *random walk time to live*. This strategy also uses the same rules applied to gossip to prevent a given *Nesos* from being visited twice. The goal is to minimize the message cost associated with query dissemination, while simultaneously striving to maximize the number of *Nesoi* that receive, and process, the query.

IV. Evaluation

A. Experimental Setting

We conducted an extensive experimental evaluation of both the overlay maintenance protocol and query routing strategies in the PeerSim simulator [8] using its event driven engine. To do this we have implemented the Overnesia protocol in this simulator as well as all query routing strategies proposed in this paper. To serve as a comparative baseline, we have

implemented an overlay network of (non-replicated) super-peers, using an extension of the Scamp protocol [5] (operating on top of TCP). The decision to use Scamp was based on the following three arguments: *i)* the complete specification of Scamp is published; *ii)* Scamp maintains relatively stable neighboring relations, being adequate to the use of TCP as transport protocol; and *iii)* in [6] the authors propose Scamp as an adequate protocol to maintain a super-peer overlay. Moreover, to compare our approach with a DHT in terms of resilience and complexity, we also implemented Chord [13] for the simulator using the same of assumptions employed in the remaining protocols.

All results presented in this paper are an average of results extracted from several independent executions of each simulation.

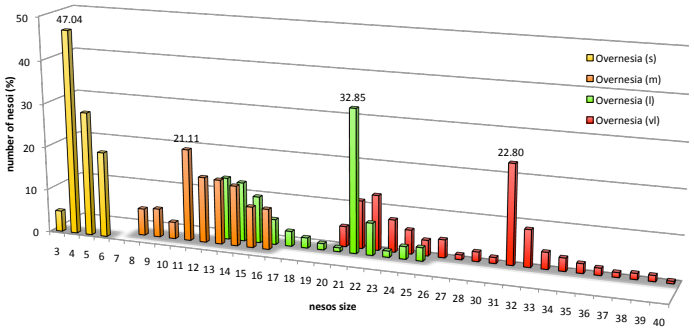
Experimental Parameters In our experiments we have a virtual clock that coordinates the delivery of events to nodes (and protocols). Message delay was configured to be uniformly distributed between 1.000 and 2.000 time units (TU). All experiments were conducted in a system composed of 10.000 nodes, with the values depicted in Table I for the relevant protocol parameters. We tested 4 different Overnesia configurations, considering different *Nesos* sizes. These configurations, which we refer to as *small* (s), *medium* (m), *large* (l), and *very large* (vl), are characterized by increasing values of the target and threshold values for *Nesos* size as reported in Table II. The reasons for using these distinct scenarios are twofold. Firstly, it allows us to demonstrate the flexibility of Overnesia. Secondly, it permits to observe the behavior of query routing strategies for distinct *Nesos* sizes. Similarly, we also experimented with 4 Chord configurations, where the number of backup successors was set to 30 (similar to the size of pView used for Overnesia), and the number of fingers maintained by each node was configured with the $\mathtt{NS}^T$ parameter used in each Overnesia configuration.

Parameter	Value
ΔT_1	20.000 TU
ΔT_2	20.000 TU
ΔT_3	10.000 TU
π	$0 - 20.000$ TU
probability ρ	0.1
TTL	10
pView size	30

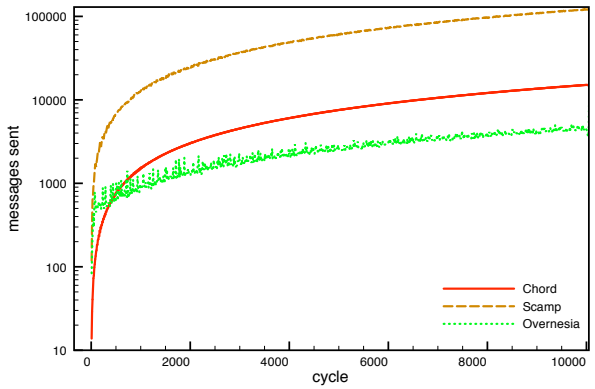
TABLE I
COMMON PARAMETERS FOR OVERNESIA

	small	medium	large	very large
$\mathtt{NS}^T$	3	10	20	30
$\mathtt{NS}^{MAX}$	6	16	25	40
$\mathtt{NS}^{MIN}$	1	6	10	15
θ	8	3	2	2

TABLE II
DIFFERENT TARGET *Nesos* SIZE



(a) *Nesos* Size Distribution



(b) Overlay Join Message Cost

Fig. 2. Overlay properties

B. Overlay Characterization

One of the goals of Overnesia is to allow members of any *Nesos* to act as a single, replicated, super-peer. Replication increases the dependability of the overlay and the potential for distributing the load of query processing among the members of a *Nesos*. To that end, the size of *Nesoi* should respect the configuration of the protocol. Moreover, to ensure the efficiency of the protocol, the complexity in terms of messages required to achieve the Overnesia topology should be as low as possible.

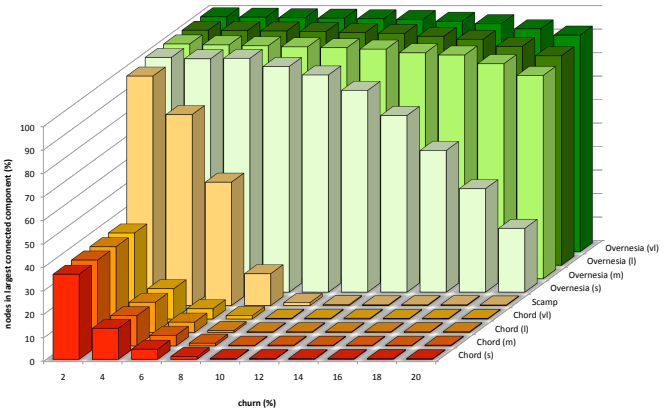
To evaluate such aspects of Overnesia, we conducted simulations where each node joins the overlay in sequence, using a random peer already present in the overlay as contact. Figure 2(a) depicts the distribution of *Nesos* size for the 4 distinct Overnesia configurations. Notice that for all configurations, the most common size for *Nesoi* is the target value of $\mathtt{NS}^T + 1$. As expected, no *Nesos* surpasses the size of $\mathtt{NS}^{MAX}$. Interestingly enough, several *Nesoi* in all configurations present a size below that of $\mathtt{NS}^T + 1$. These *Nesoi* are a result of the *Nesos* division procedure. Notice that, for all configurations of Overnesia, the second most common *Nesos* size is $\mathtt{NS}^{MAX}/2$.

This gives us additional insights over the ideal configuration for the Overnesia protocol. Namely *Nesos* size will be typically distributed between $\mathtt{NS}^{MAX}/2$ and $\mathtt{NS}^{MAX}$. Moreover, to promote stability in the overlay topology, the parameter $\mathtt{NS}^{MIN}$ should be set below $\mathtt{NS}^{MAX}/2$. Finally, to increase

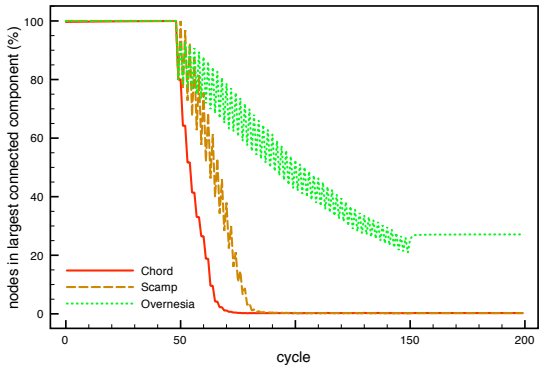
the number of *Nesos* with a size of $\texttt{NS}^T+1$, parameter $\texttt{NS}^{MAX}$ should be set close to $2\ \texttt{NS}^T + 1$, in order to take advantage of the *Nesos* division procedure.

Figure 2(b) show the number of sent messages in each simulation cycle when nodes are joining the overlay. Notice that in each cycle a new node is added to the system. Results are presented for Overnesia, Scamp and Chord. Because message cost is somewhat independent from the protocol configuration, we only depict values for the *very large* Overnesia and *small* Chord configurations. Overnesia presents much lower values than Scamp or Chord. This is mostly due to the design of Overnesia, that relies on low cost mechanisms to build and maintain the overlay, and also due to its soft constraints over the topology (*i.e.* without the hard constraints typical of DHTs).

C. Fault Tolerance



(a) Final Connectivity



(b) Connectivity Under 20% Churn

Fig. 3. Overlay Connectivity Under Churn

Any P2P overlay designed to operate in a large scale environment (*e.g.* over the internet) should ensure that its connectivity is not hampered by the presence of churn.

Experiments were performed by first creating the overlay (having nodes join as described before) and then, after a stabilization period of 50 cycles, different quantities of churn were induced (ranging from 2% to 20%) during a period of 100 simulation cycles. To induce a given percentage p of churn in the overlay, in every other cycle during the churn period p percent of nodes fail simultaneously, after which the same number of nodes join the overlay concurrently. When the churn period is over, a stabilization period of 50 cycles is executed and the percentage of nodes that belong to the largest connected component of the overlay is measured. Results are summarized in Figure 3(a).

Chord connectivity drops to values bellow 40% (for all configurations) with only 2% churn. Furthermore, it drops to connectivity values below 10% for churn values above 6%. This happens because Chord mechanism to locate fingers is slow. Therefore, it cannot operate in such a dynamic environment without degenerating into several disconnected rings. Scamp on the other hand can maintain 60% of the overlay connected for churn values of 6%, reaching connectivity values below 10% with churn values above 10%. Scamp uses a lease time that enforces nodes to rejoin the overlay in order to recover from failures. Such lease time cannot be too small, otherwise the overlay becomes unstable. In sharp contrast, the reactive nature of Overnesia, that leverages on TCP as an unreliable failure detector, combined with fast recovery mechanisms, is able to sustain high quantities of churn. All but the *small* configuration are able to maintain a connectivity above 90%, even for churn values as high as 18%. The *small* configuration, due to the smaller number of neighbors kept by each node, presents values below 80% for churn values above 16%.

To better explain what happens in these scenarios, Figure 3(b) plots the percentage of nodes in the largest connect component during the simulations where 20% churn was induced in the system. For simplicity, we only plot the worst configuration of Overnesia (*small*) and the best configuration of Chord (*very large*). During the initial 50 stabilization cycles, all protocols have a connectivity of 100%. During churn period, the connectivity of both Overnesia and Scamp drops every time churn is applied to the system, and recovers during the following cycle (where no churn is applied). Overnesia connectivity is less hampered than that of Scamp, due to its highly connected topology. Additionally, Overnesia is able to recover more connectivity than Scamp in each stable cycle. On the other hand, Chord connectivity is severely hampered as soon as churn is introduced, and it is unable to recover any of its lost connectivity in a single simulation cycle, due to the slow process to update and fix its fingers.

D. Query Routing Performance

In this section we assess the benefits that can be obtained by using query routing strategies described in Section III on top of the Overnesia overlay. For this purpose, we have executed several thousands of simulations in which, after forming the overlay, we launch queries from random nodes in the super-peer overlay. In each experiment, 100 individual queries are issued.

We evaluated, for each Overnesia configuration, for each query routing protocol configuration, and for each individual query, the following three distinct performance metrics:

Query Hit Rate (QHR): The ratio of individual *Nesoi* that receive (and process) a given query. The value varies between 0 and 1. A value of 1 indicates that all consolidated indexes were searched (i.e., each and every *Nesos* was visited by the query), and therefore the query returns fully accurate results.

Query Processing Rate (QPR): The ratio of individual nodes that are required to process a query. The goal is to lower this number as much as possible without compromising the QHR (to promote a good load balancing between peers).

Message Cost (MC): The total number of messages sent to disseminate the query. This value should also be as low as possible to promote the efficiency of the search mechanism.

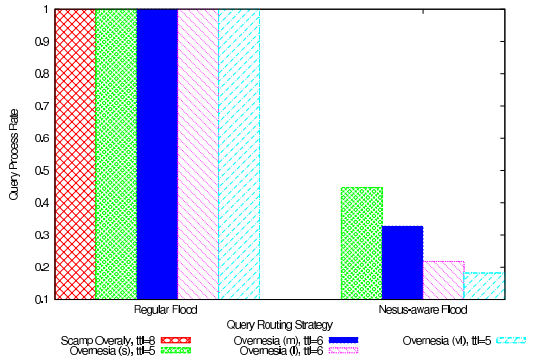
We do not plot results for Chord given that results presented in [15] show that blind search can be performed over Chord (in steady state) with a MC equal to N-1 (where N is the number of nodes in the system). Furthermore, this solution requires that all nodes process each query, generating a constant QPR value of 1. Finally, results presented earlier in this paper, have shown that Chord has a poor performance in highly dynamic environments.

***Nesos*-aware Flooding** In this section we compare the performance of regular flooding and *Nesos*-aware flooding. Due to lack of space, in this section we only depict results for TTL values which allow to achieve a QHR of 1.0 with the lowest MC value. Figure 4(a) and Figure 4(b) present respectively the query processing rate and query message cost for two flood based protocols: *i)* a regular flood protocol; and *ii)* the *Nesos*-aware flood protocol described in Section III. As expected, regular flood requires every node in the overlay to process the query. On the other hand, the *Nesos*-aware flooding protocol that leverages on Overnesia topology, achieves remarkably lower QPR values, below 0.2 for the *very large* configuration.

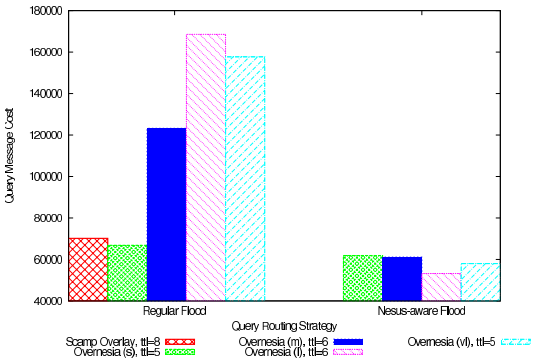
In terms of message cost, performing regular flood on the Overnesia overlay generates much more messages than flooding in a Scamp overlay. This is due to the clustering imposed by the existence of *Nesoi*. Moreover, the negative impact of flooding becomes more visible for larger *Nesos* sizes (notice that the results shown for the *very large* configuration uses a lower TTL value). Our *Nesos*-aware flooding protocol however is able to operate with message cost values smaller than flooding in the Scamp overlay. This happens because it avoids to forward messages inside each *Nesos*.

***Nesos*-aware Gossip** We now compare the performance of regular gossip, *Nesos*-aware gossip, and improved *Nesos*-aware gossip. In these experiments we explore a wide range of values for the parameters of the considered query routing protocols. Namely we tested the regular gossip and *Nesos*-aware gossip protocols with a TTL value that goes from 4 to 7, and the improved *Nesos*-aware gossip protocol with a TTL value that ranges from 3 to 6 combined with a random-walk TTL that goes from 1 to 3. Additionally, these parameters were combined, in all protocols, with a fanout that ranges from 4 to 10. As before, we only present, for each configuration, results for the set of parameters that allowed a QHR of 1.0 with the lowest cost (both message cost and query processing ratio).

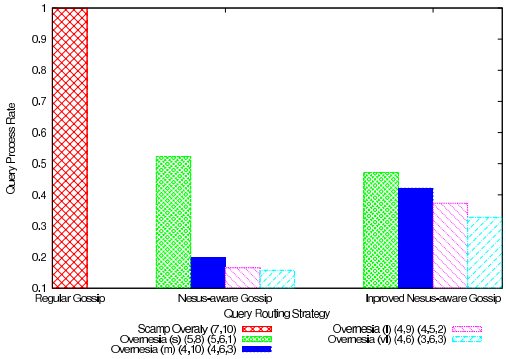
Figure 4(c) and Figure 4(d) depict, respectively, the query



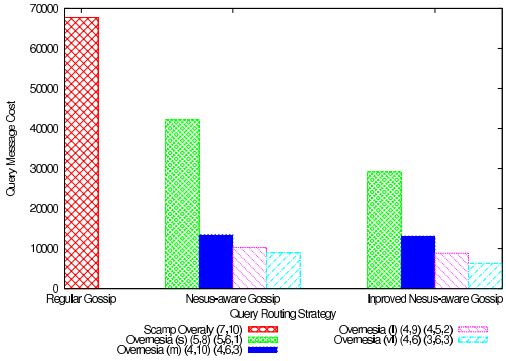
(a) Flood Strategies QPR



(b) Flood Strategies MC



(c) Gossip Strategies QPR



(d) Gossip Strategies MC

Fig. 4. Performance of Query Dissemination Strategies

processing rate and query message cost for a regular gossip protocol operating on top of Scamp, our *Nesos*-aware gossip protocol, and its improved version. In these figures we depict the configuration tuples (TTL, fanout) and (TTL, fanout, random-walk TTL) used for each protocol.

Regular gossip over Scamp is used as a baseline. As expected it presents a QPR of 1, as a result of all nodes being required to process each query, and a high message cost, due to the necessity of using a high fanout value (8). This fanout is required due to the unbalanced degree of each node, that limits the epidemic mechanism in the initial steps of the query dissemination.

Notice that for every configuration of Overnesia (with the exception of the *small* configuration) there is a trade-off in our protocols. Both protocols can improve the query processing rate and the message cost with relation to regular gossip (and regular flood). The use of the improved version of the *Nesos*-aware gossip protocol can slightly lower the message cost while increasing the query processing ratio; this happens because the improved version of the protocol allows to use more conservative fanout values, since the final random walks can, in a cheaper way, compensate for the smaller breadth. However, a portion of these random walks visit nodes in *Nesoi* which already processed the query increasing the QRP.

V. Related Work

Gnutella [7] second version uses a two-tier overlay network based on a super-peer architecture. Similarly to other super-peer architectures, regular peers connect to a super-peer which integrates a consolidated index of the resources maintained by all regular peers connected to it (as well as its own index). Nodes which are super-peers organize themselves in an unstructured overlay network. Additionally super-peers exchange their consolidated index among neighbors in the overlay, creating replicated entries that may be used to limit the query flooding in the last hop.

When a (regular) node wishes to perform a query, it forwards the query to its super-peer which, in turn, floods the query with a given time to live value trough the super-peer overlay. When a super-peer finds in its index information a match to the query, it sends an answer directly to the source of the query. Unlike our work, Gnutella does not address fault tolerance. Whenever a super-peer fails, regular peers have to resend their index to a new super-peer, which has to re-execute the consolidate operation. Moreover, a query has to be processed by every super-peer which forwards it.

SOSPNet [6] maintains a super-peer overlay which exploits semantic similarity among content maintained by peers. Regular peers maintain a cache of super-peers that are suitable targets for processing queries. When making a query, they select the target super-peer using a local preference value. This preference value is computed based on the quality of results provided by those peers in response to previous queries. In SOSPNet, the index of a regular peer is not maintained by a single super-peer. Instead, super-peers decide to replicate portions of the indexes of regular peers based on the utility of these indexes for solving past queries. Unfortunately, this also means that portions of regular indexes can never be stored in the super-peer level. Therefore, some resources are never found by queries.

Similar to our work, SOSPNet aims at distributing the load among super-peers, however our approaches operate at different levels with different goals. Whereas we distribute the load of processing and replying to queries by exploiting the topology of Overnesia, SOSPNet balances the number of queries initiated by regular peers received by each super-peer by dropping some queries. Our work can be seen as complementary to that of SOSPNet, as our work can be used to replicate the state of super-peers in the SOSPNet overlay, allowing the overlay to keep the information gathered during the operation of the system, even if super-peers leave the system.

Gia [3] is a system based on a non-hierarchical overlay that adapts the topology according to the node capacity. Overnesia is not driven by any specific node characteristic. Gia enforces index replication to all one hop neighbors. Like Gia, Overnesia also replicates super-peers indexes to a sub set of one-hop neighbors, the nView. However, unlike Gia, we have a tighter control on the number of nodes that replicate a given index and support more sophisticated mechanisms to control these replicas.

Some query routing protocols, such as [14], route queries using biased random walks, that rely in additional information provided by the system, to increase the probability of locating resources that match a given query. In [1] the use of bloom filters to provide information for query routing in P2P Systems is suggested. These techniques are complementary to and can be combined with our work.

VI. Conclusions

In this paper we have presented Overnesia, a protocol that maintains a replicated super-peer overlay network, which is highly connected, and offers improved fault tolerance. Additionally, we proposed a set of simple query routing strategies based on both flooding and gossip on the super-peer overlay which leverage on the unique characteristics of Overnesia, allowing to distribute the load of query processing among super-peers.

As future directions of work, we want to design more complex routing strategies that can better exploit the Overnesia replicated topology, such as efficient informed query routing strategies. It is our belief that many query mechanisms that have been previously proposed in the literature can offer better performance if adapted to benefit from the replication of super-peers provided by Overnesia.

References

[1] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[2] Nuno Carvalho, Joseé Pereira, Rui Oliveira, and Luís Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proc. of the 37th DSN*, Edinburgh, UK, June 2007.

[3] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM*, pages 407–418, 2003.

[4] Patrick Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM TOCS*, 21(4):341–374, 2003.

[5] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[6] Pawel Garbacki, Dick. H. J. Epema, and Maarten van Steen. Optimizing peer relationships in a super-peer network. In *Proc. of the 27th ICDCS*, page 31, Toronto, canada, 2007.

[7] Gnutela website. http://www.gnutella.com.

[8] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[9] Vana Kalogeraki, Dimitrios Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. of the 11th CIKM*, pages 300–307, 2002.

[10] João Leitão, José Pereira, and Luís Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *Proc. of the 37th DSN*, pages 419–429, Edinburgh, UK, 2007.

[11] Jenn-Wei Lin, Ming-Feng Yang, and Jichiang Tsai. Fault tolerance for super-peers of p2p systems. In *Proc. of the 13th PRDC*, pages 107–114, Melbourne, Australia, 2007.

[12] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *Proc. of the USENIX Annual Technical Conf.*, pages 10–10, 2004.

[13] Ian Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[14] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *Proc. of the 3rd P2P*, page 102, Linkoping, Sweden, 2003.

[15] Vladimir Vishnevsky, Alexander Safonov, Mikhail Yakimov, Eunsoo Shim, and Alexander D. Gelman. Scalable blind search and broadcasting over distributed hash tables. *Comput. Commun.*, 31(2):292–303, 2008.

[16] Di Wu, Ye Tian, and Kam-Wing Ng. Analytical study on improving dht lookup performance under churn. In *Proc. of the 6th P2P*, pages 249–258, Cambridge, UK, 2006.

[17] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of the 22nd ICDCS*, page 5, Viena, Austria, 2002.

[18] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. *Data Engineering, 2003. Proc.. 19th Intl Conf. on*, pages 49–60, March 2003.

[19] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA, 2001.