
Overnesia: an 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

Dec 2008
Abstract

Unstructured P2P networks have been widely used to implement resource location systems that support complex queries semantics. Unfortunately these systems usually rely on blind search algorithms, based on 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, overloads or unavailability of super-peers have clearly a critical impact on the connectivity of the overlay. These issues may be addressed using the replication of super-peers, for both load balancing and fault-tolerance purposes. This paper proposes a novel algorithm to construct Overnesia, an overlay network connecting replicated super-peers. The paper also proposes techniques to perform query routing based on the unique properties of that overlay, in such a way that query processing load is distributed among replicas.

Keywords: Peer-to-Peer, Protocols, Distributed & Networked Systems, Resource Location, Fault-tolerance & Dependability
**Overnesia: an 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 blind search algorithms, based on 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, overloads or unavailability of super-peers have clearly a critical impact on the connectivity of the overlay. These issues may be addressed using the replication of super-peers, for both load balancing and fault-tolerance purposes. This paper proposes a novel algorithm to construct Overnesia, an overlay network connecting replicated super-peers. The paper also proposes techniques to perform query routing based on the unique properties of that overlay, in such a way that query processing load is distributed among replicas.

**Keywords:** Peer-to-Peer, Protocols, Distributed & Networked Systems, Resource Location, Fault-tolerance & Dependability

1. Introduction

There are two main approaches to build peer-to-peer (P2P) overlays: structured and unstructured approaches. Structured approaches such as DHTs [15, 19] are very efficient to support exact queries but may exhibit poor performance in face of churn [14, 17] and do not have significant advantages for very complex queries [5, 8]. Unstructured approaches have the advantages of having a simpler design due to the lack of constraints in node location on the overlay topology and being potentially more resilient both to churn and node failures. Usually, queries in unstructured P2P overlays are implemented through blind search algorithms, which usually rely on some sort of flooding [1, 12].

An efficient way to minimize the cost of query flooding in unstructured P2P networks is the use of super-peers [8, 2]. An unstructured P2P network using super-peers has a two tier hierarchical structure: at the higher level, super-peer 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 the consolidate index(es) need to be rebuilt. Also, 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 the super-peers. If done appropriately, replication of super-peers may bring several advantages. To start with, it helps to make the overlay more robust and harder to attack, as one needs to disrupt all the replicas of a super-peer to force 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

---

*This work was partially supported by project “Redico” (PTDC/EIA/71752/2006).*
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 is more connected, as a result of the existence of more 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 connected among each other. We have called this overlay Overnesia.

This paper has two 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 an even distribution of super-peers in clusters, such that each virtual super-peer has approximately the same number of replicas. Finally, it 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 in the resulting Overnesia overlay. 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 evenly 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 2 and the query routing strategies introduced in Section 3. The performance of all these algorithms is analyzed in Section 4 and later compared with related work in Section 5. Finally, Section 6 concludes the paper and gives some pointers for future work.

2. Overnesia

2.1. 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 unique identifier named cID); ii) each peer knows the identity of all other members of its Nesos (denoted by the nView) and maintains a link to each of these neighbors (thus, Nesoi are

---

\(^1\)From the greek word for island.
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:

**Nesos Size (NS):** The target desirable number of Nesos members. Members of a Nesos attempt to prevent the further growth of the Nesos as soon as they know NS+1 Nesos neighbors.

**Nesos Max Size (NSMAX):** If the size of a Nesos becomes larger than threshold $NS^\text{MAX} + 1$, Nesos members coordinate to split into two distinct (smaller) Nesoi.

**Nesos Minimum Size (NSMIN):** If the size of a Nesos becomes smaller that threshold $NS^\text{MIN} + 1$, Nesos members gradually abandon the Nesos to join a (larger) Nesos, causing the graceful degradation of the 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 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 $\theta$ is also a protocol parameter). As it will become clear later in the text, the fact that each Nesos has a unique identifier 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 motivated by two reasons: i) it allows the communication between peers to be network friendly as we leverage in TCP flow control mechanisms [13] moreover, we can model the system without considering message losses between peers; ii) TCP, similarly to what we proposed in [11], is used as an unreliable failure detector. Failure detection is used, for instance, to improve the accuracy of queries, by allowing to remove from the overlay replicas of the indexes of failed nodes in a more timely manner.

### 2.2. Algorithm

**Algorithm 1:** Overnesia Protocol Overview

1: upon event Init do
2: trigger Join Procedure

3: every $\Delta T_1 + \text{random}(\pi)$ do
4: if $\#\text{Nesos} > NS^{MAX}$ then
5: trigger Divide Procedure
6: else if $\#\text{Nesos} < NS^{MIN}$
7: trigger Collapse Procedure

8: every $\Delta T_2$ do
9: if $\#\text{external neighbors} < \theta$
10: trigger External Neighboring Procedure

11: every $\Delta T_3$ with a probability $\rho$ do
12: trigger Anti-entropy Procedure

**Overview** We start by providing a macroscopic perspective on the operation of Overnesia. 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 captures these components of the Overnesia protocol.

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 ($\theta$). Finally, due to the 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 $\rho$, 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 of these 5 components of the Overnesia protocol, illustrating the operation of these components with simplified pseudo-code when adequate.

Join Procedure  In order to join the overlay, a new node forwards 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 $\text{NS}$ 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 the joining node, which uses the information on the message to update its cID identifier and to establish neighboring relations with all remaining nodes in the Nesos by sending NEIGHBORINGREQUEST messages. This procedure is depicted in Algorithm 2.

```
Algorithm 2: Join Procedure

1: upon event Init do
2:  trigger SendJOIN, contact

3: upon event Receive(JOIN, newNode) do
4:  if $\#nView < \text{NS}$ then
5:    trigger SendJOINREPLY, sender, cID, nView
6:  nView ← $nView \cup \$sender$
7:  else
8:    n ← $\epsilonView_n \lor \epsilonView_n \in \text{nView}$
9:    trigger Send(FORWARDJOIN, n, sender, TTL)

10: upon event Receive(FORWARDJOIN, sender, newNode, ttl) do
11:  ttl ← $ttl - 1$
12:  if $\#nView < \text{NS}$ or $ttl = 0$ then
13:    trigger Send(JOINREPLY, newNode, cID, nView)
14:  nView ← $nView \cup \text{newNode}$
15:  else
16:    n ← $(n \in \epsilonView_n \lor n \in \text{nView}) \land n \neq \text{sender}$
17:    trigger Send(FORWARDJOIN, n, newNode, TTL)

18: upon event Receive(JOINREPLY, sender, id, view) do
19:  cID ← id
20:  $\forall n \in \text{view}$ do
21:    trigger Send(NEIGHBORINGREQUEST, n, cID)
22:  nView ← $nView \cup n$

23: upon event Receive(NEIGHBORINGREQUEST, sender, id) do
24:  if cID = id then
25:    nView ← $nView \cup \text{sender}$
26:  else
27:    trigger Send(DISCONNECTREQUEST, sender)

28: upon event Receive(DISCONNECTREQUEST, sender) do
29:  if sender ∈ nView then
30:    nView ← $nView \setminus \text{sender}$
31:  else if sender ∈ eView then
32:    eView ← $eView \setminus \text{sender}$
33:    pView ← $pView \setminus \text{sender}$
```

Divide Procedure  This procedure is executed when the size of a Nesos exceeds the threshold $\text{NS}^\text{MAX}$ 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 (eventually unitary) Nesoi. By dividing a large cluster in two, similar to what living cells do, one can generate two stable independent Nesoi.

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 $\text{NS}^\text{MAX}$. The initiator generates two new

\(^2\)The first node to join the overlay is an obvious exception, that only has to generate a random cID.
random Nesos identifiers (namely ID

a and ID

b) and divides the current Nesos membership in two sets a and b and sends this to the remaining Nesos nodes 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 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 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 other Nesos. 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 disband a Nesos whose size has fallen below the threshold parameter NS

MIN, by migrating its members to other, more suitable, Nesos. 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 Nesoi, 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 similar manner to the join procedure described above. 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.

External Neighboring Procedure To ensure that the 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 senders Nesos nor to Nesoi to which that node is already connected.

In the special 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 keep its degree).

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 p, every node n selects another peer p in the Nesos and sends to him a message containing its own view of the Nesos current filiation. This allows p to detect missing peers in its nView. Moreover if p detects some missing nodes in n 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 which 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. Note that this gossip mechanism is only executed by nodes with an empty pdv set. This prevents anti-entropy to operate on a cluster that is about to execute a divide procedure.

3 p can be small; in our experiments we determined that a ρ value of 0.1 is adequate.
Algorithm 3: Divide Procedure

1: upon event CheckNesosSize Timer do
2:   if cID ≠ ⊥ and pdv = #null then
3:     if #nView ≥ \frac{M}{2M+1} and \beta: n \in #nView: n.nID < n.ID then
4:       ID_n = get new unique id
5:       ID_\beta = get new unique id
6:     a ← {myself} \cup SelectHalf(nView)
7:     b ← nView \setminus a
8:     pdv ← NESOSDIVISION (myself, cID, ID_n, ID_\beta, a, b)
9:     ∀ n \in nView do
10:       trigger Send(NESOSDIVISION, n, cID, ID_n, ID_\beta, a, b)
11:     setup timer (ExecuteNesosDivision Timer, RTT + 2)

12: upon event ExecuteNesosDivision Timer do
13:   if pdv ≠ #null then
14:     s ← s \in pdv and \beta: x \in pdv \land x.sender.nID < x.sender.nID
15:     if myself ∈ s.a then
16:       ∀ n \in nView do
17:         if \exists n.a then
18:           trigger Send(NESOSUPDATE, n, cID, s.ID_n, true)
19:     else if position(myself, s.a) = position(n, s.b) then
20:       trigger Send(NESOSUPDATE, n, cID, s.ID_n, false)
21:     nView ← nView \setminus n
22:     eView ← eView \cup n
23:     else
24:       trigger Send(DISCONNECTREQUEST, n)
25:     nView ← nView \setminus n
26:     cID ← s.ID_\beta
27:     else if myself ∈ s.b then
28:       ∀ n \in nView do
29:         if \exists n.b then
30:           trigger Send(NESOSUPDATE, n, cID, s.ID_\beta, true)
31:     else if position(myself, s.b) = position(n, s.a) then
32:       trigger Send(NESOSUPDATE, n, cID, s.ID_\beta, false)
33:     nView ← nView \setminus n
34:     eView ← eView \cup n
35:     else
36:       trigger Send(DISCONNECTREQUEST, n)
37:     nView ← nView \setminus n
38:     cID ← s.ID_\beta
39:     pdv ← #null
40: upon event Receive(NESOSUPDATE, sender, ID_n, ID_\beta, isNesos) do
41:   if isNesos = true then
42:     if cID ≠ ID_n then
43:       if pdv = #null or cID ≠ ID_\beta then
44:         trigger Send(DISCONNECTREQUEST, sender)
45:     nView ← nView \setminus sender
46:     else
47:       if sender ∈ eView then
48:         if sender.cID ≠ ID_n then
49:           if ID_n = cID then
50:             eView ← eView \setminus sender
51:         nView ← nView \setminus sender
52:       else update local information on cID of sender
53:       if pdv = #null or sender \notin nView then
54:         trigger Send(DISCONNECTREQUEST, sender)
55:     nView ← nView \setminus sender
56:     if pdv = #null then
57:       trigger ExecuteNesosDivision Timer
58: upon event Receive(NESOSDIVISION, sender, ID, ID_a, ID_b) do
59:   if cID = ID then
60:     if pdv = #null then
61:       setup timer (ExecuteNesosDivision Timer, RTT + 2)
Algorithm 4: Collapse Procedure

1: upon event CheckNesosSize Timer do
2:   if #nView < H N M N then
3:     with a probability of: \((1 - \frac{\#nView}{N \times M N})\) do
4:       n ← n ∈ eView or n ∈ pView or n ∈ nView
5:       trigger Send(RELOCATE_REQUEST, n, myself, cID, TTL)
6: upon event Receive(RELOCATE_REQUEST, sender, node, ID, ttl) do
7:   ttl ← ttl - 1
8:   if cID ≠ ID and #nView ≤ N S then
9:     nView ← nView ∪ node
10:    if ttl > 0 then
11:      trigger Send(RELOCATE_REPLY, node, cID, nView)
12:     else if ttl > 0 then
13:       n ← n ∈ eView or n ∈ nView
14:     trigger Send(RELOCATE_REQUEST, n, node, ID, ttl)
15: upon event Receive(RELOCATE_REPLY, sender, id, reloc_view) do
16:   if #nView < H N M N then
17:     ∀ n ∈ nView do
18:       trigger Send(DISCONNECT_REQUEST, n)
19:       nView ← nView \ n
20:     ∀ n: n ∈ eView \ n.cID = id do
21:       trigger Send(NESOS_UPDATE, n, cID, id, false)
22:     eView ← eView \ n
23:     if nView = ∅ then
24:       cID ← id
25:     ∀ n: n ∈ reloc_view \ n ∈ nView do
26:     trigger Send(RELOCATE_REQUEST, n, cID)
27:     else
28:       trigger Send(DISCONNECT_REQUEST, sender)

Algorithm 5: External Neighboring Procedure

1: upon event CheckExternalConnectivity Timer do
2:   if #eView < θ then
3:     clusters ← ∅
4:     ∀ n ∈ eView do
5:       k ← clusters ∪ n.cID
6:     d ← d ∈ eView or d ∈ nView or d ∈ pView
7:     if eView = ∅ then
8:       trigger Send(EXTERNAL_REQUEST, d, myself, cID, k, true, TTL)
9:     else
10:        trigger Send(EXTERNAL_REQUEST, d, myself, cID, k, false, TTL)
11: upon event Receive(EXTERNAL_REQUEST, sender, node, ID, k, empty, ttl) do
12:   ttl ← ttl - 1
13:   if cID ≠ ID and cID ∈ k and #eView < θ and  \( \forall n \in eView: n.cID = ID \) do
14:     eView ← eView ∪ node
15:     trigger Send(EXTERNAL_REPLY, node, cID, ID)
16:   else if ttl > 0
17:      d ← d ∈ eView or d ∈ nView
18:     trigger Send(EXTERNAL_REQUEST, d, node, ID, k, empty, ttl)
19:    else if empty = true
20:      n ← n ∈ eView
21:     trigger Send(DISCONNECT_REQUEST, n)
22:     eView ← eView \ n
23:     eView ← eView ∪ node
24:     trigger Send(EXTERNAL_REPLY, node, cID, ID)
25: upon event Receive(EXTERNAL_REPLY, sender, ID, ID_k) do
26:   if #eView = θ then
27:     trigger Send(DISCONNECT_REQUEST, sender)
28:    else
29:     eView ← eView ∪ sender
30:   if ID_k ≠ cID then
31:     trigger Send(NESOS_UPDATE, sender, ID_k, cID, false)
2.3. Increasing the Fault-Tolerance

To increase the fault-tolerance of Overnesia, we use an approach similar to the one described in [11], i.e., we augment the state of each peer with a random, unbiased, partial view of the entire overlay. This view, called the *passive view*, or simply *pView*, is maintained using a low cost background protocol[11]. Whenever a node has to remove a correct peer from its nView or eView, or receives a request sent by a peer which is not in one of those sets, that peer identifier can be added to the pView. Additionally, nodes can exchange *shuffle* messages among them selves in order to update those views, and probabilistically remove failed, or inactive, peers from that partial view.

2.4. 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 eager broadcast or anti-entropy protocols as 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.

3. 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 to thoroughly cover all possible query strategies; this is a very rich research topic on its own[5, 10, 18] that will be addressed in future work. Instead, we just aim at showing that one can indeed leverage on the 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 load balancing the query processing load 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 it 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 use a time-to-live (TTL) parameter to prevent the query to loop forever in the network. Notice that 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, [4] for the advantages of such approach). Further gossip steps attempt to forward the query to *f* external neighbors from *Nesos* 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 [6]) the protocol can be parameterized with a time-to-live parameter and a fanout. 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, and using the same rules applied to gossip to prevent a given Nesos from being visited twice. This strategy aims at minimizing the message cost associated with query dissemination, while simultaneously striving to maximize the number of Nesoi that receive, and process, the query.

4. Evaluation

4.1. Experimental Setting

We conducted an extensive experimental evaluation in the PeerSim simulator [9] using its event driven engine. To do this we have implemented the Overnesia protocol for 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 to the Scamp protocol [7] (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, somewhat, stable neighboring relations, being adequate to the use of TCP as transport protocol; and iii) in [8] the authors propose Scamp as an adequate protocol to maintain a super-peer overlay.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta T_1$</td>
<td>20.000 TU</td>
</tr>
<tr>
<td>$\Delta T_2$</td>
<td>20.000 TU</td>
</tr>
<tr>
<td>$\Delta T_3$</td>
<td>10.000 TU</td>
</tr>
<tr>
<td>$\pi$</td>
<td>0 – 20.000 TU</td>
</tr>
<tr>
<td>probability $\rho$</td>
<td>0.1</td>
</tr>
<tr>
<td>TTL</td>
<td>10</td>
</tr>
<tr>
<td>pView size</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 1. Common parameters for Overnesia

<table>
<thead>
<tr>
<th>NS</th>
<th>small</th>
<th>medium</th>
<th>large</th>
<th>very large</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NS_{\text{MAX}}$</td>
<td>6</td>
<td>16</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>$NS_{\text{MIN}}$</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>$\theta$</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2. Different target Nesos size

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 1 for the relevant protocol parameters. We tested 4 different Overnesia configurations, considering a 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 2. The reasons for using these distinct scenarios are twofold. First, it
allows us to demonstrate the flexibility of Overnesia. Secondly, it permits to observe the behavior of query routing strategies for distinct Nesos sizes.

4.2. Overlay Characterization

In order to extract the topological characteristics of Overnesia, we executed for each configuration, simulations where each node joins the system in sequence, using a random contact node (that is already part of the overlay). In Table 3 we depict values obtained for the average clustering coefficient and shortest path for each overlay. Values for a Scamp overlay of 10,000 nodes are also presented to provide a comparative measure.

One can observe that Overnesia has a larger average clustering coefficient when compared to Scamp. This is expected, as a Nesos is essentially a totally connected cluster of nodes; as the size of Nesos increases, the clustering coefficient also increases. As a result of the imposed clustering, the average shortest is also larger than Scamp, as the table shows. However, two facts are worth of notice: i) the average shortest path is larger in Overnesia than Scamp but just by a minimal amount and; ii) Contrary to what could be expected, as the Nesos size increases above the large configuration, the average shortest path tends to decrease. This phenomenon is due to the effects of the anti-entropy procedure. In fact, in our experiments we observed that this simple gossip mechanism balances with high success the external connections of each Nesos. As a consequence, it is very rare for a Nesos to have more than one link to the same remote Nesos.

![Figure 2. Nesos Size Distribution](image)

Figure 2 depicts the distribution of Nesos size for the 4 distinct Overnesia configurations. Notice that for all configurations, the most common size for Nesos is the target value of NS+1. Also, and as expected, no Nesos surpasses the size of NS MAX.

A discussion of the relevance of the graph properties, and their impact in message dissemination, can be found in [11].

<table>
<thead>
<tr>
<th></th>
<th>Average clustering coefficient</th>
<th>Average shortest path</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>0.02987</td>
<td>4.16965</td>
</tr>
<tr>
<td>medium</td>
<td>0.41907</td>
<td>4.56062</td>
</tr>
<tr>
<td>large</td>
<td>0.65003</td>
<td>4.60371</td>
</tr>
<tr>
<td>very large</td>
<td>0.74739</td>
<td>4.20471</td>
</tr>
<tr>
<td>Scamp</td>
<td>0.01443</td>
<td>3.77285</td>
</tr>
</tbody>
</table>

Table 3. Graph Properties
Interestingly enough, several Nesoi in all configurations present a sizes below that of $NS+1$. These Nesoi are a result of the Nesos division procedure. Notice that, for all configurations of Overnesia, the second most common Nesos size is $NS_{MAX}/2$.

This gives us additional insights over the ideal configuration for the Overnesia protocol. Namely Nesos size will be distributed between $NS_{MAX}/2$ and $NS_{MAX}$. Moreover, to promote stability in the overlay topology, the parameter $NS_{MIN}$ should be set below $NS_{MAX}/2$. Finally, to increase the number of Nesos with a size of $NS+1$, parameter $NS_{MAX}$ should be set close to $2NS + 1$ (in order to take advantage of the Nesos division procedure).

4.3. Fault Tolerance

One of the goals of Overnesia is to allow members of the same Nesos to act as a single, replicated, super-peer. Replication increases the dependability of the overlay given that the information maintained by the super-peer is not lost due the failure of a single node. However, to support a robust operation, the overlay must also: i) remain connected despite the occurrence of large number of node failures; and ii) reconfigure itself after such failures in order to maintain a useful number of replicas in each Nesos. We now show that Overnesia also owns these two good properties.

Experiments were performed by first creating the overlay (having nodes join as described before) and then, after a stabilization period, inducing failures in the overlay. We run multiple experiments, with different values for the number of failures. The number of failures are measured as a percentage of the total number of nodes (we have increased the number of failures by 10% in each experiment). The goal is to assess the largest failure percentage that each Overnesia configuration is able to sustain without losing connectivity. For comparison, we also present results for the (non-replicated) super-peer overlay maintained by Scamp. Results are depicted in Figure 3(a). It can be observed that 10% of node failures are enough to partition the overlay created by Scamp: in Scamp the node degree is not constant and nodes with small degree get easily disconnected. On the other hand, Overnesia requires simultaneous node failures between 40% and 60% to partition. Furthermore, since Overnesia embeds the fault-tolerance mechanism described in [11], it is able to recover from these failures in a small interval.

Finally, to evaluate the reconfiguration capacity of Overnesia, we depict in Figure 3(b) the Nesos size distribution of the overlay composed by the surviving nodes some time after the simultaneous failure of 80% of all nodes. We can observe that the the size of Nesoi are within the expected interval, given each configuration (described in Table 2). Additionally, no Nesos has a size below that of $NS_{MIN}$. This shows that Overnesia presents the capability of reconfiguring itself in dynamics environments.

4.4. Query Routing Performance

In this section we evaluate the performance benefits that can be obtained by using in the query routing strategies described in Section 3 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 sent.
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.

**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. 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 3. As expected, regular flood requires every node in the overlay to process the query. On the other hand, Nesos-aware flooding protocol leverages on Overnesia topology, achieving remarkably lower QPR values.

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 large configuration use a lower TTL value). Our Nesos-aware flooding protocol however is able to operate with message cost values smaller than flooding the Scamp overlay. This happens because it avoids to forward messages inside each Nesos.
**Nesos-aware Gossip**  In this section we compare the performance of regular gossip, Nesos-aware gossip, and improved Nesos-aware gossip. Also, 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. Similar to results depicted 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 processing rate and query message cost for a regular gossip protocol operating on top of Scamp, and both 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), using 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.

5. Related Work

Gnutella [1] 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 through 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 [8] 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 similarity 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 balancing the load among super-peers, however our approaches operate at different levels with different goals. Whereas we balance 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 [5] 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 replica a given index and more sophisticated mechanisms to control these replicas.

Some query routing protocols, such as [16], 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 [3] 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.
6. 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 balance 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 the query mechanisms that have been proposed in the literature can offer better performance is adapted to benefit from the replication of super-peers provided by Overnesia.

Acknowledgments

The authors wish to express their gratitude to Tiago Luís and David Matos, for their help with the GRID system that was used to run the simulations presented in this paper, and to Paolo Romano for his comments on early drafts of this paper.

References