MOve: Design of An Application-Malleable Overlay*

Sébastien Monnet
IRISA/University of Rennes I
Sebastien.Monnet@irisa.fr

Ramsés Morales
University of Illinois at Urbana-Champaign
rvmorale@cs.uiuc.edu

Gabriel Antoniu
IRISA/INRIA
Gabriel.Antoniu@irisa.fr

Indranil Gupta
University of Illinois at Urbana-Champaign
indy@cs.uiuc.edu

Abstract

Peer-to-peer overlays allow distributed applications to work in a wide-area, scalable, and fault-tolerant manner. However, most structured and unstructured overlays present in literature today are *inflexible* from the application viewpoint. In other words, the application has no control over the structure of the overlay itself. This paper proposes the concept of an *application-malleable* overlay, and the design of the first malleable overlay which we call *MOve*. In MOve, the communication characteristics of the distributed application using the overlay can *influence* the overlay's structure itself, with the twin goals of (1) optimizing the application performance by adapting the overlay, while also (2) retaining the large scale and fault tolerance of the overlay approach. The influence could either be explicitly specified by the application or implicitly gleaned by our algorithms. Besides neighbor list membership management, MOve also contains algorithms for resource discovery, update propagation, and churn-resistance. The emergent behavior of the implicit mechanisms used in MOve manifests in the following way: when application communication is low, most overlay links keep their default configuration; however, as application communication characteristics become more evident, the overlay *gracefully* adapts itself to the application.

Keywords

Peer-to-peer overlay, adaptability, malleable, group membership, volatility-resilience.

*This work was supported by NSF CAREER grant CNS-0448246 and UIUC-INRIA collaboration grant.

1 Introduction

Today, peer-to-peer (P2P) overlays fall into two categories - (1) structured (i.e., *Distributed Hash Table*-based) overlays such as Pastry and Chord [13, 15], and (2) unstructured (i.e., gossip- or flooding-based) overlays such as Freenet, Gnutella, KaZaA [5, 11, 20]. These P2P overlays offer reliability in the face of massive failures and churn (node join and leave), as well as scalability to hundreds and thousands of nodes.

However, both these types of overlays have the common disadvantage that they are *inflexible* from the application viewpoint. The rules and invariants for selecting and maintaining neighbor nodes in the overlay, as well as for resource discovery, are all dictated in a rigid fashion (e.g. using the result of a hash function), without taking into account the application's communication patterns. This usually means that the developer of a distributed application has a limited number of options – either go with the provided overlay, or design a new overlay from scratch. Furthermore, overlays are application-dependent and Internet-independent [12], so allowing an application to explicitly influence the overlay is a logical next-step.

In this paper, we propose the concept of an *application-malleable* overlay. An application-malleable overlay is defined as an overlay where the communication characteristics of the distributed application using the overlay can *influence* the overlay's structure itself. The twin goals of a malleable overlay are: (1) to optimize application performance from the overlay, while also (2) retaining the scale and fault tolerance of the overlay approach.

In order to realize and evaluate our design philosophy, we build a specific malleable overlay called *MOve* (for *Malleable OVErlay*) that combines elements of an *unstructured overlay* with application characteristics. In MOve, the structure and behavior of the overlay is influenced both by the underlying default unstructured overlay, and by application characteristics. The influence could either be (1) explicitly specified by the application or (2) implicitly gleaned by our algorithms.

In a P2P overlay, each node maintains a separate neighbor list - this is a membership list that specifies the *who knows whom* relationship. This neighbor list is partial in the sense that it contains only some of the nodes in the system [4, 6, 15]. MOve contains algorithms for neighbor list maintenance, efficient message propagation, and churn-resistance (i.e., resilience to nodes joining and leaving asynchronously). The most interesting feature of MOve is its emergent behavior. When application communication is low, MOve autonomically evolves to keep most of the overlay links in the default state so that most of the system looks like an unstructured overlay. However, as application communication characteristics become more and more evident, the overlay autonomically *gracefully* adapts itself to the application, but without forgetting its default structure.

To focus our approach on a particular class of applications, we choose *collaborative applications*, such as distributed whiteboard platform, an audio/video conferencing service, a replicated data-sharing service, or a distributed-gaming platform. All these applications rely on the notion of *application groups* - each process belongs to one or more groups, and interacts with other processes in common groups. For instance, the members of the same group may share a distributed state that needs to be updated (the whiteboard, the game-board, the replicas of a mutable piece of data, etc.). Alternatively, the set of replica managers for a particular data item would form a group. Managing groups at overlay level allows multiple applications to take advantage of this optimization without having to explicitly handle it.

MOve allows such *group*-based applications to influence the underlying overlay, that may be common to multiple, coexisting, collaborative applications. In the process of the neighbor list maintenance, MOve has the following two goals: (1) (*connectivity*) keep a low diameter for the overlay so that unstructured queries can be quickly propagated; and (2) (*volatility-resilience*) combat volatility arising from rapid node arrival and failure (i.e., “churn”).

The basic idea in the MOve approach is to have each node maintain a neighbor list that, by default, consists of *non-application neighbors*, i.e., randomly selected neighbors. However, with the formation of more and more application groups, some of these non-application neighbors are automatically replaced by *application-aware* neighbors (shortly: *application neighbors*). A non-application neighbor may either change status and turn into an application neighbor (if the neighbor belongs to a common group), or be replaced by a new application neighbor. We have implemented MOve, and our experiments show (1) that the system achieves logarithmic overlay path lengths; (2) that it gracefully manages transitions between application and non-application neighbors as the number of groups increases and decreases. In addition, MOve shows good scalability and volatility-resilience.

The next section presents research efforts related to the various aspects involved in this context. Then Section 3 describes a scenario explaining why it is important to have efficient communication within groups of nodes. Section 4 gives a general overview of our approach, and provides an analysis. Section 5 presents some simulation results. Finally, Section 6 discusses the contribution and the future work.

2 Related Work

In the past few years, many research efforts have focused on building overlays for peer-to-peer networks, essentially for large-scale immutable file-sharing. For this kind of application, predicting which node is going to communicate with which node is not trivial. Therefore, most algorithms for building overlays do not take communication patterns into account.

In unstructured (i.e., gossip- or flooding-based) P2P overlays, such as [11] or [16], neighbor lists are usually built and maintained by randomly selecting a subset of the neighbors' neighbors.

In structured (i.e., DHT-based) overlays, the *who knows who* relation is usually defined by means of a given topology (typically a ring); the position of each node in this given topology is determined by a hash function on its IP address [13, 15].

Even if some of these previous proposals take into account certain criteria while building the overlay, (e.g., physical locality in the case of [13]), they do not take into account the application-related relations between nodes, which can express interaction patterns that may result from the way the overlay is solicited by the application.

Very few recent research efforts take into account these relations. Semantic overlay networks [7, 18] exploit the semantic relations between peers (based on the set of files they share). They propose solutions allowing to improve the efficiency of the search mechanisms for large-scale file-sharing applications, by creating shortcuts between peers which are semantically close. Efficient search is also the goal addressed by the *path-caching* technique, which consists in keeping data references along a given search path, in order to improve the efficiency of subsequent search operations.

However, for the group-based applications we target in this paper, such as distributed shared whiteboard (or gaming) platforms, or replicated data-sharing services, search efficiency is not the only property to optimize. In such applications, the members of a same group share some data (a whiteboard, a replicated piece of data, a game state, etc.), which they all potentially read and write. When a peer writes this shared data, it is important for the updates to be efficiently propagated to the other members of the group. Consequently, the peers belonging to a same group have to be close to each other in the overlay (i.e., a few hops away), to enable the application to efficiently maintain the consistency of the members' views of the shared state.

The issue of update propagation in large-scale systems has been studied in [14]. This system proposes an efficient multicast scheme based on multicast trees built on top of the Pastry overlay. The problem we address in this paper is different, since every member of the group can be the source of multicast in our target applications. The approach we propose is also different, since it does not construct a membership mechanism based on an already *existing overlay*. Our goal is to build an *emergent and adaptive overlay* based on patterns derived from the application usage. To achieve this goal, our work is based on an unstructured overlay. This provides the ability to dynamically change links to adapt the overlay to the application needs without breaking the overlay structure (in structured overlays, links have to follow strict rules, usually based on a hash function).

The closest work related to ours is [9], which addresses the problem of building an adaptive overlay based on different criteria: topology, semantic proximity, bandwidth, etc. The problem is addressed in a generic way: the target scale and the target applications are not specified. The issue we address is more specific: it regards applications that need efficient updates within groups of nodes. Consequently, multiple criteria have to simultaneously be taken into account and controlled: application-dependent node relations, but also physical locality, as well as the connectivity of the resulting graph (expressed through the degree of clustering).

Finally, [17] is a work that has been started concurrently. The goal of [17] is offering an efficient overlay dedicated to publish/subscribe applications providing the ability to express range-based subscriptions. To achieve this goal, it focuses on clustering nodes with similar subscriptions.

3 Scenario

To motivate our work, we consider a large scale distributed gaming platform (represented by Figure 1) as a sample application. This application may involve tens of thousands of nodes spread around the Internet. For efficiency reasons, the number of neighbors that a peer must know has to be bounded, since the related information requires monitoring and state updating. Therefore, each node only has a partial view of the system. However, this should not have a negative impact on the application's desired properties, such as *connectivity*, *message propagation efficiency* and *volatility resilience*, which are important for collaborative applications, such as gaming platforms.

Connectivity. An overlay is said to be *connected* if there is a path (succession of edges or links) between every pair of nodes. This property is very important in an overlay as it provides the guarantee for a node to be able to communicate with all the other ones in the overlay.

Some particular node (for instance the black node in Figure 1) may have to lookup for a specific game instance in the platform (e.g., Game A). This game may involve only a small subset of nodes (few tens). The neighbors contained in this particular node's neighbor list may not be involved in this particular game. The platform has to be connected to make it possible for a node to reach somehow (even through a quite long path) all the other nodes. The lookup of a node that is participating in Game A is application dependent: it could be done by visiting a website, or querying a custom search engine, or by flooding a search query on the overlay.

For efficiency purposes, the diameter of the overlay should be as small as possible, even with partial neighbor lists. To achieve this, the graph formed by the nodes and links needs to have well distributed degrees. Note that it is enough for a new player to find only one player for the wished game in order to be able to reach the other ones.

Efficient Message Propagation. While a game is running, the players store object replicas which represent the current state of the game (depending on the application, this can correspond to a shared white board, etc). Each time a player plays, his node updates the state of its local game board version (i.e., its replica). In order for the other players to be able to play, they have to be notified of the changes in the game board. Therefore, messages need to be propagated in an efficient manner within a group (e.g., Game A or Game B in Figure 1). To enable efficient message propagation, the overlay should minimize the number of hops between two peers belonging to the same group.

Volatility Resilience. Among several thousands of nodes spread over the Internet, it is likely that, from time to time, some nodes fail or get disconnected. At the global level (the entire platform), failures and disconnections may not lead to break down the whole graph connectivity. At the level of a given game such events should not stop the game, which means that the remaining players have to remain connected together. Furthermore, the departure of one player may break some path in the group graph. The longest path between two nodes (the subgraph diameter) is likely to grow; however, the update propagation mechanism has to stay efficient.

4 Design

To address the issues described in the previous section, we propose the concept of Malleable Overlay that combines elements of an unstructured overlay with application characteristics. In this section, we describe the design of MOve, a system which illustrates the proposed concept.

The first purpose of an overlay is to connect nodes together. Therefore, the first property to fulfill is the *connectivity* of the constructed graph. On the over hand, Section 3 highlighted the importance of providing the ability to perform efficient updates among groups of nodes within the overlay. This may be favored by introducing some *clustering*. Both *connectivity* and *clustering* have to be preserved while taking into account the dynamic nature of the environment.

Random Graph Benefits. Graph theory shows that *random graphs* have good properties in terms of *connectivity* and *degree distribution*. For instance, in a random graph, if each node has at least $log(N)$ uniformly random neighbors (where N is the total number of nodes) the random graph will be connected with high probability [2]. Estimating the size (i.e., N) of a large scale dynamic distributed system has also been addressed in previous studies [10]. In our case, the scale is not infinite (we target thousands to a few tens of thousands of nodes), therefore safe bounds can be assumed instead. For instance, 50 links per node will provide a large safety margin to theoretically connect 5×10^{21} nodes. On the other hand, random graphs also have the benefit of leading to a good degree distribution. An overlay based on a random graph may take advantage of this for *load distribution*. For these reasons, MOve's algorithms try to keep part of the overlay close to a random graph.

In our design, nodes maintain a neighbor list, containing links to the node's neighbors. For each node, an upper bound (*l*) is set on the size of the neighbor list. This bound is first set according to an initial approximation of the network size (while observing the condition $l > log(N)$). Then, during the execution, this value can be increased when necessary, if allowed by the available resources (see below).

The neighbor list is composed of two kinds of links: *non-application links* and *application links*. Figure 2 represents a node's neighbor list.

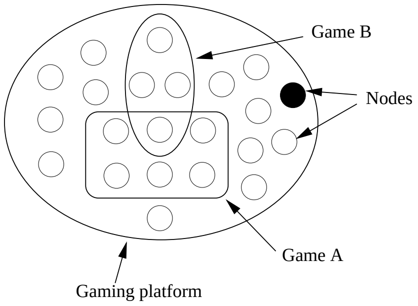

Figure 1: A gaming platform

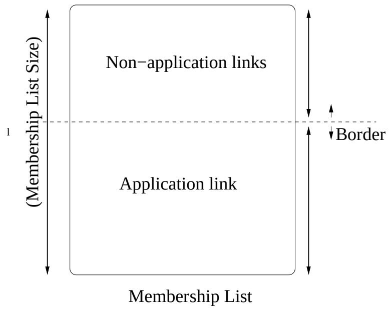

Figure 2: The neighbor list on each node

Non-Application Links. *Non-application links* are responsible for maintaining a global overlay, close to a random graph, with a low degree of clustering. If the application is in a state that does not need clustering (e.g., at initialization), the neighbor list will contain only non-application links. Remember that nodes don't need full knowledge about the network, and the number of non-application links may vary from node to node.

Application Links. To cluster together nodes that belong to a group i, each member of the group creates k_i *application links* to other randomly-chosen members of the same group. This clustering will help fast propagation of state updates among the members of the group. It will also favor an efficient propagation of application-level multicast messages. Parameter k_i is determined by the application, and it must be at least $\lceil ln(|R_i|)\rceil$, where R_i is the number of members of group i. Essentially, the goal is to create a strongly connected graph for group i (i.e., there is a path that connects every pair of nodes).

Replacement Policy. When an application link needs to be created, it will be added to the neighbor list following four different ways. Assume that we want to create an application link for group i, that points to node n. If the size of the neighbor list is smaller than l, and if there is no non-application link pointing to n, then a new link will be added to the list (1). If the size of the neighbor list has already reached l , but the node has enough resources available to maintain a larger neighbor list, then the node increases l to accommodate a new link(2). If the node decides not to grow the list, then a non-application link will be dropped, and the application link will be added (3). Finally, if there is a non-application link pointing to n, then it will become an application link (4).

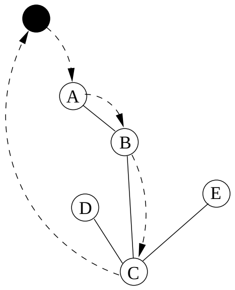

Figure 3: Join mechanism

4.1 Addressing the Connectivity Issue

The tradeoff between the good properties of random graphs and those enabled by favoring clustering between related nodes can be tuned by setting some bounds. The first bound is the size l of the neighbor list, which is managed as explained above. The second bound is k_i, which limits the number of links that are involved in the group i. If $l - \sum_i k_i$ is large enough (a few tens for the scale we are targeting), i.e., if the neighbor list contains enough non-application links, the good properties of random graphs are approximated in spite of the little clustering induced by taking the topology into account. On the other hand, it is important to notice, that $\sum_i k_i$ is in fact greater than the number of *application links*. This is explained by the fact that some *application links* may be shared by multiple groups when group intersections are not empty.

A node joins the overlay (e.g., the black node in Figure 3) by contacting any current overlay member. If the peer that receives the join request has space available in its neighbor list, it will reply with its current neighbor list, and will add a link to the joining node to its non-application links. If the neighbor list is full (which is the case for Nodes *A* and *B* on Figure 3), the join request will be forwarded to a randomly chosen node. The forwarding of a join request is associated with a time to live (TTL). If all nodes that receive the forwarded request have full neighbor lists, the TTL will reach 0, and the last node to receive the forward will forcibly add a link to the new node to its non-application links. It will then reply with its current neighbor list (e.g., Node *C*). We do this to ensure that the in-degree of a node is always above 0. The new node will use the received neighbor list to create its own list.

The **failure detection protocol** is based on the SWIM [6] protocol. Each protocol period, of length T seconds[1], each node sends a ping message to one of its neighbors. The target node is selected by sequentially traversing an array that represents the random permutation of the neighbor list. Once the array is completely

[1] Protocol periods are asynchronous at different process, although it is assumed that they have the same T.

traversed, a new permutation is computed. The node expects a reply to the ping message within a timeout of $t < T$ seconds. If the reply is not received on time, an indirect ping is sent to y nodes. These nodes will then send a ping to the intended target node, and, if they receive a reply, the reply will be sent back to the node that originated the ping. The intention of the indirect ping is to sidestep transient network problems. If no reply is received before the next protocol period, the ping target will be *suspected* of having failed. At the beginning of each protocol period, any node that has been suspect for one protocol period will be dropped from the neighbor list, i.e., declared dead.

To achieve an overlay with a low clustering coefficient[2] and evenly distributed in-degree, every U protocol periods, each node verifies if its non-application membership list has been modified. If no modification has been made after U periods, it issues a join message to a random node. With the membership list it will receive as a reply to its join message, the node will try to replace a fraction[3] of its own membership list. Note that the smaller U is, the more aggressive the replacement will be, and the faster the protocol will take the overlay to a stable low-clustering coefficient.

4.2 Group Communications

As previously explained, when a new application link is created, it will result in the substitution of a non-application link, unless the node has enough resources to grow its neighbor list. In this way we keep the maintenance cost constant at the node. On the other hand, an application link can be *shared*. For instance, assume Node a belongs to Groups i and j. If Node b joins Groups i and j, it creates a single application link to a, knowing that this link is *shared*. A sharing count is maintained for such links.

Random Walk for Application Links. To cope with the dynamic nature of the infrastructure and avoid pathological topologies that may be induced by failures, it is important to periodically refresh the links. This is also useful in order to guarantee a small path between any two nodes in a given group. To this effect, we rely on another result from random graph theory [2]: adding $O(n)$ non-application links to a graph with n vertices will reduce the diameter to $O(log(n))$. [4] This result only applies to undirected graphs. Therefore

[2] The clustering coefficient measures how many neighbors of a node are neighbors among themselves. Lower coefficient means more randomness in the graph.

[3] $f = 50\%$ in our evaluations

[4] Although this result was found for Erdos-Reiny random-graphs, and our application links are not trying to achieve a strict Erdos-Reiny random-graph, the overlay is random enough for the result to hold, as our experiments show.

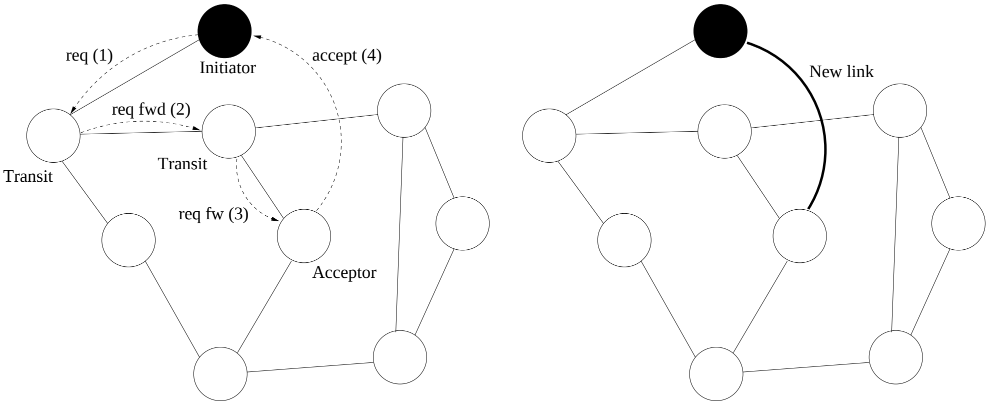

Figure 4: The random-walk mechanism

we add the restriction that all application links are *bidirectional*. When an application link is created from a to b, b will also create a link to a. If node b deletes the link, so will node a. When an application link is shared, it will be maintained until the sharing count reaches 0. When an application link stops being used as such, it is changed to a non-application link. This simulates an undirected graph inside the application group.

The graph is periodically refreshed, by having every node in an application group execute the following steps:

(1) Launch a random walk to get a new neighbor. The random walk hops at most TTL times, using application links that belong to the group.

(2) Drop an old link when the new link is created.

Although it is assumed that the timeout to launch the random walk is an application parameter, note that nodes that belong to a group are not synchronized. Also note that the bidirectionality of the links is always enforced.

4.3 Analysis

To show the benefits of link-sharing among application links (for the different application groups) and non-application links, we present an analysis of a variant of the MOve system. This variant does not impose limits on neighbor list sizes, and allows them to grow indefinitely. *Without* link-sharing, this indefinite approach would have the neighbor list size of each node grow linearly as the sum of the number of its neighbors for each application group the node belongs to, and the number of its non-application neighbors.

With our MOve approach, the number of links saved is significant, as shown by our analysis below. This results in a reduction of the overhead induced by link maintenance.

Formally, in an overlay of size N, at a node p, let $Nbrs_N(p)$ represent the set of application links at p. Assume that p belongs to k groups R_i ($i = 1$ to k). Let $Nbrs_{|R_i|}(p)$ represent the set of neighbors that p has in group R_i. Now, let $f(N) = |Nbrs_N(p)|$ and for each $i = 1$ to k, let $f(R_i) = |Nbrs_{|R_i|}(p)|$. Note that in our implementation, $f(x) = O(log(x))$, however our results are more general.

Theorem 1: Assume that: (1) each application group consists of members selected uniformly at random, (2) at node p, each non-application link for a group is selected uniformly at random among the group members and (3) at node p, each application link for a group is selected uniformly at random from among the group members[5] Then: (a) the expected number of links saved by MOve is positive and (b) it grows linearly as the number of non-application links is increased, and (c) it is proportional to the total number of non-application links if all groups at node p are equi-sized.

Proof: Without the MOve approach, the total expected number of neighbors maintained at node p is: given by

$$Nbrs_{Worst-Case}(p) = f(N) + \Sigma_{i=1}^{k} f(R_i) \tag{1}$$

Now, with the MOve variant we are analyzing, the total expected number of neighbors for a node can be formally represented as union of $k + 1$ sets as:

$$Nbrs_{MOve}(p) = |Nbrs_N(p) \cup Nbrs_{R_1}(p) \cup Nbrs_{R_2}(p) \cup \ldots \cup Nbrs_{R_k}(p)|.$$

This can be written as:

$$\begin{aligned}
= \quad & |Nbrs_N(p)| + \Sigma_{i=1}^{k} |Nbrs_{R_i}(p)| \\
& -[(\Sigma_{i=1}^{k} |Nbrs_N(p) \cap Nbrs_{R_i}(p)|) + (\Sigma_{i \neq j, 1 \leq i,j, \leq k} |Nbrs_{R_i}(p) \cap Nbrs_{R_j}(p)|)] \\
& +[(\Sigma_{i \neq j, 1 \leq i,j, \leq k} |Nbrs_N(p) \cap Nbrs_{R_i}(p) \cap Nbrs_{R_j}(p)|) \\
& \quad + (\Sigma_{i \neq j \neq l \neq i; 1 \leq i,j,l \leq k} |Nbrs_{R_i}(p) \cap Nbrs_{R_j}(p) \cap Nbrs_{R_l}(p)|)] \\
& - \ldots \\
& \pm |Nbrs_N(p) \cup_{i=1}^{k} Nbrs_{R_i}(p)|
\end{aligned} \tag{2}$$

[5] The uniformly at random assumption (3) is reasonable since our neighbor list maintenance protocols achieve such random neighbor lists.

In order to simplify this, consider an individual term of the type

$|Nbrs_{A_1} \cup \ldots \cup Nbrs_{A_m}|$, where each A_j is either a unique R_i or N. Now consider an arbitrary neighbor q of p that is in group A_1. Consider the event E that for a given $j (\neq i)$, the same neighbor q (1) also belongs to group A_j and (2) is a neighbor of p in group A_j (i.e., appears in $Nbrs_{A_j}(p)$).

Due to assumptions (2) and (3) in the above theorem, we have that the probability of the above event E is simply

$$Pr[E] = \frac{|A_j|}{N} \cdot \frac{f(A_j)}{|A_j|} = \frac{f(A_j)}{N}$$

Thus, the individual term of the type of the type $|Nbrs_{A_1} \cup \ldots \cup Nbrs_{A_m}|$ in fact has a value of:

$|Nbrs_{A_1} \cup \ldots \cup Nbrs_{A_m}| = f(A_1).\Pi_{j=2}^{m}(\frac{A_j}{N}) = \frac{\Pi_{j=1}^{m}(A_j)}{N^{m-1}}$ (3)

Substituting equation (3) into equation (2) and using equation (1) above, we get

$$\begin{aligned}
Nbrs_{MOve}(p) \quad = \quad & Nbrs_{Worst-Case}(p) \\
& -\frac{1}{N}.[(\Sigma_{i=1}^{k}(f_N(p).f_{R_i}(p)) + (\Sigma_{i\neq j,1\leq i,j,\leq k}(f_{R_i}(p).f_{R_j}(p))] \\
& +\frac{1}{N^2}.[(\Sigma_{i\neq j,1\leq i,j,\leq k}(f_N(p).f_{R_i}(p).f_{R_j}(p))) \\
& \qquad\qquad + (\Sigma_{i\neq j\neq l\neq i;1\leq i,j,l\leq k}(f_{R_i}(p).f_{R_j}.f_{R_l}(p)))] \\
& - \ldots \\
& \pm[f_N(p).\Pi_{i=1}^{k}(f_{R_i}(p))]
\end{aligned}$$

By exchanging the $Nbrs$ terms, and taking $f_N(p)$ common on the other side, we simplify to calculate the number of links saved by using MOve as:

$Nbrs_{Worst-Case}(p) - Nbrs_{MOve}(p)$, which is:

$$\begin{aligned}
= \quad & f_N(p).[\frac{1}{N}.\Sigma_{i=1}^{k}f_{R_i}(p) - \frac{1}{N^2}.\Sigma_{i\neq j,1\leq i,j,k}(f_{R_i}(p).f_{R_j}(p)) + \ldots \pm \frac{1}{N^{k-1}}.\Pi_{i=1}^{k}(f_{R_i}(p))] \\
& +[\frac{1}{N^2}.\Sigma_{i\neq j,1\leq i,j,k}(f_{R_i}(p).f_{R_j}(p)) - \ldots \mp \frac{1}{N^{k-1}}.\Pi_{i=1}^{k}(f_{R_i}(p))] \\
= \quad & f_N(p).[1 - \Pi_{i=1}^{k}(1 - \frac{f_{R_i}(p)}{N})] + [\Pi_{i=1}^{k}(1 - \frac{f_{R_i}(p)}{N}) - 1 + \Sigma_{i=1}^{k}(\frac{f_{R_i}(p)}{N})]
\end{aligned}$$

(4)

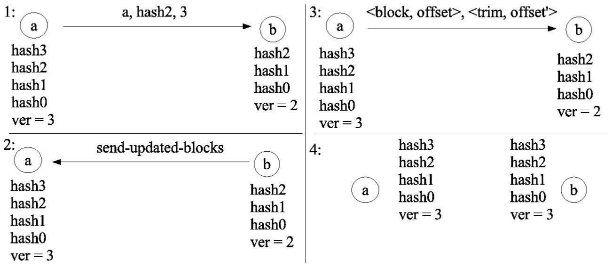

Figure 5: 1: a sends an update message to b. a contains 4 hashes on its hash list, and b contains 3 hashes on its hash list that coincide with a's three oldest hashes. 2: the hash included in the message coincides with the newest hash at b, so b requests an updated blocks transfer. 3: a sends the updated blocks, and in this example part of the update is the deletion of bytes after offset `offset'`. 4: after the update is applied, both nodes have the same hash list and version numbers.

The above result consists of two terms (each within square braces). The second of these two terms can be shown to be ≥ 0 (by using telescoping), and the first term is clearly positive. Finally, the first term is linear in $f_N(p)$, as desired. This proves (a) and (b). To prove (c) for equi-sized groups at node p, substitute $f_{R_i}(p) = f_R(p)$ for all i in equation (4) above. Then, we get that $(Nbrs_{Worst-Case}(p) - Nbrs_{MOve}(p))$ is:

$$\begin{aligned}
&= f_N(p).(1-(1-\frac{f_R(p)}{N})^k) + (1-\frac{f_R(p)}{N})^k - 1 + \frac{k.f_R(p)}{N} \\
&\simeq f_N(p).\frac{k.f_R(p)}{N} - \frac{k.f_R(p)}{N} + \frac{k.f_R(p)}{N} \\
&= f_N(p).\frac{k.f_R(p)}{N}
\end{aligned}$$

This proves (c). $\ll$

4.4 Sample Update Propagation Mechanism.

Thanks to the design described above, propagating updates within a group simply consists in sending update messages over all the outgoing *application-links* associated to this group. Below we describe a possible update propagation algorithm.

The update propagation algorithm we propose is designed to propagate and apply updates under a dynamic set of nodes and object replicas, and it provides causal consistency [8]. The update is transferred across members of the group. Although the update mechanism detects update conflicts, the actual resolution is deferred to the application layer, which receives an alarm with the conflict.

When a node updates a replica, a secure hash is computed over the entire replica contents, and an object version number is incremented. The hash is added to a list of hashes of previous replica versions –this list is ordered chronologically. The node then broadcasts an update message using the *application links* associated with the group. The message contains the IP number of the sender, the new version number, and the replica hash that corresponds to the replica before the update. When a node receives the message it verifies the version number, if it is lower than the local version number then the message is ignored. If the message is not dropped, it is forwarded on all outgoing *application links* that correspond to the group. After forwarding the update message, the node compares its current replica hash with the one received on the message: if they are equal then the node will request the upstream node to forward the updated blocks of the replica. If the hash differs, the node will send its current replica hash to the upstream node. The upstream node will search for this hash in its hash list, and if found, the upstream node will copy the entire replica to the downstream node along with the list of hashes, otherwise a conflict alarm will be sent to the application layer. Figure 4.4 shows a common case example.

5 Experimental Evaluation

Our algorithms are implemented in Java, as a discrete event simulation. The GT-ITM [3] random topology generator, following the stub-transit model, is used to provide an underlying internetwork to our simulations. Ten transit nodes are used and each stub node joins the overlay. The end-to-end latency of a message corresponds to the shortest path between the sender and receiver nodes.

Non-Application Link Clustering. For this experiment we used a topology with 520 nodes, a protocol period for the failure detection mechanism of 1 minute. Parameter U, which determines the number of protocol periods that a node will allow an unchanged list before randomly refreshing it, has a value of 1. Each node stores a strict maximum of 50 links. Figure 6 shows how the clustering coefficient changes with time. After only 50 minutes, the links among the nodes show a very low degree of clustering.

Connectivity. As the number of groups increases and becomes large, there is a possibility of overlay partitioning. In this experiment, we try to break the connectivity of the overlay by taking it to an extreme scenario. The basic parameters are the same as in the previous experiment. Figure 7 measures the size of the

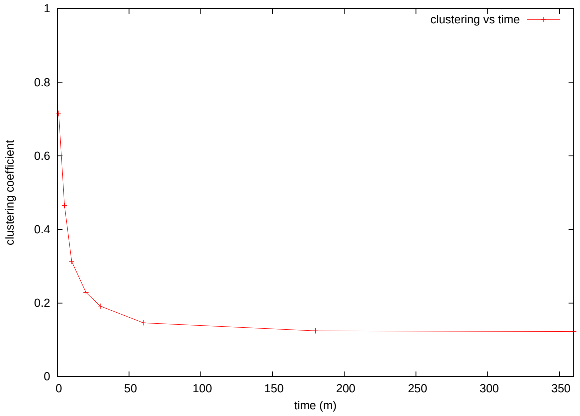

Figure 6: Clustering coefficient vs time.

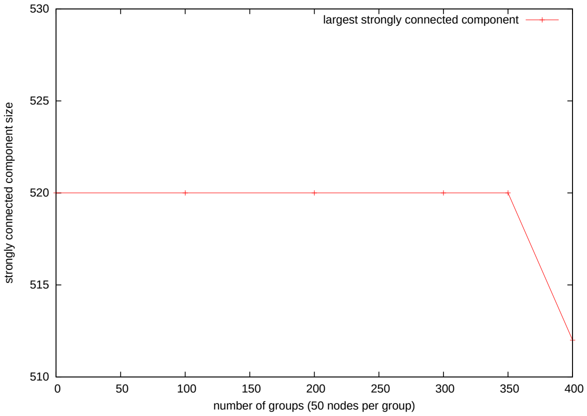

Figure 7: The largest connected component is the overlay itself, until it reaches an overutilization with 400 groups composed of 50 nodes each.

largest strongly connected component in the overlay. This size is equal to the total number of nodes when the overlay is not partitioned. We vary the number of groups from 0 to 400 (each group is composed of 50 nodes, the k parameter is set to $\lceil ln(50) \rceil$). As the plot shows, the overlay maintains strong connectivity until the number of groups approaches 400. In this case, we notice a small decrease in size of the largest component, which is due to the overlay partition.

Application Link Clustering. For this experiment, we use a 1000 node network and an application running for 2 hours with one group. In order to evaluate the quality of the graph constructed by the application links in terms of distance (in hops) between group members, we measure the characteristic path length of this group [19]. The characteristic path length is the average of the shortest path over all node pairs. The experiment is run several times varying the group size from 5 to 500 members. The k parameter (i.e., the number of application links for this group on each node) is set to $\lceil ln(groupsize) \rceil$. Figure 8(a) shows that the characteristic path length grows slowly with the group size. Even for 500 nodes it is only 3.27. This shows that the creation of one non-application link at each node of the group, using random walks, achieves its objective of providing a small number of expected hops between any pair of nodes of the group. Note also that characteristic path length follows closely the logarithm with base k. Figure 8(b) shows the same experiment, without network topology, using five thousand nodes and varying the group size up to two thousand nodes.

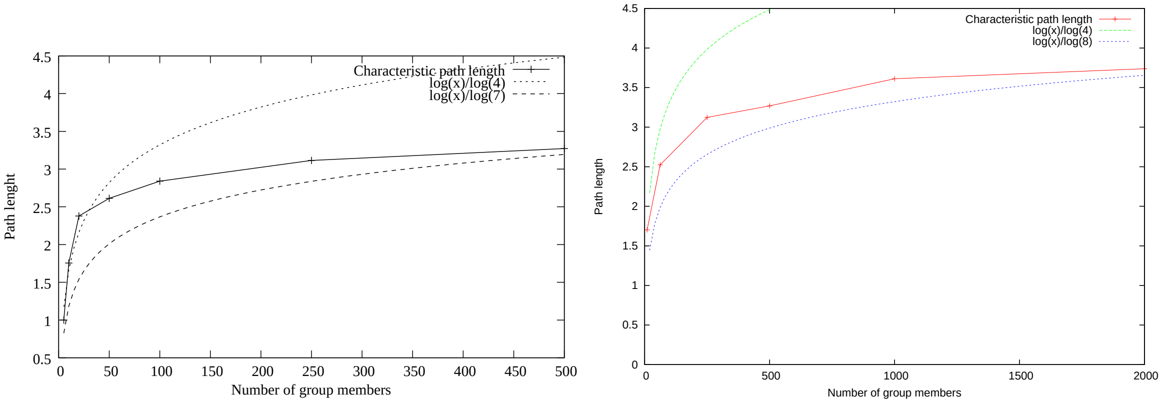

Figure 8: The characteristic path length of a group graph is $log(groupsize)$ as expected. The left plot shows a one thousand node network, and the right plot a five thousand node network without underlying topology.

Benefits of Link Sharing. When the platform contains many groups, the probability of non-empty group membership intersections grows. In this case, application links at each node can be shared by multiple groups; e.g., node A belongs to groups i and j, and its peer, node B, belongs to groups i and j, allowing them to use only one application link between them for communications related to i and j. However, different distributions of nodes across groups may give different link sharing benefits. Figure 9(a) shows the results of simulations upon 520 nodes with 60 groups of 100 nodes each. Nodes are uniformly distributed across available groups. Parameter k is set to 5. For readability, only 100 nodes are shown in the figure. The figure shows that the real number of existing application links per node is much lower than the worst case (which is k times the number of groups to which a node belongs). This is evident since the worst-case envelope is well above the real-case envelope in the figure. This result is due to link-sharing across group intersections, which allows the overlay to use fewer application links when it is solicited by the application. Figure 9(b) repeats the experiment with a different distribution: the nodes are distributed among the groups following a normal distribution with mean 260 and a standard deviation of 104. This case shows that the number of application links at each node grows at a lower rate than the worst case, thanks to link sharing.

Update Propagation. This experiment is done on a 5000-node network. No stub-transit topology is used, and inter-node latency varied randomly between 10ms and 50ms. The goal is to evaluate the scalability of the subgroup overlay to propagate data and, in this case, the scalability of the proposed update mechanism. Figure 10 shows little variation in update propagation using 62, 250, 500, 1000, and 2000 members per group. The latency increase is small thanks to the characteristic path length of the group, which always

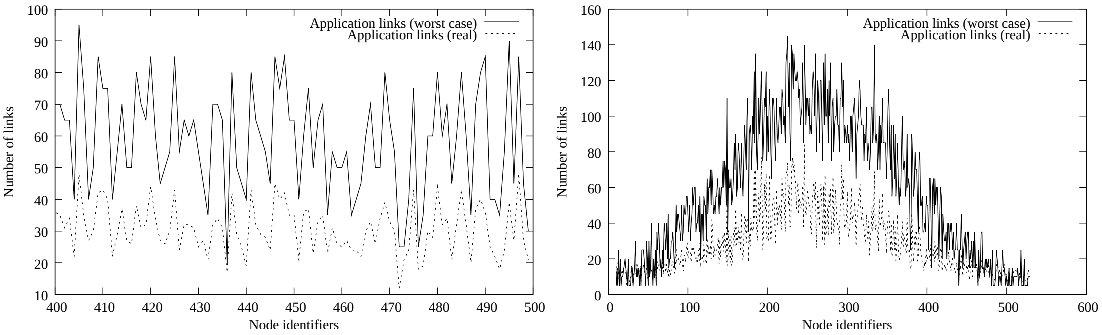

Figure 9: Link sharing among groups. The left plot shows how link sharing is below the worst case when nodes are uniformly distributed across groups. The right plot shows the case when nodes are distributed across groups following a normal distribution.

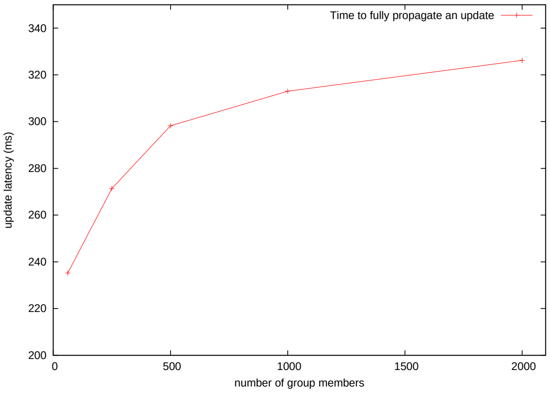

Figure 10: Time taken to propagate an update on all replicas. Small latency is achieved due to small characteristic path length.

stays below 4.

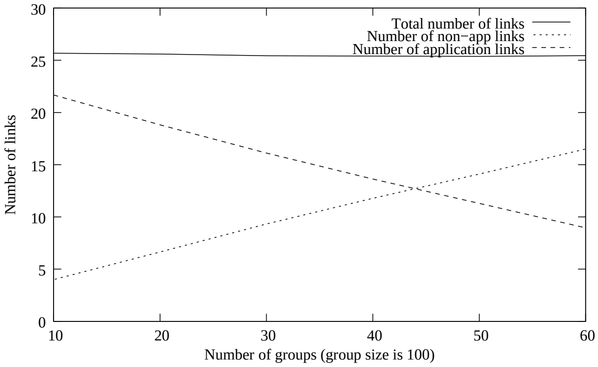

Figure 11: Controlling the clustering

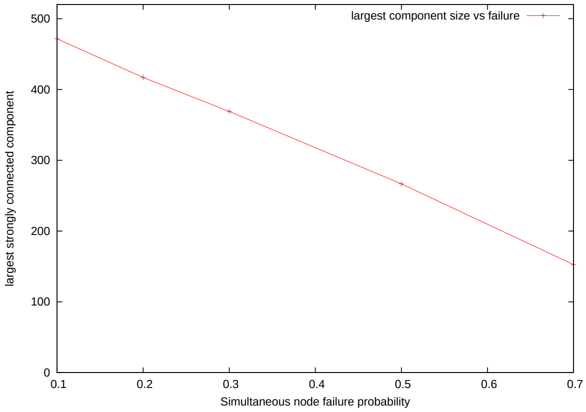

Figure 12: Resilience to simultaneous failures. Overlay partitioning was experienced only when the node failure rate was 0.70

Twisting the Overlay. We have also analyzed how the overlay reacts to the application needs (which may differ in the number and size of groups to be created). Simulations were run on a 520-node network, with a varying number of groups having a fixed group size set to 100 (and 5 as k parameter). The results (Figure 11) show that the total number of links is almost constant, while the border between application links and non-application links moves. The creation of groups leads to an increase of the number of application links, which progressively replace the non-application links.

Resilience to Node Failure. In this experiment, after 2 simulated hours, each node was subjected to a crash with some probability (all crashes occur simultaneously). The probabilities used were 0.10, 0.20, 0.30, 0.50 and 0.70. We measure the size of the largest strongly connected component immediately after killing the nodes. The points on Figure 12 are each the average over 5 simulations. Below a crash probability of 0.70, the largest strongly connected component is always the size of the remaining overlay (i.e., the overlay remains strongly connected). With 0.70 death probability, we experienced a small degree of partitioning: 2 or 3 nodes were disconnected on some of the runs.

6 Conclusion

The peer-to-peer approach is getting more and more attractive for building today's distributed applications, especially thanks to its very good properties in terms of fault tolerance and scalability. The way the P2P network (i.e., the P2P overlay) is built is important for P2P application performance. This paper describes *MOve*, a malleable overlay that applications can "twist": while retaining the scale and fault tolerance of the P2P approach, the overlay adapts, allowing the application communication to be optimized. MOve takes into account the *application topology*, which is defined as a set of groups of nodes that are supposed to have frequent interactions with each other. MOve allows such *group*-based applications (e.g., collaborative applications) to influence the underlying overlay, by replacing existing inter-node links with *application links*, in order to keep related nodes close to one another, to favor efficient data update propagation. However, the proposed algorithms still maintain a good connectivity.

Our experiments show that the proposed algorithms, which allow the overlay to adapt to the application, do enable efficient group communications. We show, on the other hand, that this optimization does not have a negative impact on connectivity: the graphs remain connected and have a good degree distribution (which is generally important for fault tolerance). Furthermore, the proposed refresh mechanism (which allows each node to periodically renew its list of neighbors) provides a good volatility-resilience.

We intend to further experiment the algorithms presented in the context of a large-scale data-sharing service using replica groups. We plan to implement *MOve* within the *JuxMem* grid data-sharing service [1] using the update mechanism to perform data replication. This will provide the ability to perform extensive experimental evaluations. Furthermore, it would be interesting to study how to adapt our approach to structured overlays, e.g. by adding (instead of substituting) *application links* to existing links of the overlay.

References

[1] Gabriel Antoniu, Jean-François Deverge, and S´ebastien Monnet. How to bring together fault tolerance and data consistency to enable Grid data sharing. *Concurrency and Computation: Practice and Experience*, (17), 2006. To appear.

[2] Bela Bollobas. *Random Graphs, Second Edition*. Cambride University Press, United Kingdom, 2001.

[3] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[4] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, number 2009, pages 46–66, Berkeley, CA, USA, July 2000.

[6] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *Proc. DSN 02*, Washington DC, June 2002.

[7] Sidath Handurukande, Anne-Marie Kermarrec, Fabrice Le Fessant, and Laurent Massouli´e. Exploiting semantic clustering in the eDonkey p2p network. In *Proc. SIGOPS European Workshop*, pages 109–114, Leuven, Belgium, September 2004.

[8] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. ICDCS*, pages 302–311, 1990.

[9] Màrk Jelasity and Ozalp Babaoglu. T-man: Fast gossip-based contruction of large-scale overlay topologies. Technical Report UBLCS-2004-7, University of Bologna, Mura Anteo Zamboni 7 40127 Bologna (Italy), May 2004.

[10] D. Kostoulas and D. Psaltoulis et al. Decentralized schemes for size estimation in large and dynamic groups. In *Proc. IEEE NCA*, July 2005.

[11] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[12] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[13] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, Heidelberg, Germany, November 2001.

[14] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. Networked Group Communication*, pages 30–43, 2001.

[15] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM 2001*, pages 149–160, San Diego, CA, August 2001.

[16] Spyros Voulgaris and Maarten Van Steen Daniela Gavida. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), june 2005.

[17] Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec, and Maarten van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *IPTPS'06: the fifth International Workshop on Peer-to-Peer Systems*, Santa Barbara, USA, FEB 2006.

[18] Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proc. 11th Euro-Par*, LNCS, pages 1143–1152, Lisboa, Portugal, August 2005. Springer-Verlag.

[19] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, (393):440–442, June 1998.

[20] KaZaA. `http://www.kazaa.com/`.