On Overlay Construction and Random Node Selection in Heterogeneous Unstructured P2P Networks

Vivek Vishnumurthy and Paul Francis
Department of Computer Science, Cornell University
{vivi, francis}@cs.cornell.edu

***Abstract*— Unstructured p2p and overlay network applications, including several that the authors wish to build, often require that a random graph be constructed, and that some form of random node selection take place over that graph. A key and difficult requirement of many such applications is heterogeneity: peers have different node degrees in the random graph based on their capacity. Using simulations, this paper compares a number of techniques—some novel and some variations on known approaches—for heterogeneous graph construction and random node selection on top of such graphs. Our focus is on practical criteria that can lead to a genuinely deployable toolkit that supports a wide range of applications. These criteria include simplicity of operation, support for node heterogeneity, quality (uniformity) of random selection, efficiency and scalability, load balance, and robustness. We show that all these criteria can more-or-less be met by all the approaches. Our novel approach, however, stands out as the best from a practical perspective because of its simplicity: it achieves the criteria while requiring each node to set only a single tuning parameter, its desired relative load.**

I. Introduction

Many unstructured P2P and overlay networks are based on random graphs of one sort or another. There must of course be some procedure to create and maintain these graphs. In addition, P2P applications often require that nodes randomly select other nodes. These two requirements—building a random graph, and doing random node selection within the graph—typically share a common mechanism: the *random walk*.

This paper is motivated by the fact that we (the authors) wished to build several new unstructured P2P applications, e.g., overlay multicast and proximity addressing, that require the construction of **heterogeneous graphs, and** random node selection on top of these graphs. We decided that it would be preferable to build a single toolkit for graph construction and random selection, and use it for all the projects. Indeed, we feel that such a toolkit could potentially serve other P2P applications as well. When we searched the literature, however, we could not find a complete solution that satisfied our requirements. Since existing approaches were studied in very different contexts, there was also no apples-to-apples comparison of them. We also wanted to avoid structured (DHT) approaches, in spite of the fact that they can potentially be adapted to be used for heterogeneous overlay construction([1], [2]) and random selection, because unstructured P2P applications have been so successful, and because our applications are well-suited to unstructured graphs [1].

The work reported in this paper makes two broad contributions: (i) We design graph building and random node selection algorithms that are practical to deploy and that are functional over a wide range of requirements. Towards that end, we considered variations on existing approaches as well as new approaches. (ii) Using simulations, we compare the various approaches and identify our novel technique Swaplinks as the most attractive graph construction mechanism. The simulation results also help in the basic understanding of the approaches, so that applications that have requirements not satisfied by our techniques may nevertheless gain from this knowledge.

A. Motivation and Requirements

We develop our set of requirements by discussing the applications that drive the need for random graph building and node selection.

In this paper, we refer to the target applications as 'P2P applications', but we use the term P2P broadly to include overlays or any distributed applications where nodes (end computers or communications devices) must organize into a graph. We also assume that the large majority of nodes in the graph are able to form links with each other. In other words, the nodes are connected to a physical network that allows communications between virtually all participating nodes.

Clearly, among the foremost of requirements of an algorithm that supports widely used p2p applications is that it be *scalable*. All P2P applications also require a graph that is robust against partition. Our basic approach to building graphs is for each node i in the graph to establish a fixed number of links K_i, called *outlinks*, with randomly selected nodes in the graph[2]. The reason different nodes would have different outdegrees is to accommodate heterogeneity in node capacities. As a result, K_i simultaneous failures are needed in general to partition node i from the network, and many more are needed to partition a group of nodes. If fewer than K_i simultaneous failures of a node's neighbors occur, the node can discover new neighbors to reestablish the constant number of outlinks. In the worst case, if all of a node's neighbors simultaneously fail, the node can rejoin the network from scratch. Given all this, and the fact that the graphs we build exhibit good control over node degree so do not have "soft

[1] Having said that, our intent is certainly to follow this work up with a solid comparison of our best unstructured approach, Swaplinks, and a DHT-based approach.

[2] We make only a logical distinction between outlinks and inlinks, for control over degree distribution. In particular, message flow can occur in either direction over any link.

spots" in the form of very few nodes that have very large degrees, from a practical perspective, our random graphs are robust to partition, and we do not feel a need to further address this issue.

One of our requirements, both for building graphs and for random node selection, is that *the load on nodes be well-balanced and controllable*. For instance, all nodes should carry roughly the same load if uniform load balance is desired. On the other hand, in many cases certain nodes should carry more load than others, for instance because they have more capacity or higher access link bandwidth. This desire for control over load manifests itself in several ways. These include the number of messages a node handles, and the number of links the node obtains. Since every node can control its number of outlinks, this means that our graph building algorithm should give us *control over the number of inlinks* (and indeed it should be roughly the same as the number of outlinks). Note that we do not assume perfect control over load: after all there is a significant random component in the algorithms and graphs. Rather, we expect statistical control, along the lines of what would be possible with true random selection.

Note that control over node degree is important for several reasons. One is that we assume that there is a certain cost to maintaining a link—for instance in the periodic keep-alive messages used to determine if a neighbor node is still active. Also important is the fact that the application load on the node may be proportional or otherwise closely related to its node degree. An example of such an application is file search in unstructured file sharing networks like Gnutella, Kazaa, or GIA [3]. Accordingly, the authors of [3] propose that node degree be related to capacity, in order to not stress low-capacity nodes.

Once a graph is built, with control over load and node degree, we also require similar *control over the walks taken on the graph.* Here, we want control over the probability that a node will be *visited* and *selected* in random walks. A node is selected when a random walk ends at that node. A node is visited when a random walk traverses that node during a walk.

Control over visits is important for two reasons. First, nodes experience load every time a node visits them. If we wish to have control over load, then we correspondingly need to have control over how often nodes are visited. Second, some applications execute application functionality every time a node is visited during a walk. For instance, in many file search algorithms ([4], [5], [3], [6]), each node visited is searched for the desired key words. Note that much of this effect can be obtained by establishing the appropriate node degree in the graph.

A number of applications require random node selection as a way of configuring application-specific topologies. Examples of these include overlay multicast or file distribution applications [7], [8], [9], the file sharing applications mentioned above [5], [3], and "proximity addressing" applications [10]. (The latter is an application where nodes form addresses that can be used to indicate how close nodes are to each other in the network.)

"Random selection in overlay multicast" changed below
Note that some applications(e.g., overlay multicast applications like Yoid [7]) might need two separate graphs: one for normal operation (e.g., the multicast graph), and the other a random graph of the kind we discuss in this paper. The first graph here might have been built keeping constraints like network proximity in mind, and thus is not completely random, and therefore not as robust as a random graph. Maintaining the second (random) graph thus makes the application more robust to partition, while also giving the application the ability to select random nodes in the graph.

BitTorrent is a P2P file distribution protocol whereby nodes feed each other blocks of a file [9]. BitTorrent uses a central node called the tracker that keeps track of existing participants, and provides downloaders with a random subset of participants with the file they are looking for. While this approach works reasonably well, in environments where even a tracker cannot adequately scale, random selection as described in this paper may be used.

Finally, a key requirement that permeates all of our work is that of simplicity. This requirement goes beyond the basic notion that, all other things being equal, simple is better than complex. We believe that algorithmic simplicity is central to achieving scalability. Our intuition is that, as networks grow, more complex algorithms will exhibit more failure modes and ultimately limit scalability even where the basic algorithms scale according to traditional measures such as memory and message overhead. Indeed we would be willing to pay a small penalty, say in the uniformity of random selection, if we can gain significant simplifications in doing so.

To summarize, our requirements for a random graph building and node selection mechanism are: scalability (realistically millions of nodes), simplicity, robustness, selection independence, control over node selection probability, control over node degree, and control over message load. The first five are hard requirements, whereas the last two are strong desirables. A mechanism satisfying these properties can then serve as a foundation or an accessory for numerous unstructured p2p applications like file-sharing, overlay multicast, and proximity addressing.

edited!! The rest of the paper is organized as follows. Section II describes related work. Section III describes how joining nodes get to know of already existing nodes in the graph. Sections IV and V describe the four basic approaches for both building graphs and walking them. Section VI presents detailed results of simulations used to evaluate the performance of the four approaches. Section VII concludes and outlines next steps.

II. Related Work

No discussion about Narada
GIA [3] is an unstructured file sharing system that uses random walks rather than flooding to do file searching. Its goal is to give high capacity nodes more of the application load than low capacity nodes, both by giving high capacity nodes higher degrees and more information to store, and by routing more

search queries to them. GIA does not give direct control over degree or load, and indeed [3] does not indicate how much load each node gets, and only gives very limited data on node degree. As such, GIA is tailored to the file sharing application and its random walks cannot be used as a general purpose graph building or node selection mechanism. Our approaches, on the other hand, while being simple, provide an amount of control that GIA does not currently have. While it is clear that GIA requires more functionality than our approaches provide (flow control, for instance), it may well benefit from this work.

SCAMP [11] uses random walks to build graphs suitable for gossiping. An interesting goal of SCAMP was to build graphs where the average node degree is proportional to the log of the number of nodes While the ability to tie node degree to graph size is a desirable property for some applications, we wanted more control over node degree than SCAMP allowed and so did not find it useful (though we do simulate it as a point of comparison).

Bullet multicast [8] uses a random selection mechanism called RanSub [12]. RanSub operates in waves of network-wide coordinated phases, where in each phase lists of random nodes are distributed through the network. The nodes learned by a given node at a given time are not random relative to nodes learned by another node at the same time. The phases must be run multiple times if different nodes are to ultimately select different sets of other nodes. This approach lacks flexibility. For instance, even if given nodes do not need to select other nodes, they must anyway continue to learn of other nodes. An approach where any given node, at any time of its own choosing, can discover one or more random nodes without burdening other nodes too much, is preferable.

Araneola [13] builds almost regular graphs that could potentially be used for random selection. But [13] does not discuss the case of heterogeneity, and makes the assumption that the existing nodes contacted by newly joining nodes are uniformly picked; this might not be the case in practice. Araneola also needs to run constant background protocols like gossip of membership views, exchange of neighbor degrees, etc., and we would like to avoid this kind of complexities if we can.

Law and Siu [14] give a distributed mechanism to construct regular random graphs, but their scheme is vulnerable to unexpected node departures. Our SwapLinks approach builds on the spirit of this approach to handle unexpected node departures and do away with the dependence on the rigid Hamilton cycles structure in [14].

Two random walk approaches that we closely study in this paper are Self-loops [15] and Iterative Scaling [16]. These methods are not suitable to use, as is, for graph construction or to accomodate heterogeneity. In section IV, we discuss how we extend these techniques to adapt them to our setting.

III. Initial node discovery

Any new node that wants to join the graph needs to know at least one already existing member in the graph. While our algorithms work with any scheme that helps new nodes discover existing nodes, a practical and simple approach we use for this purpose is to establish a rendezvous node at a well known location (a DNS name or IP address). Joining nodes first contact the rendezvous node, which tells the joining node of previously joined nodes. The rendezvous node could tell joining nodes about the same small set of joined nodes, but this puts an undue load on those joined nodes. The rendezvous node could remember all joining nodes, and tell the joining node of some small random subset, but this puts an undue burden on the rendezvous node.

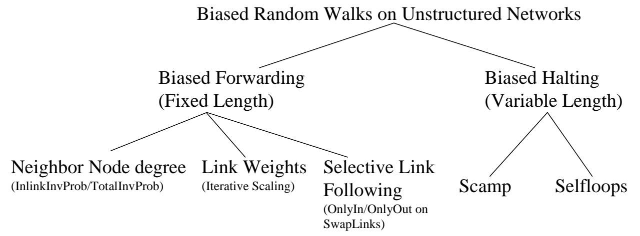


Fig. 1. Classification of Biased Random Walks.

Therefore, we assume a very light-weight approach whereby the rendezvous node remembers a small set(10) of the most recently joined nodes. New nodes enter the network by contacting some (or all) of these nodes. This approach effectively spreads the load of node discovery. Note, however, that even this approach requires caution, because with a naive graph-building scheme, this approach can lead to "long-thin" (high diameter) networks. Nevertheless, in the remainder of this paper, we assume this style of node discovery. This discovery mechanism can be made more robust by having the rendezvous node remember an additional small set of stable random nodes known to be up with high probability (which the rendezvous node can discover with selection walks). The scenarios we test in this paper however do not need such a measure, so we do not use the more robust rendezvous scheme here.

Note that the rendezvous node can be replicated, and one can be selected by the joining node using DNS or even an Internet Protocol (IP) form of delivery called IP anycast. In this case, however, the rendezvous nodes must take care to keep each other informed of the initial joining nodes so as to avoid a graph partition in the early stages of its formation.

IV. Algorithms for graph construction

A truly random walk, whereby each node selects uniformly randomly among its neighbors, will select high degree nodes proportionally more often than low degree nodes simply because more links lead to those high degree nodes. Therefore, unless the graph has perfectly uniform node degrees, the random walk must somehow be biased against high degree nodes.

While this is true both for walks used for the purpose of selecting nodes to build the graph (*build walks*), and for walks used for other node selection (*selection walks*), the problem is more severe for build walks. This is because any favoring of high-degree nodes by build walks will compound itself as the network grows. If a node obtains a slightly higher than average node degree, the subsequent joining nodes will select

it more often and choose it as their neighbor, thus giving it an even higher node degree, thus making it a target for yet more neighbors. Indeed, it is not enough for build walks to simply negate the effect of node degree, so that selection is uniform. The reason for this is that early joining nodes participate in more "selection trials"– they get more chances to be selected as neighbors by joining nodes than do later nodes. Therefore, there must be additional bias or mechanisms to ensure high degree nodes do not keep collecting more links.

Actually, our situation is even more difficult than this. In addition to the above, our requirement of heterogeneous node capacities requires random graphs where higher capacity nodes have proportionally higher node degrees than lower capacity nodes. Further we require that walks visit and select nodes in proportion to their capacities. Our basic approach to heterogeneity is that high-capacity nodes establish more outlinks than low-capacity nodes. For instance, if the lowest capacity node establishes 5 outlinks, a node with twice that capacity will establish 10 outlinks. Our build must therefore operate in such a way that nodes obtain roughly as many inlinks as they have outlinks (within random variations). We refer to this as the *expected node degree* or *expected indegree*.

There are two fundamental approaches to counteract the effects of early joiners obtaining more inlinks, and the self-reinforcing trend of high-indegree nodes becoming even higher-indegree nodes[3]. One approach is to simply endow build walks with an even stronger bias against high-indegree nodes, so that nodes never get high indegrees. There are several ways to do this, which are shown in the taxonomy of Figure 1. The other approach is to actively manage each node's indegrees, so that nodes explicitly shed inlinks when they get too many. The basic mechanism, which we call *Swaplinks*, is for nodes with high indegrees to move an inlink to nodes with low indegrees. We next describe the taxonomy of the biased walk approaches, and then go on to describe each of the biased walk approaches we study in this paper. We then discuss the Swaplinks approach in Section IV-E.

Taxonomy of Biased walks:
In our biased walk approaches, the basic graph building mechanism is for a joining node i to establish and maintain a constant number of outlinks K_i with nodes discovered by taking K_i random build walks. If an outlink is lost, for instance because the neighbor crashes or leaves the network, the node reestablishes the outlink by taking another biased random walk and adding an outlink to the discovered neighbor.

Note that in all our biased walk approaches, a node never has the option of refusing a request to create an inlink. One could easily imagine a scheme where we could do this, for instance by not terminating a build walk at a node if its indegree-to-outdegree ratio is above some constant. We chose not to consider such approaches in part because the bias tends to prevent the need for this, and in part because we wanted to keep our approaches so that we could better understand their fundamental characteristics.

[3] In general, when we say "high indegree", we mean "higher-than-expected indegree".

Note also that any given random walk may fail, for instance because of packet loss or sudden node failure. In this paper, we assume that any node initiating a walk will repeat it if it does not succeed within some short time.

Looking at the taxonomy in Figure 1, we see that there are two fundamental ways to bias a walk, which we call biased-halting and biased-forwarding. In biased-halting, the next hop at a node is picked uniformly at random from among all of the links at the node – there is no weighting in this regard. Instead, the walk is ended at each node with a random probability that is weighted inversely to the degree of the node. The result is that the length of each walk is variable, though the average length can be fixed. We discuss the *Selfloops* style of biased-halting walk in section IV-A. SCAMP, discussed in section II, also uses biased halting walks to find neighbors for newly entering nodes.

In biased-forwarding, the random selection of the next hop in the walk is weighted against high degree nodes. In these walks, the number of hops is set at a fixed constant H, which must be long enough to allow the walk to *mix* into the network – a constant times the diameter of the network[4]. The biased-forwarding walks we study are *InlinkInvProb*, *TotalInvProb*, and *Iterative Scaling*, discussed in sections IV-B – IV-C.

There are trade-offs between the biased-halting and biased-forwarding approaches. On the one hand, biased-forwarding requires nodes to exchange state about their neighbors–their node degree or a more general weighting. Biased-halting requires no special knowledge of neighbors. On the other hand, biased-halting walks tend to unfairly load high-degree nodes, because walks tend to be forwarded to high-degree nodes, only to continue on with high probability.

A. Selfloops

Biased-halting approaches are ideal in settings where the graph is not under one's control, and the cost of calculating weightings is high. Indeed, the biased-halting approach we use is based on work by Bar-Yossef et al [15], who used it to select web pages with uniform probability. Their approach, which we call *SelfLoops*, is elegant and intuitively appealing. The basic idea is to emulate a graph with perfectly uniform node degrees by adding virtual links to oneself (i.e. self loops!) For example, say that the target uniform node degree (the virtual degree)is 100. A node with 90 real links would add 10 virtual links to itself. A node with 25 real links would add 75 virtual links to itself. Subsequently during a walk, each "link" is selected with equal probability, and the virtual walks are of "fixed length", though the real walks are not. In practice, for uniform selection, the virtual degree is set to a large constant at each node, and the value used for the virtual hop length can be set such that the average real hop length is as needed.

The Bar-Yossef approach, as defined, does not support heterogeneity or provide the needed bias for build walks.

[4] Given that the diameter grows slowly with the size of the graph, and given the range of network sizes and node degrees this paper examines, we can simply pick a conservative value like $H = 10$.

We modified the Bar-Yossef approach as follows to make it useful in our setting. For selection walks, the virtual degree of each node is made directly proportional to its outdegree. For build walks, the virtual degree is directly proportional to the square of the outdegree and inversely proportional to the indegree ($\frac{od^2}{id}$) (see Table I). This modification of the virtual degree for build walks leads to the desired situation where the expected indegree of each node is equal to its outdegree. To see why, assume a well refreshed graph that has reached stable degree distribution [5]. Now examine the change in indegree of node i when another node r performs a refresh. Since the steady state has been reached, the net change in the expected indegree of i due to the refresh is zero. Now the probability that i was an out-neighbor of r before the refresh is given by $c \cdot indeg(i) \cdot outdeg(r)$ where c is a constant [6]. So the probability that i loses an inlink because of the refresh is given by $c \cdot indeg(i)$. The probability that i gains an inlink because of the refresh is given by $c' \frac{outdeg(i)^2}{indeg(i)}$, where c' is a constant. Thus we get $indeg(i) = c'' \cdot outdeg(i)$, where the constant of proportionality is of course 1. We show later through simulations that the linear dependence on outdegree is achieved even without refreshes.

With this modification though, it gets much harder to estimate the virtual hop length to use to achieve a desired average real hop-length during graph construction. A conservative option is to use a large enough value, but this results in a larger average hop-length. In our experiments, we use trial and error to estimate the virtual hop-length. This lack of tight control on the average hop-length is one of the drawbacks of the Selfloops approach.

Note that one of the problems with biased-halting walks is that any given walk can be quite short. For instance, if the walk length is set to terminate after an average of 10 hops, then there is a very small chance that a walk will end at one hop, a bigger chance the walk will end within two hops, and so on. Such short walks clearly do not mix well, so we experimented with a hybrid approach where if the expected walk length was h hops, the walk could not terminate within $h/2$ hops. For the first $h/2$ hops, we use one of the biased-forwarding walks described below (specifically the TotalInvProb walk), and for the later half we use SelfLoops. We call this hybrid TotalInvProb-SelfLoops (Hyb-TIP-SL).

B. The Inverse-Probability walks

In this style of biased-forwarding walks, the bias in forwarding a walk is directly proportional to the outdegree of the node and inversely proportional to either its indegree (*InlinkInvProb*, or IP) or the total degree (*TotalInvProb*, or TIP). The former produces a stronger bias, and is used for build walks. The latter is used for selection walks.

Note that one could invent any number of inverse weightings

Name	Link Weighting	
SwapLinks(SW)-Normal/Outlink-loss	$w_N^A = \frac{1}{indeg(A)}$ if $N \in$In-nbrs(A)	
SwapLinks(SW)-Inlink-loss	$w_N^A = \frac{1}{outdeg(A)}$ if $N \in$Out-nbrs(A)	
InlinkInvProb(IP)	$w_N^A \propto \frac{outdeg(N)}{indeg(N)}$	$\Sigma_{N \in Nbrs(A)} w_N^A = 1$
TotalInvProb(TIP)	$w_N^A \propto \frac{outdeg(N)}{totaldeg(N)}$	$\Sigma_{N \in Nbrs(A)} w_N^A = 1$
SelfLoops(SL)-Selection	$w_N^A \propto \frac{1}{virt-deg(A)}$	$virt - deg(A) \propto outdeg(A)$
SelfLoops(SL)-Build	$w_N^A \propto \frac{1}{virt-deg(A)}$	$virt - deg(A) \propto \frac{outdeg(A)^2}{indeg(A)}$
Iterative Scaling(IS)-Selection	$\Sigma_N w_N^A = 1$	$\Sigma_N (outdeg(N) \cdot w_A^N) = outdeg(A)$
Iterative Scaling(IS)-Build	$\Sigma_N w_N^A = 1$	$\Sigma_N \left(w_A^N \cdot \frac{outdeg(N)^2}{indeg(N)} \right) = \frac{outdeg(A)^2}{indeg(A)}$

TABLE I

SUMMARY OF THE DIFFERENT WALK STRATEGIES FOR A WALK AT NODE A; NODE N IS A NEIGHBOR OF A, AND w_N^A IS THE PROBABILITY THAT A WALK AT A IS FORWARDED TO N. $virt - deg$ DENOTES THE VIRTUAL DEGREE.

derived from neighbor node degree (square of the degree, square root of the degree, etc.). Though we did explore these variations, we found the above approaches (IP and TIP) to be adequate for our purposes, and therefore do not report any of the other variations in this paper.

C. Iterative Scaling

In the other style of biased-forwarding walk, an iterative distributed computation is executed across all nodes that allows each node to assign weights to all links. The computation, called *Iterative Scaling* (IS), is based on a technique used to derive the elements of a matrix when the row and column sums are known [16]. SCAMP applied this iterative scaling technique to random walks as a means of randomly selecting an "introducer" node that helps a newly entering node join the network [17]. To employ the Iterative Scaling scheme in a graph setting such as ours, each node (say A) assigns outgoing and incoming weights to each of its links, where the outgoing weight of a link corresponds to the probability that the link is picked during a random walk from A, and the incoming weight corresponds to A's perception of the probability that A is picked during a random walk from the other end of the link.

Nodes periodically normalize their weights by scaling their incoming (outgoing) weights so that the incoming (outgoing) weights add to 1, and exchange weights through updates: when node A receives a weight update from neighbor B for the link A-B (denoted l), A would set $wt_{in}^A(l) = wt_{out}^B(l)$ and vice-versa. ($wt_{in}^A(l)$ denotes the incoming weight assigned by A to link l). The weight scalings and updates are intended to bring the system to a state where at every node both the incoming and outgoing weights add to 1, so a sufficiently long random walk is equally likely to end at any node.

To accommodate heterogeneity and the different biases for

[5] A refresh is where a node discards one of its outlinks and chooses another; a refreshed graph is one where all nodes have performed a large number of these refreshes. Refreshes are discussed in more detail in Section IV-D.

[6] This follows from the assumption that each node's degree is negligibly small when compared to the total number of links in the network

build and select walks, we modified the Iterative Scaling approach similarly to how we modified the Bar-Yossef approach. When used for selection, the ideal probability that a node is selected is proportional to its outdegree. When used for building, the ideal value is directly proportional to the square of the outdegree and inversely proportional to its indegree. So, when weight updates are performed at a node A, the incoming weight for each link A-B is scaled by the estimated probability of a walk reaching B (which is $k \cdot outdeg(B)$ for selections and $k \cdot \frac{outdeg(B)^2}{indeg(B)}$ for graph build) before the normalization is performed.

D. Some Issues with Biased Walk Approaches

Exchanging neighbor information: Given that the biased-forwarding schemes require nodes to have knowledge about their neighbors–explicit with inverse probability (IP), implicit with iterative scaling (IS)–we must address the question of how this knowledge is obtained. At one extreme, with IS we could run the distributed computation to steady state every time there is a link change somewhere. This is obviously not practical, as links may come and go at a rapid rate, and not really necessary either because in any event the effect of a link change diminishes rapidly with distance from the link. With IP or IS we could have each node send a message to all of its neighbors every time it experiences a link change. This is still somewhat heavyweight, but certainly reasonable. A third approach is to simply piggyback the neighbor information on the random walk messages. This will result in less accuracy, but is simpler and more efficient.

Note that it may or may not be possible to piggyback neighbor information on the periodic keep-alive messages used by nodes to determine if neighbors are still up. The reason is that, for high-degree nodes (or for nodes that belong to a large number of low-degree graphs), it is easy to imagine an optimization whereby only a few of a node's many neighbors probe for liveness. These few neighbors would then tell the remaining neighbors if the node went down. In this case, the node obviously cannot convey periodic information to most of its neighbors.

Graph refreshes: As described above, build walks have a stronger bias in order to counteract the effect of early joining nodes having more opportunities to obtain neighbors. One of the effects of this bias is that joining nodes have a higher probability of attaching to more recently joined nodes than old nodes, thus removing some of the randomness from the graph. And, in spite of the bias, older nodes inevitably accumulate more links(as described earlier); this too detracts from the randomness of the graph. One way to counteract this is for nodes to periodically remove an outlink and replace it with another randomly selected node. We call this process *refreshing*. As our results show, refreshing can have a strong improvement on the quality of the graph.

Refreshing has a number of negative aspects though. One is its overhead. Another is that graph changes may negatively affect the application using the graph (though to be fair in none of our example applications is this a problem). A third is simply that it introduces a new engineering requirement into the system. With refreshing, one now has to ask how often to refresh, when it is no longer necessary to refresh, and so on. All things being equal, it is better not to have to ask and answer these questions.

Note that churn, where nodes leave the network, has the same effect as refreshing.

E. SwapLinks

SwapLinks is inspired by, but quite different from, the approach used to build random graphs by Law and Siu [14]. The basic idea in [14] is that when a joining node A adds an outlink to a node B discovered during a build walk, one of the inlinks of node B is transfered to node A. This has the effect of maintaining a constant number of inlinks at node B, and of giving the joining node A the same number of inlinks as outlinks, which is our goal. Indeed, if a graph only grows (nodes never leave), then every node will have identical indegrees and outdegrees.

The wrinkle to this approach is when nodes leave. If we want to maintain the invariant of all nodes having exactly the expected indegree, as Law and Siu do, then the procedure becomes quite complex. Law and Siu outline an approach, but it is not robust to abrupt node departures. Their basic approach is that each departing node would help all of its neighbors form new links so that the invariant is maintained after the departure as well. To make this robust against abrupt departures, we might need to have each node know some or all of its neighbors' neighbors, but then this will fail in the presence of simultaneous multiple departures. Dealing with all of this would require additional mechanisms not specified by [14], and makes this approach unattractive.

However, if we relax the constraint of having to maintain the perfect indegree invariant at all points of time, then the problem of handling churn becomes much more tractable. Before we discuss how our Swaplinks technique handles churn, we need to provide definitions of two kinds of walks used solely with Swaplinks:

OnlyInLinks: This is one type of random walk that is essentially a biased-forwarding walk, but in fact requires no knowledge about the neighbors. In this fixed-length walk, each node chooses uniformly randomly among its inlinks only. The idea here is: when the indegrees of nodes are close to the outdegrees, walking only inlinks results in selection roughly proportional to each node's outdegree. OnlyInLinks itself though cannot be used to build graphs, because the rendezvous server would return a list of the most recently joined nodes, and since all links point from new nodes to older nodes without refreshes, walking only the inlinks would never take the walk outside this set of recent nodes. The end result would be a "long and skinny" network, one with a large diameter, and therefore not desirable.

OnlyOutLinks: This is the analogous walk where each node chooses uniformly randomly among its outlinks.

The Swaplinks approach works as follows. When a node joins, it follows the procedure described above–for every

node with which it forms an outlink, it steals one randomly selected inlink. The build walk used for selecting the node is OnlyInLinks. This works in this case because the swapping of links mixes the graph sufficiently to completely avoid any trend towards newly joined nodes.

If a node A loses an outlink (due to node deletion), then it replaces the outlink with a new neighbor O discovered with an OnlyInLinks build walk. Unlike the case of a new node join though, now O does not donate any of its inlinks to A, as A is not looking for inlinks here. Analogously, when a node B loses an inlink due to a node departure, B checks if its indegree is less than its outdegree. If so, it needs to establish a new inlink. It does this by launching an OnlyOutLinks walk to discover a node I that has high indegree (with high probability). A randomly selected in-neighbor of I now discards its outlink with I, and forms a new outlink with B, thus pushing both B's and I's indegree toward its ideal value[7].

Now consider a sequence of node deletions. Assuming that the indegrees of the deleted nodes are close to the respective outdegrees, we will have rougly the same number of broken outlinks and broken inlinks as a result of the deletions. Now when a node A repairs its broken outlink, it forms a new outlink to a new neighbor O, thus increasing O's indegree, in turn increasing the likelihood that O is chosen for the purpose of repairing a broken inlink by some other node, which results in O's indegree dropping back to its earlier value! Thus the churn-handling mechanism described above ensures that the degree distribution never gets too far from the desired distribution, even after a long sequence of node departures. (Section VI has the related results)

Although the biased walk approaches have a certain elegance to them, SwapLinks has a certain engineering appeal. In particular, there are no engineering decisions required about how to exchange information between nodes (as in biased-forwarding), and how often to refresh (as in both biased-forwarding and biased-halting), and no uncertainty about how long walks may take (as in biased-halting). Perhaps the only negative of SwapLinks is that there is extra overhead when a node leaves, because sometimes two walks must be taken (to replace both outlinks and inlinks) instead of just one.

V. Selection Walks

The previous section focused on graph building. The four walks described, however, can be used for selection over any of the graphs – how a graph is walked is independent of how it is built (assuming that the necessary neighbor information is exchanged during building). To summarize, they are Total Inverse Prob (TIP), Iterative Scaling (IS), SelfLoops (SL), and the hybrid TIP-SL(Hyb-TIP-SL).

There is an important limitation to the SL and hybrid TIP-SL approaches that result from the fact that SL is a biased-halting scheme and therefore has variable length walks. Specifically, the file sharing applications described in Section I-A require long walks where work is done (a local file search) at each node visited. SL walks, however, do not exhibit uniform selection during the walk, as each step is unbiased. Rather, they only exhibit uniform selection upon ending.

While the file sharing application is an important one, more generally the notion of a node starting a walk from the node where the last walk ended, instead of from itself, is useful. We refer to these types of walks as *cursor* walks, due to the fact that the last node visited can be seen as a cursor pointing to where to start next. The cursor walk works as follows: the node initiating the walk remembers the previously selected node *P*, and when the next selection is to be performed, takes a short (1 to a few hops) walk from *P*, instead of starting each walk from itself. The first random selection here is performed in the usual non-cursor manner, and the subsequent selections are performed using the cursor.

In addition to being suitable for applications like the file-sharing application, the cursor approach reduces the imposed load and latency by an order of magnitude, at the cost of maintaining information about the cursor. Further, by spreading the selection load uniformly across the network, it improves the load balance in scenarios where a small set of nodes initiate the majority of the random walks, whereas in the non-cursor approach the initial load during any random walk is necessarily borne by nodes close to the initiating node.

It should be noted, however, that individual cursor selections are not very random relative to the immediately preceding cursor selections (see Section VI-F). Over a long walk, however, the selection does tend towards uniform distribution.

VI. Experimental Results

We start by describing the simulations used to evaluate the various approaches. We use static (non-time based) simulations. When simulating node additions or deletions, each node is fully added or deleted before the next node is added or deleted. Likewise, there is no notion of packet loss. While the simulations are not therefore fully realistic, we believe that they reflect the basic characteristics of the various approaches, and allow them to be legitimately compared. We believe this in part because of the random nature of our techniques—neither the order of events or the timing of events are very important.

We examine two graph building scenarios:
(i) *Shrink:* A graph is built with a given number of nodes N-without any churn until all nodes have joined- and then nodes start leaving one at a time until the graph shrinks to 25% its original size
(ii)*Churn:* An N-node graph is built - without any churn until all nodes have joined - and then there are $2N$ churn-events, where a churn-event consists of either a single node kill or a single node join, with the same probability. The expected network size after this sequence of events is N. In all our measurements, unless otherwise mentioned, we set N to 5000.

When the network only grows, i.e., when nodes only enter without leaving, SwapLinks' degree distribution (by design) is perfect, and therefore is not a fair comparison; we do not present these results here. On the other hand, the other schemes

[7]Note that a walk is initiated here only if some node departure led to a link loss; in the above instance, I will not launch any walks as a result of its losing its inlink to B.

perform worse during the grow-only phase than they do under churn, because of the refreshing nature of churn(see below).

To measure the quality of random selection, we run $10.M$ selection walks using the algorithm to be evaluated, where the underlying graph has M nodes at the time of selection (i.e., after the churn or shrink has completed), and look at the distribution of the selected nodes, and the selection load balance.

To model heterogeneity in our measurements, we use the following setting: Each of the N nodes in the graph is a *default-degree* node with probability 0.5, and a *heterogeneous* node with probability 0.5. Each default-degree node has an outdegree of 5. Each heterogeneous node chooses its outdegree uniformly randomly from the range [2,50]. As before, churn or shrink is performed on the graph after all nodes have joined and formed all their outlinks.[8]

The default setting we use in our experiments is: N=5000 nodes, build walk length of 10 hops, and, except in case of heterogeneity, a constant outdegree of 5 at every node. Ten hops was chosen because they produced better results than shorter walks, but longer walks did not perform significantly better than 10-hop walks. (In Section VI-E, we show that 10-hop build walks are sufficient for a wide range of network sizes.)

For simplicity, we ensure that all build walks only find nodes that are not already neighbors of the initiating node, by storing the initiator's neighbor-list in the walks. This could be easily simulated in a real implementation by having the initiator retry if a build walk ended at a node that is already a neighbor.

Given that we have four graph-building techniques, four selection walks, heterogeneity, cursor walks, graphs of different sizes, and numerous parameters to measure, we need a way to prune down the data set. We do this by first evaluating the four graph construction techniques in terms of the "goodness" of the graphs they generate. We look at graph construction when all nodes have the same outdegrees, i.e., the *homogeneous* case in section VI-A. We evaluate the performance of all the graph construction algorithms in conditions of heterogeneity in section VI-B. Looking at these results, we pick the most promising graph building algorithm, which is SwapLinks, and do most of our subsequent experiments on that graph. We examine the quality of random selection: first we execute the four selection schemes over a homogeneous SwapLinks graph in section VI-C, and then test all the selection walks over heterogeneous graphs (in this case over all the build methods) in section VI-D. We next look at the scaling behavior of the SwapLinks algorithm in section VI-E. Finally, we evaluate the cursor mechanism in section VI-F.

A. Graph Building

In this section we compare the different graph building algorithms in terms of the following parameters: degree distribution, network diameter and average distance between nodes, and distribution of the load placed on the network by the build walks. The graphs we study here are all homogeneous. We evaluated both graphs with and graphs without refreshes (except for SwapLinks, which does not benefit from refreshes). The refreshes are performed after the churn or shrink as described above has completed. For IS and IP graphs, we evaluated both the case where all immediate neighbors are informed immediately of any link change (*1-hop updates*) and the case where neighbor information is only piggy-backed on build walk messages (*Piggybacking*). We include in the comparison graphs built using SCAMP, and TrueRandom graphs, where each node forms 5 outlinks with distinct uniformly chosen nodes in the network.

Ideally, at any given time, the load caused by the entry of new nodes or departure of nodes should be spread uniformly over the existing nodes in the network. We verify this property in the following manner: 10 new nodes are added to the system and the load placed on each previously existing node(barring the last 10 joiners[9]), in form of the number of messages received by it, is logged. This is repeated a total of 100 times with the load summed over the 100 times, and finally the average load per node *Avgbload-add* and standard deviation of the load values *Dev(BLoad-Add)* of all nodes is computed. We chose the comparatively small number of nodes added (10) here, as we want to focus on the load placed on already existing nodes: with increase in the number of nodes added, there is an increase in the load placed on the new nodes themselves. Since this method of testing imposes the same load on the network irrespective of the size of the graph, the per-node load values are going to be higher for smaller graphs. To evaluate the load caused by node departures, we select $M/5$ nodes randomly, where the graph contains M nodes, and delete them (one by one) from the graph, and log the resulting load placed on the remaining nodes. We then compute the average load per node *Avgbload-kill* and standard deviation of the load *Dev(BLoad-Kill)* caused by the deletions.

Table II shows the results for the homogeneous-capacity graph building simulations. A noticeable trend is that all parameters improve with refreshes, the improvement with a churned graph being more noticeable than that with a shrunk graph. This is because the effects of shrink ensures that each node will have refreshed its out-neighbor set multiple times with high probability, so a shrunk graph is effectively equivalent to a refreshed graph.

Another key thing to note from the results is that they are almost all reasonably good as far as the degree distribution is concerned. For instance, the standard deviation in node degree for TrueRandom is 2.23, and the only graph that did significantly worse than that was InlinkInvProb where neighbor information was only piggy-backed. Most did better than TrueRandom.

SwapLinks' policy of neighbor replacement ensures it has

[8]By contrast, GIA simulated heterogeneity spanning three orders of magnitude. While indeed node capacities vary by this much in measured Gnutella networks, we do not believe that a node with 1000 times the capacity of a dial-up would be willing to devote all of that capacity to file sharing!

[9]The last 10 joiners would be unfairly heavily loaded because of the rendezvous scheme.

		Dev (Deg)	Indeg-95pc	MaxIndeg	Diam	Dist	Dev(BLoad-Add)	Dev(BLoad-Kill)	Avgbload-add	Avgbload-kill
Grow	TrueRandom	2.23	9	15.03	5.06	3.97	-	-	-	-
N=5K	SCAMP*	6.97	28.24	44.68	5.34	3.45	7.81	-	10.6	-
	IP-Norefs	2.23	9	15	5.19	3.98	12.32	7.08	15.27	33.68
	IP-10refs	1.82	8	13.2	5.03	3.98	6.28	6.93	17.56	33.84
Churn	IS-Norefs	2.04	8.05	13.4	5.27	3.99	13.81	6.95	15.49	33.47
N=5K	IS-10refs	1.57	8	11.8	5.03	4.01	6.88	6.74	17.58	33.4
	SL-Norefs	2.03	8	13.34	5.3	4	5.54	5.36	9.51	14.87
	SL-10refs	1.55	8	11.66	5.03	3.99	4.6	4.63	9.58	12.58
	SW-NoRefs	1.31	7	11.66	5.01	3.99	4.11	5.19	9.63	17.86
	IP-Norefs	1.83	8	12.65	4.75	3.38	18.85	6.95	69.05	33.84
	IP-10refs	1.84	8.05	12.5	4.73	3.37	19.11	6.93	69.16	33.85
Shrink	IS-Norefs	1.58	8	11.1	4.77	3.38	21.18	6.7	70.75	33.31
N=5K to	IS-10refs	1.57	8	10.95	4.75	3.37	20.94	6.63	70.73	33.24
N=1.25 K	SL-Norefs	1.55	7.94	11.02	4.78	3.39	16.25	5.14	47.82	15.22
	SL-10refs	1.56	7.92	10.86	4.75	3.38	15.24	4.62	39.07	12.56
	SW-NoRefs	1.5	7.7	11.64	4.75	3.37	14.97	5.27	39.02	17.6
Piggy-	IP-Churn	5.31	12.15	75.45	5.02	3.86	16.66	11.84	6.61	10.98
back	IS-Churn	2.24	9	15.85	5.19	3.98	8.44	5.71	7.68	11.12
Only	IP-Shrink	2.74	9.5	27.7	4.83	3.38	18.4	4.62	32.53	11.18
NoRefs	IS-Shrink	1.85	8	12.9	4.74	3.38	20.16	4.87	34.93	11.15

TABLE II

BUILD PARAMETERS: COMPARISON OF DEGREE DISTRIBUTION, DIAMETER, AND BUILD-LOADS OF THE DIFFERENT MECHANISMS. ALL GRAPHS EXCEPT SCAMP HAVE EXACTLY 5 OUTLINKS PER NODE, AND USE 10-HOP NEIGHBOR DISCOVERY WALKS. *Diam* AND *Dist* ARE THE AVERAGE ESTIMATED DIAMETER AND THE AVERAGE DISTANCE BETWEEN NODES, ESTIMATED USING A SAMPLE SET OF 20 NODES WHERE THE FARTHEST DISTANCE NODE FROM EACH NODE IN THE SAMPLE SET IS FOUND TO GET THE ESTIMATED DIAMETER, AND THE AVERAGE DISTANCE BETWEEN THE NODES IN THE SAMPLE SET IS USED AS THE ESTIMATED AVERAGE DISTANCE. *Dev(Deg)* IS THE STANDARD DEVIATION OF DEGREES, *Indeg-95pc* IS THE AVG. 95TH PERCENTILE VALUE, AND *MaxIndeg* IS THE AVG. MAXIMUM VALUE OF THE INDEGREE. (*)SCAMP'S 95TH PERCENTILE AND MAXIMUM DEGREE VALUES CORRESPOND TO THE TOTAL DEGREE, SINCE ITS OUTDEGREE IS NOT A CONSTANT.

the best indegree distribution[10].SwapLinks also has the best load distribution during node addition, mainly because its neighbor discovery walks use only inlinks and thus do not distinguish between nodes based on their degrees, since all nodes have the same outdegree. SelfLoops unfairly loads high-degree nodes because it does not bias among links during walk forwarding, while InlinkInvProb and Iterative Scaling end up loading low-indegree nodes unfairly heavily as a result of their random walk weightings. InlinkInvProb and Iterative Scaling end up with high message load overheads anyway when they use 1-hop updates. The diameter and distance estimates are more or less the same for all the four building strategies.

The load during node deletion is the only parameter here that is worse for SwapLinks than for some of the other strategies. The reason here is SwapLinks' higher aggregate load during node deletions: neighbor discovery walks are initiated for in-neighbors as well as out-neighbors. Nevertheless, the Dev(BLoad-Kill) parameter with SwapLinks is still quite close to the other strategies. And, considering that neither refreshes nor neighbor information is required, SwapLinks may after all be more efficient as well as simpler.

SCAMP here has the worst degree distribution, partially due to its larger average total degree of 15.7. We did not run churn or shrink on SCAMP since SCAMP does not explicitly discuss handling of unannounced departures.

[10]In SwapLinks, the entry of new nodes negates, to a certain extent, the bad effects of prior node deletions, since each new node entry can only improve the degree distribution.

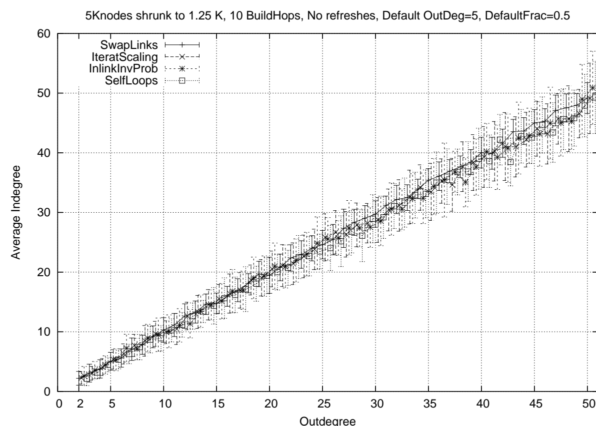
B. Graph construction under heterogeneity

In this section we study how well the different schemes adapt to heterogeneity. The setting we will be using here is one where a 5000 node graph is shrunk or churned. Each of the 5000 nodes has, with a probability of 0.5 the default degree of 5, and with a probability of 0.5 a uniformly picked degree from the range [2,50]. We present results of the shrink case without refreshes; all the other cases, namely, shrink with refreshes, churn with and without refreshes give similar results, which are not shown here. Graphs built using InlinkInvProb and Iterative Scaling make use of one hop updates.

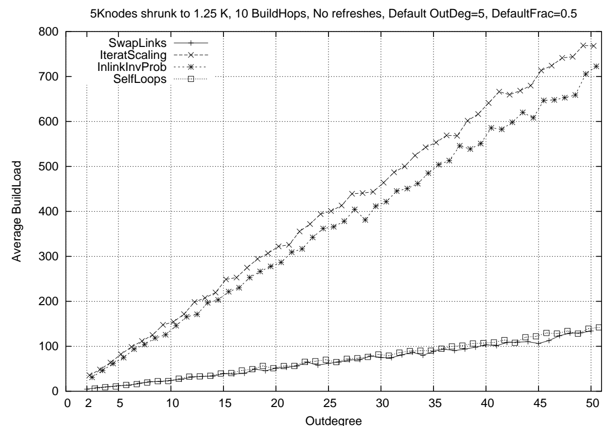
We show here the average indegree and the build load during addition as a function of the outdegree. For each outdegree, we get the set of nodes with that outdegree, and compute averages from that set to get the figure for the particular outdegree. We use the same model to measure build load during node addition as we did in section VI-A. The distribution we want to achieve is one where all relevant parameters are directly proportional to the outdegree.

Fig. 2 shows the variation of the indegree and the build load during addition of new nodes. All strategies result in a linear dependence of both the indegree and the load on the outdegree, demonstrating that the modifications made to the walk probabilities indeed work as intended. In separate experiments, we found that the load during node deletion (not shown here) also grows linearly with the outdegree.

In the figure, the IS and IP load curves are much higher than the other two because of the 1-hop update load: each node that gains an inlink during the test needs to let all of its



(a) Avg. Indegree vs. Outdegree



(b) Avg. BldLoad-Add vs. Outdegree

Fig. 2. Heterogeneity : Variation of Build Parameters with Outdegree

neighbors know, and with an expected total degree of 31 here, this results in a significant overhead. Note here that we could reduce the frequency of updates to achieve a smaller message overhead, but this comes at the cost of reduced accuracy of the maintained state. We do not evaluate this trade-off in this paper. If we altogether drop the use of 1-hop updates with IS or IP, we will have to use proactive methods like planned refreshes, or exchange of neighbor information, or both, to generate good graphs; these result in overheads of their own.

Nevertheless, all build strategies do exhibit good control over heterogeneity, but we prefer the SwapLinks strategy over the others. There are two main reasons. The first is that it performs well under all conditions. Second, and just as importantly, it seems the easiest to engineer: Swaplinks has just one parameter to set, namely the outdegree of each node[11]. With the other strategies, in addition to setting the outdegree, we need to worry about the frequency of exchanging neighbor information (with IP or IS), or about setting the virtual hop-length to achieve a target average hop-length(with SL), and the frequency of refreshing(IP, IS, SL). While none of these tasks is inherently difficult, it is nice to be able to avoid them since we can.

C. Quality of Random Selection on Homogeneous Graphs

Having picked SwapLinks as the most promising algorithm to build graphs (from sections VI-A and VI-B), we now evaluate the quality of random selection of the four selection schemes running over a SwapLinks graph. We use two parameters to measure the quality of selection: the distribution of the selected nodes, and the distribution of load imposed by the selection walks. The selection strategies TotalInvProb and Iterative Scaling make use of only piggybacked information sent over build walks, so these do not incur any extra overhead in terms of state maintenance. We do not employ piggybacking on the selection walks here because the number of selection walks we use in the experiments is comparatively large, so piggybacking on even the selection walks would lead to an undesirable artificial improvement in the measured quality of selection.

[11]Strictly speaking, all the strategies need to also set the walk length to some value optimal to the number of nodes. Practically speaking, however, this can be set by default to a conservative large value such as 10 hops–see VI-E.

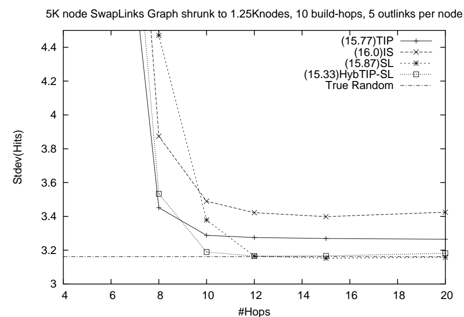


Fig. 3. Std.dev(hits) vs. #Hops for the SwapLinks graph. Numbers in parentheses indicate the 95th percentile value of hits at 10 hops (the average is 10 hits).

We refer to the node selected by a random walk as the node *hit* by the walk. To evaluate selection quality, we start a set of random walks from a *single* node, and log the number of hits each node receives : we use a single start point to avoid the artificial smoothing introduced by having multiple start nodes. The number of walks executed is equal to 10 times the current number of nodes in the graph. We use the standard deviation of hits as the metric to measure selection quality.

We show the results of the shrink scenario here; results for the SwapLinks churn graph are similar. The results shown here correspond to a 5000 node graph before the shrink is performed. All nodes here have the same outdegree of 5.

Fig. 3 shows the average standard deviation of hits as a function of the length of the walk. Once again, the main thing

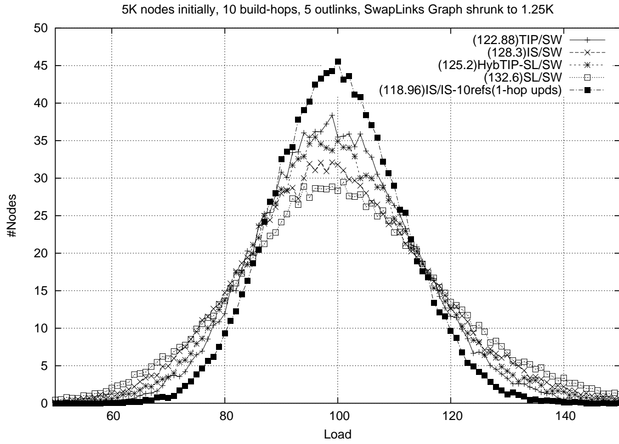


Fig. 4. Selection load distribution over the SwapLinks Graph at 10 hops. Numbers in parentheses indicate the 95th percentile value of the load.

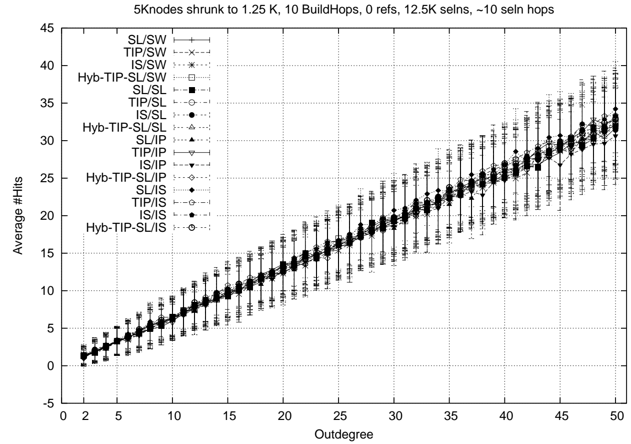


Fig. 5. Heterogeneity: Average Hits vs. Outdegree

to note is that all of these walks perform satisfactorily well. The number of hits at the 95th percentile is similar for all approaches, and not that far from the average of 10. Hyb-TIP-SL gives the best hit distribution on the SwapLinks graph, and this is very close to the best hit distribution using any mechanism on any other graph. TotalInvProb's selection also is good, though it stabilizes at a distribution slightly different from TrueRandom's distribution. Iterative Scaling's distribution is not as good as the others because only piggybacking on the build walks is insufficient to bring the weights to the required state of convergence. Because SelfLoops is a variable walk-length strategy, its performance when the number of hops is small is poor since quite a few of its walks would be very short and end very close to the start point. As a benchmark for selection quality, we use True Random selection, where each hit node is uniformly randomly picked from the entire population. True Random selection is just an instance of the Balls and Bins problem, resulting in a Possion distribution of selection hits; its standard deviation of hits is given by the square root of the mean number of hits each node receives.

We measure the selection load seen by a node as the number of selection walks that pass through or end at the node (Fig. 4). For selection load, we again execute a given number of walks (again the number of walks is 10 times the number of nodes in the graph), this time with the origins of the walks distributed across the graph such that every node in the graph is selected as the start node an equal number of times. The idea here is that the load distribution should be uniform when all nodes are involved in about the same number of walks - note that if some nodes start more of the walks than the others, there will be an unavoidable skew in the selection load seen by the nodes very close (within 3-4 hops) to the given start nodes. Fig 4 shows the bell curves of the selection load distribution when the walk-length is set at 10.

Note here that we have added one curve that is not based on the SwapLinks graphs: this is IS selection on an IS graph with neighbor information exchange and ten refreshes. We show this curve as a point of comparison because it is the best of all selection/build combinations. Among the remaining, TotalInvProb gives the best load distribution here, while Hyb-TIP-SL's selection load curve is slightly worse. Both of these curves themselves are reasonably close to the best (IS/IS) curve. Iterative Scaling as a selection mechanism on top of SwapLinks again suffers to some extent due to its imperfect piggybacked state. SelfLoops is the worst in terms of load-balance, as here the number of walks that pass through a node increases with its degree.

The decision of which algorithm to use to perform selection on the SwapLinks graph depends on the application. If each node performs selections relatively infrequently, then the algorithm to use would be either TotalInvProb or Hyb-TIP-SL, which are very close to each other here in terms of performance. If, on the other hand, selection walks are more frequent, then by using piggybacking on top of the selection walks, Iterative Scaling will be able to converge to the required state faster, so it would be the strategy to use. Generally, on any graph, if Iterative Scaling is given enough time to stabilize, it gives good selection in terms of both the hit distribution and load balance.

D. Selection with Heterogeneity

We now look at the quality of selection when nodes have different outdegrees. We use the same setting we used for evaluating graph building under heterogeneity (section VI-B), i.e, a 5000 node graph subjected to shrink, and the same expected outdegree distribution. We present the results of running 12,500 random selection walks using each of the four selection algorithms on top of all the four different graphs. We measure the distribution of selection hits as a function of the outdegree.

Fig 5 contains the results. The selection hits vary linearly with outdegree for all combinations of selection strategies and build methods. The selection load curve follows a similar pattern: linear, with smaller variance, which we do not show here for lack of space. So with regard to heterogeneity, we

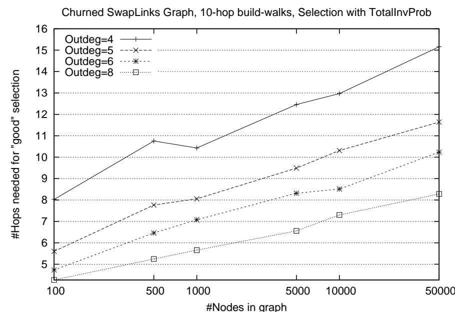


Fig. 6. Variation of the required selection walk-length for a range of network size and average degrees

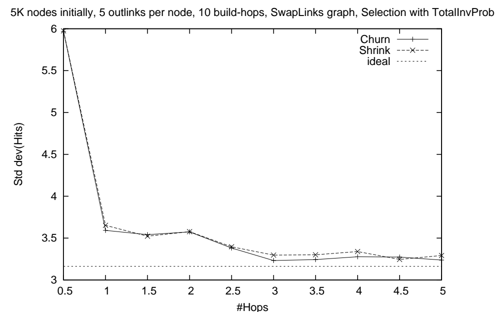


Fig. 7. Std Dev(Hits) vs. #Hops in each Cursor Walk

Deg	Dev(Deg)	95pc(Deg)	MaxDeg	Diam	Dist
4	1.22	6.0	11.0	7.0	5.65
5	1.32	7.0	14.0	6.15	4.93
6	1.39	8.0	15.0	6.0	4.63
8	1.52	11.0	16.0	5.05	4.13

TABLE III

GRAPH PARAMETERS FOR 50,000 NODE CHURNED GRAPHS.

are able to engineer all of the selection methods to function satisfactorily well.

E. Scaling to larger sizes

In this section we evaluate the scaling behavior of SwapLinks over a wide range of network sizes and average degrees: we vary the network size from 100 to 50,000, and the outdegree per node from 3 to 8, and measure the number of hops it takes to obtain a random selection distribution whose standard deviation is within 5% of that of true random distribution. The graphs are churned before the selections are performed. We use TotalInvProb as the selection mechanism here. All build walks are 10 hops in length. Fig. 6 shows the results.

With only 3 outlinks per node, TotalInvProb was not able to consistently reach the required quality of selection when the system size grew beyond 1000, so these results are not shown. When the outdegree is more than 3 though, TotalInvProb reaches the desired quality. The number of hops needed grows with the logarithm of the network size, and, as can be expected, decreases as the average degree increases. The rate of change of the number of required hops as the system size increases is very small. From a practical perspective, this would allow someone deploying a P2P application to select a conservative but reasonable value for number of hops given their largest expected user population.

To verify that our SwapLinks builds good graphs even at large scale, we show in Table III the indegree distribution and the estimated diameter and average distance for 50,000 node churned graphs for different values of outdegree per node. These results show that the graph building mechanism and the selection walk procedures both scale well.

F. The Cursor Approach

In this section, we evaluate the cursor walk described in Section V. Fig 7 shows the variation of the quality of hit distribution with increase in the expected walk-length[12]. The total number of cursor walks initiated here is equal to ten times the network size. The result shows that the approach is indeed viable, with about 3 hops on each small walk needed to approach the uniform distribution. When the walks are shorter than this length, the probability of revisiting already visited nodes increases, affecting the selection distribution. A trend that can be noticed is that even numbered hops are local maxima in the plot. We believe this is because with an even hop length, the probability of the walk backtracking and returning to the originating node increases.

VII. Conclusions and future work

Our next step is to implement the SwapLinks algorithm, and test it in a real setting (i.e. planetlab). We plan to compare this strategy with a random selection strategy that uses DHTs. In particular, we have focused on unstructured approaches because of the success of unstructured P2P applications, and because we ourselves are building unstructured P2P applications. Our intuition is that unstructured random selection will be easier to implement and will scale better. But this is only an intuition: it needs to be tested.

Another important piece of work that needs to be done is to consider misbehaving nodes. Although not reported, we ran experiments with the biased-walk approaches where misbehaving nodes would terminate every build walk at themselves. Even without creating any additional outlinks, these nodes were able to obtain inlinks with almost every every other node in the graph! We need to explore simple mechanisms for preventing this.

A third area we need to explore is that of establishing proximal neighbors (those with low latency or high bandwidth) in addition to random neighbors. While this could in theory be left to the application (once it has a random network to "explore"), it seems that providing this capability as part of the toolkit would be broadly useful.

[12]Here a fractional walk-length of say 1.5 hops corresponds to the set of cursor random walks where each walk is independently of length 1 or 2, with probability 0.5 each.

The broad conclusion that we draw from this work, however, is that our original goal—to find a simple and scalable algorithm for building random graphs and doing random selection, with good control over heterogeneity—is certainly satisfied! Specifically, our SwapLinks approach lets us construct graphs while requiring the setting of only a single parameter by each node, namely its desired node degree, and enables the desired random selection on top of the graphs thus built. We are honestly delighted with the results, and feel confident that we and others can base a number of interesting P2P applications on this foundation.

References

[1] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, "Bandwidth-efficient management of DHT routing tables," in *Proc. NSDI*, 2005.

[2] M. Castro, M. Costa, and A. Rowstron, "Debunking some myths about structured and unstructured overlays," in *Proc. NSDI*, 2005.

[3] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like p2p systems scalable," in *Proc. ACM SIGCOMM*, 2003.

[4] L. A. Adamic, R. M. Lukose, B. Huberman, and A. R. Puniyani, "Search in power-law networks," *Phys. Rev. E*, vol. 64, no. 046135, 2001.

[5] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *In ICS'02*, 2002.

[6] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, ser. LNCS, vol. 2009. Springer-Verlag, 2001, pp. 46–66.

[7] P. Francis, "Yoid: Extending the internet multicast architecture," in *Unrefereed report*, 2000.

[8] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *Proc. ACM SOSP*, 2003.

[9] "Bittorrent, http://www.bittorrent.com/."

[10] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *Proc. ACM SIGCOMM*, 2004.

[11] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Scamp: peer-to-peer lightweight membership service for large-scale group communication," in *Proc. 3rd Intnl. Wshop Networked Group Communication (NGC '01)*, 2001, pp. 44–55.

[12] D. Kostic, A. Rodriguez, A. B. Jeannie Albrecht, and A. Vahdat, "Using random subsets to build scalable network services," in *Proc. USITS*, 2003.

[13] R. Melamed and I. Keidar, "Araneola: A scalable reliable multicast system for dynamic environments," in *Proc. NCA 2004*, 2004.

[14] C. Law and K.-Y. Siu, "Distributed construction of random expander networks," in *Proceedings of the IEEE Infocom '03 Conference*, 2003.

[15] Z. Bar-Yossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz, "Approximating aggregate queries about web pages via random walks," in *Proc. VLDB*, 2000.

[16] I. Csiszár, "Information theoretic methods in probability and statistics," *IEEE Information Theory Society Newsletter*, vol. 48, pp. 21–30, 1998.

[17] A. J. Ganesh, A.-M. Kermarrec, and L. Massouli, "Peer-to-peer membership management for gossip-based protocols." in *IEEE Trans. Computers 52(2)*, 2003, pp. 139–149.