On Random Node Selection in P2P and Overlay Networks

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

Abstract

Distributed peer-to-peer 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 core component of both of these requirements is the random walk, which can be used to select random neighbors when building a graph, and can be used to select random nodes over the built graph. While there are numerous studies that look at specific aspects of random walks, the literature ultimately did not provide a comprehensive and satisfactory approach that would work with predictable results over a range of applications. Using simulations, this paper compares a number of techniques—some novel and some variations on known approaches—for building random graphs and doing random node selection over those 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 be met, and that while no approach is superior against all criteria, our novel approach broadly stands out as the best approach.

1 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 P2P applications that require random node selection. We decided that it would be preferable to build a single random node selection mechanism, and use it for all the projects. Indeed, we feel that such a mechanism, packaged and distributed as 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.

The work reported in this paper has two aims. One is to 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. The second aim is to compare various approaches using simulation so that applications that have requirements not satisfied by our techniques may nevertheless gain an understanding of the basic characteristics of the various approaches.

We have been successful in-so-far-as we have a design that is very simple, and, as far as we can tell from simulations, appears to operate well over a range of criteria. Now we can move on to implementing and testing our design with reasonable confidence that it will work, and that we have made a good choice.

1.1 Motivation, requirements, and some previous work

We develop our set of requirements by discussing the applications that drive the need for random graph building and node selection. Along the way, we discuss the techniques and shortcomings of approaches that have been tried in the context of specific P2P and overlay network applications. (In the subsequent section, we review approaches that have been studied outside the context of P2P applications.)

We refer to these 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.

We preface this discussion by first pointing out that all P2P applications require a graph that is robust against partition. If the membership of a graph is small enough, then this can be achieved by simply gossiping or broadcasting the membership of the entire graph to all other nodes [2]. In these cases, this is a fine approach—it is simple and robust—and we would use it ourselves were we not planning to design P2P applications for very large user populations. There are applications, however, that have tens of thousands of nodes connecting over broadband or thinner access links for which the full-membership approach won't do. SCAMP, for instance, is a gossip approach designed to prevent nodes from requiring full membership knowledge [7]. Indeed the approach of graph building used by SCAMP is one of the options we consider in this paper.

Fortunately it is well-known that many types of random graphs are robust against partition ([13] and references therein). 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. The reason different nodes would have a different number of outlinks is to accommodate heterogeneity in node capacity. As a result, K_i simultaneous failures are needed to partition node i from the network, and many more failures are required 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, from a practical perspective, our random graphs are robust to partition, and we don't 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 don't 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 a 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 to its node degree. An example of such an application is file search unstructured file sharing networks like Gnutella or Kazaa. For instance, [3] shows that random walked searches can scale better than broadcast searches where node degrees are power-law distributed, and [9, 11] propose that node degree be related to node capacity in order to achieve this without unduly stressing 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 the file search algorithms mentioned above ([3, 9, 11]), each node visited is searched for the desired key words. Freenet, an anonymizing file distribution and discovery network, also uses random walks in this manner [14]. Note that much of this effect can be obtained by establishing the appropriate node degree in the graph.

GIA [11] in particular 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 search load than low capacity nodes, both by giving high capacity nodes higher degree, and by routing more queries to them. GIA doesn't give direct control over degree or load, and indeed [11] doesn't 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 can't be used as a general purpose graph building or node selection mechanism. How valuable our mechanisms are to GIA, on the other hand, is a question that we don't directly answer. Our approaches are simple, however, and provide an amount of control that GIA doesn't 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.

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 [1, 4, 5], the file sharing applications mentioned above [9, 11], and "proximity addressing" applications [6], The latter is an application where nodes form addresses (for instance, a cartesian coordinate) that can be used to indicate

how close nodes are to each other in the network. In approaches such as Vivaldi [6], which don't use special positioning nodes (beacons or landmarks), each node selects a certain number of random other nodes, and measures its network distance to those nodes. This selection should follow a uniform probability distribution.

Random selection in overlay multicast is slightly different. Here nodes randomly discover other nodes in order to form links in the multicast overlay itself—that is, links over which the multicast stream is transmitted. Note that these multicast links are not necessarily the same as the links in the random graphs discussed up to now. The multicast links should exhibit locality—nodes near each other in the physical network should form multicast links so as to minimize load on the network and to make the multicast itself efficient. This means that the multicast graph is not random, and therefore won't have the same robustness to partition that a random graph has. For this reason, Yoid [1] maintained two graphs, one for multicast, and the other a random graph for keeping the multicast group from partition. Yoid used a naive approach for its random walks in that it doesn't bias against high-degree nodes. This approach is known to produce highly skewed node degree distributions (power-law), and corresponding skewed random selection. Our own simulations bear this out.

Bullet multicast [4] uses a random selection mechanism called RanSub [19]. RanSub operates in waves of network-wide coordinated phases, where in each phase lists of random nodes are distributed through the network. It operates in such a way that multiple nodes learn about the same set of other nodes in a given phase. 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 don't need to select other nodes (because they already have enough multicast links), 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 independent of other nodes in the network, is preferable.

BitTorrent is a P2P file distribution protocol whereby nodes feed each other blocks of a file [5]. BitTorrent uses a central node called the seeder that keeps track of the participants, and tells joining nodes about a random subset of existing participants. While this approach works reasonably well, in environments where even a seeder can't adequately scale, random selection as described in this paper may be used. Note, however, that in any event a boot-strapping procedure, whereby a joining node learns of at least one existing member, is required (for all P2P applications).

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.

1.2 Outline

Having now motivated the problem and presented some approaches inadequate for our needs, Section 2 describes the four basic approaches for both building graphs and walking them, describes the contexts in which the approaches were used, and why they too are inadequate for our needs. Section 3 presents results of simulations used to evaluate the performance of the four approaches. Section 4 concludes and outlines next steps.

2 Algorithms, and some previous work

In this section, we describe the basic approaches we use in building graphs and taking random walks. Along the way, we describe previous work that inspired some of the approaches we take, and why that work taken as is is not sufficient for our needs.

2.1 Initial node discovery

Our description of the join algorithm in the previous section assumed that a joining node knows of at least one existing node in the graph. The basic approach 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.

Therefore, we assume an approach whereby the rendezvous node remembers a small set of recently joined nodes, as well as a small set of random nodes known to be up with high probability (which the rendezvous node can discover with selection walks). Joining nodes will try initially to contact the recently joined nodes, and if this fails they can try

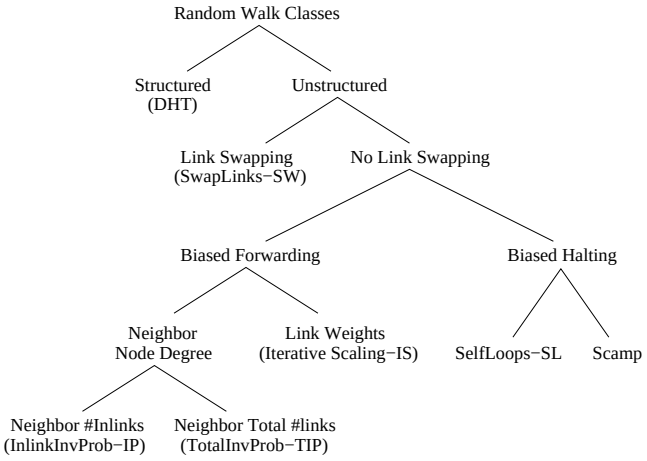


Figure 1: Classification of Random Walks.

the other nodes. This approach effectively spreads the load of node discovery. In the remainder of this paper, we assume this style of node discovery.

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.

2.2 Structured graphs

Figure 1 gives a classification of the various styles of graph building and random selection for P2P networks. The top level splits between structured and unstructured graphs. While in this paper we only analyze unstructured (random) graphs, we must address the fact that one way to do random selection is to build a structured graph, such as a DHT (Distributed Hash Table) network like Pastry and others [16, 17, 18]. Assuming that the node IDs of DHT members are randomly distributed over the DHT's address space, random node selection can be done by simply selecting a random address from the space and routing to the selected address. To accommodate heterogeneity, high-capacity[1] nodes can replicate themselves in the DHT multiple times. Since DHTs are designed for P2P applications, why then does this not satisfy our goals?

In P2P applications where a DHT is not otherwise needed, the use of DHTs for random node selection alone is not nearly as simple as it could be. It is much simpler to build and maintain a network using weighted random walks as outlined above than to build a DHT network. Likewise, it is simpler to do a random walk than to route to a specific DHT address. We believe that this simplicity is a very important factor in being able to realistically deploy very large P2P applications. For this reason, we believe that, in applications where DHT functionality is not otherwise needed (which is the case for multicast, coordinate proximity addressing, and unstructured file sharing applications), with the following caveat, the random network approach is far preferable. We don't further consider the use of structured networks.

The caveat, however, is that this simplicity argument may fall apart in the face of untrusted nodes that wish to establish artificially high node degrees. An adversary node might want a high node degree as a denial-of-service attack, or to defeat anonymity. Without additional invariants, such an adversary node could establish as many outlinks as it wanted. The invariants required to prevent this may (or may not) make the random network approach as or more complex than a DHT. This paper assumes an environment of trusted and correctly operating nodes, and, other than a brief discussion at the end, leaves the important problem of adversaries to future work.

2.3 Unstructured graphs

As mentioned in the previous section, 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. The reason is because any favoring of high-degree nodes in the build walk selection process 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"—there are more opportunities for subsequent joining nodes to select them than there are for later joining nodes. We found in our early simulations that, when walks were biased to select uniformly, early joining nodes had more inlinks than later joining nodes. Therefore, there must be additional bias or mechanisms to select against high degree nodes.

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 node degree. Our basic approach

[1]By high-capacity, we mean nodes that should receive more load. Of course it may not be desirable, for instance out of fairness considerations, for high-capacity nodes to actually receive higher load than low-capacity nodes

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[2]. These two approaches are reflected in the next layer of the taxonomy of Figure 1 (Link Swapping and No Link Swapping). One approach is to simply endow build walks with an even stronger bias against high-indegree nodes, so that nodes never get high indegrees. As can be seen in the taxonomy, there are several ways to do this, which are described in the next section. The other approach is to activity manage each node's indegrees, so that nodes explicitly shed inlinks when they get too many. The basic mechanism is for nodes with high indegrees to move an inlink to nodes with low indegrees. We discuss this swaplinks approach in a subsequent section.

2.3.1 Biased walk approaches

There are several basic methods to bias a random walk. These methods apply whether the walks are build walks or selection walks—the difference is essentially in the amount of bias.

In 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 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 they could do this, for instance by not terminating a build walk at themselves if their 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.

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 doesn't not succeed within some short time.

Looking again 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-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—around two or three times the diameter of the network[3].

In biased-halting, the next hop is picked uniformly at random among the inlinks, among the outlinks, or among both inlinks and outlinks —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.

There are a couple of basic trade-offs between biased-halting and biased-forwarding. 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.

Biased-halting: Biased-halting approaches are ideal in settings where the graph is not under ones' control, and the cost of calculating weightings is high. Indeed, the biased-halting approach we used is based on work by Bar-Yossef et al [15], who used it to select web pages with uniform probability. Bar-Yossef's 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 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 our experiments, we ensure that the average number of real walks taken by SelfLoops is as needed.

To accommodate heterogeneity and the different biases for build and select walks, we modified the Bar-Yossef approach as follows. 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 ($\frac{od^2}{id}$). 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 this is, assume a well refreshed graph that has reached stable degree distribution. [4] Now examine the change in indegree of node i when another node r performs

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

[3]In practice, given that the diameter grows slowly with the size of the graph, we can simply pick a conservative value like $H = 10$.

[4]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 the subsequent text.

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 $c \cdot d_i \cdot K_r$ where c is a constant and d_i the indegree of i.[5] So the probability that i loses an inlink because of the refresh is given by $c \cdot d_i$. The probability that i gains an inlink because of the refresh is given by $c' \frac{k_i^2}{d_i}$, where c' is a constant. Thus we get $d_i = c'' \cdot K_i$, 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.

Biased-halting was also used with SCAMP [7] as a way 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 (specifically $(c+1)log(N)$, where c is a selectable constant). Nodes in SCAMP have indegrees and outdegrees that vary with time. To join the network, a new node u contacts a randomly selected "introducer" node already in the network. The introducer sends out messages to its out-neighbors as a result, each message resulting in a new inlink gained by the new node. The number of messages sent by the introducer is equal to its outdegree plus the constant c. The new node forms a sole outlink to the introducer. Subsequent node additions result in forwarding of messages to u through links pointing to u, and each received message increases u's outdegree with some probability. As a result, the average outdegree grows as the desired log function of N. While the ability to tie node degree to graph size is a sweet property for some applications, we wanted more control over node degree than SCAMP allowed and so didn't find it useful (though we did simulate it as a point of comparison).

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 don't 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).

Biased-forwarding: We experimented with two types of biased-forwarding walks. In one, the bias 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 flatter is used for selection walks.

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

Note that one could invent any number of inverse weightings derived from neighbor node degree (square of the degree, square root of the degree etc.). Though we fooled around with these a bit, we found the above approaches to be good, and therefore don't report any of these variations in this paper.

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 [21]. SCAMP applied this iterative scaling technique to random walks as a means of randomly selecting the "introducer" node described above [8]. Here 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 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 sum of incoming weights at each node is proportional to its outdegree. When used for building, the ideal sum is directly proportional to the square of the outdegree and inversely proportional to its indegree.

When weight updates are now 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.

Exchanging neighbor information: Given that these 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 aliveness. These few neighbors would then tell the remaining neighbors if the node went down. In this case, the node obviously can't 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. 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 random selection in a graph.

Refreshing has a number of negative aspects. 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 is it 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.

OnlyInLinks: There is one type of random walk which is essentially a biased-forwarding walk, but in fact requires no knowledge about the neighbor. In this fixed-length walk, each node chooses uniformly randomly among its inlinks only. If a node has no inlinks, then the walk terminates at that node (thus allowing it to obtain an inlink, in the case of build walks). The idea here is that, since walking an inlink selects a node in proportion to its outdegree, the result is uniform selection proportional to each node's outdegree.

OnlyInLinks itself though cannot be used to build graphs : here newly joining nodes are strongly steered towards very recently joined nodes. This is because the rendezvous server would return a list of recently joined nodes, and since all links point from new nodes to older nodes without refreshes, walking only the inlinks would drag the walk towards the most recent nodes. The end result would be a very "long and skinny" network—one with a large diameter, and therefore poor for random selection.

We mention this approach only because it is useful in SwapLinks style of graph building, discussed next.

2.3.2 SwapLinks

SwapLinks is inspired by, but quite different from, the approach used to build random graphs by Law and Siu [12]. The basic idea in [12] 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 in the case of multiple simultaneous failures. The basic approach is that each node would know some or all of its neighbors neighbors. If one of its neighbors leaves (which it may do abruptly without notice), then it must establish a link with one of the neighbor's neighbors. However, these nodes may also fail at roughly the same time, and so on. the method for dealing with all of this looks a lot like the successor nodes of Chord or the leaf sets of Pastry. Not surprisingly, we make the same argument as with DHTs—that the pure Law-Siu approach is significantly more complex than our other unstructured approaches.

However, if we forget about trying to maintain the perfect indegree/outdegree invariant, then things can get a lot simpler. Our 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. Note that 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[6] 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]. As we

[6] A biased-forwarding random walk that selects the next hop randomly among a nodes outlinks, thus biasing towards high indegree nodes.

[7] Note that a walk is initiated here only if some node departure led to a

will show later, among the techniques we studied, SwapLinks gives the most suitable graph for random selection.

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 with the biased-forwarding approaches), and no uncertainty about how long walks may take (as with the biased-halting approaches). Perhaps the primary 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.

2.4 Selection Walks

The previous sections 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.

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 1 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. Over a long walk, however, the selection does tend towards uniform distribution.

3 Experimental Results

We start by describing the simulations used to evaluate the various approaches. We used 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:* A 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[8]. Note that the expected network size after this sequence of events is N. In all our measurements, unless otherwise mentioned, we set N to 5000.

We do not bother to report on graph building scenarios where only nodes are added because SwapLinks behaves perfectly in this case, and it therefore isn't a fair comparison. For the other schemes, the grow-only scenarios behave similarly to the shrink and churn scenarios.

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[9].

link loss; In the above instance, I will not launch any walks as a result of its losing its inlink to B.

[8]Note that with 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.

[9]By contrast, GIA simulated heterogeneity spanning three orders of mag-

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 (see Section 3.5).

Given that we have four graph-building techniques, four selection walks, heterogeniety, 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 3.1. We evaluate the performance of all the graph construction algorithms in conditions of heterogeneity in section 3.2. 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 3.3, and then test all the selection walks over heterogeneous graphs (in this case over all the build methods) in section 3.4. We next look at the scaling behavior of the SwapLinks algorithm in section 3.5. Finally, we evaluate the cursor mechanism in section 3.6.

3.1 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 measured both with and without refreshes (except for SwapLinks, which doesn't benefit from refreshes). The refreshes are performed after the churn or shrink as described above has completed. Finally, for IS and IP graphs, we measured both where all neighbors are informed of any link change, and where neighbor information is only piggybacked on build walk messages.

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, in form of the number of walks each node has to forward, is logged. This is repeated a total of 100 times with the load summed over the 100 cycles, and finally the standard deviation of the load values of all the nodes is computed. We refer to this as *Dev(BLoad-Add)*. 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. To evaluate the load caused by node departures, we select 1000 nodes randomly and delete them (one by one) from the graph, and log the resulting load placed on the remaining nodes. We then compute the standard deviation *Dev(BLoad-Kill)* of the total load caused by the deletions.

Table 1 shows the results for the homogeneous-capacity graph building simulations. The key thing to note about these results is that they are almost all good! For instance, the standard deviation in node degree is for TrueRandom is 2.23. The only graph that did significantly worse than that was Inverse Probability where neighbor information was only piggy-backed. Most did better than TrueRandom. Likewise, the indegrees (95th percentile and max) are smaller than for TrueRandom. Given this, we believe there are no bad choices here. Some, however, are slightly better than others.

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 all nodes will have refreshed its out-neighbor set multiple number of times with high probability, so a shrunk graph is effectively equivalent to a refreshed graph.

SwapLinks' policy of neighbor replacement ensures it has the best indegree distribution. 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. InlinkInvProb and Iterative Scaling end up loading low-indegree nodes unfairly heavily as a result of their random walk weightings, while SelfLoops unfairly loads high-indegree nodes because here each node chooses all of its links (in and outlinks) with the same probability, leading to high degree nodes attracting more walks. The diameter and distance estimates are more or less the same for all the four building strategies.

The build load distribution during kill is the only parameter here that is worse for SwapLinks than for 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. Of course an important metric to consider here is the aggregate build load, which is slightly higher for SwapLinks when there are nodes leaving the system. On the other hand, neither refreshes nor neighbor information is required, which means that 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 15.5. We did not run churn or shrink on Scamp since Scamp does not discuss explicit handling of unannounced departures.

nitude. While indeed node capacities vary by this much in measured gnutella networks, it seems absurd to us that a node with 1000 times the capacity of a dialup would be willing to devote all of that capacity to file sharing!

		Dev(Deg)	Indeg-95pc	MaxIndeg	Diam	Dist	Dev(BLoad-Add)	Dev(BLoad-Kill)
Churn N=5K	IP-Norefs	2.22	9	15.1	5.23	3.96	7.78	4.54
	IP-10refs	1.82	8	13.3	5.02	3.95	4.81	4.22
	IS-Norefs	2.01	8.1	13.3	5.18	3.96	9.55	5.23
	IS-10refs	1.56	8	11.5	5.01	3.98	5.5	4.64
	SL-Norefs	2.03	8	13	5.26	3.96	5.39	5.34
	SL-10refs	1.55	8	11.5	5.03	3.97	4.67	4.62
	SW-NoRefs	1.31	7	12.1	5.01	3.96	4.09	5.19
N=5K	TrueRandom	2.23	9	15.03	5.06	3.97	-	-
	Scamp	7.12	28.15*	45*	6.09	3.49	2.34	-
Shrink N=5K to N=1.25 K	IP-NoRefs	1.85	8	12.5	4.73	3.42	18.17	4.26
	IP-10refs	1.81	8	12	4.71	3.39	16.89	4.19
	IS-NoRefs	1.59	8	11.1	4.8	3.39	20.72	4.81
	IS-10refs	1.57	8	11.1	4.71	3.36	20.35	4.71
	SL-NoRefs	1.55	7.9	11.3	4.77	3.38	16.28	5.14
	SL-10refs	1.57	8	11	4.73	3.38	15.25	4.59
	SW-Norefs	1.49	7.5	11.1	4.78	3.42	15.13	5.21
Piggy-backed Info	IP-Churn	5.31	12.3	79.8	5.02	3.86	18.04	12.32
	IS-Churn	2.26	9	15.9	5.20	3.96	8.60	5.71
	IP-Shrink	2.68	9.2	26.7	4.85	3.39	18.58	4.63
	IS-Shrink	1.89	8	13.5	4.75	3.37	20.86	4.90

Table 1: 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.

3.2 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. As before graphs built using InlinkInvProb and Iterative Scaling make use of one hop updates.

We compute 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. The distribution we want to achieve is one where all relevant parameters are directly proportional to the outdegree.

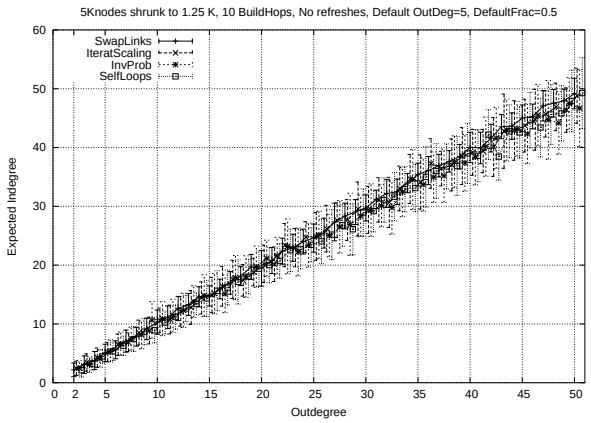
Fig. 2 shows the variation of the indegree, and the build load during addition of new nodes. We use exactly the same model to measure build load during addition as we did in section 3.1 All strategies result in the desired behavior where both indegree and load grow linearly with the outdegree. This demonstrates that the modifications made to the walk probabilities indeed work as intended. Thus any of the four build strategies studied is suitable to be used under heterogeneity.

The build load curves with InlinkInvProb and Iterative Scaling are slightly lower than the other two for the following reason: Both InlinkInvProb and Iterative Scaling are "low-indegree seeking" walks, so they tend to load those nodes that have just come in, and are yet to gain inlinks. This means that during the course of build load tests for addition of new nodes, a new node A that just entered would spend a non-negligible portion of its walks among the few nodes that entered just before A did, i.e, nodes that were added as part of test. Some of these walks end prematurely on reaching nodes with zero indegree. So the aggregate load placed on the already existing nodes by the new nodes is smaller in case of Iterative Scaling and InlinkInvProb.

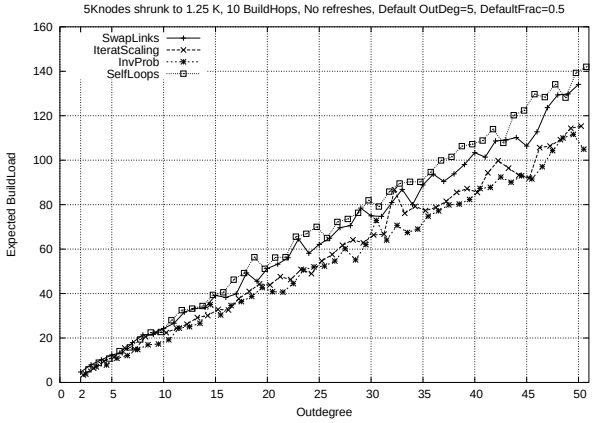
The main point, however, is that once again all build strategies exhibit good control over heterogeneity. Nevertheless, we prefer the SwapLinks strategy over the others. There are two main reasons. The first is that it performs well under all conditions. Second, and just as importantly, it seems the easiest to engineer. We don't have to worry about the process of exchanging neighbor information or about refreshing the graph. While neither of these tasks are difficult, it is nice to be able to avoid them since we can.

3.3 Quality of Random Selection on Homogeneous Graphs

Having picked SwapLinks as the most promising algorithm to build graphs (from section 3.1), we now evaluate the quality of random selection of the four selection schemes running over a SwapLinks built 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. Note here that the selection strategies TotalInvProb and Iterative Scaling make use of only piggybacked informa-



(a) Avg. Indegree vs. Outdegree



(b) Avg. BldLoad-Add vs. Outdegree

Figure 2: Heterogeneity : Variation of Build Parameters with Outdegree

tion 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 in such a scenario would lead to an undesirable artificial improvement in the measured quality of selection.

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 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

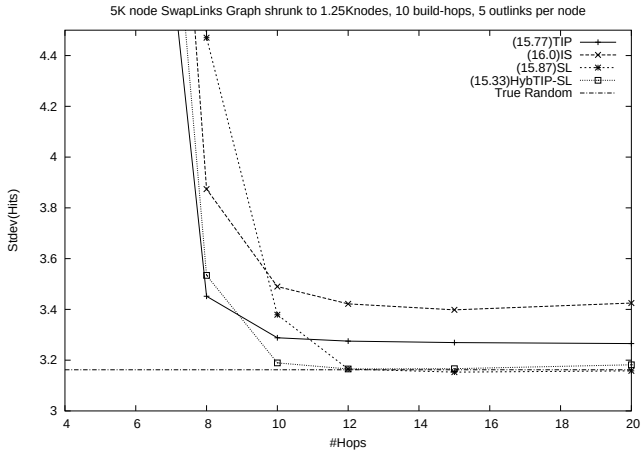


Figure 3: Std.dev(hits) vs. #Hops for the SwapLinks graph. Numbers in parentheses indicate the 95th percentile value of hits at 10 hops (the average is 10 hits).

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. We can view this as an instance of the Balls and Bins problem, where the number of bins is given by the number of nodes in the graph, and the number of balls is given by the number of selection walks. This reduces to the Poisson distribution and its standard deviation 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 the node forwards (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

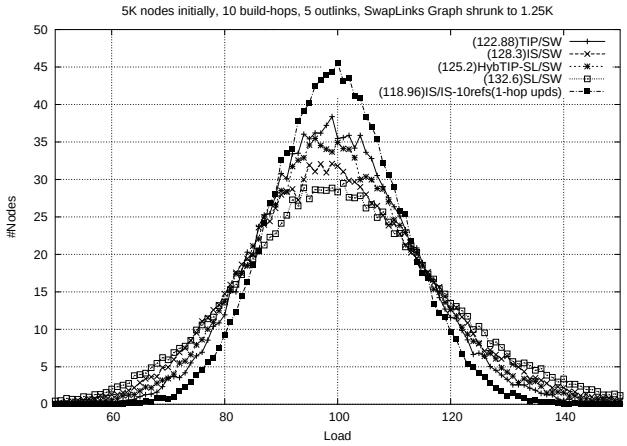


Figure 4: Selection load distribution over the SwapLinks Graph. Numbers in parentheses indicate the 95th percentile value of the load.

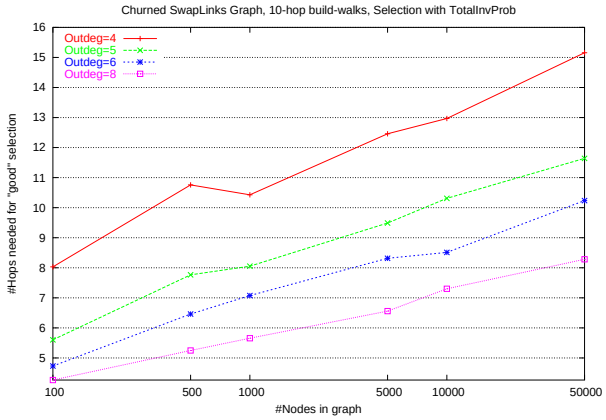


Figure 6: Variation of the required selection walk-length for a range of network size and average degrees

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 2: Graph parameters for 50,000 node churned graphs.

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 selectin 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.

3.4 Selection with Heterogeneity

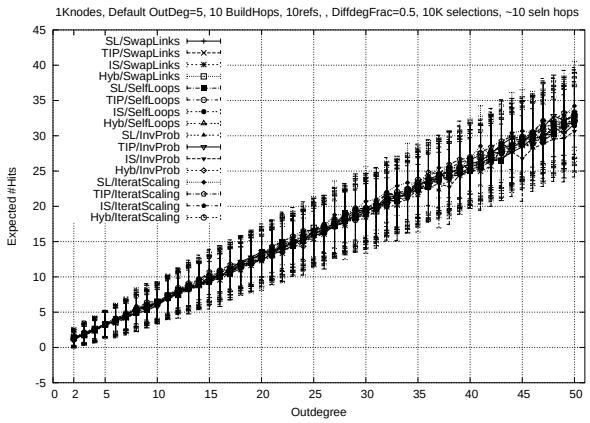
We now look at the quality of selection when nodes have different outdegrees. We use the same setting we used for evaluating graph building under heterogeneity (section 3.2), i.e, a 5000 node graph subjected to shrink, and the same expected outdegree distribution. We present the results of running random selection walks using all the four selection algorithms on top of all the four different graphs. We will measure the distribution of selection hits and selection load as a function of the outdegree.

Fig 5 contains the results. Both selection hits and load vary linearly with outdegree for all combinations of selection strategies and build methods. So with regard to heterogeneity, all of the methods we study perform satisfactorily well.

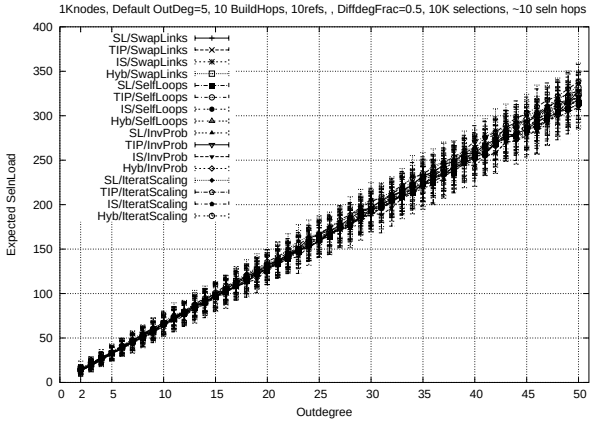
3.5 Scaling to larger sizes

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 by randomly killing a node or adding a new node a total of $2N$ times, where N is the network size. 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



(a) Average Hits vs. Outdegree



(b) Avg Selection Load vs. Outdegree

Figure 5: Heterogeneity : Variation of Selection Parameters with Outdegree

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, Total-InvProb 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 still builds good graphs even at large scale, Table 2 shows 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.

3.6 The Cursor Approach

In this section, we evaluate the cursor walk described in Section 2.4.

Fig 7 shows the variation of the quality of hit distribution with increase in the expected walk-length[10]. 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

[10]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.

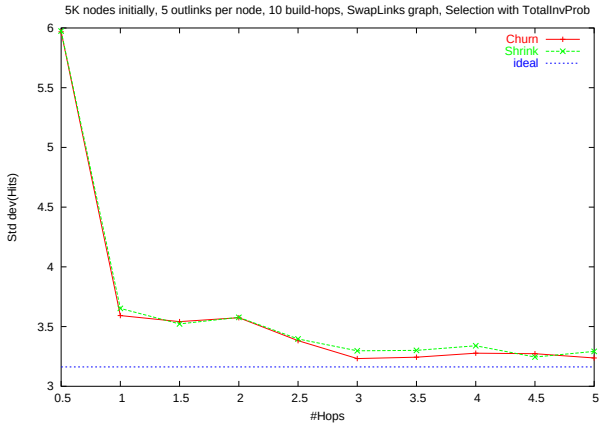


Figure 7: Std Dev(Hits) vs. #Hops in each Cursor Walk

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.

4 Conclusions and future work

The broad conclusion that we draw from this work is that our original goal—to find a dead-simple and scalable algorithm for building random graphs and doing random selection, with good control over heterogeneity—is certainly satisfied! 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.

The next step is to design and build the algorithm, and test it in a real setting (i.e. planetlab). We expect to use SwapLinks to build the graphs, and probably total inverse

probability to do selection over the graph.

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.

In addition, there may still be improvements we can make on the basic techniques explored in this paper. One might be to keep a history of visited nodes during a walk, to avoid revisiting the same nodes. This might be done using a bloom filter. One reason to do this is for file sharing applications, where a single long cursor walk is used for a given search. Revisiting a node is completely redundant in this case.

An optimization might be to continuously calculate the number of nodes N in the graph [22], and then use this information to optimize the length of walks.

Another small improvement might be to allow semi-broadcast walks. For instance, a walk may contain a parameter that it is to be replicated X times. Each node that replicates the walk would decrement the parameter accordingly, so that the walk would soon become X parallel walks. Such a walk would reduce load on nearby nodes somewhat (compared to X separate walks).

5 Acknowledgments

The authors would like to thank Robbert VanRenesse for his participation and help early in this project. We would also like to thank Emin Gun Sirer and members of the Copano group (Cornell P2P and Network Overlay Group) for their valuable comments and suggestions.

References

[1] Paul Francis, *Yoid: Extending the Internet Multicast Architecture*, Unrefereed report, April, 2000

[2] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang, *A Case for End System Multicast* In ACM Sigmetrics, Santa Clara, CA, USA, 2000.

[3] Lada A. Adamic, Rajan M. Lukose, Bernardo Huberman, and Amit R. Puniyani *Search in Power-Law Networks*, Phys. Rev. E, 64 46135 (2001)

[4] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat, *Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh*, In Proc. ACM SOSP 2003

[5] http://bitconjurer.org/BitTorrent/

[6] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris *Practical, Distributed Network Coordinates* HotNets 2003

[7] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Laurent Massoulie, *SCAMP: peer-to-peer lightweight membership service for large-scale group communication*, In Proc. 3rd Intnl. Wshop Networked Group Communication (NGC '01), pages 44–55. LNCS 2233, Springer, 2001

[8] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Laurent Massouli: *Peer-to-Peer Membership Management for Gossip-Based Protocols.* IEEE Trans. Computers 52(2): 139-149 (2003)

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

[10] Christos Gkantsidis, Milena Mihail, and Amin Saberi, *Random Walks in Peer-to-Peer Networks*, to appear in IEEE Infocom 2004

[11] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker, *Making Gnutella-like P2P Systems Scalable*, In Proc. ACM SIGCOMM 2003, Karlsruhe, Germany, Aug 2003.

[12] C. Law and K.-Y. Siu, *Distributed construction of random expander networks*, In Proc. IEEE Infocom 2003

[13] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal, *Building low-diameter p2p networks*, In STOC 2001, Crete, Greece, 2001

[14] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, *Freenet: A distributed anonymous information storage and retrieval system*, In Proc. International Workshop on Design Issues in Anonymity and Unobservability, volume 2009 of LNCS, pages 46–66. Springer-Verlag, 2001

[15] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz, *Approximating Aggregate Queries about Web Pages via Random Walks*, In Proc. VLDB 2000.

[16] A. Rowstron and P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany, November 2001.

[17] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A Scalable Content-Addressable Network*, In Proc. ACM SIGCOMM 2001

[18] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, In Proc. ACM SIGCOMM 2001, San Diego, CA, Aug. 2001.

[19] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat, *Using Random Subsets to Build Scalable Network Services*, In Proc. USITS 2003.

[20] S. Saroiu, P. K. Gummadi, and S. D. Gribble. *A measurement study of peer-to-peer file sharing systems*, In Proc. MMCN 2002, San Jose, January 2002.

[21] I Csiszár, *Information theoretic methods in probability and statistics*, IEEE Information Theory Society Newsletter 48 (1998), 21-30.

[22] David Kempe, Alin Dobra, and Johannes Gehrke. *Computing Aggregate Information using Gossip.* In Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science. Cambridge, MA, October 2003