Apocrypha: Making P2P Overlays Network-aware

Prasanna Ganesan Qixiang Sun Hector Garcia-Molina
Stanford University, Stanford, CA 94305, USA
{prasannag, qsun, hector}@cs.stanford.edu

***Abstract*—Many distributed systems built on peer-to-peer principles organize nodes in an overlay network, in order to enable communication between nodes. In general, this overlay network may have nothing to do with the location of nodes on the physical network. We propose a generic mechanism called Apocrypha to make any P2P overlay "network-aware", and thus optimize internode communication. We show how Apocrypha adaptively trades off the quality of optimization against the cost of optimization under dynamic conditions. We demonstrate the applicability and utility of Apocrypha on two different P2P systems: the Chord system which uses a deterministic overlay topology, and the Gnutella system which operates on an ad hoc topology. We also describe new, improved routing protocols for the Chord system, and introduce the long-circuit phenomenon in Gnutella.**

I. Introduction

Many distributed systems, ranging from file-sharing and distributed file systems to cooperative caching and multicast systems, are being developed using peer-to-peer (P2P) principles [gnu], [kaz], [lim], [RKCD01], [DKK+01]. In a P2P distributed system, components are autonomous, and there is no central control. Component failures are viewed as the norm rather than as an exception. Consequently, systems designed for a P2P environment provide a high degree of robustness and "self-healing".

A price that P2P systems pay for this robustness and autonomy is a loss in efficiency as compared to a tightly coordinated distributed system. A lot of recent effort has therefore gone into improving the efficiency of P2P systems without sacrificing robustness. One important area for optimization in P2P systems is *communication efficiency*. This paper offers a solution for improving the communication efficiency in any P2P system, by adapting the communication structure to the underlying physical network.

To explain the problem, we first discuss communication in P2P systems. All P2P systems organize participant *nodes* in an *overlay network*, where each node maintains a transport-layer link with only a small set of other nodes. Communication between nodes then takes place on top of this overlay network. The use of an overlay network is dictated by robustness considerations; it is considered too expensive for each node to know, and directly communicate with, all other nodes in the system. Different P2P systems use different mechanisms to construct and utilize such overlay networks. For example, Gnutella [gnu] uses flooding on this overlay network to propagate messages, while Chord [SMK+01] uses the overlay network to route messages from one node to another.

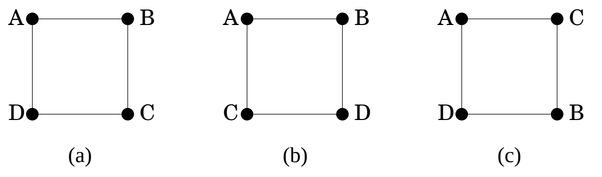

Fig. 1. All valid overlay networks for the 4-node hypercube topology

An overlay network thus generated may, however, have little to do with the location of nodes on the physical network. For example, two nodes on the same LAN may not necessarily be anywhere near each other on the overlay network and, conversely, nodes that are far apart on the physical network may be neighbors on the overlay. It has been observed that P2P systems would benefit if the overlay network were somehow constructed to "resemble" the physical network, i.e., by making neighbors on the overlay be nearby on the physical network. Such an overlay network is expected to improve on a network-ignorant construction, both by reducing the latency on the overlay links, and by reducing bandwidth usage in the internet core (which translates into money for ISPs).

There are clearly a vast number of overlay networks that may be constructed on a given set of nodes, but each P2P system imposes its own rules to determine which of these overlay networks are *valid*; we call this set of valid overlay networks the *overlay topology*. To illustrate with a simple example, imagine a P2P system on a static set of four nodes that requires a "hypercube topology", which is simply all overlay networks shaped like a square, as shown in Figure 1. Let us say that nodes A and B are very close to each other on the physical network, while C and D are far away from both A and B, as well as from each other. In such a case, overlay networks (a) or (b) would be more desirable than (c), since both (a) and (b) have a fast overlay link between A and B.

This paper studies a generic mechanism called **Apocrypha**[1] for identifying and constructing the "best" valid overlay network for any P2P system. Since P2P networks may be extremely dynamic, with nodes joining and leaving all the time, a critical feature of Apocrypha is its ability to effectively *maintain* such a "network-aware" overlay network as the system evolves dynamically.

Returning to our simple four-node example, Apocrypha operates by exploiting the fact that the P2P system must construct one of the three overlay networks in Figure 1. Say the P2P system initially constructs overlay network (c). Apocrypha can

[1]A P2P Overlay ConstRuction to Yield Proximal neigHbors (Allegedly)

then transform (c) into (a) by a simple operation: swapping the "letters" B and C in the picture of overlay network (c), which corresponds to node B breaking its overlay link with node D and connecting to node A instead, and node C performing the reverse operation. In general, Apocrypha may have to perform a number of these "swaps" to end up with a good overlay network, and our algorithm is based on this simple idea.

Of course, such an algorithm would require that we somehow *know* that a particular swap is "desirable". In our example, we knew that it was desirable to form an overlay link between A and B and therefore performed a swap to create it. Our solution assumes the existence of a "black box" that can compute a distance between two arbitrary nodes on the physical network. This black box may be implemented in many different ways, e.g., [FJP+99], [RHKS02], [XTZ03], [CDK+03], and is used by our solution.

It may not be immediately obvious that the swapping idea is applicable to any P2P system; one might wonder whether performing such swapping can result in an invalid overlay network. In Section II, we will explain what swapping really translates into, when applied to a P2P system, and why it is a "safe" operation on all P2P systems.

Extending the intuition presented so far into a practical, generic solution for constructing a network-aware overlay poses many challenges, including:

Identifying nodes to involve in swaps: We need Apocrypha to operate in a distributed fashion and identify pairs of nodes to swap while minimizing the number of invocations of the black box to compute proximity.

Minimizing swap overhead: The swapping of two nodes may be necessitate swapping additional application state in some P2P systems. For example, with distributed hash tables (DHTs), swapping two nodes may also require swapping of the content they store, and we need optimizations in order to reduce such overhead.

Adapting to dynamism: Finally, a major challenge in the design of Apocrypha is to enable it to adapt to dynamic node arrivals and departures. When nodes join and leave an "optimized" overlay network, the quality of the network may be adversely affected and, consequently, work needs to be performed to reoptimize the network. The higher the rate of dynamism, the more expensive it can be to maintain an efficient overlay network. Consequently, Apocrypha adaptively trades off the cost of maintaining and optimizing the overlay network against the quality of the resulting network.

To the best of our knowledge, Apocrypha is the first solution for constructing network-aware overlays for many kinds of P2P systems, specifically systems with deterministic, randomized or ad hoc topologies. (We explain what these topologies are in Section II.) Apocrypha also appears to be the first solution that explicitly trades off optimization cost against network quality in the face of varying rates of dynamism. In addition, we believe that a generic mechanism such as Apocrypha helps the application developer by detaching the problem of choosing an overlay topology from that of making the structure adapt to the underlying physical network.

Evaluating the utility of Apocrypha on large overlay networks requires a realistic model of physical-network latencies. We evaluate Apocrypha both on well-known internet models, as well as using real data gathered on PlanetLab. We demonstrate that our algorithms are (surprisingly) robust and perform well on a variety of physical-network models, as well as on the real data.

We apply and evaluate Apocrypha on two different P2P systems: (a) Chord, which uses a hypercube-like structured topology, and (b) Gnutella, which operates on an ad hoc topology. In the process, we introduce new routing algorithms for Chord that considerably improve query latency in the Chord system. On the Gnutella system, we demonstrate a surprising phenomenon that we call *long circuits*. We note that these routing algorithms, and the long-circuit phenomenon, are interesting in their own right even outside the context of Apocrypha.

Our paper is organized as follows. In Section II, we classify P2P systems based on their topology and explain how Apocrypha is applicable to all of them. In Section III, we define the problem of creating a network-aware overlay for a static P2P system, describe a simple centralized algorithm for it, and evaluate it using graph-theoretic metrics. Section IV describes how to translate the centralized algorithm into a distributed one. Section V discusses the dynamic version of the problem where nodes may join and leave the P2P system. Section VI discusses the application of Apocrypha to Chord, and evaluates the utility of Apocrypha in terms of query latency on Chord. Section VII discusses an evaluation on Gnutella, and describes the long-circuit phenomenon. We discuss related work in Section VIII and conclude in Section IX.

II. Classifying P2P Systems and Understanding the Swap

In this section, we classify P2P systems based on their topology, explain what the swap operation translates into in each class, and justify the safety of this operation.

Before we do so, we make two observations. First, the swap operation merely transforms an overlay network into another isomorphic overlay network, i.e., the "shape" of the overlay network remains unchanged by the swap. Second, the justification of the safety of the swap lies in an important underlying property of P2P topologies that we will prove in this section:

Validity Closure: For any P2P system, if a given overlay network is valid, then all overlay networks isomorphic to it are valid.

If this property holds for a P2P topology, it is clear that the swap is a safe operation. We now classify P2P topologies and demonstrate that they all possess the validity-closure property.

Deterministic Topologies: A deterministic topology is one in which every valid overlay network is isomorphic to a specified graph, that we call the *basis graph* B of the topology. Our "hypercube topology on 4 nodes" is a deterministic topology whose basis graph is the square. Validity closure clearly holds for deterministic topologies: If N_1 is isomorphic to B and N_2 is isomorphic to N_1, then, by transitivity, N_2 is isomorphic to B.

In fact, the converse is also true, i.e., if two overlay networks N_1 and N_2 are valid, they must be isomorphic to each other.

Many P2P systems, e.g., CAN [RFH$^+$01] and Chord [SMK$^+$01], use deterministic topologies. In these systems, each node is assigned a unique id, or *label*, chosen at random from some space of labels. The basis graph B of the topology is a function of the chosen set of labels. In other words, the P2P system dictates what pairs of labels have an edge. Each valid overlay network corresponds to an assignment of labels to nodes, with an overlay link between a pair of nodes if and only if their corresponding labels have an edge in B. Our solution starts with an initial assignment of labels to nodes (and a corresponding overlay network), and then let nodes *swap* labels, while always maintaining an overlay network consistent with the labeling of nodes. Performing a "correct" series of such swaps will lead us to the best possible valid overlay network.

Nondeterministic Topologies: A nondeterministic topology is one where every valid overlay network is isomorphic to *any* one of a set of basis graphs. The validity-closure property continues to be true by exactly the same argument as earlier. However, the converse is no longer true; even if an overlay network N_2 is not isomorphic to a valid network N_1, N_2 could still be valid. An extreme form of nondeterministic topologies is an *ad hoc topology*, where every overlay network is considered valid.

P2P systems such as Pastry [RD01] and Tapestry [ZKJ01] use nondeterministic topologies. Each node again has a unique label but, now, the overlay topology corresponds to any one of multiple basis graphs on these labels. Again, Apocrypha starts with some assignment of labels to nodes, with an overlay network corresponding to one of the multiple basis graphs, and reassigns labels to nodes while preserving the same basis graph. Apocrypha can thus no longer explore the space of *all* valid overlay networks, since it confines itself to overlay networks corresponding to just one basis graph.

Gnutella [gnu] and its super-peer variants like KaZaa [kaz] and LimeWire [lim] are examples of P2P systems built on ad hoc topologies. Again, Apocrypha works the same way and does not explore the full space of valid overlay networks. However, we note that none of these systems *desire* a truly ad hoc topology. For example, all of them would ensure that the overlay network forms a connected graph, and rely on randomized topologies to provide such properties, as we discuss next.

Randomized Topologies: Randomized topologies are a special form of nondeterministic and ad hoc topologies. While there are still multiple basis graphs available in such topologies, the P2P system can no longer choose *arbitrarily* among these different basis graphs. Instead, there is a specific probability with which each basis graph ought to be chosen in order to guarantee important properties of the system such as connectedness, low diameter, fault tolerance, degree balance and efficient routing.

P2P systems such as Symphony [MMP03] and low-diameter constructions on Gnutella-like systems [PRU01] use randomized topologies. Since the "important properties" of the system all rely on the initial basis graph being chosen by a randomized protocol, it is critical to "preserve" this basis graph when transforming the overlay network, which is exactly what Apocrypha does.

III. Apocrypha: The First Cut

We begin by defining the problem of constructing a network-aware overlay for an arbitrary *static* P2P system:

Problem: Given (a) an initial valid overlay network with each node assigned a unique label, and (b) a black box d that computes the "distance" between any two nodes, create a new overlay network by re-assigning the original set of labels among the nodes (and constructing appropriate overlay links), such that the average distance, computed over all overlay links using d, is minimized.

Note that our objective of minimizing average link distance may not correspond directly to an application metric. We evaluate Apocrypha in terms of application metrics in later sections. In this section, we devise a centralized algorithm for solving this problem. In later sections, we show how to adapt it into a practical, distributed mechanism for solving the same problem, and modifications to deal with dynamic overlay networks.

A. A Centralized Algorithm

We now discuss how to solve the above problem assuming that all the input is available in a single location. The problem can be shown to be NP-hard by a simple reduction from the graph-isomorphism problem, and we adopt a hill-climbing approach to solve it. We sketch this basic hill-climbing algorithm in Algorithm 1. The term $nbrs(i)$ is shorthand for the set of neighbors of node i on the overlay network.

Algorithm 1 Apocrypha(OverlayNetwork N, DistanceFunction d)

```
1:  while not converged do
2:    for each node $i$ do
3:      Pick a random node $j$.
4:      InitialCost= $\sum_{m \in nbrs(i)} d(i,m) + \sum_{n \in nbrs(j)} d(j,n)$;
5:      NewCost= $\sum_{m \in nbrs(i))} d(j,m) + \sum_{n \in nbrs(j)} d(i,n)$;
6:      if NewCost < InitialCost then
7:        Swap Label($i$) and Label($j$);
8:        Break edges between $i$ and its neighbors
9:        Break edges between $j$ and its neighbors
10:       Create edges between $i$ and $j$'s old neighbors
11:       Create edges between $j$ and $i$'s old neighbors
12:     end if
13:   end for
14: end while
```

At each step, the algorithm selects each node i in turn and considers whether it is beneficial to swap the labels of i and a randomly chosen node j. Note that performing this check can be expensive, as it requires computing the distance d between i and every neighbor of j, and between j and every neighbor of i. If it is determined that the swap is beneficial, i and j proceed to exchange their labels. This step corresponds to i and j exchanging their *set* of neighbors on the overlay network. (In the special case when the two nodes are already neighbors, they also continue to remain neighbors after the swap.) The algorithm terminates when there is no more, or very little, progress made.

B. Evaluation

We now evaluate the effectiveness, and the cost, of the above hill-climbing algorithm as it operates on different kinds and sizes of overlay networks. Our evaluation requires a model of the physical network that captures the distance between every pair of overlay nodes. We begin by using a physical-network (internet) model generated from the standard GT-ITM topology generator, that enables us to simulate the algorithm on overlay networks of large sizes. Later, we describe results on a smaller-scale overlay network, where the overlay nodes are machines on PlanetLab, with pair-wise distances between overlay nodes being measured directly based on ping times.

1) Evaluation using GT-ITM: We use the GT-ITM topology generator [ZCB96] to generate transit-stub models of the physical network. Overlay nodes in the overlay network are attached at random to stub nodes in the GT-ITM network model. The distance between a pair of overlay nodes is set as the latency between them in the internet model. (We will henceforth use the words “distance” and “latency” interchangeably.)

We used GT-ITM to generate multiple internet models with the same average graph degree but different proportions of transit and stub domains; we assigned latencies of 5, 20 and 100ms to stub-stub, stub-transit and transit-transit links respectively, with the link from an overlay node to its corresponding stub node having a latency of 1ms. We also experimented with adding different levels of Gaussian noise to these latencies. In general, we found that the results obtained were fairly robust and insensitive to the exact parameters used in our internet models. The results presented here were all obtained on one representative model.

It has been observed that the distribution of latencies as obtained from GT-ITM is somewhat different in its characteristics from latency distributions observed in the real world [GGG$^+$03]. Interestingly, we experimented with simple variants of the GT-ITM model, using non-uniform distributions of overlay nodes to different transit domains. The resulting latency distributions appear to be much closer to observed, real-world latencies than with the plain GT-ITM model. Due to its digressive nature, we consign our description of these internet models, and the results obtained on them, to an extended technical report [GSGM03]. Our evaluation on these internet models suggests that Apocrypha performs much better in these modified models than on plain GT-ITM. Therefore, the results presented here offer a conservative estimate of the likely benefits of Apocrypha.

We choose two different overlay networks on which to evaluate the application of Apocrypha: Gnutella and Chord. The detailed structure of these networks do not concern us immediately, and will be described in Sections VI and VII. Our Gnutella network consists of a 24702-node snapshot of Gnutella obtained in 2001 [Cli], while our Chord network is

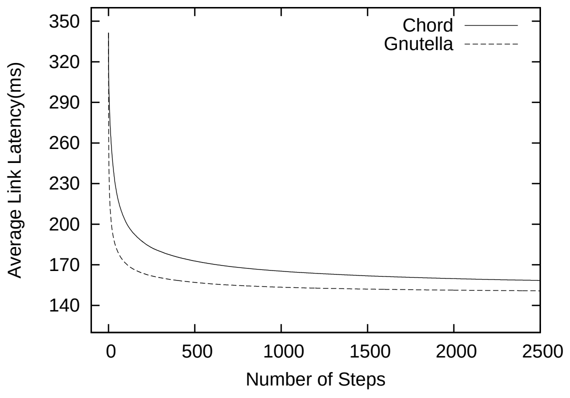

Fig. 2. Convergence of Apocrypha on Chord and Gnutella

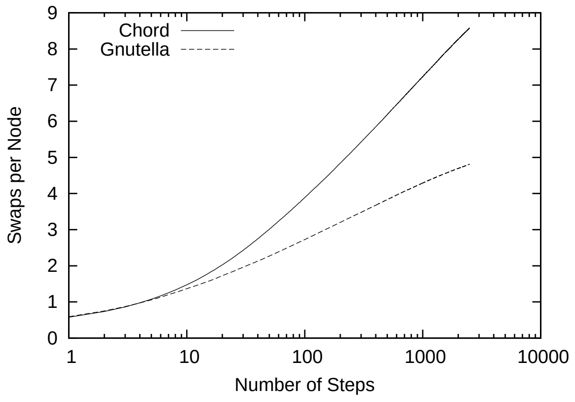

Fig. 3. Number of Swaps performed by Apocrypha

constructed on 25000 nodes in the standard fashion as described in [SMK$^+$01]. We start out measuring the average link latency, as well as the number of swaps made by Apocrypha in its optimization.

Average Link Latency: Figure 2 plots the *quality* of the overlay network, i.e., the average of the overlay-link latencies, as a function of the number of steps that Apocrypha runs for. The initial average link latency is about 341ms, corresponding to the average latency between a pair of nodes in our internet model. On the Chord network, the link latency drops to 172ms after 500 steps, and to 158ms after 2500 steps, resulting in an improvement by roughly a factor of 2 compared to the initial state. On the Gnutella network, the average latency drops to just 161ms after 250 steps and eventually falls to 150ms after 2500 steps, thus demonstrating slightly greater improvement than Chord. This is not too surprising, since the Gnutella network has a lower average degree, and greater degree skew, which makes it less “constrained” and easier to optimize. We note that although our algorithm is only guaranteed to find a local minimum, the observed network quality is actually very close to optimal, as we discuss in our extended technical report [GSGM03].

Number of Swaps: A more surprising result is that Apocrypha uses very few swaps to achieve the rapid convergence observed in Figure 2. Figure 3 plots the number of swaps initiated per node as a function of the number of steps. Note that

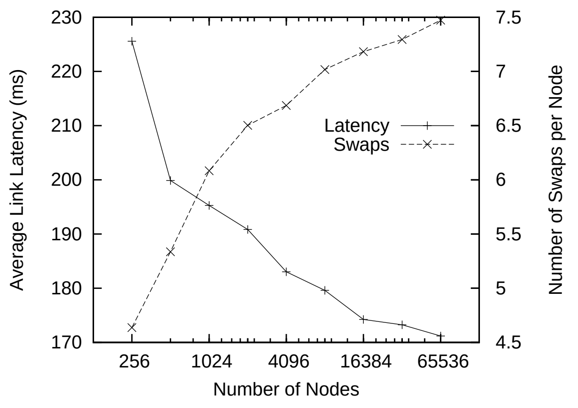

Fig. 4. Quality and Number of Swaps as a function of Network Size

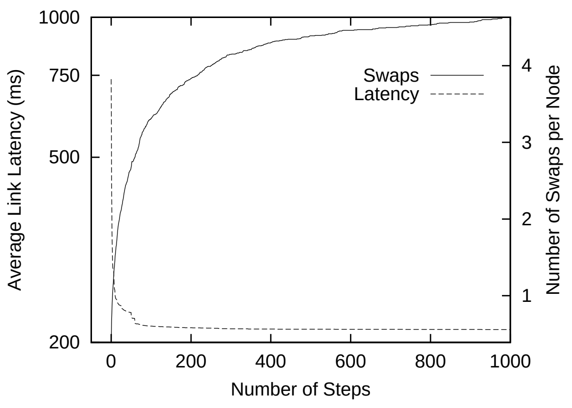

Fig. 5. Quality and Number of Swaps for a 223-node Chord network on PlanetLab

the x-axis is plotted on a logarithmic scale. For both Chord and Gnutella, we see that the number of swaps increases logarithmically with the number of steps, with about half the swaps occurring in the first 100 steps. Moreover, the total number of swaps performed is extremely small, being slightly greater than 8 for Chord, and under 5 for Gnutella, even after 2500 steps.

Practical Implications: The above two graphs suggest that this basic hill-climbing approach could be the basis for a practical solution. First, we need very few steps to converge from a completely arbitrary overlay network to a good one. In practice, Apocrypha would run incrementally and will never need to start from an arbitrary configuration. Second, the total number of swaps necessary per node is extremely low, suggesting that each step could be performed pretty "fast" in a distributed implementation.

Varying the Number of Nodes: We now study how the quality of the overlay network, and the number of swaps performed, evolves as a function of the size of the overlay network. We present experiments on the Chord overlay network, in which the degree of each node is roughly $2 \log n$ where n is the number of nodes in the network. Figure 4 depicts both the quality (avg. link latency) of the overlay network, and the number of swaps performed, for different-sized Chord networks, after 1000 steps of the hill-climbing algorithm. Note that the x-axis is in logarithmic scale.

We see that the average link latency decreases as the network gets larger. There are two reasons for this phenomenon. First, a smaller Chord network is much more "dense" than a larger network, meaning that a large fraction of all possible edges are present in the network. Consequently, a small network is more constrained and harder to optimize. Second, as the number of nodes increases, there are more pairs of nodes that are close to each other on the physical network, enabling better optimization of overlay links.

Interestingly, the number of swaps performed per node increases roughly *as a logarithmic function of the number of nodes*. (We note that the number of swaps becomes even closer to a logarithmic function if the hill-climbing algorithm is run to "convergence within ϵ" for each network size, rather than for 1000 steps in all cases.) Note that the degree of each node is also a logarithmic function of the number of nodes. Consequently, we see that the number of swaps per node is proportional to the average degree of nodes in Chord, suggesting that each swap "optimizes" a constant number of links. We have observed the same "swaps proportional to degree" behaviour on other kinds of overlay networks besides Chord, too.

2) Evaluation on PlanetLab: We consider a Chord overlay nework implemented on PlanetLab computers. PlanetLab [plaa] consists of machines distributed at a few hundred sites across the world. We assigned 223 nodes, distributed across 128 sites, to a Chord network and gathered the measured ping times [plab] between every pair of nodes. We then simulated Apocrypha on this Chord network, using the measured ping time as the distance between a pair of nodes. Pings between some pairs of nodes do not succeed, and time out on a consistent basis; we set these ping times to be ten seconds.

Figure 5 plots both the average link latency and the total number of swaps performed, as a function of the number of steps of the hill-climbing algorithm. We note that the average link latency drops very steeply (notice that the y-axis is in log scale) initially, even more so than in the earlier experiments on the GT-ITM topology. This steep drop is due to the quick adaptation of the network in avoiding the extremely bad links with very large latencies. We also observe that convergence to the final value is relatively fast, and the final average latency is a factor 3.5 smaller than the initial average. The number of swaps performed is practically identical to the number of swaps in the GT-ITM topology for comparable Chord networks (Figure 4), and each node initiates slightly more than 4 swaps in total.

Overall, we see that the relative quality improvement we obtain on PlanetLab is higher than on the GT-ITM topology. Since a large fraction of the quality improvement could potentially have been obtained by simply eliminating the "extremely bad" links between nodes that do not respond to each other's pings, we repeat the above experiment on a *smaller* 182-node Chord network, where the nodes which do not respond to more than 10% of pings are eliminated. This smaller network has less than 0.2% of all pairwise ping times being larger than 5 seconds.

Figure 6 plots the quality and the number of swaps as a function of the number of hill-climbing steps for this 182-node network. We observe that the initial average latency is only 133ms,

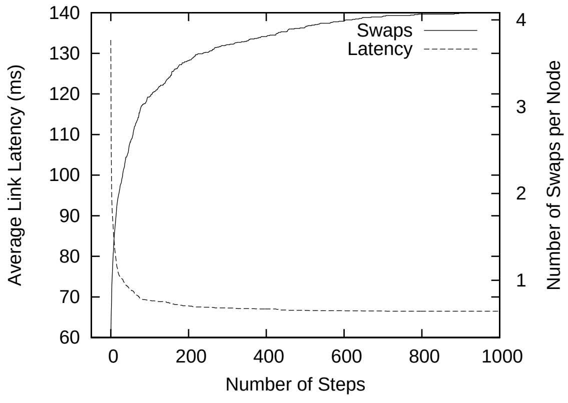

Fig. 6. Quality and Number of Swaps for a 182-node Chord network on PlanetLab

which is considerably smaller than the initial average of 739ms in the earlier 223-node case. The final latency after optimization drops to 66ms, still a factor of 2 improvement over the initial latency. This improvement is actually larger than that for a correspondingly sized Chord network operating on GT-ITM, which may have two explanations: (a) the GT-ITM model overestimates the distance between nodes, or (b) nodes on PlanetLab are nearer each other than arbitrary nodes on the internet. We observe that the number of swaps performed is once again very similar to that in the earlier cases, being slightly larger than 4 per node.

IV. A Distributed Solution

We now design a distributed solution for the problem of creating network-aware overlays, in which each participant node runs an algorithm in order to evolve the (static) overlay network into an optimal state. One simple way to design a distributed algorithm is to just let each node execute steps 3-12 of Algorithm 1 in a periodic fashion. There are two difficulties that need to be overcome for this idea to work: (a) A node needs to be able to find another random node j in step 3 of the algorithm. (b) We need to lower the overhead of running the algorithm on a continuous basis. We presently discuss how each of these two issues are tackled.

A. Finding a Random Node

To find another random node in a distributed manner as in step 3 of Algorithm 1, each node i initiates a *probe*, a random walk originating at itself and proceeding for a pre-determined number of steps, say W. The termination point j of the random walk is chosen by i as the candidate it considers swapping labels with. Nodes i and j can then collaborate to decide whether exchanging labels and neighbors is beneficial to the overlay, just as in the original algorithm. Of course, the quality of our algorithm depends on the choice of W, the length of the random walk. The ideal choice of W depends on the mixing properties of the overlay network, but our experiments indicate that setting W to be about half the diameter of the overlay graph is a good idea; using smaller values of W may lead to results inferior to that obtained by the centralized algorithm, and for higher values of W, there is practically no marginal improvement on most practical P2P topologies.

Note that these probes can made more efficient in many systems by piggybacking them on other traffic. In DHTs, the path taken by a query for a key translates into a random walk on the overlay network. In systems using random walks for queries, the same random walk can be used to find a random node. In flooding-based systems like Gnutella, the *source* of a query can be treated as a random point by a node receiving the query (so long as the TTL on the received message is low). Thus, finding random nodes in the network can often be made inexpensive.

Biasing probes: Instead of using a purely random walk to find a candidate, we could “bias” the random walk to seek candidates more likely to swap places with the probe initiator. The policy we consider is to assign every node a *dissatisfaction index* which is equal to the average of its overlay-link latencies. A probe then returns the most dissatisfied node encountered on the random walk, instead of returning the termination point. The danger in using such a policy is that the likelihood of converging to a local minimum is higher than with random hill climbing. Our experiments later on in this section evaluate whether such biasing is a good idea.

B. Minimizing Overhead

The second challenge we face is to lower the overhead of each node running the algorithm on a continuous basis. Each time a node executes steps 3-12 of the algorithm, it needs to (a) initiate a probe, (b) check whether a swap is beneficial, (c) possibly perform a swap by breaking and forming overlay links, and (d) possibly swap application state if necessary. We have argued that the overhead of step (a) can be made pretty small. The overhead of step (c) is not too big a concern either since we have seen that the number of swaps performed per node is extremely small and decreases with time. Step (d) can be expensive on some P2P systems and we discuss how its overhead may be minimized in Section V-A. We are left with the overhead of step (b) which we now discuss.

Every time a node finds a candidate by probing, it needs to check whether a swap is beneficial, which requires invoking the black box to compute latency between the node and each of the candidate’s neighbors. For some black box implementations, like the Euclidean vectors suggested in [RHKS02], [CDK$^+$03], the invocation is cheap and there may actually be little overhead associated with this step. However, other implementations of the black box, such as active ICMP pings, might be much more expensive to invoke. We would therefore like to somehow reduce the number of probes initiated, both while actively improving the overlay network, and, more importantly, after converging to a point where very little improvement is being achieved. Of course, detecting this global convergence in a distributed way is non-trivial.

We can think of the following simple solution: each node continuously monitors the “improvement” in its dissatisfaction index as a result of each of its probes over the last τ minutes.

If this improvement is very low, the node realizes that the overlay network has converged to a near-optimal state and can stop initiating probes. This idea does not quite work because each successful swap is only guaranteed to improve the *overall objective function* for the entire network, not an individual node's dissatisfaction index.

A better solution is for each node to monitor its *state fluctuation*, the difference between its best and worst dissatisfaction index over the last τ minutes. A node then continues to "wake up" periodically, but initiates a probe only if its state fluctuation is greater than some threshold ϵ. If the fluctuation is less than ϵ, the node decides to avoid probing, a step we call *probe quenching*. For robustness, a node is also given a small nonzero probability to initiate a probe even if the fluctuation is less than ϵ. So, once the network evolves into a "good" state, the fluctuation is likely to be very low, and the probe initiation rate essentially corresponds to the small probability of initiating a probe for robustness. For reasons that will later become apparent, we will call this probe-quenching mechanism the *hot-stove policy*. Note that nodes always respond to a probe *initiated by some other node*.

C. Evaluation

To see how the use of probes, biasing probes and probe quenching affect the quality of the overlay generated by Apocrypha, we ran the distributed version of Apocrypha on a 25000-node Chord network, using the GT-ITM internet model. The results on other physical-network models, on PlanetLab data, and using the Gnutella network, are along similar lines.

Nodes consider initiating a probe once a minute, and send out the probe if it is not quenched. (Our choice of time unit is arbitrary. In practice, the time unit may itself be a function of the state of the network as we discuss in our section on dynamism.) For the quenching criterion, we use $\tau = 20$ and $\epsilon = 1$ms, i.e., we quench a probe if the dissatisfaction index measured over the last 20 minutes fluctuates by less than 1ms[2] Again, recall that we are starting from a completely arbitrary initial network whereas, in practice, we would maintain the overlay network incrementally and always hover around a near-optimal state.

Figure 7 depicts the convergence of distributed Apocrypha (using walks with $W = 10$) as a function of time; the centralized algorithm is used as a baseline for comparison. Notice that it is hard to make out the different lines because they all more or less coincide, which illustrates exactly what we want to show: Distributed Apocrypha, with or without the use of biasing and quenching, still performs almost identically to the centralized algorithm. We see on the bottom right of the graph that the use of quenching makes the network converge to a slightly inferior state than without quenching. This is only to be expected, since quenching avoids investing too much effort in obtaining marginal improvements in overlay quality. We have not shown the number of swaps used by each of the algorithms but they, too, are almost identical.

[2]A point of concern might be that the distance d between the same two nodes might vary over time and that our choice of ϵ is too low. Such cases are handled identically to the way we handle dynamism in the next section.

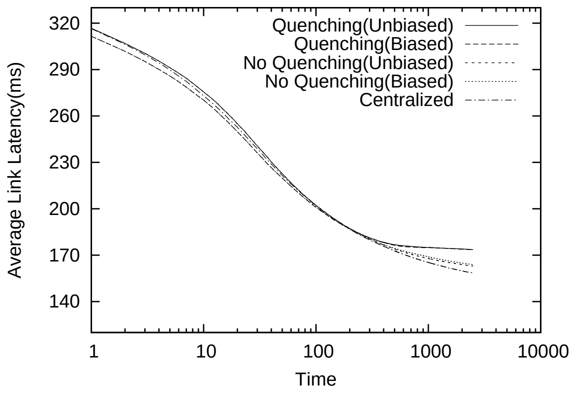

Fig. 7. Convergence of distributed Apocrypha on the Chord network

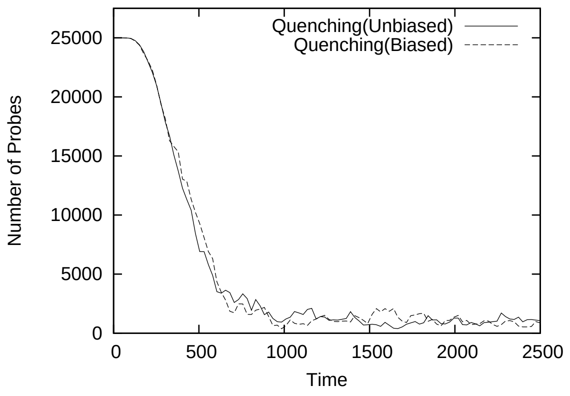

Fig. 8. Number of probes initiated at each time step

Figure 8 shows the number of probes initiated in the network every minute. Note that we always have 25000 probes a minute without quenching. We see that the use of probe quenching is effective in reducing the number of probes after very little time. With quenching, each node initiates a probe about once every 25 minutes, once the overlay network converges to a reasonable state. We also see that biasing does not appear to either help or hurt in reducing the probes. The real benefit of using biased walks will be seen in our experiments on dynamic overlay networks. Experiments on the Gnutella network produced similar results and are omitted.

V. Dealing with Dynamism

So far, we have ignored the fact that the overlay network changes continuously as nodes join and leave the P2P system. How well Apocrypha adapts to dynamism is dependent on how much the overlay topology itself changes when nodes join and leave. For example, there isn't much that the algorithm can do if the entire overlay structure changes every time a node joins or leaves. However, most P2P systems do not behave this way and allow the overlay structure to change smoothly. For many P2P systems building distributed hash tables, the number of edges that need to be added or removed due to a node arrival or departure is usually a small logarithmic fraction of the total edges in the system. For Gnutella, the departure or arrival of a node affects only a constant number of edges.

Apocrypha adopts the following approach to dealing with dynamic node arrivals and departures: The P2P-system protocol is permitted to execute just as it normally does when nodes join and leave the system. As a result, new overlay links may be created and some old overlay links may be deleted. In consequence, the average overlay link latency may become worse (or better, if we got lucky). However, since a majority of the overlay links do not change, we still expect the overlay network to be in reasonably good shape. Apocrypha can then continue to initiate probes and perform swaps on this new overlay network to improve it, and restore it to its former glory.

Even at extremely high rates of dynamism (with mean node lifetimes less than one hour), our experiments show that Apocrypha (with each node considering initiating one probe per minute) successfully improves the overlay network to a quality as good or better than in the static case. The problem is that our hot-stove policy for probe quenching does not help at all in reducing overhead; with a high rate of node dynamism, it is very likely that a node receives at least one new neighbor in the interval of time τ we used as our window. This one new neighbor is likely to throw the state fluctuation over the threshold of ϵ that we set earlier. Consequently, the probe-quenching mechanism hardly ever decides not to perform a probe. Apocrypha continuously performs a lot of work to keep up with all the changes happening in the system in order to improve the system, perhaps more than we want it to. At high rates of dynamism, we would like the option of performing probes at a lower rate while perhaps settling for a lower-than-optimal quality overlay network.

Our solution to dealing with dynamism is based on a slightly counter-intuitive observation. If the overlay network is dynamic, nodes need to be less aggressive in initiating probes. The reason is that we desire to permit some "slack" for the network quality to get slightly worse before we fix it. If no slack is allowed, we will only be wasting a lot of effort making small improvements that will soon disappear if there is more dynamism. Moreover, by allowing the network to deteriorate slightly, it becomes possible to make larger improvements more efficiently. This leads us to the idea that we should "slow down time" when the network is dynamic, and let nodes react as if in slow motion.

Of course, since we have a distributed system where a node can only observe the dynamism among its neighbors, nodes need to make *local* decisions without any knowledge of the global rate of dynamism. Despite nodes making such local decisions, we require that the cumulative behaviour of the nodes be a function of the globally observed rate of dynamism.

We devise a new probe-initiation policy, called the *Pretty-Girl Policy*[3], which provides the above behaviour by using a three-state automaton. Let us define a node to be *young* if it joined the network within the last τ minutes, and old otherwise. Our new probe-initiation policy now becomes:

- If k of a node's neighbors are young with $k > 0$, the node is in *dynamic mode* and initiates a probe with a probability $a + b \cdot k$ for some constants a and b.
- If none of a node's neighbors are young, and the node was in dynamic mode earlier, it switches to *tentative-static* mode. It slowly increases the probability of initiating a probe until either (a) p probes are sent out in tentative-static mode or (b) one of its neighbors is young. In case (a), the node switches to *static mode*. In case (b), the node switches back to dynamic mode.
- In static mode, a node uses our originally suggested hot-stove policy for quenching.

[3]The basis of this name lies in Albert Einstein's work on understanding external sensory input on time dilation [Ein38].

Some intuition into the above policy can be obtained as follows: It is clear that the aggression of a node in initiating probes should be a function of the dynamism it observes *among its neighbors*, since nodes are required to make local decisions. If very few of a node's neighbors are newly joined nodes, then it ought to be less aggressive and let the newly arrived nodes do the work to improve the system. But if a lot of its neighbors are new, then the node itself needs to step up and initiate probes so that the network does not deteriorate too much.

The pretty-girl policy helps us deal, not only with different rates of dynamism, but also with bursts of dynamism interspersed with periods of calm. When a burst of dynamism ends, nodes cautiously send out probes and slowly step up the frequency of probing until they are sure the system is indeed static. Once they realize that it is static, they go back to their usual quenching policy.

Figure 9 depicts the effects of the pretty-girl policy on the convergence of the network in the face of different rates of dynamism. We again start with an arbitrary, initial Chord overlay network of 25000 and have δ nodes joining and δ nodes leaving the system every minute (for a total of 2δ nodes changing in every minute). Each node considers initiating a probe once a minute and, of course, may not do so if the pretty-girl policy so dictates. We show the evolution of the network for different values of the dynamism δ. The topmost curve depicts running Apocrypha (at $\delta = 100$) without the use of biased walks, and can be seen to be vastly inferior to using biased walks at the same rate of dynamism. The remaining curves all use biased walks, and we see that, at low dynamism, the system evolves more or less similarly to the static case. When the dynamism is higher, the network tends to stabilize at a higher average latency rather than climbing all the way down to a quality near that of the static case. The reason is that the pretty-girl policy concludes that it is too expensive to maintain a high-quality network at that continuous rate of dynamism.

Figure 10 shows how many probes are being initiated at each rate of dynamism, again as a function of time. The spikes on the left end of the curve may be ignored, as they are artifacts of the system starting with all nodes joining at the same time. We see that the static case performs the least probes, the case $\delta = 2$ performs more probes, and there are even more probes at $\delta = 25$. However, what is interesting is that *fewer* probes are performed at $\delta = 100$ than $\delta = 25$, because Apocrypha realizes that it is not worth performing more probes at that high rate of dynamism. We note that the hot-stove policy would quench practically no probes at all under any rate of dynamism (at $\delta = 100$ for example, it would perform about 24900 probes

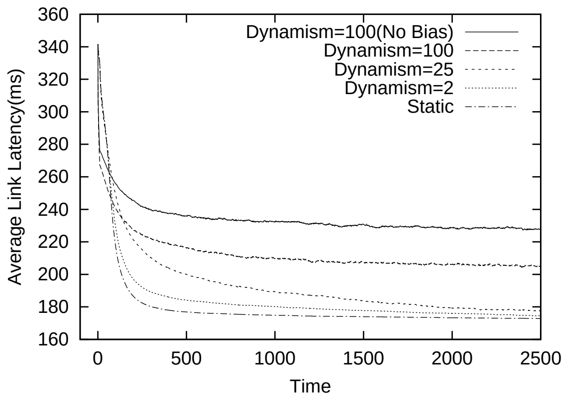

Fig. 9. Convergence as a function of time

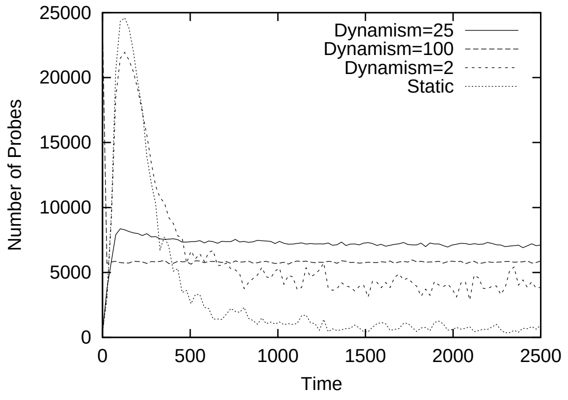

Fig. 10. Number of probes quenched

every minute). At all rates of dynamism, the number of probes initiated per minute using the pretty-girl policy is much lower than that with the hot-stove policy.

Figure 11 shows what happens when the dynamism in the network is bursty. We again start with an initial unoptimized network, and there is no dynamism except for three bursts in the intervals $[500, 700]$, $[1500, 1700]$ and $[2500, 2700]$. During these intervals, there are $\delta = 100$ nodes joining and $\delta = 100$ nodes leaving the network each minute. The figure shows evolution of both the quality of the network and the number of probes initiated. We see that the number of probes stays relatively low during the bursts of dynamism, while the quality starts deteriorating. But the deterioration does not get arbitrarily bad (as evidenced by the "flat top" of the humps in the latency curve). When the dynamism ends, the number of probes drops initially (just before each big spike) while nodes are in tentative-static mode and are sending out "feelers" to check if the network is indeed static. Once nodes detect the absence of dynamism, they start probing aggressively to improve the network quickly. When the improvement flattens out, they once again reduce the number of probes initiated. Over the long term, we see that the average latency of the network declines steadily.

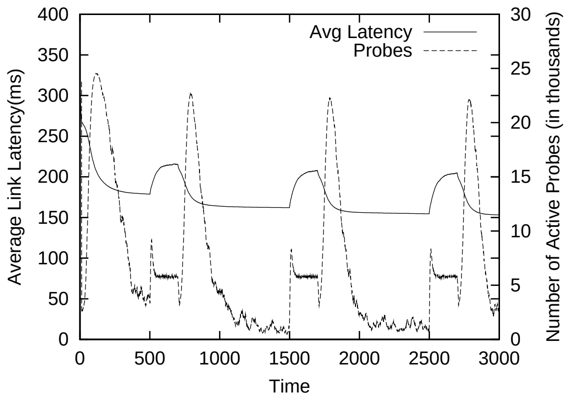

Fig. 11. Effects of burstiness

A. Using Proxies

Finally, we show how to minimize the cost of making nodes swap labels. We have already seen that Apocrypha requires very few swaps per node, but even this many swaps may be expensive since nodes may have to swap application state associated with the labels. In Gnutella, or systems using overlay networks for efficient broadcast, swapping is free since there is no associated application state. Similarly, in P2P multicast systems built on structured overlay networks [RKCD01], swapping is practically free. On the other hand, in P2P distributed hash tables (DHTs), a node's label determines the content it stores and, therefore, swapping labels entails having to swap the associated content too.

One simple technique to reduce the overhead in DHTs is as follows: When two nodes swap labels, they do not swap their associated content. Instead they set up *proxy links* to each other, so that a query directed to one node for the other's content may be answered by forwarding the query along the proxy link. (Note that new content inserted into the system should be stored at the correct node, and can be reached without using the proxy link.) This approach has many advantages.

When a node n joins the system, the P2P system gives it a label and requires the node to "take over" a portion of the hash space that some other node m is currently responsible for. Node n can tentatively establish a proxy link to m instead of actually moving content over. Since n has not been in the system very long, it is likely that it has a high dissatisfaction index and, consequently, Apocrypha will make n swap labels multiple times before n "settles down" with some label. Each time n swaps labels with some other node n', it can hand over its current proxy link to n', and create a proxy link to n' for itself. When n is no longer involved in swaps for a while, it can move the content over from whichever node its proxy link points to. Thus, the amount of content that needs to be swapped can be reduced considerably.

Second, in many DHT systems, index data stored by a node is designed to "expire" after a certain amount of time. This expiration time is often very low, and the proxy link will quickly become unnecessary as all the data that it points to will soon expire. In the meanwhile, a node can lazily cache all results to queries that are sent over the proxy link, thus improving perfor-

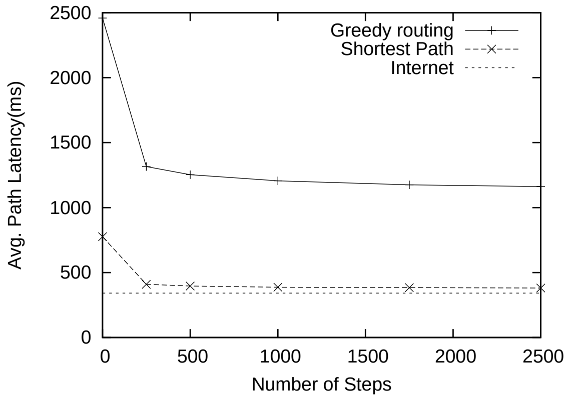

Fig. 12. Performance of queries on Chord using greedy routing

mance and avoiding copying data that is not queried for.

VI. Applying Apocrypha to Chord

So far, we have described and evaluated the quality of overlay networks optimized by Apocrypha, using a generic graph metric: the average link latency. The real question, however, is to understand the impact of an "improved" overlay network on the actual applications that use the overlay network. To answer this question, we study two P2P search applications, Chord (this section) and Gnutella (the next section), and evaluate Apocrypha using application metrics such as the average query latency, and the fraction of query results received.

We first describe Chord, and the hash-table functionality that it provides. Chord is a distributed hash table storing key-value pairs. Keys are in the circular range $[0, 2^N)$, and each node is assigned a unique "id" (or label, in our terminology) which is simply a randomly chosen integer in the range. A node is then designated responsible for storing all keys less than or equal to its id, and greater than the next smaller existing node id. A node with id x maintains links to the nodes responsible for $x + 2^k$ for all integers $0 \leq k < N$, with arithmetic modulo 2^N.

Answering a query for a given key is equivalent to forwarding the query to the node responsible for that key, using a sequence of overlay links. This forwarding is achieved using Chord's greedy routing protocol. Greedy routing is simple: a query is forwarded continuously in the clockwise direction to the neighbor whose id is closest to that of the query. (Note that we are not permitted to "overshoot" the query key.) If nodes choose ids randomly, the size of the key space that each node is responsible for is about the same, and, assuming that queries are initiated equally at every node and for every key, the average query latency is equal to the average latency in routing a message between every two nodes using Chord's greedy routing protocol. We refer to the latency along some specific path from one node to another as the *path latency*. Thus, the average query latency is simply the average path latency when paths are chosen using greedy routing.

Figure 12 shows the average path latency (for different kinds of paths) on Chord as a function of time as Apocrypha optimizes a static 25000-node Chord network on a GT-ITM topology. The horizontal line at the bottom represents the average latency between pairs of nodes in our internet model, and is thus a lower bound for any overlay-routing protocol[4] The top curve depicts the average of the path latency between every pair of nodes using greedy routing; note that the average path latency drops from about $2500ms$ on the initial unoptimized network to about $1160ms$ by the use of Apocrypha. The middle curve plots the average latency of the *shortest overlay path* between pairs of nodes. We see that this shortest-overlay-path latency between two nodes is very close to the internet latency (380ms after 2500 steps, compared to an internet latency of 341ms). The trouble is that greedy routing does not appear to use this shortest path and instead uses a path of much higher latency for routing. Luckily for us, greedy routing is not the only way to route on the Chord network. We explore three new routing protocols for Chord which all progressively improve on greedy routing and come closer to finding the shortest overlay path between nodes. We then evaluate the performance of these routing protocols both on the GT-ITM topology and on the PlanetLab data.

A. *Choosing paths wisely with Solomon*

The Chord structure offers multiple alternative paths from any node to any other node. For simplicity, imagine that there are exactly 2^N nodes with labels $0..2^N - 1$. Greedy routing is then equivalent to expressing the distance to the destination as a binary string, and "fixing" the 1s left-to-right. For example, we would cover a distance of length 56 (111000 in binary) by three successive hops of length 32, 16 and 8, which is equivalent to left-to-right bitfixing on the binary string 111000 to convert the 1s to 0s. We observe that any routing protocol that fixes all these bits will also require the same number of hops, irrespective of the order in which the bits are fixed. The PRS scheme of [GGG$^+$03] uses this insight and suggests that all neighbors corresponding to fixing a "1" in the distance be considered *candidates*, and that a node should forward a message to the candidate with the *lowest latency* from itself.

The above idea does not quite work when the number of nodes is not exactly 2^N. For example, let us say we wanted to get from node 0 to node 54 (distance 111000) just as earlier; say we want to take a hop of length 8 to fix the right 1 in the binary string. If there is no node with id exactly 8, the Chord system requires node 0 to maintain a link to the node with the smallest id that is greater than 8. Say, node 0 has a link to node 9. Taking this hop of length 9 would lead to a remaining distance is $56 - 9 = 47$, with a binary representation 101111, which leaves us five more 1s to fix in the distance. In general, the number of hops necessary to reach the destination may not be logarithmic.

One simple way to fix the problem is to consider a neighbor as a candidate for forwarding only if the remaining distance from that neighbor has fewer 1s than the current distance. If no

[4]Our GT-ITM model produces pairwise latencies that obey the triangle inequality. When latencies do not obey the triangle inequality, like in our experiments on PlanetLab, the average pairwise internet latency is not necessarily a lower bound.

such neighbors are available, we switch to the standard, greedy routing for the remainder of the route. This modified protocol uses at most $2 \log n$ hops to forward a message to a destination. However, such a protocol is too restrictive and does not offer enough choices in selecting paths; so, we relax the condition to also ignore l lower-order bits in the distance. For example, say node 0 needs to route to node 84(distance 1010100). The number of 1s in the distance is 3. If 0 forwards the message to node 17, the remaining distance would be 67(1000011) which has the same number of ones as the original distance. However, if we ignore the last $l = 3$ bits in the distances, the number of ones actually decreases from 2 to 1, and node 17 would be considered a candidate. Ignoring these low-order bits thus increases the number of routing options, and increases the maximum number of hops necessary only by a constant. We call this routing protocol *Solomon* (Selection Of LOcal neighbor from Multiple Overlay Neighbors). The number of bits to ignore, l, should ideally be set to $N - \log n + \log \log n$ where n is the number of nodes in the system and N is the number of bits in the identifiers.

B. Exploiting Bidirectionality: Duomon

Recall that both greedy routing and Solomon use links only in the clockwise(forward) direction. However, most P2P systems implement overlay links as TCP connections, which are inherently bidirectional. In consequence, links can also be used in the anti-clockwise(backward) direction to good effect. For example, if we need to send a message from node 1 to node 0, we could use the insight of Christopher Columbus and simply use the overlay link between 0 and 1 in the anti-clockwise direction. A trivial way to exploit this observation is to trade off bandwidth for latency by simultaneously routing two copies of a query, one clockwise and the other anti-clockwise, using the Solomon protocol[5]. The destination receives two copies of the query, one earlier than the other, and can throw away the duplicate copy. Note that the cost of transmitting the query is also doubled in the process. We call this simple protocol Duomon.

C. Using One-Lookahead: Ebola

Duomon is a simple means of exploiting bidirectional links. However, we can actually do much better, using just one copy of the query, by combining two powerful ideas: the use of both clockwise and anti-clockwise links along the same route, and the use of lookahead.

To illustrate the first idea, imagine we had to cover a distance of 7(111). Clockwise routing requires 3 steps of 4, 2 and 1, while anti-clockwise routing would require even more steps on a network with more than 64 nodes. Observe that we could actually cover this distance in *two* steps by combining a forward step of 8 with a backward step of 1. (On a "perfect" Chord network with 2^N nodes, we show that an extension of this idea leads to routing in just $\frac{1}{3} \log n + o(1)$ steps on average in [GM04].)

[5]The performance using anti-clockwise links is not identical to that using clockwise links because the in-degree distribution of Chord nodes is much more skewed than the out-degree distribution.

The second idea is to use one-lookahead, i.e., make a node aware not only of the quality of links to its neighbors, but also of the links from each neighbor n to n's neighbors. Thus, a node can decide which neighbor to forward a message to, by studying *two steps* of the routing path and evaluating its quality. The advantage of one-lookahead is that nodes can avoid being seduced by attractive-looking neighbors that may not offer a good second step in routing to the destination.

Alternatively, we can think of one-lookahead as an approximation of infinite lookahead; with infinite lookahead, each node can figure out and use the shortest overlay path to route to any destination. As has been observed in [MMP03], the use of one lookahead does not incur much overhead; the lookahead information can be piggybacked on other traffic, and is updated lazily at very low cost when nodes join and leave.

We combine lookahead with bidirectional use of overlay links to produce a routing protocol called *Ebola*(Exploiting Bidirectionality using One LookAhead). Define the *minHamming* distance to the destination as either the clockwise or anti-clockwise distance to the destination, whichever has fewer 1s in its binary representation. The key idea in Ebola is to consider a pair of steps, each of which may be clockwise or anti-clockwise, such that the resulting minHamming distance has fewer 1s than the original minHamming distance. Once all such pairs of steps have been identified, we use the pair that has the lowest total latency[6]. Just as in the case of Solomon routing, if no such pair of steps exists, we revert to greedy routing, along the clockwise or anti-clockwise direction whichever is shorter. Again as in the case of Solomon routing, we will ignore the l least significant bits in distances, while computing minHamming distance.

D. Evaluation

We now evaluate the improvement obtained in query latency by using each of these three new routing protocols on a static Chord network as Apocrypha optimizes it.

1) Evaluation on GT-ITM: We begin by considering a static 25,000-node Chord network with the physical network modeled using GT-ITM. Figure 13 plots the average path latency between all pairs of nodes (which, as mentioned earlier, is equal to the average query latency) for the three new routing protocols, as well for greedy clockwise routing. Note that we have normalized the average path latency on the y-axis by the average internet latency to obtain the *stretch*. So, a stretch of 2 implies that it takes twice as long, on average, to route from one node to another on the overlay network, compared to just directly routing between the nodes on the internet.

Note that the initial stretch, before Apocrypha performs any optimizations, is 7 for the standard greedy protocol (which is not too surprising, since greedy routing takes about 7 overlay hops on average, and each overlay hop should initially correspond to a random hop on the internet). On the other hand, Ebola has an initial stretch of under 4, Duomon has a stretch 4.7 and Solomon 5.3, thus demonstrating their utility even when

[6]We eliminate pairs of steps from node i that actually leads to a direct neighbor of i.

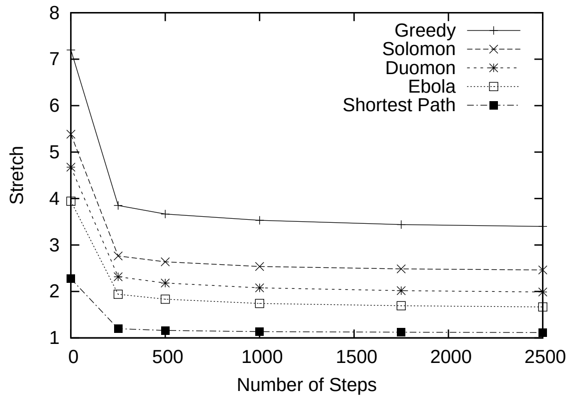

Fig. 13. Average Path Latency on Chord with different routing protocols

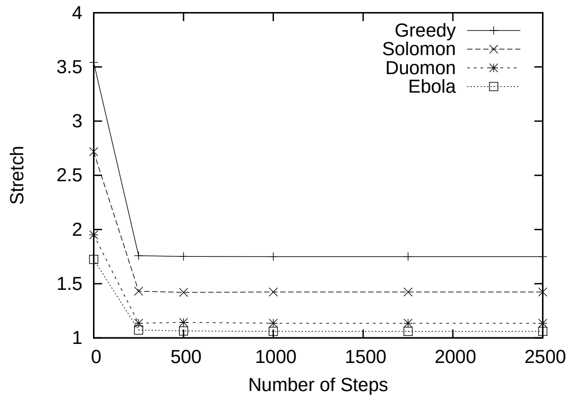

Fig. 15. Average Path Latency on Chord with different routing protocols

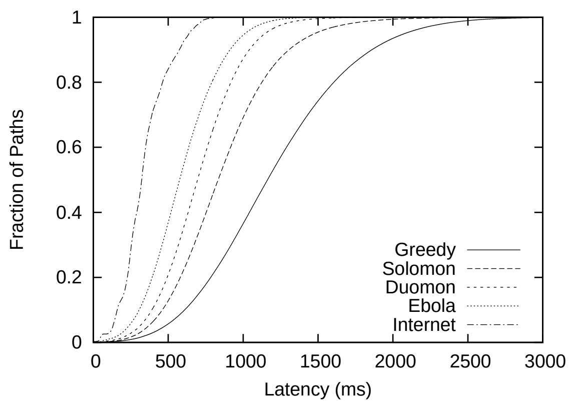

Fig. 14. Cdf of query latencies using different routing protocols

Apocrypha is not used to optimize the Chord network. When used in conjunction with Apocrypha, all of our routing protocols continue to perform considerably better than greedy routing. For example, the use of Ebola reduces the stretch to under 1.7 on the static Chord network.

Figure 14 shows the cumulative distributions of the path latencies when using the different routing protocols. We chose to compute path latencies when routing on the Chord network optimized for 1000 time steps, which corresponds to the quality of the network seen under typical rates of dynamism (as seen in Figure 9). The left-most curve represents the distribution of the latencies between pairs of nodes in the internet model. For greedy routing (bottom-most curve), we see that the percentage of paths with a latency under 1 second is only about 30%. On the other hand, for Ebola, more than 93% of the paths have a latency under 1 second. Greedy routing has a much higher variance than our three protocols, which is not surprising since greedy routing is deterministic and independent of the individual link characteristics.

2) Evaluation on PlanetLab: We now evaluate the routing protocols on Chord, this time on the PlanetLab data. As earlier, we construct a 182-node Chord network, with inter-node latencies being determined by ping times on PlanetLab. Figure 15 depicts the stretch for the different routing protocols as a function of the number of hill-climbing steps.

Greedy routing, as depicted in the top line, has a stretch of 3.5 on the unoptimized Chord network, which is the average number of hoprs required for routing on a 182-node Chord network. After 250 steps, the stretch drops to 1.75 and remains constant at that value. The use of Solomon reduces the initial stretch to 2.7, and the stretch on the optimized network falls below 1.5. With the use of Ebola, the initial stretch is 1.7, and drops to less than 1.06 after optimization, which is very close to the optimal achievable stretch.

Overall, we observe that the stretch incurred by the routing protocols is lower than in the experiments on the GT-ITM topology, again suggesting that it is more difficult to optimize a network in the GT-ITM model than on PlanetLab. We conjecture that the underlying reason for this phenomenon is that there are more nodes that are "close" to a given node on PlanetLab than in the GT-ITM model.

VII. Applying Apocrypha to Gnutella

In this section, we study the impact of Apocrypha on queries in the Gnutella network. Gnutella uses flooding with a TTL in order to find answers for a query. Unlike in Chord where there is exactly one node which provides the answer to a query, there may be multiple nodes in Gnutella each of which has some answers to a query. Thus, the metric of concern in Gnutella is the number of answers obtained for a query as a function of the time taken to obtain that many answers. Since different queries may have different numbers of answers available, and the answers may be distributed across nodes in different ways, we use a slightly simpler metric: the number of nodes a query is propagated to, as a function of the time since the query was initiated. Ideally, we would like the query to propagate to the most nodes in the least amount of time.

We simulated the Gnutella network on top of our GT-ITM network model. Since we expected bandwidth to be an issue in a flooding-based system such as Gnutella, we imposed bandwidth constraints on the physical tail-links connecting each of the nodes to a router in our network model. Low-degree nodes were assumed to be attached to the network via cable modems, higher-degree nodes through 10Mbps LANs and the highest-degree nodes via 100Mbps LANs. We also introduced limited buffering on the tail links, and each node dropped network

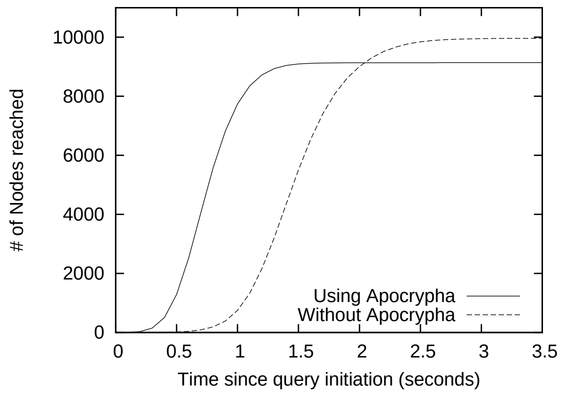

Fig. 16. Number of nodes reached in Gnutella as a function of time.

packets when the buffer overflowed. In our simulation, each node generated queries by a Poisson process with a rate of one query/minute. Queries were flooded using an initial TTL of 5. The CPU time taken to process a query is considered negligible compared to the time taken for transmitting messages over the overlay network.

We simulated both an unoptimized Gnutella network with nodes randomly located on the internet, and a Gnutella network optimized by Apocrypha. Since the Gnutella network converged very quickly with the use of Apocrypha, we show only one representative state of the Gnutella network rather than showing it at various stages of convergence.

Figure 16 shows the cumulative number of nodes reached by an individual query as a function of time. We see that, initially, the query reaches more nodes in less time on the optimized network compared to the unoptimized one. After one second, the query has reached about 9000 nodes on the optimized network comparted to fewer than 1000 nodes in the unoptimized case. Eventually, however, the use of Apocrypha *reduces* the total number of nodes reached (9000 with its use versus 10000 otherwise). One may initially believe that the optimized network may result in more load imbalance, hence causing more nodes to become bandwidth bottlenecks. Upon closer inspection, we found that the real reason for reaching fewer nodes is due to a phenomenon that we call *long circuits.* We now illustrate this phenomenon.

A. The Long-circuit phenomenon

The Gnutella protocol uses a TTL field to limit the broadcast of a query to a certain depth h. One would assume that all nodes within h hops would receive the query if there were no bottlenecks. Unfortunately, due to the way nodes throw away duplicate query messages, not all nodes within h hops may receive the query. We illustrate this problem in Figure 17, which shows a Gnutella-network fragment with numbers on the edges representing latencies. Say a query is initiated from node S with a TTL of 5. We observe that node T has two paths from S, one of length two and the other of length three. If the query from S reaches T via the 3-hop path first, as it does here, the query would be forwarded from T with a TTL of 2, which means that

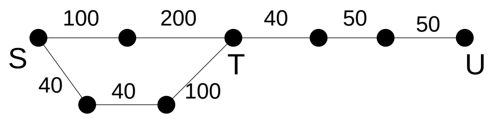

Fig. 17. An example of a long circuit

node U will not receive a copy of the query. When a duplicate copy of the query is received via the 2-hop path, it would simply be discarded. Thus, the query fails to reach U, despite U being within five hops of the source S. When there are two paths between S and T with the longer path being faster, we say S and T have a *long circuit*.

Long circuits occur naturally in Gnutella, even without the application of Apocrypha. However, the use of Apocrypha exacerbates the problem, because of its side-effect of increasing the variance in link latencies for a given node. After Apocrypha optimizes the overlay network, a typical node has five or six fast links and one or two slow links. Following two of these slow links may lead to a node that has a faster alternative route with more hops, thus creating a long circuit. In consequence, fewer nodes end up being reached by the query. The increase in the long-circuit phenomenon is the primary culprit in causing queries on the optimized network to reach fewer nodes.

There are different solutions possible for eliminating long circuits. We could (a) alter the protocol to retransmit a duplicate copy of the query on detecting a long ciruit, (b) *modify* the topology to eliminate long circuits, (c) alter Apocrypha to optimize a different objective function: the sum of the maximum latency links for each node, or (d) use new ad hoc topologies in combination with new search protocols to avoid long circuits. Space constraints prevent us from offering more details on these alternatives.

B. Other Ad Hoc Networks and Extensions

Supernode networks are very similar to Gnutella networks except that all the supernodes tend to have high bandwidth and more homogeneous “capacities”. So, we expect supernode networks to be more like random regular graphs than power-law graphs. Again, we omit our evaluations on such graphs due to space constraints. An alternative to Gnutella flooding is random-walk based searches on random regular graphs. In this case, the application metric of minimizing query latency is exactly the same as our hill-climbing metric of minimizing the average overlay link latency.

So far, we have assumed that all nodes have the same capabilities and that any pair of nodes can swap their set of neighbors, irrespective of the relative sizes of these neighbor sets. In reality, some nodes are capable of maintaining connections with a large set of neighbors, while others operating via a cable modem or a DSL line might not be able to support too many neighbors. It is easy to modify Apocrypha to take these simple *capacity constraints* into account. Recall that distributed Apocrypha requires two nodes to agree on whether to swap their labels or not. We could simply let a node refuse to swap labels if doing so leads to it having an unreasonably high degree. Again, we omit details.

VIII. Related Work

There have been different approaches adopted for making P2P overlays network-aware. For deterministic topologies, [RFH$^+$01] suggested carefully selecting labels in CAN to reflect geographical locations of nodes. However, this approach may be problematic because many properties of CAN rely on the set of labels being randomly chosen. Reference [RHKS02] suggests using improved routing protocols specific to CAN; we use the same general principle of exploiting multiple paths in designing Solomon. Reference [GGG$^+$03] also suggests a routing scheme similar to Solomon as discussed earlier; their routing scheme appears to be for "regular" networks on 2^N nodes.

For nondeterministic topologies, both Pastry [RD01], [CDHR02] and Tapestry [ZKJ01] exploit the nondeterminism inherent in their topology to allow nodes to choose physically close neighbors to connect to. Nondeterministic variants of Chord, using ideas similar to Pastry and Tapestry, have been proposed in [DKK$^+$01], [GGG$^+$03], [ZGG03]. The quality of the resulting overlay networks appear comparable to the overlay network quality we achieve on deterministic networks. However, the techniques used for producing the non-deterministic overlay are a function of the exact overlay network being constructed, and are not directly applicable to optimizing deterministic or randomized topologies. Also, a price to be paid for nondeterminism is a weakening of the guarantees on structural balance and other topological properties that may be offered with random selection of neighbors.

There has been relatively little work on building network-aware ad hoc networks, except in the context of overlay-multicast protocols [CMB00], [RHKS02]. These works use heavyweight protocols designed for use among a small number of nodes. Moreover, these protocols do not provide any guarantees on the topological properties of the resulting overlay network. Gia [CRB$^+$03] is a Gnutella-like system that performs some topology adaptation by breaking and forming links in order to enable high-capacity nodes to have high degree. The goals of topology adaptation in Gia are different from ours and, again, does not offer guarantees on topological properties like Apocrypha does.

The idea of using the swap to perform isomorphic transformations is fairly common. Reference [SCK$^+$03] uses the swap in order to construct multicast trees. The metric being optimized in this work is different from ours, as it is designed to optimize a tree rather than an arbitrary graph. Moreover, their proposed algorithms are designed for small-scale, static networks, rather than a highly dynamic P2P system. A significant difference between these algorithms and Apocrypha is the quenching policies used by Apocrypha to enable efficient distributed convergence even under dynamic conditions.

IX. Conclusions

We have presented Apocrypha, a generic mechanism to make P2P overlays network-aware. We showed how this mechanism may be applied to different P2P systems, and evaluated it both using graph-theoretic and system-specific metrics. We offered new, improved routing protocols for Chord and evaluated their effectiveness both while using Apocrypha and in isolation. We also introduced the long-circuit problem in Gnutella.

References

[CDHR02] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks, 2002.

[CDK$^+$03] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical distributed network coordinates. In *HotNets Workshop*, 2003.

[Cli] Clip2.com. Clip2 gnutella crawl files. Private collection.

[CMB00] Y. Chawathe, S. McCanne, and E. Brewer. An architecture for internet content distribution as an infrastructure service, 2000.

[CRB$^+$03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Gia: Making gnutella-like p2p systems scalable. In *Proceedings ACM SIGCOMM*, 2003.

[DKK$^+$01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, pages 202–215, 2001.

[Ein38] Albert Einstein. On the effects of external sensory input on time dilation. *Journal of Exothermic Science and Technology*, 1938.

[FJP$^+$99] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An architecture for a global internet host distance estimation service. In *Proc. IEE INFOCOM*, 1999.

[GGG$^+$03] K. Gummadi, R. Gummadi, S. Gribble, S.Ratnasamy, S.Shenker, and I.Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM*, 2003.

[GM04] P. Ganesan and G. S. Manku. Optimal routing in Chord. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[gnu] Gnutella. Website http://gnutella.wego.com.

[GSGM03] P. Ganesan, Q. Sun, and H. Garcia-Molina. Apocrypha: Making p2p overlays network-aware. Technical report, Stanford University, 2003. Available upon request.

[kaz] Kazaa. Website http://www.kazaa.com.

[lim] Limewire. Website http://www.limewire.com.

[MMP03] G.S. Manku, M.Bawa, and P.Raghavan. Symphony: Distributed hashing in a small world. In *Proc. USITS*, 2003.

[plaa] Planetlab. Website http://www.planet-lab.org.

[plab] PlanetLab All-Pairs-Pings. http://pdos.lcs.mit.edu/∼strib/pl_app/.

[PRU01] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter peer-to-peer networks. In *Proc. FOCS*, 2001.

[RD01] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP '01*, 2001.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 149–160, 2001.

[RHKS02] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. IEEE INFOCOM'02*, 2002.

[RKCD01] A. I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. 3rd Intl. Networked Group Communication Workshop*, 2001.

[SCK$^+$03] S.Banerjee, C.Kommareddy, K.Kar, B.Bhattacharjee, and S.Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *Proceedings IEEE INFOCOM*, 2003.

[SMK$^+$01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.

[XTZ03] Zhichen Xu, Chunqiang Tang, and Zhen Zhang. Building topology-aware overlays using global soft-state. In *Proc. ICDCS'03*, 2003.

[ZCB96] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, March 1996. IEEE.

[ZGG03] Hui Zhang, Ashish Goel, and Ramesh Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proceedings ACM SIGMETRICS*, 2003.

[ZKJ01] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, 2001.