Cooperative Peer Groups in NICE*

Rob Sherwood Seungjoon Lee Bobby Bhattacharjee
Department of Computer Science, University of Maryland, College Park, Maryland
{*capveg, slee, bobby*}*@cs.umd.edu.*

Abstract

We present a distributed scheme for trust inference in peer-to-peer networks. Our work is in the context of the NICE system, which is a platform for implementing cooperative applications over the Internet. We describe a technique for efficiently storing user reputation information in a completely decentralized manner, and show how this information can be used to efficiently identify non-cooperative users in NICE. We present a simulation-based study of our algorithms, in which we show our scheme scales to thousands of users using modest amounts of storage, processing, and bandwidth at any individual node. Lastly, we show that our scheme is robust and can form cooperative groups in systems where the vast majority of users are malicious.

Keywords: Reputation Base Trust, P2P, Distributed Algorithms

1 Introduction

NICE[1] is a platform for implementing *cooperative* applications over the Internet. We define a cooperative application as one that allocates a subset of its resources, typically processing, bandwidth, and storage, for use by other peers in the application. We believe a large class of applications, including on-line media streaming applications, multi-party conferencing applications, and emerging peer-to-peer applications, can all significantly benefit from a cooperative infrastructure. However, cooperative systems perform best if all users do, in fact, cooperate and provide their share of resources to the system. In this paper, we present techniques for identifying cooperative and non-cooperative users. Using our schemes, individual users can assign and infer "trust" values for other users. The inferred trust values represent how likely a user considers other users to be cooperative, and are used to price resources in the NICE system.

We focus on distributed solutions for the trust inference problem. We decompose the distributed trust inference problem into two parts: a local trust inference component that requires trust information between principals in the system as input and a distributed search component that efficiently gathers this individual trust information to be used as input for local inference algorithms. There already exist systems, e.g., e-bay[21], that have a centralized user-evaluation system. Other resource bartering systems, e.g., MojoNation[22], have also implemented centralized trust inference solutions. Our goal is to enable open applications where users do not have to register with an authority to be a part of the system. Centralized solutions do not scale in open systems, since malicious users can overwhelm the central "trust" server with spurious transactions. The most widely used decentralized trust inference scheme is probably the PGP web of trust [25], which allows one level of inference. We present a new decentralized trust inference scheme that can be used to infer across arbitrary levels of trust. There is no trusted-third-party or centralized repository of trust information in our scheme. Users in our system only store information they explicitly can use for their own benefit. We show that our algorithms scale well even with limited amount of storage at each node, and can be used to efficiently implement large distributed applications without involving explicit authorities. Further, our solutions allow individual users to compute local trust values for other users using their own inference algorithm of choice, and thus can be used to implement a variety of different policies.

1.1 Cooperative Systems

The notion of a cooperative system is not unique in networking; in fact, packet forwarding in the Internet is a cooperative venture that utilizes shared resources at routers. Our overall goal in NICE is to extend this notion to include end-applications and provide an incentive-based framework for implementing large distributed ap-

*This work is supported by a NSF CAREER award (ANI0092806).

[1]NICE is a recursive acronym for "NICE is the Internet Cooperative Environment" (See http://www.cs.umd.edu/projects/nice).

plications in a cooperative manner. Clearly, an immense amount of distributed resources can be harvested over the Internet in a cooperative manner. This observation is key in the recent surge of peer-to-peer (p2p) applications, and we believe the next generation of such p2p applications will be based upon the notions of cooperative distributed resource sharing.

A number of interesting distributed algorithms for p2p systems, most notably in the area of distributed resource location, have recently been introduced. All of these schemes, however, assume that all peers in the system implicitly cooperate and implement the underlying protocols perfectly, even though it may not be explicitly beneficial to do so. Consider the following examples:

- In Gnutella [11], peers forward queries flooded on behalf of other users in the system. Each forwarded message consumes bandwidth and processing at each node it visits.

- In Chord [20], a document is "mapped" to a particular node using a hash function. Thus, a peer serves a document that is, in fact, owned by some other node in the system. Thus, peers in the system expend their own resources to serve documents for other nodes in the system. This situation is not unique to Chord; all hash-based location systems, including CAN [16], Bayeux [24], Pastry [18], have this property. It is possible to build a system in which nodes only serve a pointer to the document data and also to implement various load balancing schemes; however, even in the best load-balanced system, there can be temporary overloads when a large amount of local resources are expended due to external serving.

- A number of relay-based streaming media protocols have been developed and demonstrated. In these protocols, nodes devote resources such as access bandwidth for serving their child nodes.

In each example above, any individual user may choose *not* to devote local resources to external requests, and still get full benefit from the system. On the other hand, the integrity and correct functioning of the system depends on each user implementing the entire distributed protocol correctly and selflessly. However, experience with deployed systems, such as Gnutella and previously Napster, show that only a small subset of peers offer such selfless service to the community, while the vast majority of users use the services offered by this generous minority [3]. The goal of this work is to efficiently locate the generous minority, and form a clique of users all of whom offer local services to the community.

1.2 Model

In this paper, we assume that a (p2p) system can be decomposed into a set of of two-party transactions. A single transaction can be a relatively light-weight operation such as forwarding a Gnutella query or a potentially resource intensive operation such as hosting a Chord document. Next, we assume that the system consists of a set of "good" nodes that always implement the underlying protocols correctly and entirely, i.e. good users always fulfill their end of a transaction. The goal of our work is develop algorithms that will allow "good" users to identify other "good" users, and thus, enable *robust* cooperative groups. These are peer groups in which, with high probability, each participant successfully completes their end of each transaction. Specifically, we propose a family of distributed algorithms which can be used by users to calculate a per-user "trust" value. The trust value for node B at a node A is a measure of how likely node A believes a transaction with node B will be successful. In our system, users store a limited amount of information about how much other users trust them, and we present algorithms for choosing what information to store and how to retrieve this trust information. Once relevant information has been gathered, individual users may use different local inference algorithms to compute trust values.

It is important to note that we assume that good nodes are able to ascertain when a transaction is successful. Clearly, in many cases, it is not possible to efficiently determine whether a transaction fails (e.g. when a node sometimes does *not* serve Chord documents that it hosts). It is even more difficult to determine whether a transaction fails because of a system failure or because of non-cooperative users. For example, consider the case when all users are cooperative but a document cannot be served due to a network failure. We believe this problem is inherent in any trust-inference system that is based on transaction "quality". We discuss different policies for assigning values to transaction quality in Section 4.

The overall goal of this work is to identify cooperative users. An ideal trust inference system would, in one pass, be able to classify all users into cooperative or non-cooperative classes with no errors. However, this is not possible in practice because non-cooperative users may start out as cooperative users. The specific goals of our work are as follows:

- Let the "good" nodes find each other quickly and efficiently: Good nodes should be able to locate other good nodes without losing a large amount of resources interacting with malicious nodes. This will allow NICE to rapidly form robust cooperative

groups.

- Malicious nodes and cliques should not be able to break up cooperating groups by spreading misinformation to good nodes. Specifically, we want to develop protocols in which malicious nodes are rapidly pruned out of cooperative groups. Further, we assume malicious nodes can disseminate arbitrary trust information, and the cliques formed of good nodes should be robust against this form of attack.

In Section 4, we describe algorithms that achieve our goals with low run-time overhead, both in terms of processing and network bandwidth usage. We believe this algorithm is the first practical, robust, trust inference scheme that can be used to implement large cooperative applications.

The rest of this paper is structured as follows: in the next section, we discuss prior work in distributed trust computations. In Section 3, we present an overview of the NICE system, and describe how distributed trust computations are used by NICE nodes. We describe our algorithms and local node policies in Section 4, present simulation results of the trust search in Section 5, and simulated results of a NICE resource trading system in Section 6. We discuss our conclusions in Section 7.

2 Related Work

In this section, we discuss prior work in trust inference and present a brief overview of systems that are based on notions of trust and incentive.

The concept of "trust" in distributed systems is formalized in [14] using social properties of trust. This work considers an agent's own experience to obtain [-1, 1]-valued trust, but does not infer trust across agents. Abdul-Rahman et al. [1] describe a trust model that deals with direct experience and reputational information. This model can be used, as is, in NICE to infer trust. Yu et al. [23] propose a way to compute a real-valued trust in [-1, 1] range from direct interactions with other agents. A product of trust values is used for reputation computation, and undesirable agents are avoided by having an observer of bad transactions disseminate information about the bad agent throughout the network. This work is primarily about using social mechanisms for regulating users in electronic communities, and the techniques developed here can be used in NICE. In this paper, we focus on algorithms for efficiently storing and locating trust information.

Another scheme [2] focuses on management and retrieval of trust-related data, and uses a single p2p distributed database which stores complaints about individuals if transactions with them are not satisfactory. When an agent p wants to evaluate trust for another agent q, it sends a query for complaint data which involves q, and decides q's trustworthiness with returned data, using a proposed formula. However, this system implicitly assumes that all participants are equally willing to share the communal data load, which may not be true in many p2p systems [3]. Such a system is also vulnerable to DoS attacks, as there is no preventative measure from inserting arbitrary amounts of complaints into the system.

PGP [25] is another distributed trust model that focuses on proving the identity of key holders. PGP uses user defined thresholds to decide whether a given key is trusted or not, and different introducers can be trusted at finite set of different trust levels. Unlike NICE, trust in PGP is only followed through one level of indirection; i.e. if A is trying to decide the trust of B, there can be at most one person, C, in the trust path between A and B. There are also a number of popular web sites, e.g. e-bay and Advogato (see www.advogato.org) that use trust models to serve their users. However, all data for these sites is stored at a trusted centralized database, which may not be ideal for open systems, and lead to the usual issues of scalability and single point of failure.

The Eigentrust [12] system focuses on taking pairwise trust values, i.e. the trust $c_{i,j}$ for all pairs i, j, and attempts to calculate a single global trust value for each principal. It accomplishes this task by computing in a distributed manner the principal eigenvector of the entire pair-wise trust matrix. In the current Eigentrust system, when node i and node j have not interacted, $c_{i,j}$ is assumed to be zero. The protocol presented in this paper is complementary to Eigentrust, and can be used to infer unknown $c_{i,j}$ values.

A system similar to NICE is Samsara[13]. While NICE attempts to solve distributed resource allocation problems in resource neutral manner, Samsara focuses strictly on remote file storage. In Samsara, nodes exchange chunks of objects, and query each other to verify that nodes are correctly storing the objects they claim to. If a particular node fails a query, other nodes in the system begin to probabilistically drop the node's objects. The probability of a given object being dropped increases as the number of consecutive failed queries increases. The argument is made that it is not possible with current bandwidth limitations for a malicious node to replace dropped objects fast enough to counteract the dropping rate. However, it is not clear how Samsara's bandwidth restriction defense fairs against a more sophisticated attacker that employs forward error correction (FEC) to entangle multiple objects. Samsara does not have a notion of trust inference.

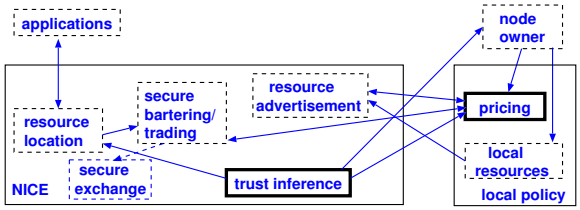

Figure 1: NICE component architecture: the arrows show information flow in the system; each NICE component also communicates with peers on different nodes. In this paper, we describe the trust inference and pricing components of NICE.

3 Overview of NICE

In this section, we present a brief overview of the NICE platform. Our goal is to provide context for the distributed trust computation algorithms presented in Section 4. NICE is a platform for implementing cooperative distributed applications. Applications in NICE gain access to remote resources by bartering local resources. Transactions in NICE consist of secure exchanges of resource *certificates*. These certificates can be redeemed for the named (remote) resources. Non-cooperative users may gain "free" access to remote resources by issuing certificates that they do not redeem.

NICE provides a service API to end-applications, and is layered between the transport and application protocols. The NICE component architecture is presented in Figure 1, with this papers contributions in bold. Applications interact with NICE using the NICE API, and issue calls to find appropriate resources. All of the bartering, trading, and redeeming protocols are implemented within NICE and are not exposed to the application. These sub-protocols share information within themselves and are controlled by the user using per-node policies. NICE peers are arranged into a signaling topology using our application-layer multicast protocol [4]. All NICE protocol-specific messages are sent using direct unicast or are multicast over this signaling topology. The exact details of the remaining sub-protocols, e.g., bartering, resource location, secure exchange, and the NICE API itself remain the subject of future work. For the purpose of building and inferring trust, we treat resources and transactions as abstractions throughout the paper.

3.1 NICE Users and Pricing Policies

Until now, we have used the terms user and node in an imprecise manner. In NICE, each user generates a PGP style [25] identifier which includes a plaintext identification string and a public key. The key associated with a NICE identifier is used for signing resource certificates, for trading resources, and for assigning trust (Section 4) values.

It is important to note that neither the NICE identifier nor the associated key needs to be registered at any central authority; thus, even though NICE uses public keys, we do not require any form of a global PKI. Thus, NICE can be used to implement open p2p applications without any centralized authority. Since there is no central registration authority in NICE, a single user can generate an arbitrary number of keys and personas. However, in NICE, pricing is coupled with identity, i.e., *new users have to pay more for services* until they establish trust in the system. Thus, it is advantageous for nodes to maintain a single key per user and not to change keys frequently. This property makes NICE applications robust against a number of cheap identity based denial-of-service attacks that are possible on other p2p systems[10].

The goal of the default policies in NICE is to limit the resources that can be consumed by cliques of malicious users. These policies work in conjunction with the trust computation which is used to identify the misbehaving nodes. In practice, NICE users may use any particular policy, and may even try to maximize the amount of resources they gain by trading their own resources. The primary goal of the default policies is to allow good users to efficiently form cooperating groups, and not lose large amounts of resources to malicious users. The pricing and trading policies are used to guard against users who issue spurious resource certificates using multiple NICE identities. We use two mechanisms to protect the integrity of the group:

- **Trust-based pricing**

  In trust-based pricing, resources are priced proportional to mutually perceived trust. Assume trust values range between 0 and 1, and consider the first transaction between Alice and Bob where the inferred trust value from Alice to Bob is $T_{Alice}(Bob) = 0.5$, and $T_{Bob}(Alice) = 1.0$. Under trust-based pricing, Alice will only barter with Bob if Bob offers significantly more resources than he gets back in return. Note however that as Bob conducts more successful transactions with Alice, the cost disparity will decrease. This policy is motivated by the observation that as Alice trades with a principal with lower trust she incurs a greater risk of not receiving services in return, which, in turn, is reflected in the pricing.

- **Trust-based trading limits**

In these policies, instead of varying the price of the resource, the policy varies the *amount* of the resource that Alice trades. For example, in an scenario with Alice and Bob, Alice may trade some small, trivial amount of resources with Bob initially, but once Bob has proven himself trustworthy, Alice increases the amount of resources traded. This policy assures that when trading with a principal with relatively low trust, Alice bounds the amount of resources she can lose. The simulated results presented in Section 6 use trust based trading limits.

4 Distributed Trust Computation

We assume that for each exchange of resources, i.e., a *transaction*, in the system, each involved user produces a signed statement (called a *cookie*) about the quality of the transaction. For example, consider a successful transaction t between users Alice and Bob in which Alice consumes a set of resources from Bob. After the transaction completes, Alice signs a cookie c stating that she had successfully completed the transaction t with Bob. Bob may choose to store this cookie c signed by Alice, which he can later use to prove his trustworthiness to other users, including Alice [2]. As the system progresses, each transaction creates new cookies which are stored by different users. Clearly, cookies have to be expired or otherwise discarded; the algorithms we present later in this section require constant storage space.

We will describe the trust inference algorithms in terms of a directed, weighted graph T called the trust graph. The vertices in T correspond exactly to the users in the system. There is an edge directed from Alice to Bob if and only if Bob holds a cookie from Alice. The value of the Alice→Bob edge denotes how much Alice trusts Bob and depends on the set of Alice's cookies Bob holds. Note that each transaction in the system can either add a new directed edge in the trust graph, or relabel the value of an existing edge with its new trust value.

Assume that a current version of the trust graph T is available to Alice, and suppose Alice wishes to compute a trust value for Bob. If Alice and Bob have had prior transactions, then Alice can just look up the value of Alice→Bob edge in T. However, suppose Alice and Bob have never had a prior transaction. Alice could potentially *infer* a trust value for Bob by following directed paths (ending at Bob) on the trust graph as we describe below.

[2]It is also possible for Alice to keep a record of this transaction instead of Bob. In this alternate model of trust information storage, users themselves store information about whom they trust, and can locally compute the trust of the remote nodes they know of. This model, however, is susceptible to a denial of service attack that we describe later in this section.

4.1 Inferring Trust on the Trust Graph

Consider a directed path $A_0 \rightarrow A_1 \rightarrow \ldots \rightarrow A_k$ on T. Each successive pair of users have had direct transactions with each other, and the edge values are a measure of how much A_i trusts A_{i+1}. Given such a path, A_0 could infer a number of plausible trust values for A_k, including the minimum value of any edge on the path or the product of the trust values along the path; we call these inferred trust values the *strength* of the $A_0 \rightarrow A_k$ path. The inference problem is somewhat more difficult than computing strengths of trust paths since there can be multiple paths between two nodes, and these paths may share vertices or edges. Centralized trust inference is not the focus of our work, but it is important to use a robust inference algorithm. We have experimented with different inference schemes, and we describe two simple but robust schemes. In the following description, we assume A (Alice) has access to the trust graph, and wants to infer a trust value for B (Bob):

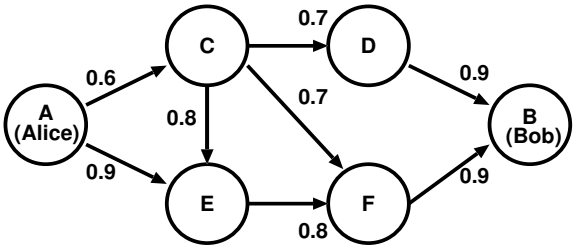

Figure 2: Example trust graph: the directed edges represent how much the source of edge trusts the sink.

- **Strongest path:** Given a set of paths between Alice and Bob, Alice chooses the *strongest* path, and uses the minimum trust value on the path as the trust value for Bob. The strength of a path can be computed as the minimum valued edge along the path or the product of all edges along a path. Given the trust graph, this trust metric can easily be computed using depth-first search. In the example shown in Figure 2, we use the min. function to compute the strength of a path. In this example, the strongest path is $AEFB$, and Alice infers a trust level of 0.8 for Bob.
- **Weighted average of strongest disjoint paths:** Instead of choosing only the strongest path, Alice could choose to use contributions from all disjoint

paths. The set of disjoint paths is not unique, but the set of strongest disjoint paths (modulo equi-strength paths) is and can be computed using network flows with flow restrictions on vertices. Given the set of disjoint paths, Alice can compute a trust value for Bob by computing the weighted average of the strength of all of the strongest disjoint paths. The weight assigned to the Alice→X→ ... →Bob path is the value of the Alice→X edge (which represents how much Alice directly trusts X). In the example in Figure 2, $ACDB$ is the other disjoint path (with strength 0.6), and the inferred trust value from Alice to Bob is 0.72.

Both these algorithms are robust in the sense that no edge value is used more than once and trust values computed are always upper-bounded by the minimum trust on a path. Before any of these local algorithms can be used, the trust graph has to be realized in a scalable manner, and (edge) values have to be assigned to cookies.

Also note that the trust graph is not necessarily connected. For example, if Alice is new to the system, then there exists no path from her to Bob in the graph. In this case, we must have some application specific policy to assign a *default* level of trust. This policy is highly application specific, for example the level of trust for an unknown node in a file sharing application should be higher than the default trust in a medical professional referral service. Additionally, if we assume that each user maintains more than $\ln N$ cookies, where N is the number of users in the system and that the trust graph approximates an r-regular random graph[7], then with high probability the random graph remains connected. While a random graph is not necessarily a good model for the trust graph itself, we use it to motivate our simulation parameter selections, and show in Section 5 that it performs well.

Note that in order to infer trust for Bob, Alice does not need to access the entire trust graph, but only needs the set of paths from her to Bob. In the rest of this section, we describe schemes to store the trust graph and to produce sets of paths between users in a completely decentralized manner over an untrusted infrastructure. We begin with a discussion of different techniques for assigning cookie values, and describe our distributed path discovery protocol in Section 4.3.

4.2 Assigning Values to Cookies

Ideally, after each transaction, it would be possible to assign a real number in the [0,1] real-valued interval to the quality of a transaction and assign this as the cookie value. In some cases, transactions can be structured such that this indeed is possible: e.g. assume that Alice transcodes and serves a 400Kbps video stream to Bob at 128Kbps, and according to a prior agreement, Bob signs over a cookie of value 0.75 to Alice. The same transaction may have resulted in a cookie of value 0.9 if Alice had been able to serve the stream at 256Kbps. In many cases, however, it is not clear how to assign real-valued quality metrics to transactions. For example, in the previous example, Alice could claim that she did serve the stream at 256Kbps, while network congestion on Bob's access link caused the eventual degradation of the quality to 128Kbps. It is, in fact, easy to construct cases when it is not easily feasible to check the quality of service. In most cases, however, we believe it is somewhat easier to assign a $\{0,1\}$ value to a transaction, i.e. either the transaction was successful, or it was not. As applied to the previous example, Bob and Alice could negotiate a threshold rate (say 64Kbps) at which point he considers the entire transaction successful, and assigns a 1-valued cookie to Alice, regardless of whether the data was delivered at 64.5Kbps or 400Kbps. Further, for many transactions, such as streaming media delivery, it is possible for one party to abort the transaction if the initial service quality is not beyond the 0-value threshold.

In the rest of this paper, we assume that cookies are assigned values on the [0,1] interval. However, it is possible to assign arbitrary labels to cookies, and to conduct arbitrary policy-based searches as long as the requisite state is kept at each user. For example, it is possible to construct a system where cookies take one of four values (e.g., "Excellent", "Good", "Fair", and "Poor"), and users search for "Excellent"-valued cookies that are less than one week old. All of the NICE path enumeration and inference schemes work correctly as long as cookies have a comparable value, regardless of how users assign these values, and what range these values take.

One issue is who decides cookie values: by direct user intervention or by program. There is a trade-off in the two approaches between the latency in assigning values versus the accuracy of values. Cookies values that are assigned automatically, e.g., by heuristic or rule, do not have to wait for an interactive user before being posted to the trust network. However, in many cases it is possible to trick a heuristic into giving good cookie values for marginal service. For example, Alice offers Bob 10GB of disk space for one hour in exchange for one hour of computing time on Bob's machine. Bob takes the disk and allows Alice to use his computer, but runs additional jobs on his computer such that Alice's job is slowed. An automated cookie valuation might give a high valued cookie for this transaction, but a human valuation would catch

Bob's cheating, and return a low valued cookie.

4.3 Distributed Trust Inference: Basic Algorithm

In this section, we describe how users locate trust information about other users in our system. This distributed algorithm proceeds as follows: each user stores a set of signed cookies that it receives as a result of previous transactions. Suppose Alice wants to use some resources at Bob's node. There are two possibilities: either Alice already has cookies from Bob, or Alice and Bob have not had any transactions yet.[3] In the case Alice already has cookies from Bob, she presents these to Bob. Bob can verify that these indeed are his cookies since he has signed them. Given the cookies, Bob can now compute a trust value for Alice.

The more interesting case is when Alice has no cookies from Bob. In this case, Alice initiates a search for Bob's cookies at nodes from whom she holds cookies. Suppose Alice has a cookie from Carol, and Carol has a cookie from Bob. Carol gives Alice a copy of her cookie from Bob, and Alice then presents two cookies to Bob: one from Bob to Carol, and one from Carol to Alice. Thus, in effect, Alice tells Bob, "You don't know me, but you trust Carol and she trusts me!" In general, Alice can construct multiple such "cookie paths" by *recursively* searching through her neighbors. In effect, Alice floods queries for Bob's cookies along the cookie edges that terminate at each node, starting with her own node. After the search is over, she can present Bob with an union of directed paths which all start at Bob and end at Alice. Note that these cookie paths correspond exactly to the union of directed edges on the trust graph which we used for centralized trust inference. Thus, given this set of cookies, Bob can use any centralized scheme to infer a trust value for Alice.

This basic scheme has several desirable properties:

- If Alice wants to use resources at Bob, *she* has to search for Bob's cookies. This is in contrast with the analogous scheme in which nodes themselves keep records of their previous transactions. Under such a setting, if Bob did not know Alice, *he* would have to initiate a search for Alice through nodes he trusted. A malicious user Eve could mount an easy denial-of-service attack by continuously asking other nodes to search for Eve's credentials. In our system, nodes forward queries on behalf of other nodes only if they have assigned them a cookie, and thus, implicitly trust them to a certain extent.
- Alice stores cookies which are statements of the form "X trusts Alice". Thus, *Alice only devotes storage to items that she can use explicitly for her own benefit*, and thus, there is a built-in incentive in the system to store cookies. In fact, if Bob assigns a low-value cookie for Alice, she can discard this cookie since this is, in effect, a statement that says Bob does *not* trust Alice. In general, users store the cookies most beneficial to their own cause, and do not forward messages on behalf of users they do not trust.
- The transaction record storage in the system is completely distributed, and if two nodes conduct a large number of spurious transactions, only they may choose to hold on to the resultant state. In contrast, in a centralized transaction store, these nodes could easily mount a denial-of-service attack by overwhelming the transaction store with spurious transaction records.

Note that a malicious node may choose to drop or corrupt trust queries sent to it. However, as the network evolves, trust lookup queries are sent primarily to nodes that are *already* trusted, so dropping a trust lookup query is less common. A possible modification to the protocol is to change the trust lookups from a recursive process to an iterative process. Specifically, Alice queries Cathy for any of Bob's cookies, Cathy returns a cookie for Doug, which Alice then uses to query Doug for Bob's cookies, iteratively. However, this iterative protocol has higher network overhead, so we do not consider it further.

4.4 Refinements

While the flooding-based scheme we have described is guaranteed to find all paths between users and has other desirable properties, it is not a complete solution. Flooding queries is an inefficient usage of distributed resources, and as pointed out before, malicious nodes can erase all information of their misdeeds simply by throwing away any low valued cookies they receive. We next describe three refinements to the basic scheme that address these issues.

4.4.1 Efficient Searching

The recursive flooding procedure described above does find all cookies that exist for a given principal. However, it is extremely inefficient, since it visits an exponentially growing number of nodes at each level. Further, unless

[3] There is yet a third possibility in which Alice has discarded cookies from Bob, but we assume that this case is equivalent to Alice having no cookies from Bob

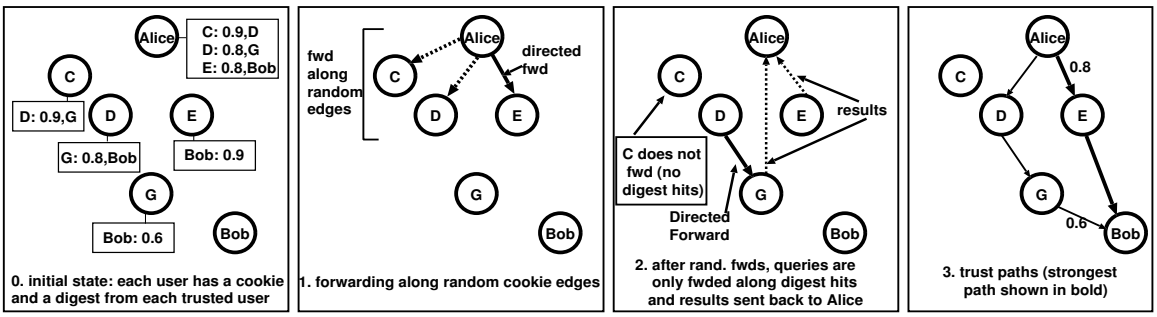

Figure 3: Different stages in the operation of the Alice→Bob search protocol. Edges in this figure represent message flow. It is important to note that corresponding edges in the trust graph point in the *opposite* direction.

the flooding is somehow curtailed, e.g., by using duplicate suppression or by using a time-to-live field in queries, some searches may circulate in the system forever.

It is obvious to consider using a peer-to-peer search structure, such as a distributed hash table (DHT) [20, 18], to locate cookies. However, this is not possible since in NICE because we do not assume the existence of anything more sophisticated than plain unicast forwarding. NICE is the base platform over which other protocols, such as Chord, can be implemented. The NICE protocols are much like routing protocols on the Internet: they cannot assume the existence of routing tables etc., and must be robust against packet loss and in the case of NICE, against malicious nodes. Thus, we must employ other mechanisms to make the cookie searches more efficient.

Instead of flooding to all neighbors in the trust graph, nodes forward queries to a random subset of their neighbors (typically of size 5). However, the resulting search still increases exponentially at each hop, only with a smaller base! Thus, additionally we add the following extension to our base protocol: whenever node receives a cookie from some other node, it also receives a *digest* of all other cookies at the remote node. Since, in our implementations, the number of cookies at each node is quite small (typically around 40 for a 2048 node system), this digest can be encoded using around less than 1000 *bits* in a Bloom filter[4] [6]. Thus, the storage space required for the digests are trivial (around 128 bytes), but they allow us to *direct* the search for specific cookies with very high precision. The idea of using digests for searches has been used previously, e.g., in lookaround caching [5] and summary caches [9]. It is, in fact, a base case of probabilistic search using attenuated Bloom filters [17]; in our experiments, we found that we did not need to use full attenuated Bloom filters — only one level of filters was sufficient. Lastly, each node also keeps a digest of recently executed searches and uses this digest to suppress duplicate queries.

In our implementation, when choosing nodes to forward to, we always choose nodes whose digests indicate they have the cookie for which we are searching. However, it is possible that there are no hits in any digest at a node; in this case, we once again choose nodes to forward to uniformly at random. However, we only forward to randomly chosen nodes if the query is within a predetermined number of hops away from the query source. Thus, in the final version of the search, a query spreads from the source, possibly choosing nodes at random, but the flooding is quickly stopped unless there is a hit in a next-hop digest.

Note that for most applications, the cost of keeping the digests fresh will be small. With respect to Alice, digests are only kept at nodes that hold a cookie from her. Thus, when Alice updates her local cookie cache, she need only contact nodes that are one hop away from her on the trust graph. Rather than force Alice to maintain a list of nodes who hold her cookies, nodes could periodically contact Alice to refresh their digests.

Example Before we describe other extensions to the base protocol, we illustrate the digest-based search procedure with an example (corresponding to Figure 3). Alice wants to use resources Bob has, but does not have a cookie from him. She initiates a search for a cookie path to Bob. In Figure 3-0, we show the initial state of cookies and digests at each user, e.g., Alice has a cookie of value 0.9 from C, and her digest from C shows that C has a cookie from D. For this example, we assume the search out-degree is 3, and the random flooding hop limit is 1. Alice first sends a query not only to nodes with a digest

[4]Such a filter, with only eight hash functions, would have a false positive rate of 3.16×10^{-5}.

hit (e.g. E), but also to random nodes (e.g. C and D) as illustrated in Figure 3-1. After receiving the query, E finds Bob's cookie and returns the query to Alice. When C receives the query, he finds that none of his neighbors have a digest hit for Bob, so does not forward the query further. On the other hand, D does forward the query to G (Figure 3-2) who has a digest hit for Bob, and G returns the query to Alice with the cookie she received from Bob. Figure 3.3 shows two paths Alice finds, with the strongest path in bold.

4.4.2 Negative Cookies

A major flaw with the original scheme is that low-valued transactions are potentially not recorded in the system. Consider the following scenario: Eve uses a set of Alice's resources, but does not provide the negotiated resources she promised. In our original scheme, Alice would sign over a low-valued cookie to Eve. Eve would have no incentive to keep this cookie and would promptly discard it, thus erasing any record of her misdeed.

Instead, Alice creates this cookie and stores it *herself*. It is in Alice's interest to hold on to this cookie; at the very least, she will not trust Eve again as long as she has this cookie. However, these "negative cookies" can also be used by users who trust Alice. Suppose Eve next wants to interact with Bob. Before Bob accepts a transaction with Eve, he can initiate a search for Eve's negative cookies. This search proceeds as follows: it follows high trust edges out of Bob and terminates when it reaches a negative cookie for Eve. In effect, the search returns a list of people whom Bob trusts who have had negative transactions with Eve in the past. If Bob discovers a sufficient set of negative cookies for Eve, he can choose to disregard Eve's credentials, and not go through with her proposed transaction. It is important to note that Bob only initiates a negative cookie search when Eve produces a sufficient credible set of credentials; otherwise, Bob is subject to a denial of service attack where he continuously searches for bad cookies. For efficiency, if Eve presents a cookie directly from Bob, Bob need only do a local search for negative cookies. For example, if Bob issued Eve a high valued cookie, and Eve later cheats Bob, Bob would then store a negative cookie for Eve locally. Afterward, when Eve presents Bob with the original high-valued cookie, Bob need only check his local cache for a negative cookie about Eve before rejecting the transaction.

In our implementation, we keep a set of digests for negative cookies as well, but perform Bloom filter-directed searches for these negative cookies only on neighboring nodes.

4.4.3 Preference Lists

In order to discover potentially "good" nodes efficiently, each user keeps a *preference list*. Intuitively, Alice's preference list contains nodes that she has yet to contact, but which have a higher probability of being good than purely random nodes. Nodes are added to a preference list as follows: suppose Alice conducts a successful cookie search for Bob, and let P be the cookie path that is discovered between Alice and Bob. If the transaction with Bob goes well, Alice adds all users in P who have very high trust value (1.0 in our implementation) to her preference list. Alice knows that these nodes have a higher probability of being good than purely random nodes, because she now trusts Bob, and Bob trusts them, creating an implicit trust path. Obviously, only users for whom Alice does not have transaction records are added to her preference list.

In summary, the NICE distributed trust valuation algorithm works as follows:

> *Nodes that request resources present their credentials to the resource owner. Each credential is a signed set of certificates which originate at the resource owner. Depending on the set of credentials, the resource owner may choose to conduct a reference search. The trust ultimately computed is a function of both the credentials, and of the references.*

There are a number of other pragmatic issues pertaining to cookies that we address in NICE, e.g., cookie revocations, and cookie time limits. Specifically, cookies issued from departed nodes no longer have value to the system. Additionally, nodes may change the behavior over time, so cookies should reflect their most current behavior. To address these concerns, cookies are created with an expiration timestamp, and nodes periodically flush expired cookies. The length of the timestamp is application-dependent, and creates a trade off between lookup efficiency versus data freshness. Finally, if a node loses its private key, the only effect is that useless cookies will persist in the network until they expire.

5 Results

We present simulations from different sets of experiments. In the first set (this section), we analyze the scalability and robustness of our inference scheme. In these experiments, our goal is to understand how well the cookies work, without regard to how a real system might use cookies. For example, in these experiments, "good" nodes continually interact with unknown nodes, even when they already know

of a large set of other good nodes. While this assumption helps demonstrate how cookies propagate and nodes discover each other in the system, in a real system, we expect good nodes to find a set of other nodes they trust and interact with these trusted nodes much more heavily. We examine this further in section 6, we present experiments in which cookies are used to establish trust, but then nodes follow a more conventional pattern, and try to interact with their trusted peers with higher frequency.

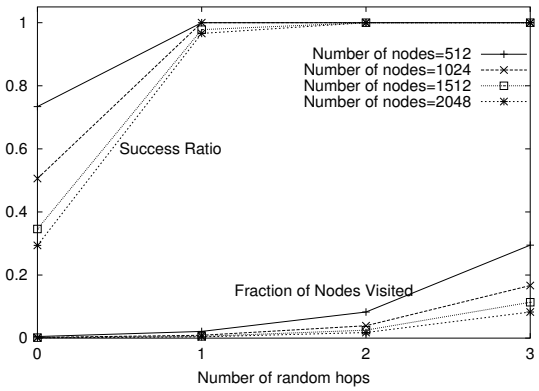

Figure 4: Success ratio and no. of nodes visited (40 cookies at each node).

Experiments with the search algorithm In the rest of this section, we present results from our simulations of the trust inference algorithm proposed in Section 4. In all our results, we use the minimum cookie value as path strength, and use the highest valued path strength as the inferred trust between users. We have experimented with other functions as well, and the results from this simple inference function are representative. Each search carries with it the minimum acceptable strength, and searches stop if no cookies of the minimum acceptable value are present at the current node. Using the minimum cookie value as the strength measure (instead of product of cookie values) consumes up to an order of magnitude more resources in the network and represents a worst-case scenario for our schemes.

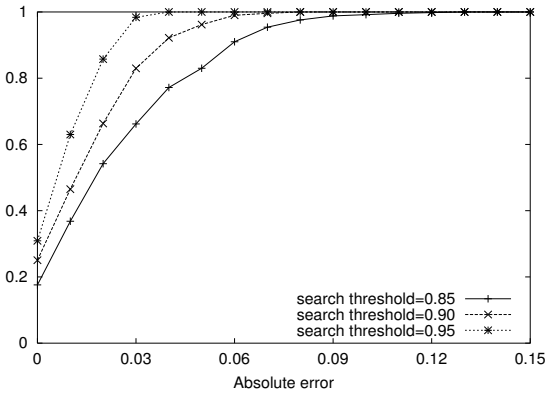

Figure 5: CDF of errors versus oracle (40 cookies at each node, out-degree set to 5) with varying thresholds.

We divide our results into two parts. First, we analyze the cost of running the path search algorithm in terms of storage and run time overhead. The storage cost is entirely due to the caching of positive and negative cookies; the run-time overhead comes from the number of nodes that are visited by each query, and the computation cost for forwarding a query. The computation cost of forwarding each query is negligible: we have to generate random numbers, compute eight MD5 hash functions, and check eight bits in a 1000-bit Bloom filter. In these experiments, the digests were assumed to be always fresh. We did not simulate updating of the digest, but we believe a periodic soft-state refreshing algorithm will work adequately. The main overhead of the search algorithm comes in terms of the number of messages sent and number of nodes visited. The bandwidth consumed by the searches is proportional to the number of nodes visited, and we report this metric in the results that follow. In the second part of our results (Section 5.2), we show that our trust inference schemes do indeed form robust cooperative groups, even in large systems with large malicious cliques and with small fractions of good nodes. We begin with an analysis of the scalability and overhead of our path searches.

5.1 Scalability

In this first set of results, we simulate a stable system consisting of *only* good users. Thus, we assume that all users implement the entire search protocol correctly. Before the simulations begin, we populate the cookie cache of each user with cookies from other users chosen uniformly at random. Each query starts at a node s chosen uniformly at random and specifies a search for cookies of another node t chosen uniformly at random. In the next section, we will show how long the system takes to converge starting from no cookies in the system, and how robust groups are formed when there are malicious users in the system.

In our first experiment, we fix the number of good cookies at each user to 40. The cookie values are exponentially distributed between [0,1], with a mean of 0.7[5]. Note that

[5]It is not clear how cookie values should be distributed. We have also experimented with uniformly distributed cookie values with similar

all nodes with constant number of cookies is a *worst case* for searching performance. If cookies were distributed unevenly, for example, with a Zipf distribution, then the average diameter of the trust graph would reduce, and the lookup times would decrease [8].

We conducted 500 different searches for cookies of value at least 0.85, where the search out-degree at each node is set to 5. In Figure 4, we plot the average success ratio and the average fraction of nodes in the system visited by the searches. The x-axis in the plot corresponds to the number of hops after which random forwards were not allowed, and the search proceeded only if there was a hit in a Bloom filter. There are four curves in the figure, each corresponding to a different system size, ranging from 512 users to 2048 users. From the figure, it is clear that only one hop of random searching is enough to satisfy the vast majority of queries, even with large system sizes. It is interesting to note that even when the system size increases, the average *number of nodes* visited remain relatively constant. For example, the average number of nodes visited with 2 hop random searches range from 42.4 (for a 512 node system) to 36.2 (for the 2048 node system). Thus, the search scheme scales well with increasing system size. As we show next, the success ratio and the number of nodes visited depend almost entirely on the number of cookies held at each node, and the out-degree of each search.

In Table 1, we fix the number of nodes to 2048 and show the effect of changing the search out-degree. Each row shows searches corresponding to a different minimum threshold ranging from 0.8 to 0.95. Each node holds 40 cookies, the average cookie value is still fixed at 0.7, and the number of random hops is set to 2. In the table, $\#N$ is the average number of nodes visited by a query and $\#P$ denotes the number of paths found on average. As expected, the number of nodes visited increases as the search threshold decreases, and also as the out-degree increases. In all cases, as the search threshold increases, the number of distinct paths found decreases. In Table 2, we show the effects of changing the number of cookies at each node. These experiments were conducted using the same parameters, except the out-degree was fixed at 5. With small numbers of cookies and high thresholds, searches do result in no paths being found. In Table 2, the 0.9 and 0.95 threshold searches had 10% and 42% unsuccesful queries respectively; all other searches returned at least one acceptable path. In our simulator, when a search returns no acceptable paths, we retry the search once more with a different random seed. The numbers of nodes visited in the results above include visits during the retries and account for why the number of nodes visited does not decrease when the search threshold is increased.

In our system, there is a clear trade-off between how much state individual nodes store (number of cookies) and the overhead of each search (fraction of nodes visited). Note that unlike in systems such as [2], users in our system do not benefit by storing fewer cookies since this effectively *decreases* their own expected trust at other nodes. There is a built-in incentive for users to store more cookies, which, in turn, increases search efficiency. Users may choose to store a large number of cookies but not forward searches on behalf of others. We comment on this issue when we discuss different models of malicious behavior in the next section. Lastly, we note that it is possible to further increase the efficiency of the searches by adjusting the two search parameters — out-degree and number of random hops — based on the threshold and results found. Such a scheme will minimize the number of nodes visited for "easy" searches (low search threshold) and find better results for searches with high thresholds. We have not implemented this extension yet.

The previous two results have shown that the number of cookies and search out-degree provides an effective mechanism to control the overhead of individual searches. However, in each case, we have only shown that each search returns a set of results. It is possible that the searches find paths that are above the search threshold, but are not the best possible paths. For example, suppose that a search for threshold set to .85 returns a path with minimum cookie value .90 . While this is an acceptable result, there may be a better path that the search missed (e.g. with minimum cookie value .95). In this case, the best path returned had an absolute error of .05. To quantify the quality of the found paths, we plot the absolute error in the paths returned by our searches as compared to an optimal search (full flooding). In Figure 5, we plot the CDF of the absolute error for the best path that we find versus the best possible cookie path given an infinitely knowledgeable oracle as the search threshold is changed. The higher threshold searches have a smaller possible absolute margin of error, and thus produce the best paths. However, very high threshold searches are also more likely to produce no results at all.

5.2 Robustness

We analyze two components of the system: how long it takes for the system to stabilize and how well our system holds up against malicious users. Modeling malicious users is an important open research question: one for which we do not provide any particular insights in this

results.

	K=3		K = 5		K = 7		K = 20	
thrsh.	# N	# P	# N	# P	# N	# P	# N	# P
.8	14.5	4.2	37.5	10.7	71.1	20	499.1	132.5
.85	14.6	3.6	36.2	8.7	68.7	16.5	380.6	88.3
.9	14.6	2.9	35.1	6.8	66.0	12.8	222.9	41.5
.95	15.7	2.0	33.2	4.2	55.9	7	89.2	11.2

Table 1: Effect of changing out-degree (K): N, P=nodes, paths traversed

	C=20		C=40		C=102	
thresh.	# N	# P	# N	# P	# N	# P
0.8	34.4	2.9	37.5	10.7	32.8	23.6
0.85	34.7	2.4	36.2	8.7	37.3	25.4
0.9	34.7	1.9	35.1	6.8	40.8	25.4
0.95	28.6	1.4	33.2	4.2	41.9	21.5

Table 2: Effect of changing number of cookies stored (C)

paper. Instead, we use a relatively simplistic user model with three different types of users:

- Good users: Good users always implement the entire protocol correctly. If a good user interacts with another good user, then the cookie value assigned is always 1.0. Good users do not know the identity of any other good (or otherwise) users at the beginning of the simulation.
- Regular users: Regular users always implement the entire protocol correctly; however, when a regular user interacts with another user, transactions result in cookie value that range exponentially between 0.0 and 1.0, with a mean of 0.7. Regular users also do not know the identities of any other users when the simulations begin.
- Malicious users: All malicious users form a cooperating clique before the simulation begins. Specifically, each malicious user always reports implicit trust (cookie value 1.0) for every other malicious user. Once a malicious user interacts with a non-malicious user, there is a 20% probability that the transaction is completed faithfully, else the malicious user cheats the other party. The intuition behind this model is that nodes that are consistently malicious, i.e., that fail transactions 100% of the time, are easily detected and defeated. With malicious nodes that periodically complete transactions, we are considering a slightly more sophisticated attack model.

At each time step in the simulation, a user (Alice), is chosen uniformly at random. Alice selects another user (Bob) from her preference list with whom to initiate a transaction. If Alice's preference list is empty, she chooses the user Bob uniformly at random. This transaction commences if Bob can find at least one path of strength at least 0.85 between himself and Alice and if Bob cannot locate a negative cookie for Alice. If no cookie path can be found, i.e., the transaction between Alice and Bob cannot proceed, Alice tries her transaction with a different user. After two unsuccessful tries, Alice chooses a random user Carol and the simulator allows a transaction without checking Alice's credentials. (Recall that in these first set of experiments, good users only want to form a large good user clique, and do not initiate transactions with other good users they know of with higher probability). When the cookie cache is full, cookies are removed from a user's cookie cache using the following rule: cookies of value 1.0 are not replaced; other cookies are discarded with uniform probability.

In the first result, we only consider good and regular users (there were 488 regular users and 24 good users in this experiment). In Figure 6, we plot the fraction of transactions between good users and the fraction of paths between good users. The x-axis shows the total number of transactions in which at least one party was a good node. (We choose this measure as the x-axis because in a real system, malicious nodes can fabricate any number of spurious transactions, and the only transactions that matter are the ones involving good nodes). The effect of the preference lists is clear from the plot: even though there is a less than 5% chance of a good node interacting with another good node, there is a path between any two good node within 1500 transactions. By 2500 total transactions, the majority of which were between good nodes and regular nodes, all good nodes have cookies from all other good nodes, and the robust cooperative group has formed. This good clique will not be broken unless a good node turns bad, since 1.0 valued cookies are not flushed from the system.

In the next set of results, we introduce malicious nodes. Figure 7 illustrates the fraction of failed transactions involving good nodes normalized by the total number of transactions involving good nodes. The curves in the figure show the number of failed transactions involving good nodes for varying numbers of bad nodes in the system, averaged over 1000 transaction intervals. For these results, we define failed transactions as those that produce a cookie of value less than 0.2. In the beginning of the simulation, the number of failed transactions are proportional to the number of bad users in the system. However, for all bad user populations, the good users identify all bad users

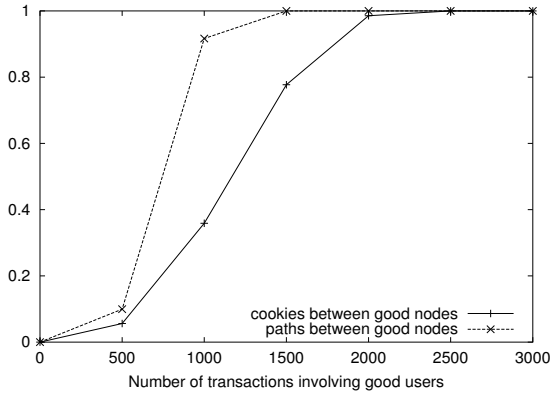

Figure 6: CDF of system initialization with good and regular users.

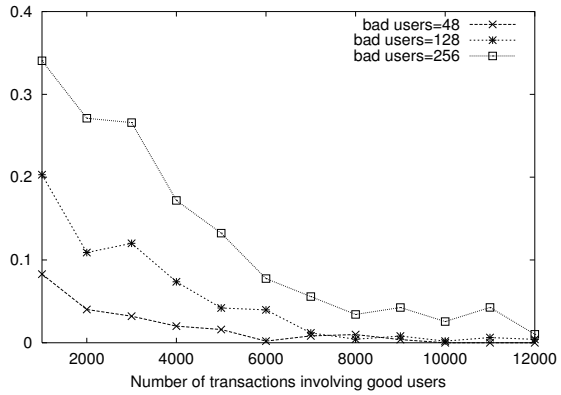

Figure 7: Fraction of failed transactions for good users (40 cookies at each node, 512 nodes total).

and the number of good-bad transactions approaches zero. The effect of the preference lists is again apparent in this experiment: recall that all bad nodes always report 1.0 trust for other bad nodes. Thus, bad nodes rapidly fill the preference lists of good nodes, but are quickly identified as malicious.

In our experiments, good users do not preferentially interact with other good users (as would be expected in a real system). Instead, if their preference lists are empty, they pick a random user to interact with. Recall that there are an order of magnitude more regular users in the system than good users. Thus, good users continue to interact with regular users and approximately 5% of good user transactions result in failures (not shown in Figure 7). In a deployed system, the fraction of failed transactions would be much smaller, since the vast majority of transactions initiated by good users would involve other good users.

It is important to note that even with many malicious users, a robust cooperative group *eventually* emerges in our system. This property is true, regardless of the number of positive or negative cookies good users keep, as long as *good users can choose random other users* to conduct transactions with and *can withstand failed transactions proportional to the fractions of malicious users*. Without random node selection, bad cliques can stop good users from ever communicating with another good user. However, as long as bad users cannot stop to whom good users communicate, a cooperative group emerges. In our simulations, this translates to assuming that the bootstrap node or topology server is trusted. However, in practice the boot strapping service could be logically implemented using a more secure system [19]. The good users eventually find and keep state from other good users, and this state cannot be displaced by malicious users. Obviously, the number of transactions required for good cliques to form depends on the number of malicious nodes in the system, but good users rapidly find other good users by using their preference lists. It is possible for a malicious node to infiltrate good cliques for prolonged periods, but as these bad nodes conduct transactions that fail, the negative cookies will cause these users to be rapidly discarded from the good clique.

We have varied other parameters in our experiments, and present a summary of our findings. We experimented with a different malicious node model in which the bad nodes do not forward queries from non-malicious nodes. The results for this model were not appreciably different from the model we have used in our above results. Also, it is not immediately clear how to choose the probability with which transactions with malicious users fail (recall that bad nodes succeed 20% of the time). If this probability is low, then malicious users can be identified relatively easily (usually after one transaction). If this probability is set too high, then in effect, the user is not malicious since it acts much like a regular user. In our experiments, as the bad nodes reduce the transaction failure probability, the number of transactions required to identify all bad nodes increases, but the total number of bad transactions remain similar. We have also experimented with models in which bad users actively publish negative cookies for good users. As these users are identified as bad by the good users, these negative cookies are rendered useless. Lastly, we note that our *good* user model is probably too simplistic. Even good users may be involved in failed transactions, possibly due to no fault of their own. However, we believe our results will still hold as long as there is a definite and marked difference between the behavior of good and bad users.

6 Simulations on a Realistic System

As noted above, we have implemented the cookies protocol in a different simulator, which admits much larger user populations ($O(10^5)$ users). The goal of the new simulator is to model a more realistic resource discovery model, attempts to model work loads, and quantify the effect of our trust based trade limits (Section 3.1). We assume that there is a single bootstrap node that keeps track of the last 100 nodes to join the system. Each node periodically queries the bootstrap node to obtain a set of neighbors (if a node already has sufficient neighbors, it does not query the bootstrap node). The bootstrap node is not part of the trust inference system, and is used only to start the simulation. Each node in the system starts with a fixed number of "jobs" that they need completed, and a fixed number of jobs that they can serve for others. Of course, malicious nodes need not complete any job they accept. After discovering each other, pairs of nodes conduct a "transaction" to trade a set of jobs.

Node model Each good node maintains the following state:

- For each jobs, a status indicating whether that job is complete or not. If completed, the node remembers which node completed the job. Finally, the node also maintains state about jobs that have been issued but not completed.
- A fixed set of good cookies and a fixed set of negative cookies. These are used exactly as described above. Recall that if a node n has a good cookie from node p, then it also has a digest of the set of bad cookies that p has recorded.
- A preference list, which is a set of nodes to whom the next set of jobs will be issued.

In the simulator, bad nodes accept jobs but do not complete them with a fixed probability. We assume that there is a post-verification protocol that allows good nodes to realize that their jobs were not completed properly.

6.1 System Behavior

The simulations proceed as follows: the nodes initially populate their preference list using random information from the bootstrap server. Each node issues a maximum number of transaction requests (nominally set at 10 for each simulation) to nodes in their preference list. Each node maintains its preference list sorted in order of fraction of successful transactions with the nodes in the list ties are broken using the actual number of good transactions, and the transactions are issued in this sorted order.

Assume that n requests a transaction to be completed at node p. Each transaction requests a specific number of jobs to be completed. The number of jobs issued from node n to p increases exponentially with the number of successful transactions. (We have also experimented with a linear increase scheme, which we present in the results). With each transaction request, n tries to present a valid cookie path to p.

Node p accepts the request from n as follows: if n cannot present a valid cookie path, p searches the network for a bad cookie for n. If a bad cookie is found, then p rejects the transaction request. If a bad cookie is not found,i.e., p has no information about n good or bad, then p will accept the request with a fixed probability, 50%, else ask n to retry its request later. If p accepts the request, then it will, initially do one job for n. Recall that the number of jobs accepted at a node will increase with each successful transactions, as per the trust-based trading limits.

After p completes a set of jobs for n, it does not accept any other jobs from n until n performs an equivalent set of jobs for p. In general, the set of jobs accepted is constrained by the number of available resources at each node, the number of actual outstanding jobs each node has, etc.

After a transaction completes, n issues a "verification" message. This is how good nodes realize that malicious nodes have not completed their tasks properly. Once n finds that it had issued jobs to a bad node (say b), it records a bad cookie for b, and marks all the previous jobs done by b as failed. Note that allows us to model a relatively broad notion of a job. For example, jobs could be data blocks stored at other nodes, or jobs could be computations conducted at other nodes.

All simulation messages have latency between 10-11ms, distributed uniformly at random. Also, nodes issue the verification message a random amount of time after the transaction has occurred.

Preference List updates The efficiency of this system (like real systems) depends on which nodes are contacted in what order when node n wants to place jobs. In the simulator, this is reflected in the way the preference lists are maintained. When node n issues a bad cookie for any node b, n takes b out of its preference list. If n issues p a good cookie, then p gives n a copy of its preference list; n integrates this information into its own preference list as follows. Initially, all new nodes in p's preference list are assigned the same trust (and transaction success parameters) as p. Then, these nodes replace existing nodes with

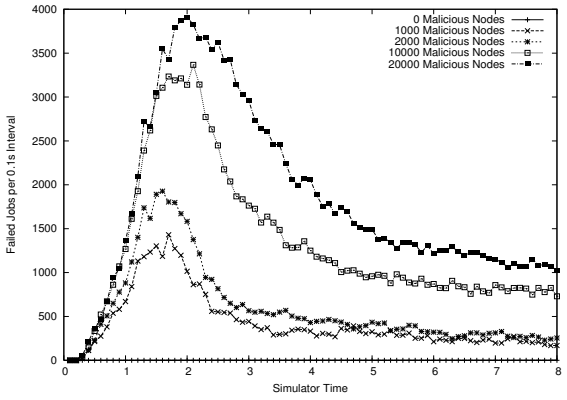

Figure 8: Failed jobs over time; 80 cookies

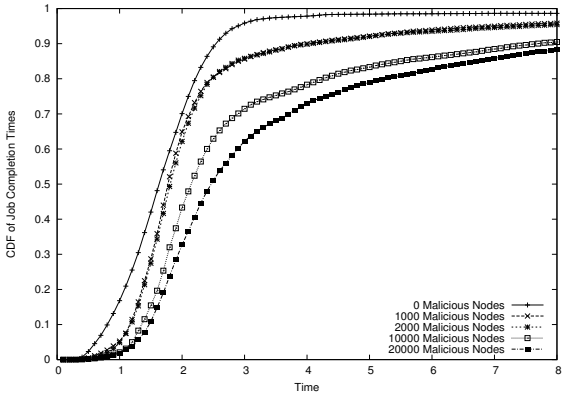

Figure 9: CDF of job completion times; 80 cookies

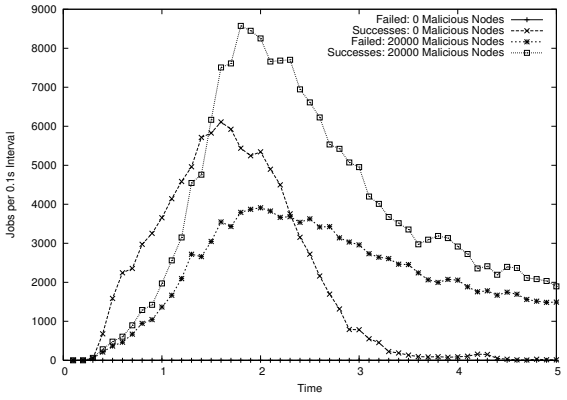

Figure 10: Succeed vs. Failed Jobs; 80 cookies

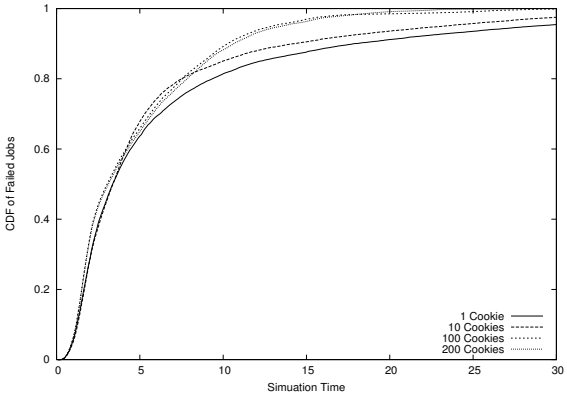

Figure 11: Cumulative distribution of failed jobs; 2000 malicious nodes

lower trust value in n's preference list.

6.2 Simulation Results

Given our system description, there are two key metrics that we chose to measure in the simulation. Specifically, we consider how quickly good nodes place all of their jobs, i.e. completion time, and how many jobs are lost to bad nodes in the process, i.e. loss rate. Then we show results where we vary the number of malicious nodes relative to "good" nodes, and we vary the amount of state each node is allowed to hold. In all experiments, transactions only involving malicious users are disregarded.

Number of Malicious Nodes In this experiment, we fix the number of good users (as defined in Section 5) at 1000 nodes. Each node carries state for 80 cookies, and joins the system distributed uniformly at random during the first second of the simulation. Each node has 100 jobs to place, and capacity to serve 100 jobs. A node stays in the system until it can place all of its jobs successfully; nodes time out after 30 seconds of inactivity. In all experiments, less than 1% of nodes time out. Malicious users fail jobs with 80% probability.

Figure 8 illustrates the number of failed jobs over time, as measured in 0.1 second intervals. As expected, the number of failed jobs initially increases as the ratio of malicious to good nodes increases. However, over time as good nodes discover each other, the number of failed jobs approaches zero. Specifically, in the case with 20,000 malicious nodes, i.e. a 20-to-1 ratio of malicious to good nodes, good nodes are still able to make progress as shown in Figure 10. By simulation time 3 seconds, over 62% of the total jobs in the system have been placed (Figure 9).

Note that in Figure 10, the number of successful jobs also reduces over time (unlike in Figure 7 from Section 5). This is because, in these simulations, good nodes leave the system after all of their jobs are satisfied. Since new nodes do not join, after a few simulation seconds, almost all the good jobs in the system are done (unlike in the previous case, where there was an unbounded number of jobs

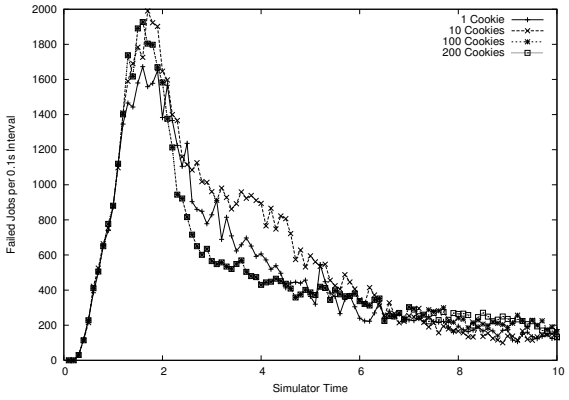

Figure 12: Failed jobs over time; 2000 malicious nodes

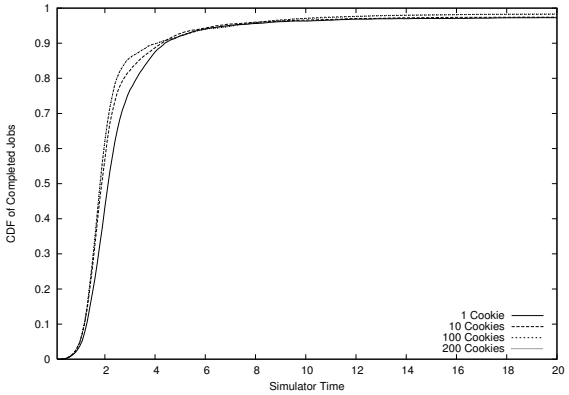

Figure 13: CDF of completed jobs; 2000 malicious nodes

in the system). We believe that a system that incorporated continuous node arrivals and departures, as would be expected in a practical system, could result in better average performance.

Amount of State Per Node Even in this more realistic system (in which good nodes visit each other preferentially), the number of cookies a node keeps is important. We have experimented with varying the amount of cookie state each node keeps. In this experiment, 1000 good nodes join uniformly at random within the first second of simulation time. The number of malicious nodes is fixed at 2000, and the amount of state varies. As above, each node has 100 jobs to place, and capacity for 100 jobs. Malicious users fail jobs with 80% probability.

Figure 11 depicts the cumulative distribution of failed job placements over time. As expected, the number of failed jobs is initially high, but reduces as time progresses and as cookies are traded throughout the system. Figure 12 represents a count per 0.1 second interval of failed job placements over time. Note that as the number of cookies increase, the completion times and the number of failed jobs decrease. Observe the benefit from doubling the number of cookies from 100 to 200 is minimal, and thus a node can achieve practically full benefit from the cookie protocol from storing merely 100 cookies. Figure 13 shows the cumulative distribution of completed jobs. Note that nodes place over 90% of their jobs in the first 7 seconds in all experiments.

7 Summary and Conclusions

The main contribution of this paper is a low overhead trust information storage and search algorithm which is used to implement a range of trust inference and pricing policies. Our scheme is unique in that the search and inference performance for the whole group increases as users store more information, and information is explicitly beneficial for the storer's cause. We have presented a scalability study of our algorithms, and have shown that our technique is robust against malicious users. We have experimented with networks with over 20,000 nodes. Our results show that the protocol presented here scales to networks of this size, however, it is not clear that the protocol presented here will be efficient on systems an order of magnitude larger. It is likely that in very large systems, a fundamentally more sophisticated solution, e.g. based upon a DHT [20, 18] or random walks [15] would be preferred. We should note that structures such as a DHT implicitly assume that all participants are trustworthy, and we expect a solution such as ours will be required as the basis for booting a trusted DHT. We also believe techniques presented in this paper are a crucial piece for building large peer-to-peer systems for deployment over the Internet.

References

[1] Alfarez Abdul-Rahman and Stephen Hailes. Supporting Trust in Virtual Communities. In *In Proceedings Hawaii International Conference on System Sciences 33*, 2000.

[2] Karl Aberer and Zoran Despotovic. Managing trust in a Peer-2-Peer information system. In Henrique Paques, Ling Liu, and David Grossman, editors, *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM-01)*, pages 310–317, New York, November 5–10 2001. ACM Press.

[3] E. Adar and B.A. Huberman. Free riding on gnutella. Technical report, Xerox PARC, August 2000.

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. ACM Sigcomm*, August 2002.

[5] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. Self-organizing wide-area network caches. In *IEEE Infocom'98*, 1998.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[7] Béla Bollobas. *Random Graphs*. Cambridge University Press, 2001.

[8] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-law Relationships of the Internet Topology. In *SIGCOMM*, pages 251–262, 1999.

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *Proc. SIGCOMM'98*, 28(4):254–265, September 1998.

[10] E. Friedman and P. Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10(2):173–199, 1998.

[11] http://gnutella.wego.com.

[12] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *World Wide Web (WWW)*, 2003.

[13] Landon P. Cox and Brian D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer storage. In *Symposium on Operating Systems Principals (SOSP)*, pages 120–132, October 2003.

[14] S. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Sterling, 1994.

[15] Ruggero Morselli, Bobby Bhattacharjee, Michael Marsh, and Aravind Srinivasan. Efficient lookup on unstructured topologies. Technical Report CS-TR-4593, University of Maryland, 2004.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *In Proceedings of the ACM SIGCOMM 2001 Technical Conference*, 2001.

[17] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *Proceedings of INFOCOM 2002*, 2002.

[18] Antony Rowstran and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[19] Narendar Shankar, Christopher Komareddy, and Bobby Bhattacharjee. Finding Close Friends over the Internet. In *Proceedings of Internation Conference on Network Protocols*, November 2001.

[20] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[21] http://www.ebay.com .

[22] http://www.mojonation.net .

[23] Bin Yu and Munindar P. Singh. A social mechanism of reputation management in electronic communities. In *Proceedings of Fourth International Workshop on Cooperative Information Agents*, 2000.

[24] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area dat a dissemination. In *Eleventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001)*, 2001.

[25] P. Zimmermann. Pretty good privacy user's guide. Distributed with PGP software, June 1993.