Using Gossip for Dynamic Resource Discovery

Eric Simonton, Byung Kyu Choi, and Steven Seidel
Department of Computer Science
Michigan Technological University
Houghton, MI 49931
Email: {ersimont, bkchoi, steve}@mtu.edu

***Abstract*— Resource discovery is the process of locating shared resources on a computer network. Previously studied examples include efficiently finding files with a given title on a file sharing system. New developments in the application of networked computers raise the issue of dynamic resource discovery, the process of locating shared resources that are always changing. An example application is peer-to-peer computing, where a user wishes to locate idle CPU time anywhere on the network.**

Peer-to-peer computing is an exciting new computing paradigm. There are vast amounts of idle CPU resources scattered through the globe. We envision a peer-to-peer system to harness those resources, where every member of the network can both share their own CPU and utilize others' CPUs. In a network of hundreds of thousands of computers, resource discovery will play an important role. To avoid debilitating amounts of excess network traffic it is imperative that an efficient resource discovery algorithm be chosen.

This paper's contribution to this topic is the use of gossip to reduce network traffic without sacrificing effectiveness. This project has investigated piggybacking gossip messages on other communications to increase the intelligence of searching protocols. The overhead of piggybacking the small amount of data needed is very small, and a case study by simulation shows that it can reduce network traffic by 71-84 percent.

I. Introduction

Peer-to-peer networks have proven to be a powerful and popular tool for file sharing. In the future, it is not hard to imagine further applications of the peer-to-peer paradigm, such as CPU cycle sharing. Projects like SETI@HOME, Grid.org, and Distributed.net have proven the immense potential of harnessing the vast amount of idle CPU resources in the world [1]–[3]. Unlike these projects, a peer-to-peer system would offer these CPU resources to anyone on the network, instead of a privileged few.

Resource discovery is the process of finding (or *discovering*) something in a network. In file sharing, this is the process of searching for files, where files are the *resource*. In peer-to-peer computing, it is the process of searching for available memory, disk space, and/or CPU cycles. Since shared files most likely never change when in the file sharing system, they are termed *static* resources, whereas CPU cycles are *dynamic* because their availability can change frequently.

Dynamic resources pose an added challenge to resource discovery, since they require keeping track of those changing resources. For this reason, in large scale peer-to-peer computing, resource discovery will be a critical component, as an inefficient protocol would unnecessarily consume a significant portion of network bandwidth as well as CPU time. In file sharing, completely flat and completely decentralized resource discovery protocols are of interest because they do not require nodes to volunteer for specialized responsibilities, and they eliminate single points of failure. Again, these benefits are multiplied when keeping track of changing resources. Avoiding single points of failure, in addition to simplifying membership management protocols, is also a good reason to use an unstructured network; unstructured networks are fairly tolerant of node failures as well as targeted attacks [4].

In general, all resource discovery protocols must be push-based, pull-based, or a combination of the two. A push-based protocol advertises resource availability throughout the network so that, upon demand, a resource requester does not have to spend time searching for it. However, it creates unnecessary overhead when demand is low. A pull-based protocol eliminates this overhead by eliminating advertisements. The drawback is that search messages take longer to find resources, since they must travel blindly through the network.

Algorithms designed for static resource discovery combine push- and pull-based techniques in clever ways to balance the tradeoffs mentioned above. However, they will not likely prove to be the best techniques for dynamic resources, as in peer-to-peer computing. For example, many of the best techniques propagate pointers to resources through the network so that a search can find any one of the pointers and be immediately directed to the resource. However, there is additional overhead for keeping such pointers up-to-date for dynamic resources, which could easily render those techniques inefficient.

There is a large domain of dynamic resource discovery possibilities to explore. Instead of inventing an entirely new protocol, this paper presents one step into that area by offering a method to modify existing pull-based protocols. This technique piggybacks advertisements, in the form of gossip messages, onto the packets generated by those protocols. Those advertisements are then used to re-direct search packets generated by the pull-based protocol toward resources that are more likely available. Hopefully this work sheds some light into the dynamic resource discovery realm to help develop a systematic new protocol in the future.

The rest of this paper is structured as follows: Section II discusses other work related to resource discovery, section III lays out the assumptions made for the system model used to evaluate this work, section IV describes the use of gossip in detail, section V shows the results of experimental evaluations, followed by concluding remarks in section VI.

II. Related Work

A. File Sharing

In considering *dynamic* resource discovery, there is much to learn from work that has been done for *static* resource discovery, usually done in the context of file sharing systems. Many of the techniques employed there can easily be modified to accommodate dynamic resource discovery, but leave room for improvement due to differences in the application environment.

Of particular interest is the work done to improve the resource discovery protocol of the Gnutella network. Gnutella became famous as a popular file sharing application, and infamous for its inefficient resource discovery. However, it proved the viability of a large, completely unstructured network.

One suggestion to improve Gnutella's resource discovery efficiency comes from the observation that unstructured networks roughly follow a power-law distribution, where few a nodes connect to many neighbors and many nodes connect to a few neighbors [5]. This work suggests that all nodes keep track of the resources held by their neighbors, so each node can answer a query for itself *and* its neighbors. Using this technique tests were run using a simple random request forwarding scheme, and results show surprising efficiency.

Another work further improved that idea using the following principle: the nodes with the highest connectivity (greatest number of immediate neighbors in the overlay network) keep track of the content of the greatest number of nodes in the network [6]. Therefore, the more high-connectivity nodes a query passes through, the more of the network it will cover. So queries are always forwarded to the neighbor with the highest connectivity. This leverages the heterogeneity in connectivity of a power-law network's nodes. Mechanisms for topology adaptation and flow control further refined this approach in [6].

Directly applying any of these search techniques to dynamic resources raises an immediate problem: how should a node keep track of its neighbors' resources? With something as volatile as CPU availability, it is clear that a simple pointer cannot be kept for very long to resources at any neighboring node. This is the exact problem this work tackles.

Another idea for resources discovery in unstructured networks is called *percolation-based* searching [7], [8]. While [8] claims this is superior to the solution described above because of greater network coverage, this is not actually very helpful in something like a cycle-sharing environment. File sharing systems query for a resource held only one or a few nodes, whereas cycle-sharing systems query for a resource that resides (at times) on very many, if not all of the nodes. If it is necessary to query large portions of the network to find this type of resource, it must be that a very large portion of the network is heavily loaded. Therefore, there will be many other hosts also looking for resources. With so many machines sending queries, any free resources will be found by nearby neighbors, eliminating the need for queries to reach far into the network.

B. Peer-to-Peer Computing

Applications like SETI@HOME [1], Grid.org [2] and Distributed.net [3] already exist that leverage idle computing resources scattered over the globe. They do not, however, address the goals of peer-to-peer computing. Instead, they provide massive amounts of parallelism to no more than a few applications at a time, which requires developers to write software with this massive parallelism in mind. This paper seeks to provide for a system true to the peer-to-peer architecture, where every member of the network can both offer resources *and* utilize those of other members.

Lanfermann *et al.* describe work which meets these criteria [9], [10]. The system they discuss allows executing entities to migrate from one machine to another, as more desirable resources become available. This concept is easily extended to the type of system this paper describes; in fact it is part of the groundwork on which it is based. The ultimate goal is to allow users' programs, or program modules, to float anywhere on the network, wherever available computational power is found. However, like the existing applications discussed above, the resource discovery activities for these systems rely on a centralized server. Of course, any time centralized servers are used, scalability is compromised since the capacity of the centralized servers will always dictate a maximum network size, and a single point of failure is introduced.

Until recently we have found very little work that addresses fully distributed, *dynamic* resource discovery. Fairly simple, local algorithms for this purpose were presented in 2001, but that work was carried no further [11]. Recent papers which address these criteria appeared in 2004 [12], [13]. These papers address the precise research problem this paper takes up. They provide a good start for finding a suitable protocol, but there is still more work to be done. They evaluated expanding-ring searches, purely advertisement-based searches, and rendezvous point searches. Based on their results, they favored the rendezvous point protocol. Although their simulation results do show excellent performance through rendezvous points, rendezvous points are more or less centralized servers. Though many may be used, which avoids a single point of failure, the practice does not fully leverage all the benefits that can be found in a pure peer-to-peer architecture, where all members of the network share equal responsibility.

More recently, Nandy *et al.* addresses the topic of resource discovery in a cycle-sharing network [14]. They use gossip to maintain distance-vector routing tables that point toward available resources. This is very promising work, which could benefit from incorporating some of the ideas presented in this paper. One weakness it possesses is the assumption that nodes only communicate with their immediate neighbors. When considering a peer-to-peer network laid over the internet, this is a very restrictive assumption.

This paper joins [14] in offering a scalable, fully decentralized, truly peer-to-peer technique for dynamic resource discovery. Unlike [14], however, it is not a stand-alone protocol. Instead it shows how to modify existing pull-based protocols to increase their efficiency in a dynamic resource environment. This work takes one step out of the realm of static resource discovery protocols into that of dynamic resource discovery, instead of proposing an entirely new approach. It is also flexible enough accommodate both walker-type protocols and flooding-type protocols. Walker-type protocols generally create much less network traffic, while maintaining equally good performance when searching for a single resource. Flooding-type protocols, however, are useful for finding many resources at once, perhaps when finding many CPUs on which to run many parallel tasks.

III. System Model

This section describes the system model used for this work.

A. The Network

- There is no packet loss.
- Every node in the network is always reachable.
- The network latency between two nodes follows a uniform random distribution for every packet, between L_{min} and L_{max} milliseconds. Note that this allows for out-of-order delivery and jitter.
- Nodes join and leave the network over time creating network churn, but stay in the network at least as long as the average time required to complete one task. Session times follow a power-law distribution, which appears to be a property of all peer-to-peer networks regardless of their application [15].
- A hypothetical membership management protocol is used which maintains a power-law distribution of nodes' out-degrees, a well-known characteristic of unstructured peer-to-peer networks [7], [16], [17]. In simulations, this hypothetical protocol instantly connects new nodes to the network and does not affect network traffic statistics.
- Nodes gain connections with time, so that the oldest nodes in the network have the largest out-degree.
- Nodes never fail; they always go through a disconnection process to leave the network.
- When a node leaves the network it does not query for new nodes to finish the tasks in its CPU queue; they are simply dropped and recorded as a *churn failure*. This simplifies the simulation but still provides a fair comparison between different resource discovery protocols. Churn failures are not reflected in this paper's results.

B. The CPU/Task Model

A very simple CPU model is used for a dynamic resource.

- Every node on the network has one CPU which it shares openly for as long as it remains in the network.
- CPU speeds follow a Weibull distribution [18] with shape parameter = 2.
- CPUs use simple, non-preemptive, first-come-first-served scheduling with no concept of task priority.
- Each CPU has a task queue with space for one waiting task in addition to one executing task. In this work a CPU's *availability* is defined as the number of tasks it can accept; i.e. zero if there are tasks both executing and waiting, one if there is only one executing, and 2 if there are no tasks in the CPU.
- Resource discovery commences whenever a task is generated at a node whose CPU queue is full.
- Tasks are represented simply by the number of computational cycles required to complete them. These sizes follow a Weibull distribution with shape parameter = 1.
- Tasks enter the network from any node, following a uniform distribution.

C. The Resource Discovery Protocol

- Any pull-based resource discovery protocol may be employed, and will be modified on-the-fly.
- All nodes' clocks are roughly synchronized.
- All nodes use the same resource discovery protocol.

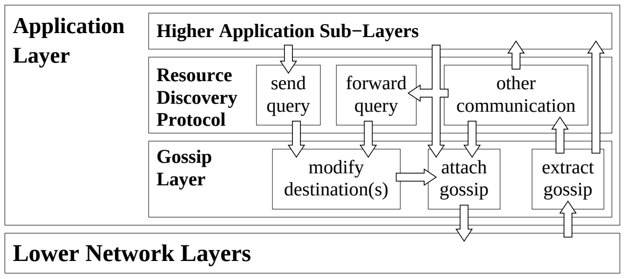


Fig. 1. The main idea this paper presents can be seen as a new sub-layer in the network protocol stack, labeled here as the "Gossip Layer".

IV. Main Idea

This paper presents the use of gossip in a sub-layer of the application layer of the network protocol stack. Section IV-A describes this sub-layer by example, section IV-B describes the contents of a gossip message and sections IV-C, IV-D and IV-E describe the algorithmic details of the sub-layer's functions. The three latter sections contain a few simplifications for the sake of clarity, which are addressed in section IV-F.

A. A Typical Scenario Using the 16-Walker Protocol

This scenario illustrates the functions of the main idea this paper presents, when applied to the K-walker protocol. Figure 1 shows the flow of packets in this scenario.

A user wishes to find an available CPU, which prompts the *Resource Discovery Protocol* to generate a query packet for K of its neighbors, chosen randomly (per the K-walker protocol). It passes that packet and its list of destinations to the *Gossip Layer*, which has already built up a list of gossip (called the *gossip queue*) extracted from packets it received previously. Using the information in that gossip queue, it modifies the list of destinations to target nodes whose CPUs are more likely available. Before sending the K packets over the network, it must attach some gossip messages to them. One of these messages is about the current state of the current node, and the others are copies made from the gossip queue.

Now consider one of those K recipients. The gossip layer at that node receives the query packet from its lower network layers and extracts the attached gossip messages, using them to update its gossip queue. It delivers the packet to the resource discovery protocol (in this case, K-walker), which determines that this node's CPU is not available. It must then forward the query packet to another node. It chooses one neighbor randomly (per the K-walker protocol), then passes it back down to the gossip layer as the new destination for the query packet. The gossip layer may then modify the destination based on information in its gossip queue. Finally it attaches gossip messages to the packet and sends it.

The recipient of that packet then extracts the gossip, modifies its gossip queue accordingly, and delivers the packet to its resource discovery layer. This node determines that it can accept the task, so it reserves a slot in its CPU task queue for twice the maximum network latency, $2L_{max}$ milliseconds. It creates a pledge packet to send to the originator of the query, then the gossip layer attaches gossip and sends it.

The originator of the query then receives the pledge after, at most, L_{max}ms. It extracts the gossip, then packages the task into a task packet to send back to the pledging node. The gossip layer attaches gossip and sends it. If other nodes along the path of any of the other $K - 1$ query packets also pledge their CPUs, this node will take no action on their pledge packets. This is of little consequence, since reservations are held for such a short time.

B. The Gossip Message

Gossip messages contain four fields:

- *node:* the node whose state information this message contains. In an actual implementation, this field may likely contain an IP address.
- *avail:* some value indicating the *availability* of the resource being shared, where a greater value indicates the resource is "more" available. In our simulations this field contains the number of open slots in *node*'s CPU queue (at the time this message was created).
- *create:* the time at which this message was created.
- *expire:* the time at which this message will expire. Simulations show that, in this work, the lifetime of gossip messages need not exceed 110 seconds (see Figure 2).

In an actual implementation, gossip messages may contain four bytes for *node*, one for *avail*, and eight each for *create* and *expire*. This is a total of 21 bytes for each gossip message. Simulations show the maximum benefit from attaching three of these messages to each packet (see Figure 3). The total of perhaps 64 bytes (including a single, one byte length field) piggybacked onto each packet in the network creates an overhead which we believe can easily be neglected.

C. Modifying Destination(s)

Function modDests(Query Packet, Node List) modifies the destination(s) of an outbound query packet according to messages in the gossip queue. To decide which of those messages most likely point to available resources, consider the relation *more-reliable*. Gossip message g1 is *more-reliable* than g2 when:

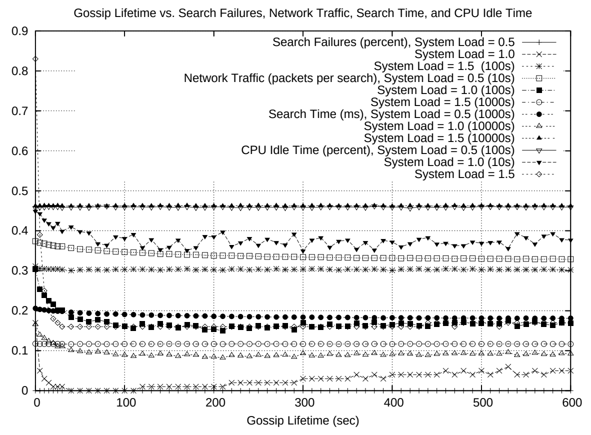


Fig. 2. This plot shows the effects of varying the lifetime assigned to gossip messages at three system loads. Most metrics stabilize at values greater than 110, and search failure increase past that point, so this is the value is chosen for further simulations. The units for the y-axis vary according to each set of data (refer to the legend).

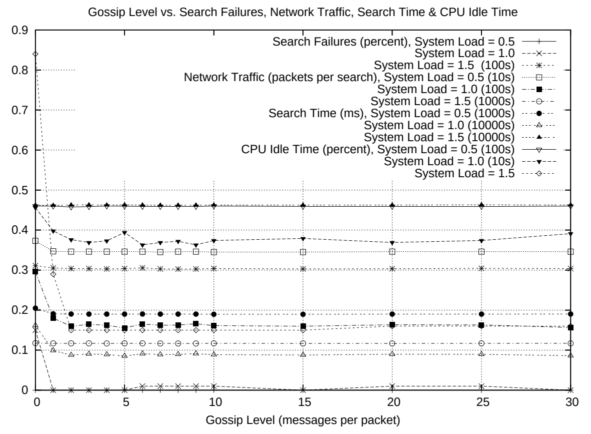


Fig. 3. This graph shows the effects of varying the number of gossip messages attached to each packet in the network at three system loads. The data show no benefit in attaching more than 3, which only costs about 64 bytes per packet. The units for the y-axis vary according to each set of data (refer to the legend).

- g1.avail $>$ g2.avail
- OR (g1.avail == g2.avail) AND (g1.expire $>$ g2.expire)
- OR (g1.avail == g2.avail) AND (g1.expire == g2.expire) AND (g1.create $>$ g2.create)

g1 is *less-reliable* than g2 when it is not more-reliable. According to simulations, this is the optimal order in which to apply these comparisons, *i.e. avail* first, *expire* second, and *create* last [19].

This function receives a list of destinations for a query packet from the resource discovery protocol. It then attempts to replace that list with its own of equal size. If there is not enough gossip to create a list that large, random elements from the original list remain to make up the difference. Since the number of destinations does not change, the flooding or non-flooding behavior of the resource discovery protocol is preserved. Notice that only gossip messages with a non-zero

```
Input    : A query packet p to send to a list of nodes
           dests
Variable: mods, an initially empty list of nodes
Variable: queue, the gossip queue kept at each node

repeat
    Let g be the "most-reliable" message in queue
    if g.avail == 0 then break
    adjust g.avail as though the query will be filled
    add g.node to mods
    if dests contains g then  remove g from dests
    else  remove a random node from dests
until mods.size == dests.size

for each n in dests do attachGossip (p, n)
for each n in mods do attachGossip (p, n)
```

Function modDests(Query Packet, Node List) This function is called by the resource discovery protocol to send a query packet to a list of nodes on the network. It is responsible for modifying the list according to gossip in the gossip queue.

value in the *avail* field are used to influence the destination of query packets. Also note that, given the nature of the *more-reliable* relation, a message reflecting zero availability will only be encountered after all non-zero values are exhausted.

D. Attaching Gossip

Function attachGossip(Packet, Node) attaches some number of gossip messages to every outbound packet, defined as *GOSSIP_LEVEL*. The attached messages reflect the current state of the node itself and gossip messages stored in its gossip queue. To decide which messages to attach consider another relation between gossip objects, *spread-better*. Gossip message g1 is *spread-better* than g2 when:

- g1.avail > g2.avail
- OR (g1.avail == g2.avail) AND (g1.create > g2.create)
- OR (g1.avail == g2.avail) AND (g1.create == g2.create) AND (g1.expire > g2.expire)

Again, simulations show that this is the optimal order in which to apply these tests [19].

When given a packet to send, this function starts the list of messages to attach with a new one reflecting the current state of the node itself. Then the spread-best objects are drawn out of the gossip queue and added to the list until either the queue is empty or the list has *GOSSIP_LEVEL* objects.

E. Extracting Gossip

Function extractGossip(Packet) extracts gossip messages from inbound packets to populate the gossip queue. The queue has a maximum size, *MAX_Q*. Also, the queue never contains multiple gossip messages about the same node, but instead retains only the most up-to-date information possible. To that end, consider a third relation between gossip objects, *outdates*, such that gossip message g1 *outdates* g2 when:

- g1.node == g2.node
- AND (g1.create > g2.create)

```
Input    : A packet p to send to a node dest
Variable: spread, an initially empty list of gossip
          messages
Variable: queue, the gossip queue kept at each node

add gossip about this node to spread
copy the "spread-best" message in queue to spread
while spread.size < GOSSIP_LEVEL do
    if everything in queue is in spread then break
    copy next "spread-best" message in queue to spread
attach spread to p
send p to the lower network layers
```

Function attachGossip(Packet, Node) This function is called by the resource discovery protocol and by modDests(Query Packet, Node List) to send packets to the lower network layers. It is responsible for adding gossip to those packets.

```
Input    : A packet p that has arrived at the node
Variable: queue, the gossip queue kept at each node

for each g in p.gossip do
    if queue contains h where h.node == g.node then
        if g outdates h then  replace h with g
    else  add g to queue;
while queue.size > MAX_Q do
    remove the "queue-worst" message from queue
deliver p to the resource discovery protocol
```

Function extractGossip(Packet p) This function is called by a lower layer to deliver a packet to the resource discovery protocol. It is responsible for extracting and processing gossip messages attached to the packet.

When given a packet this function steps through its attached gossip messages, replacing messages already in the queue with new ones that outdate them and simply adding the rest. Afterward, if there are more than *MAX_Q* objects in the queue, it trims off the least-reliable messages until the queue is no longer over-sized.

F. Remaining Details

Sections IV-C, IV-D and IV-E describe the functions of the gossip layer in detail, but with some simplifications for the sake of clarity. The algorithmic descriptions do not take into account the fact that gossip messages expire, nor that a node need not keep gossip about itself. Simulations handle the former by discarding expired messages whenever they are encountered. To correct the latter, a gossip message is never sent to the node it describes, nor are newly generated gossip messages ever copied back into their own nodes' gossip queue.

V. Evaluation

A. Simulation Setting

The main idea presented in this paper was tested by simulations run on a network of 12,000 nodes for 24 simulated hours.

Details of the simulator's design beyond what is described in this paper can be obtained from [19]. Simulations are based on a discrete event simulator which handles packet delivery between nodes. Network latencies are generated between $L_{min} = 10$ and $L_{max} = 100$ms using a uniform random distribution. To simplify simulation, there is no memory of differing latencies between differing pairs of nodes; *all* packets are delivered using the same latency distribution. Also, there is no packet loss.

Every node keeps a list of neighbors, for use by the resource discovery protocol, forming a power-law overlay topology. Neighbor connections are always bi-directional. The simulator generates and maintains these connections randomly, maintaining a power-law exponent of 2.0 throughout the simulation, which is within the ranges observed for the Gnutella network [7], [16], [17].

Simulations achieve network churn by scheduling the time between node births and deaths drawing from an exponential distribution so that session times average one hour, and very few are less than five minutes (the time required to complete one average-size task). To start each of these events the simulator randomly chooses whether it will be a birth or death, weighted to keep the network size near 12,000 nodes. If it chooses a death, it also chooses which node should die such that it approximates a power-law distribution for session times. When a node dies, it breaks down its neighbor connections, then continues forwarding packets until all gossip it spread about itself expires. In this way, nodes always die nicely; they never fail. Additionally, they simply drop any tasks they own at the time of death, rather than starting new queries for them. This also simplifies simulation without compromising the fairness of comparison between different simulations' results.

The simulator contains a single task generator which produces all the tasks that enter the network. Each task consists simply of a value indicating the number of CPU cycles that must be spent to complete it. This value is drawn from a Weibull distribution with shape parameter = 3 so that, on average, they take five minutes to execute on a 3GHz CPU. Each task is assigned to an initial node, chosen from a uniform random distribution. If possible that node accepts the task itself, *i.e.* if its CPU queue is not full. These cases generate no network traffic. When a node cannot accept a task locally, it initiates resource discovery.

Simulations use a form of the K-walker algorithm for their resource discovery protocol. Using that algorithm, a node wishing to search the network for an available CPU chooses K of its neighbors to which to send a query packet, duplicating choices whenever it has less than K neighbors. If, when a node receives a query packet, it has enough space in its CPU queue to accept another task, it reserves a slot in its CPU queue for $2L_{max} = 200$ms and returns a pledge packet to the inquirer. This guarantees that the inquirer can receive the pledge and respond before the reservation expires. If the queried node cannot accept the task itself, it chooses a random neighbor to which to forward the query packet. Each query packet can be forwarded up to 24 times, allowing it 25 hops total, according to the findings in [19]. The inquiring node sends the task to the first node from which it receives a pledge packet. In the event that none of the query packets elicit a response, the task is droppped after a set time.

B. Simulation Metrics

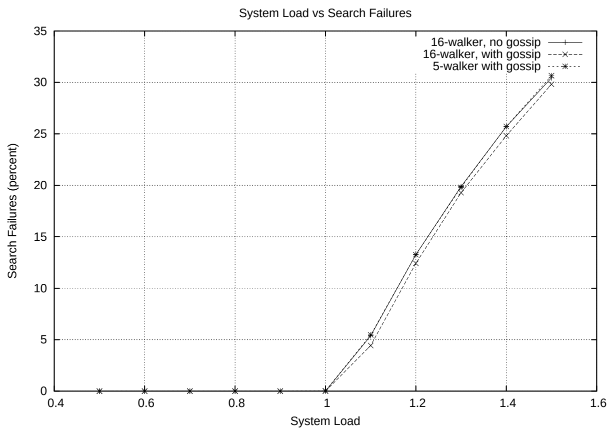
Four metrics are chosen to evaluate the success of the new gossip layer: search failure rate, network traffic, search time, and CPU idle time. A *search* encompasses all of the resource discovery activity initiated by a node at one time. So, for the K-walker protocol, all K walkers are part of the same search. *Failure rate* is the percentage of searches which do not find any available resources. *Network traffic* is measured in terms of the average number of packets generated per search. *Search time* is the time between a newly generated task arriving at a node which cannot accept it locally and entering the CPU queue of another node on the network. Searches that fail are not included in this metric. *CPU idle time* is measured as the percent of the total CPU time in the network which was spent idle. Note that lower values are more desirable for all of these metrics. The aim of these simulations is to determine how much network traffic the gossip layer could eliminate without increasing search failures.

C. Simulation Results

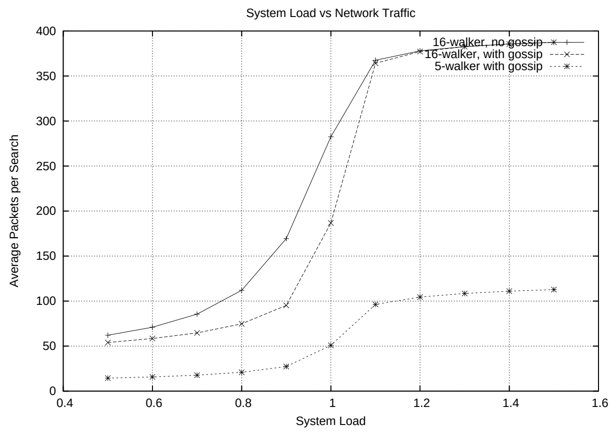
Figure 4 has four graphs containing the results of simulations that use the 16-walker protocol. 16 is within the range for K recommended by [20] for static resource discovery. Simulations indicate that by adding the gossip layer all four metrics can be slightly improved, or K can be reduced to 5 for dramatic reduction in network traffic and no increase in any other metric. Following is a discussion of each of the four graphs, then a brief overview of applying the gossip layer to flooding-style protocols.

Figure 4(a) shows the failure rates for simulations under a range of system loads. *System load* is the ratio of the total cycles required to complete all tasks to the total cycles offered by CPUs in the network. So, a system load of 0.5 indicates that the simulation generated enough tasks to consume half of the available CPU cycles in the network. The most striking feature of this graph is that the search failure rates for 16-Walkers without gossip and 5-Walkers with are virtually identical. Search failures rest on zero until the system load increases past 1.0, indicating that the protocols are very effective at finding resources even when they are relatively scarce. This does not mean, however, that they find resources *whenever* they exist. Notice from Figure 4(d) that CPUs are still 4.5 percent idle when the system load is 1.0. This is possible because 15 percent of tasks in the network are dropped because of network churn at this system load. As should be expected, the search failures in 4(a) increase almost linearly after $x = 1.0$.

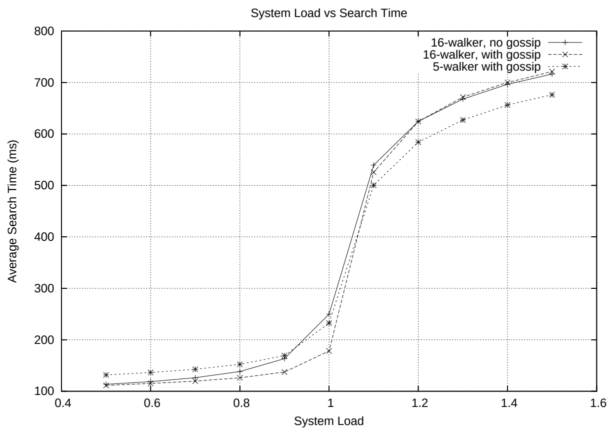
Figure 4(b) shows the network traffic generated by the three protocols under a range of system loads. When $x < 1.0$, while virtually all searches succeed, the network traffic increases following the equation $K(hops + 1) + 1$, where *hops* is the average number of hops each walker takes before finding



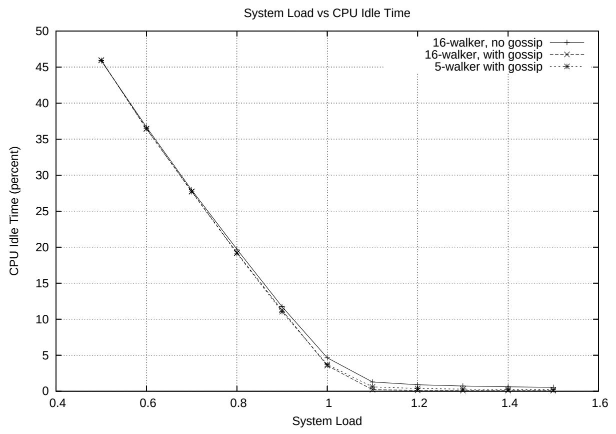
(a) System Load vs. Search Failure Rate



(b) System Load vs. Network Traffic



(c) System Load vs. Search Time



(d) System Load vs. Idle Time

Fig. 4. Under a variety of system loads, a 16-Walker algorithm without gossip and a 5-Walker algorithm with gossip yield the same failure rates (4(a)), search times (4(c)), and CPU utilization (4(d)), while a 16-Walker with gossip yields performs slightly better. The 5-Walker algorithm with gossip produces much fewer packets than both 16-Walker algorithms (4(b)).

an available resource. Each walker which finds a resource generates one packet for each hop it takes, then prompts the generation of exactly one pledge packet. So, a single walker creates $hops + 1$ packets. Since there are K walkers for every search, each one produces $K(hops + 1)$ packets, and every successful search additionally produces one task packet. Since adding the gossip layer to the 16-Walker protocol reduces network traffic, as shown in the figure, and K is held constant, the gossip layer must reduce the average number of hops each walker takes. Reducing K to five of course reduces network traffic by 11/16ths, and the data show that it actually reduces it a little further. The further reduction can be explained by the fact that fewer walkers will arrive in rapid succession at the immediate neighbors when initiating queries, so that fewer walkers will have to hop past the cpu reservations made by other walkers from the same search. When $x > 1.0$, network traffic asymptotically approaches the point where every walker fails after 25 hops, $25K$. For both 16-Walker algorithms, this asymptote is $y = 400$, and for 5-Walkers, $y = 130$.

Figure 5 shows the percent reduction in network traffic that 5-Walkers with gossip makes over 16-Walkers without gossip. The data represented is one minus the average traffic generated by a 5-Walker search with gossip divided by the average traffic generated by a 16-Walker search without gossip. It shows that the gossiping protocol yields between a 70 and 85 percent improvement, with the most improvement when resources are scarce, but still available.

Figure 4(c) shows the length of time, on average, between initiating a successful search and its task's admittance to a CPU queue. Under all system loads, the 16-Walker algorithm with gossip out-performs both others. During low system loads, the 16-Walker protocol without gossip out-performs the 5-Walker. In this case, almost every neighbor has available resources, so it is simply a one-hop race between the walkers to see which will find them first, and there will more likely be a faster hop in a set of 16 than 5. Once the system load increases past 0.9, however, the 5-Walker protocol begins out-performing the 16-Walker without gossip. The average difference between the results for those two protocols is 13 milliseconds, favoring the 5-Walker. These data show a transition between two asymptotes. At low system loads, when resources are almost always available at any node's immediate

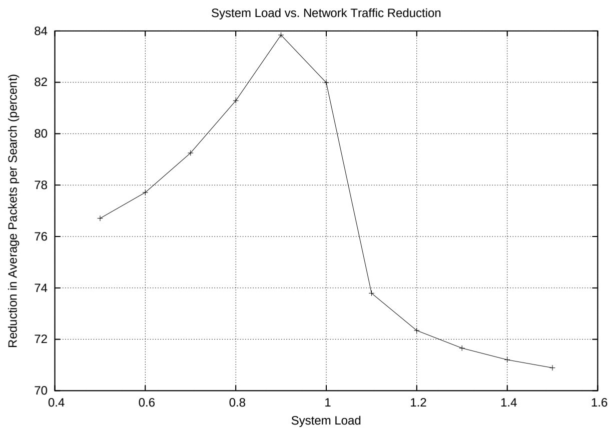


Fig. 5. The improvement in network traffic by decreasing K from 16 to 5 and adding the gossip layer, given by the equation: $100 \times$ (1 - {avg. packets per search for 16-Walker without gossip} / {avg. packets per search for 5-Walker with gossip}

neighbors, resource discovery completes after three hops: one each by a query packet, pledge packet, and task packet. Under high system loads, when resources are very scarce, so many walkers are in the network that the one that happens to visit a node at the instant its CPU becomes free wins its resource. Since we assume that resource availability cannot be predicted, this must happen completely by chance.

Figure 4(d) shows the average percent of time a CPU on the network is idle. The three sets of data are nearly the same, with the gossiping protocols slightly out-performing the non-gossiping one. The improvement comes mostly from the addition of gossip; the increase of K from 5 to 16 makes very little difference. Idle times fall linearly when $x < 1.0$, and remain at almost 0 percent when $x > 1.0$.

Random-walk-style protocols are best suited for finding one or a few resources on a network, but finding many at once for parallel tasks will be another, perhaps more important application of resource discovery for peer-to-peer computing. Simulations of a home-grown flooding-style algorithm show that the gossip layer also benefits this class of protocols. Using that algorithm gossip reduces average search failures by 7 percent, search time by 2 percent, and CPU idle time by 23 percent. Network traffic is slightly increased, but this could be considered better, since the increase comes entirely from pledge packets. When the system load is 1.0, search failures improve by 29 percent, search time by 3 percent, and CPU idle time by 26 percent. Refer to [19] for more a more detailed description of these results and the protocol that was used.

VI. Conclusion

This work presents the viability and effectiveness of inserting a new layer into the network protocol stack which reduces the amount of network traffic created by resource discovery. Using the K-walker protocol, the gossip layer is able to reduce that traffic by 71-84 percent by decreasing K from sixteen to five. This decrease does not increase search failures, search time, or CPU idle time. By all metrics it also improves results for a flooding-style protocol. Simulations only generated network traffic associated with the resource discovery activities in the network; other packets can also be passed through the gossip layer, which should improve resource discovery activities even further.

Overall, this work is one important step in the direction of a resource discovery protocol tailored for dynamic resources. We believe gossip can play an important role in such a protocol, and we welcome further improvements.

References

[1] "Seti@home: Search for extraterrestial intelligence at home." [Online]. Available: http://setiathome.ssl.berkeley.edu/

[2] "Grid.org - grid computing projets." [Online]. Available: http://www.grid.org/home.htm

[3] "Distributed.net: Node zero." [Online]. Available: http://distributed.net/

[4] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," 2001. [Online]. Available: citeseer.csail.mit.edu/ripeanu01peertopeer.html

[5] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. Huberman, "Search in power-law networks," *Physical Review*, vol. 64, pp. 46,135–46,143, March 2001.

[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like p2p systems scalable," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM Press, 2003, pp. 407–418.

[7] F. Banaei-Kashani and C. Shahabi, "Criticality-based analysis and design of unstructured peer-to-peer networks as 'complex systems'," in *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 351.

[8] N. Sarshar, P. V. Roychowdury, and O. Boykin, "Percolation-based search on unstructured peer-to-peer networks," in *IPTPS*, 2003.

[9] G. Lanfermann, G. Allen, T. Radke, and E. Seidel, "Nomadic migration: A new tool for dynamic grid computing," in *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 429.

[10] ——, "Nomadic migration: Fault tolerance in a disruptive grid environment." in *CCGRID*, 2002, pp. 280–281.

[11] A. Iamnitchi and I. Foster, "On fully decentralized resource discovery in grid environments," in *International Workshop on Grid Computing*. Denver, Colorado: IEEE, November 2001. [Online]. Available: citeseer.ist.psu.edu/iamnitchi01fully.html

[12] D. Zhou and V. Lo, "Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system," in *IEEE International Symposium on luster Computing and the Grid, 2004*, April 2004, pp. 19–22.

[13] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, "Cluster computing on the fly: P2p scheduling of idle cycles in the internet." in *IPTPS*, 2004, pp. 227–236.

[14] S. Nandy, L. Carter, and J. Ferrante, "Guard: Gossip used for autonomous resource detection," in *IPDPS*, 2005.

[15] D. Stutzbach and R. Rejaie, "Characterizing churn in peer-to-peer networks," University of Oregon, Tech. Rep. CIS-TR-05-03, June 2005. [Online]. Available: http://www.cs.uoregon.edu/˜reza/PUB/tr05-03.pdf

[16] D. Zeinalipour and Y. T. Folias, "A quantitative analysis of the gnutella network traffic," 2002. [Online]. Available: http://www.cs.ucr.edu/˜csyiazti/courses/cs204/project/html/final.html

[17] M. Jovanovic, F. Annexstein, and K. Berman, "Modeling peer-to-peer network topologies through "small-world" models and power laws," in *IX Telecommunications Forum, TELFOR 2001*, 2001.

[18] "The weibull distribution." [Online]. Available: http://www.weibull.com/LifeDataWeb/the_weibull_distribution.htm

[19] E. Simonton, MS Thesis, Department of Computer Science, Michigan Technological University, Jan 2006. [Online]. Available: http://www.cs.mtu.edu/˜ersimont/thesis.pdf

[20] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *ICS '02: Proceedings of the 16th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2002, pp. 84–95.