Comparison of Name Resolution Algorithms
cs694 Project Report

G. P. Phillips
graham@isi.edu

December 11, 1997

1 Introduction

This project is motivated by the active network research being undertaken by DARPA together with various research groups in the United States. An active network is different from contemporary networks in that code is carried in packets and executed at the network layer. Various researchers have suggested that code should be dynamically loaded on demand. This approach would allow large code to be separated into many smaller capsules, so that only the main capsule need be explicitly sent into the network (the other capsules would be demand-loaded by the main capsule). The question we would like to answer in this memo is how an efficient demand-loading mechanism could be constructed.

We separate the demand loading mechanism into two modules: a name resolution module and a transfer module. The name resolution module performs a function that maps the name of an object to its location, while the transfer module is responsible for transferring the object from the remote location. This memo considers the mapping function only. The transfer module might consist of a reliable transfer protocol such as FTP or light-weight transfer protocol proposed by the ANTS toolkit[2].

2 Model for name resolution algorithms

Our goal is to compare various algorithms that resolve a given object's name to the objects' location. Each resolver algorithm, although implementation details may differ, performs the following resolution function.

> Given the name of an object (for example, a file name), the resolver algorithm returns an address (for example, the node address) of the

object's location.

In general this function is a one-to-many mapping because objects may be replicated by various nodes. The goal of our analysis is to find an algorithm that performs this mapping with the constraints that (i) the replica is found quickly and (ii) the communication bandwidth requirement is small. Our goal is to compare various compromises between these two constraints.

2.1 Assumptions

It is assumed that all objects have globally unique names. How these objects are created and named is outside the scope of this memo. An application executing on a node makes a request for the resolution of an object's name by sending a resolve-request packet. We call the node that sends the request as the *requester* node. The resolve-request packet contains the name of the requested object and the address of the requester node.

Every object is associated with a *source* node, which maintains a stable copy of the object. The fully qualified name of an object includes its associated source node, which ensures that the requester can always locate a copy of the object.

The resolve-request packet is forwarded hop by hop and is processed at each node. Note that the requester can extract the source information from the object's name, however, this does not imply that the resolve-request packet reaches the source because the request may be serviced by other nodes.

A resolver algorithm executes on each hop and processes arriving requests. The resolver algorithm determines whether the requested object resides on the particular node. If the object does not reside on the node, the request is forwarded, otherwise a resolve-reply packet is sent to the *requester*. Once the requester has received the resolve-reply packet, it can then fetch the object from the node that generated the resolve-reply packet.

In our analysis, we investigate resolve-request packets, starting from their generation to the time that the first resolve-reply packet is generated.

2.2 Input parameters

The input parameters of our model are given as follows:

- the name of an object,
- the probability p that a node in the graph has an object replica,
- the resolver algorithm (see section 2.4),
- one or more topologies.

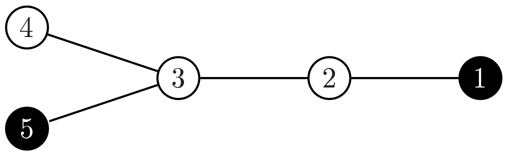

Figure 1: Simple topology

2.3 Output parameters

Given a particular set of input parameters, two performance metrics are computed:

- the number of hops from the requester node to the source and
- the total number of resolve-request packets required to resolve the given object's name.

A performance metric is computed for a single requester/source pair on a single topology. However, these metrics may be averaged in the following ways:

1. all nodes acting as a requester,
2. all nodes acting as a source and all nodes acting as a requester,
3. multiple topologies.

2.4 Description of resolver algorithms

In this section we develop five algorithms for resolving an object's name to a node address. We also provide an example of how each algorithm works and how the performance metrics are calculated for the simple topology shown in figure 1. In each example, we consider that node 4 makes a request for the object called node1/A. We call this name the fully qualified name because it consists of the node address (in this case node1) and the object name (in this case A). Our model assumes that there is at least one copy of the object at the source (in this case node 1). In the figure, the black nodes denote that a replica is present at the node.

2.4.1 Simple discovery

The simple discovery algorithm retrieves objects from the source without looking for replicas. For example, an application on node 4 requests object node1/A. Because node 4 does not have object A, it retrieves the object directly from the source (node 1). The number of hops between requester and source is 3. There is no need to send resolve-request packets because the name can be resolved locally and therefore the number of resolve-request packets sent is 0.

2.4.2 Recursive discovery

This discovery algorithm processes resolve-request packets on a hop by hop basis. Each node receiving a resolve-request packet checks whether a replica is present. If not, it forwards the request to the next hop towards the object's source. The search terminates when the request is received by a node that has a replica. For example, an application on node 4 requests object node1/A. Because node 4 does not have a replica, it forwards the request to node 3 (which is the previous hop towards node 1). Node 3 also does not have a replica, so it forwards the request to node 2. The recursive search terminates at node 1 because this node has a replica. The number of hops between requester and source is 3 while the number of resolve-request packets sent is 3.

2.4.3 Short-circuit discovery

This algorithm is similar to the previous algorithm except that the resolve-request packet is not processed at every hop towards the source. Instead the request is processed only by the first hop. In our example where node 4 sends a request, when the request reaches node 3, the request is forwarded directly to the source (node 1). The number of hops between requester and source is 3 while the number of resolve-request packets sent is 2. Note that we are counting the number of requests from node 3 to node 1 as a single request because the request is not processed at node 2.

The motivation for this algorithm is given as follows. We expect that in an active network, object will be replicated on each node from the source to requester, while in contemporary networks, objects will be replicated only at requesters. We expect that this algorithm will work well in both environments.

2.4.4 Neighbour discovery using expanding ring search

This algorithm uses an expanding ring search algorithm to find nearby replicas. In the first round, a request packet is sent to each neighbour, and in the next round, a request packet is sent to each neighbour's neighbour. In each round, requests are only sent to those nodes that have not already received a request. Our algorithm is performed in a centralized manner and we ignore the overhead required to establish a distribution tree to ensure that nodes do not receive multiple requests. The algorithm terminates when a request packet is received by a green node.

For example, node 4 performs an expanding ring search for object A. The number of hops between the requester and replica is 2 while the number of resolve-request packets sent is 4. The result of 4 requests is calculated as follows. In the first round, 1 request is sent along link (4,3), while in the second round, 3 requests are sent along links (4,3), (3,2) and (3,5) yielding a total of 4 requests.

2.4.5 Neighbour discovery with advertisements

This algorithm employs a proactive approach in that when a node obtains a replica, an advertisement of this new replica is broadcast to all other nodes in the network. Obviously this algorithm is impractical for large networks but it provides a first approximation of advertisement schemes.

In our example, we assume that when object A was replicated on node 5, its presence was advertised to all nodes in the topology. On receiving an advertisement, each node will update its reference of the location of the particular object. The reference will be updated only if the node sending the advertisement is closer than the location of the current reference. Now when node 4 requests object `node1/A`, it knows that object A is located at node 5. The number of hops between the requester and the replica is 2. The number of `resolve-request` packets sent is 4 from the 4 advertisement packets sent to each node.

3 Implementation details

3.1 Simulation or calculation?

The computation that we implemented should be classified as a *calculation* rather than a *simulation* because an event scheduler is not used. Instead, the computation performs a graph calculation by traversing a graph and calculating various values.

The reasons for not using a discrete event simulator are twofold. First, we were not initially concerned with how latency would be affected by interactions between packets sourced by different nodes. Our assumption was that the latency of resolving a name would be dominated by the way in which requests are propagated through the network, rather than by packets being delayed due to processing in nodes. Second, we were interested in comparing our algorithms for very large topologies. We concluded that larger networks could be investigated if nodes were not represented by “heavy-weight” objects with associated processes such as packet multiplexors, processing routines and forwarding routines.

3.2 Methodology

All topologies are calculated using the Georgia Tech ITM topology generator[3]. The output of the ITM generator is a description of a single topology in Stanford GraphBase format. The GraphBase topology file is then converted into a format suitable for input to our `ns` program[1].

The overall computation is summarized as follows (while pseudo code is given in figure 5). A static global routing table is constructed. Each node in the topology is either populated with an object called A or not. In particular, for each node in the graph, a random variable is extracted from a uniform distribution with the domain $[0, 1)$. If this random variable is less than p (an input parameter to the model) then the node is populated with object A.

The two performance metrics are then calculated for a given resolver algorithm. As we have stated above, the performance metrics may be averaged over multiple topologies and over one or more requester nodes. The following data structures are used, (i) the global routing table, (ii) a global structure that maintains the names of those objects that are replicated at each node and (iii) a global structure for collecting the two metrics. Note that the representation of the topology is given by the global routing table.

3.3 Modifications and contributions to ns

A modification was made to ns to speed up the processing time of the compute-routes function. We created a new function compute-global-routes that performs only the functionality that we require. This simple modification reduced the running time of our computation by more that a factor of 2.

An OTcl class was provided to John Heidemann for inclusion into the ns distribution. This class, called Histogram, allows histograms of integer values to be collected. Values may be added to a Histogram instance as a particular computation proceeds. At any time, the Histogram instance can be queried for various statistics, including mean, max and min. Furthermore, a mechanism is provided for computing arbitrary statistics, for example, variance.

3.4 Examples of particular computations

In this section we show graphs that compare the two performance metrics for the various resolver algorithms. The input parameters for the example computation are given as follows. We consider a single flat topology with 100 nodes and a single named object called A. The probability p (that each node has object A) is incremented from 0 to 1 in increments of 0.2. For each value of p, we select a single arbitrary source (node 13) and we average metrics over all nodes in the network acting as a requester.

Our motivation for modeling a single object is that we assume that interactions caused by other objects will have a small impact on the latency of resolving a given object's name. However, our assumption may be invalid if large numbers of requests are being processed by a few resolver processes, because under this condition, requests may be queued while other requests are processed. The author acknowledges that the above assumption can only be asserted using a more detailed

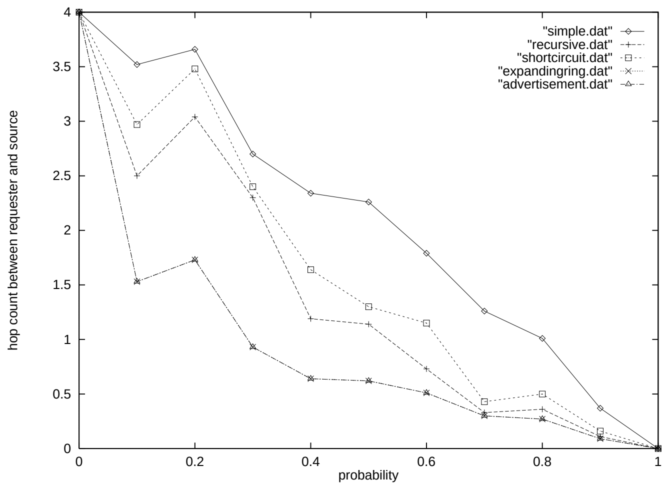

Figure 2: Number of hops

model.

Our motivation for selecting an arbitrary source is that we felt that interesting features might be smoothed out by to much averaging.

Figure 2 shows the average number of hops from requester to a replica of a single object for the 100 node flat topology. Each curve represents one resolver algorithm. Note that the *neighbour discovery using expanding ring search* is identical to the *neighbour discovery using advertisements* because both algorithms locate the nearest replica.

Figure 3 shows the total number of request messages that are used to resolve the name of a single object for a 100 node flat topology. The *neighbour discovery using advertisements* uses 99 advertisement messages. Actually, this value of 99 is a lower bound because each node (except for the node making the advertisement) will receive a copy of the message, but many more messages may be required within the network to ensure their delivery. The *neighbour discovery using expanding ring search* is very costly when the object is not widely replicated. Sparsely replicated objects will typically be far apart and therefore many rounds of the expanding ring search will be required to resolve an object's name. Consequently, there will be many resolve-request packets, particularly near the epicentre of the requester node. Figure 4 shows the same data as figure 3 but is drawn to a different scale.

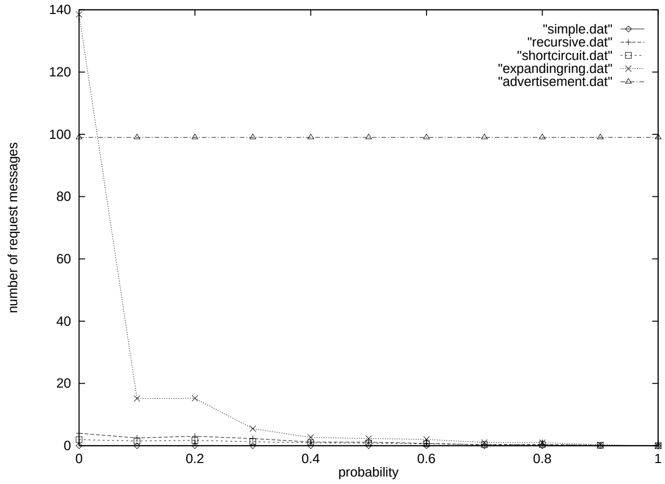

Figure 3: Number of messages

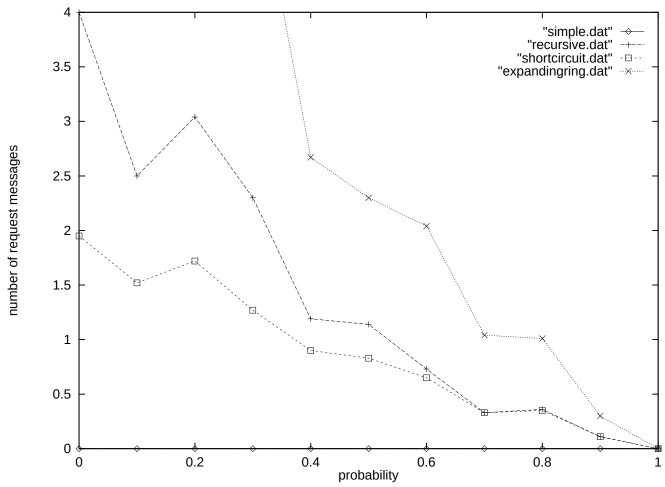

Figure 4: Number of message (same as figure 3 but drawn to a different scale)

4 Results

4.1 Hop count

Two of the resolver algorithms, *neighbour discovery using expanding ring search* and *neighbour discovery using advertisements*, provide a lower bound for the hop count metric. This lower bound is not surprising because both algorithms were constructed to have this property. Their is not a significant difference between *recursive discovery* and *short-circuit discovery* although *recursive discovery* consistently outperforms *short-circuit discovery* by a narrow margin. The reason for *short-circuit discovery* being slightly worse is because this algorithm looks only one hop away before going directly to the source. If an object replica was present two hops away (towards the source) then *recursive discovery* would locate the replica while *short-circuit discovery* would not. These two algorithms would appear to yield hops metrics no more that 2 or 3 times the lower bound. This result is encouraging but we should be careful to note that the metrics are averaged and therefore worst cases are hidden by this averaging. Further analysis should focus on worst values rather than average values.

4.2 Communication overhead

The total number of `resolve-request` packets is prohibitive for *neighbour discovery using expanding ring search* and *neighbour discovery using advertisements* and therefore refinements will be necessary before these algorithms could be deployed in real networks. One refinement could be to limit the scope of the expanding ring search and the scope of advertisement.

Obviously one cannot perform better than *simple discovery* in terms of the number of `resolve-request` packets, but this algorithm has the largest hop count of all the other resolver algorithms. The *recursive discovery* algorithm scales linearly with the diameter of the network while the *short-circuit discovery* algorithm never forwards more than 2 `resolve-request` packets for each name to be resolved. Note that we counting requests in terms of number of times the request is *processed* along the path (see section 2.4.3). We believe that the *recursive discovery* and *short-circuit discovery* algorithms are the most promising.

5 Hindsight and future work

Probably the hardest task of the project was verifying the output once the code was running without run-time errors. One can easily fall into the trap of assuming that the output is correct if one thinks the output is correct. To be sure, several errors were found while testing and again while documenting the project.

The work could be extended in several ways. Most importantly, a couple of the resolver algorithms should be selected for further refinement. For example, the *neighbour discovery using advertisements* algorithm could be refined by reducing the maximum circumference of advertisements, while the *neighbour discovery using expanding ring search* could be refined by improving discovery for sparsely replicated objects.

The work could further be extended by analyzing the interaction of multiple objects (such as queueing of requests at nodes) and modeling the access patterns for objects in a real network.

6 Code availability

The source code and our raw simulation output files are available at `<URL:http://www.usc.edu/~grahamph/contrib/cs694_97/ns_stuff>`. The source code is contained in the files,

`Discovery.tcl`	`Set.tcl`
`Histogram.tcl`	`foxTypedOpts.tcl`
`nsDiscovery`	

while the raw simulation output are contained in the files,

`advertisement.dat`	`recursive.dat`	`simple.dat`
`expandingring.dat`	`shortcircuit.dat`	

The top level (or main) file is `nsDiscovery`. This file requires various command-line options, but if no options are given, a help page is displayed.

References

[1] Kevin Fall, Sally Floyd, and Kannan Varadhan. Network simulator. http://www-mash.CS.Berkeley.EDU/ns/ns-documentation.html, 1997.

[2] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. http://www.tns.lcs.mit.edu/publications/openarch98.html, 1997.

[3] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee. GT-ITM: Georgia Tech internetwork topology models. http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html, 1997.

```
CALCULATE-METRICS($prob, $directory, $resolver, $single-source, $single-requester)
# $prob is the probability that a node has object A
# $directory is a directory in the filesystem
# $single-source is either a non-negative node ID or −1
# $single-requester is either a non-negative node ID or −1
#
foreach $topology in $directory
    clear all objects from $topology
    foreach $node in $topology
        let $rand be a uniform random variable from 0 to 1
        if ($rand < $prob) then
            $node has a replica of object A
        endif
    end
    let $n be the number of nodes in $topology
    if ($single-source ≥ 0 and $single-requester ≥ 0) then
        calculate metrics from $single-requester to $single-source using $resolver
        store metrics as a function of the $p and $n
    else if ($single-source ≥ 0) then
        foreach $node in $topology
            let $requester be $node
            calculate metrics from $requester to $single-source using $resolver
            store metrics as a function of the $p and $n
        end
    else
        foreach $node in $topology
            let $source be $node
            foreach $node in $topology
                let $requester be $node
                calculate metrics from $requester to $source using $resolver
                store metrics as a function of the $p and $n
            end
        end
    endif
end
calculate metric statistics as a function of $p and $n
print metrics
```

Figure 5: Pseudo code for calculating metrics