Reconstructing Interconnections on Disconnected Mobile Hosts

Dongho Kim and B. Clifford Neuman

Information Sciences Institute
University of Southern California

Abstract—Today’s distributed systems are very dependent on access to fixed infrastructure for their coordination, sharing, and management. Though communication can occur without central coordination, services for accessing distributed resources are very limited unless the nodes have been organized in advance. If there are nearby resources available but unknown to the nodes that need the resources, then there should be ways of letting the nodes discover and access the resources without central coordination.

This research focuses on organizing disconnected nodes so that they can operate using cached data without access to the fixed infrastructure. Each object stores relationships with other objects using unique names and tags that enable transparent access to distributed resources.

To demonstrate our approach, we applied the technique to an e-mail messaging system for intermittently connected users. In this system, not only do disconnected nodes discover one another and deliver the message directly, but messages are automatically relayed among hosts when only some users synchronize with the fixed infrastructure.

A new grouping mechanism is introduced to reduce the overhead of discovering objects among the hosts. Evaluation results show that the mechanism improves the performance of the system.

1. Introduction

Today, the predominant use of wireless computer communications is to connect mobile computing devices to fixed (central) infrastructure, such as file server systems[1]. In the very near future, we expect to see wider use of wireless communications, and the cost will drop significantly. Without radical changes in the mechanism used to organize computer systems we will fail to take advantage of the dynamic nature of the communication medium and the resulting systems will suffer from poor availability. If we redesign the underlying mechanism for organizing such systems, we can have systems that are more autonomous than their wired counterparts, with improved availability.

In today’s systems, support for distributed operations, including coordination, sharing, and management, usually requires access to fixed infrastructure. Though communication can occur without central coordination, fixed infrastructure is often necessary to translate names, or to find the objects or resources with which to communicate. While disconnected, system can operate using cached data, but only when the names and relationships were known in advance and cached along with the data itself. Coordinating changes, even among communicating disconnected systems, usually requires access to the fixed infrastructure for synchronization.

While the mechanisms for coordination and management of systems today require access to fixed infrastructure, much of the data that is actually needed to do this coordination is already cached on mobile nodes. The problem is that it is scattered across many different nodes. What is missing in today’s systems are the *assembly instructions* for putting the pieces back together when one does not have
access to all of the pieces.

Our research enables reconstruction of interconnections between information resident on multiple disconnected nodes. The nodes discover one another, and coordinate among themselves to reconstruct the relationships between the collection of cached data and replicas present on the communicating nodes. In many cases, sufficient data does exist on the nodes already, and appropriate relationships can be reconstructed to enable the same coordination between these nodes that would have been possible if they were still connected to the fixed infrastructure.

The implications of this are significant: it allows one to "drop-in" computing and information infrastructural components that organize themselves into stand-alone systems supporting functionality that systems today could only achieve with communication back to fixed nodes. When a user drops an object into the system while disconnected, the system can organize itself so that the object can be transparently accessed as if the system is connected to the fixed infrastructure.

Since enabling this for an entire enterprise is a significant undertaking, here we focus on a small scale application to demonstrate the functionality in a self-organizing storage system optimized for use in an experimental e-mail messaging system for intermittently connected users.

For example, in some of today's messaging systems, when traveling with a colleague (each with a mobile host configured with an e-mail client), messages sent between the two users are not delivered until the sender has first connected to the home system to upload the message and the recipient subsequently connected to retrieved the messages. In the system we developed, two hosts discover one another and deliver the message directly without coordination with the fixed infrastructure. Also certain messages are automatically relayed between hosts when only one user synchronizes to the fixed infrastructure.

To enhance the performance of the system, we introduced a new mechanism that groups objects, then name the groups with unique identifiers. By grouping the scattered objects over disconnected hosts, the system reduces the overhead of discovering objects among the distributed and disconnected hosts. A simulation result obtained from a commonly expected data set shows up to 93% of reduced communication bandwidth requirement with the grouping mechanism.

We start by presenting the basic concept of the system and illustrates the core idea using a web example. The next section reviews the messaging system on which the system was implemented.

The design of the system is presented in Section III. The grouping mechanism is described in Section IV. Implementation of the system is briefly presented in Section V. The performance results are shown in Section VI. Related work and conclusion are described in Section VII and Section VIII, respectively.

II. Basic concept

This section describes basic model of how we can organize disconnected nodes and let them share resources.

While the mechanisms for coordination and management of systems today require access to fixed infrastructure, much of the data that is actually needed to do the coordination is already cached on mobile hosts. However, the data is scattered across many different hosts. Today's systems are missing the "assembly instructions" that can put the pieces back together when a user has access to only some of the pieces.

In our system, the information needed to reassemble components of the infrastructure to each other are attached to the components so that when they come in contact with one another, they organize themselves into a working sub-
set of the original system. As new data objects become available to one node (upon partial connection of that node with others) the “assembly instructions” carried with the data support integration of that data into the system view maintained by other disconnected nodes.

By providing these assembly instructions, and attaching them to each piece of the infrastructure, it becomes possible to reconstruct the interconnections between those parts of the infrastructure that are available. These assembly instructions are provided through the use of URNs (Uniform Resource Names)[2], [3], [4] for the objects in the system. Each object stores a list of the relationships with other objects using unique names[5], [6]. This list of relationships is bi-directional, meaning that an object knows not only about the objects it references, but also about the objects that reference it.

Using this approach, it is possible to reconstruct a connected graph of the relationships between all accessible resources on the communicating nodes. If the relationship graph were disconnected, it might not be possible to configure or utilize some resources, even though they are on a node with which one can communicate.

As long as relationships are maintained bidirectionally, it should be possible to reconstruct a connected graph. This is the case because each of the nodes in the system that will be potentially disconnected must maintain references to objects in system infrastructure, and they will cache objects that form a complete path from central nodes in the infrastructure. Two disconnected nodes that can communicate with one another will, except in rare circumstances, maintain cached objects (and relationships associated with those objects) that will overlap, providing connectivity to the graph that is the union of the local and cached objects on both nodes.

For ease of finding objects, we use a directory service[7] which provides a set of convenient access functions for each node in the system. The directory service stores local data and maintains the relationships between those objects and objects elsewhere in the system. It maintains information about cached objects whether cached locally, or cached elsewhere if the local node references the object and it is cached on a node with which the current node communicates.

Each node carries information about relationships between the node and other nodes in the directory service. Each node has pointers to ancestors including its parents in the directory which are represented by their unique identifiers. Also, each node has a set of functions that help it reconstruct relationships when it comes in contact with other pieces.

Each node maintains a discovery directory for the resources it holds, and the resources it knows how to reach. When looking for a data object that is not in its own cache, a system consults its discovery directory looking for an entry for a communicating node that knows about the nodes containing the desired object. If it finds an entry, it contacts the node identified to locate the object. Otherwise, it broadcasts a message locally asking if any other communicating nodes have knowledge of the nodes. Section IV describes the mechanism that improves performance by reducing the number of broadcasts.

To illustrate the core idea, we show a scenario of sharing web pages in the example below. Although we describe the idea in terms of links on the web pages, the real concepts can be directly mapped to the elements in directory services which also have hierarchical links. The web example is provided only to make the idea more intuitive.

Figure 1 illustrates how we can reconstruct interconnections between disconnected hosts using discovery directory services so that the hosts can share resources. Figure 1-(a)
shows some of the web pages under www.usc.edu. Figure 1-(b) shows two mobile hosts (host A and host B) that are disconnected from the rest of the world, but still directly connected each other.

The user of host A has a link to a web page (www.usc.edu) that has links to other pages. He only has some of the pages (www.usc.edu and www.usc.edu/dept) cached on his machine. Another person on host B also happens to have the USC pages cached through the top level, but he has not cached all the reachable pages either. However, he might have cached some of the pages he tends to access a lot in the machine, such as the computer science department web page (www.usc.edu/dept/cs). The two users are already connected together and the first user would like to access the page he never had before, but the other user has.

If the two hosts can communicate with one another, and if the user on host A wishes to use data cached on host B, the system must identify this data. Ideally, the same names and same structure (e.g., www.usc.edu/dept/cs) that applies when connected to the fixed infrastructure would be used. However, host A did not cache those parts (www.usc.edu/dept/cs) of the directory service needed to identify objects on host B in advance.

In a self-organizing system, however, host A will be able to reconstruct entries from host B’s side of the directory service using data cached on host B, so long as the cached entries form a connected graph (they are connected at the shared node www.usc.edu). Since both hosts carry information about the relationship between nodes and maintain a discovery directory for the resources they know how to reach, when host A requests information on www.usc.edu/dept/cs, host B will be able to find its entry in its directory and locate the objects.

The result of this exercise is the reconstruction of a subset of the directory graph formed by the union of the cached entries. In this example, host A cached the pages circled with thin lines and host B cached the pages circled with thick line only, as shown in Figure 1. However, by cooperating with each other, host A and host B can reconstruct the complete view (in Figure 1-(a)) by themselves even though each of them has only part of the complete view.

III. DESIGN: RECONSTRUCTION

In this section, we describe how we organize disconnected objects in the messaging application by maintaining their URNs and inter-relationships among them. We first define our URN and the other tags that we keep with each object (message or mailbox) and illustrate how they support reconstruction of the interconnection using a set of representative scenarios.
A. Identifying nodes: Defining URN

To build a messaging application on top of the system described, we represent mailboxes as directories and messages as files. Each object (a mailbox or a message) has a unique URN (Uniform Resource Name). A URN of an object is unique during the object's whole lifetime. URNs are used to provide stable names for resources whose characteristics may vary over time. For instance, a URN may be used as a stable reference to a web page. The web page can be moved, replicated, or modified and still remain accessible using the same URN. In contrast to URLs (Uniform Resource Locators) [2, 8] which contain location information, the location information and other characteristics of a URN are provided by resolution servers. We define a class of URNs that embed this unique identifier.

We define our URN as the following:

urn:x-pmn:timestamp.pid.username

As defined in [3, 4], all URNs start with leading “urn:”. We assigned “x-pmn” as the Namespace Identifier (NID), where leading “x:” shows that our URN is experimental and not explicitly registered with IANA (Internet Assigned Numbers Authority) and “pmn” is an arbitrary string we assigned (short for Prospero Message Name). Our Namespace Specific String (NSS) comes after the second colon. The format for the time stamp is yyyymmddhhmss. Unix process id, and user name are appended using “.” as a delimiter. We use this format for both messages and the mailboxes. For example, in Figure 2, the message “M” has URN as “urn:x-pmn:20001221103011.1002.user1”. The information for other replicas are stored as URLs. In Figure 2, “URL0” is the URL for the replica on ton.isi.edu.

B. Tagging

In addition to URNs, the following data structures are stored with each object.

- Descendent links: Each object has meta-data defining its relationship with other objects. In a mailbox, this is a list of links to the individual messages. A sub-mailbox of a mailbox is a descendent of the mailbox. The target of the link contains the object location (corresponding to a URL) and URNs. For a message, each link contains a list of URLs for storage locations for the message, and is annotated with a URN for the message. The location information on the link points to the related object itself, or a cached copy. The location is considered a hint, since the location itself might not be accessible after disconnection, or cached items may be flushed from the identified cache.
- Back links (Parent links): In addition to the forward link from the recipients' mailbox, each message maintains back links (descendent to parent) to the mailbox of the recipients. The back links are associated with resolve tags.
- Resolve tag: This tag indicates whether the link to the object is resolved (i.e., the destination of the back link is aware of the existence of the original descendent link). It is defined as a vector, so that it can represent the resolve status of multiple back links. The first element represents the name of the Inbox to which the back link is pointing and the second element is reserved for the server. Except the first one, each element has binary values, i.e., “Yes” for resolved or “No” for unresolved. The initial value for each tag is always set to “No”, meaning that everything is assumed unresolved in the beginning. It becomes “Yes”
only when the resolution is verifiable either by triggering a transfer or by receiving a reply from the other host stating that it has the object resolved already.

We take a pessimistic approach to let the users retrieve messages before they connect to the server and synchronize. However, being pessimistic may cause lower system performance because of the required verification process. An example can be found in Figure 2. In this case, the first resolve tag is for user2's Inbox, and the second one is for user3. Both Inboxes for user2 and user3 on the server are not aware of the existence of this back link, for now. Only the Inbox for user2 on the client machine is aware of the link. An optimization by a heuristic approach could be done to enhance the performance. Figure 3 through Figure 6 illustrates the usage of the resolve tag.

- Typical message tags: These include the typical e-mail header information, i.e., subject, date, to (receiver), from (sender), status, etc.
- Authentication/security (optional): This may include authentication level, etc.

The details of how we use these data to reconstruct the interconnections are illustrated in section III-D.

C. Accessing resources at other nodes

There are several details to be considered in using resources from other disconnected nodes. First of all, how does one find nearby resources, i.e., the other disconnected nodes with which one can communicate, local printers, etc.

We use a combination of broadcast and local directory servers for discovery. Broadcast is used on the local network to announce availability of a new machine. Each machine maintains a discovery directory of the resources it holds, and the resources it knows how to reach.

When a system begins to communicate with other disconnected nodes (and periodically thereafter) it broadcasts a message with its identity, and a list of the groups of data that it knows about. This broadcast is confined to a local network, i.e., a single subnet. Systems hearing this broadcast record the data in their local discovery directories. Upon hearing messages from other nodes, the new node contacts an existing node to retrieve the contents of an existing discovery directory, using this data as the initial state for its own directory.

When looking for a data object that is not in its own cache, a system consults its discovery directory looking for an entry for a communicating node that knows about the group containing the desired object. If it finds an entry, it contacts the node identified to locate the object. Otherwise, it broadcasts a message asking if any other communicating nodes have knowledge of the group.

Data in the discovery cache are considered hints. Data in the discovery directory are aged, and there may be multiple entries per group of data. If the desired object cannot be found following the first entry matching the group of the object, then subsequent directory servers in the list may be consulted.

D. Reconstructing Interconnections

This section describes how we organize disconnected nodes and let them share resources using the the data structure described in section III-A and III-B, and discovery mechanism in section III-C.

Upon disconnection, for any new messages received which have been annotated with the mailbox URLs for a group found in the system's discovery cache, the system finds any instances of the mailbox with which it can communicate, and forwards links to the messages.

To do that, each host maintains a list of unresolved back links. By looking at the list, the host can determine unresolved links from other hosts. Other hosts become accessible either by being connected to the network or by being mutually disconnected (connected to each other but
disconnected from the rest of the world). The host can notify the other host if there are unresolved links. When the other host is notified of the existence of unresolved links, it can initiate a transfer of the message into his/her cache to update the link.

Four figures (Figure 3 through Figure 6) illustrate an example. In the example, we assume that there are three users (user1, user2, and user3) and three mobile clients (host1, host2, and host3). Host1 belongs to user1, host2 belongs to user2, and host3 belongs to user3. Each user should have two replicated mailboxes - one on the client and the other on the server. (e.g., user1 has a mailbox on host1 and the server.)

The example shows how a message can be exchanged among clients without connecting to the server. A user can transparently send a message to multiple recipients as if he/she is connected to the network. The message is directly delivered or forwarded when the involved hosts are either connected or mutually disconnected.

Fig. 3. User1 sends message M to user2 and user3 when host1 and host2 are connected to each other while isolated from the network. Then, User2 caches the message M into host2.

In Figure 3, two hosts - host1 and host2 - are connected while disconnected from the rest of the network. User1 sends a message to user2 and user3 while only host1 and host2 are connected. Although user1 sent the messages to user2 and user3, only user2 received the message at this moment. Thus, the resolve tag on the back link of the message M for the mobile client of user2 (in this case, it is host2) is set to “Yes” but the tag for the mobile client of user3 stays as “No”. Note that the server tags for both mail recipients are set to “No”, because the server is not aware of this transaction at this moment. After receiving the message, the system can replicate the message to the local storage of host2 so that the client can access the message object while disconnected.

Consider the following situation when host2 was disconnected from host1. Then, only host2 and host3 are connected as shown in Figure 4. Since the system believes that the message is not sent to user3 yet (with “No” resolve tag), host2 notifies host3 the existence of the care/of message M to user3. Here we use the discovery mechanism to find such unresolved messages, as described in the previous subsection (III-C). Forwarding occurs only when the message has not already been received by user3. Note that if both of user1 and user3 were connected to the network, or they were connected to each other before user2 and user3 were connected, user3 should have received the message M from user1 already.

After a while, host1 and host3 are connected to each other while disconnected from the network as shown in
Figure 5. After browsing the local directory of host1, the system finds that user3 does not know of the message M, because the resolve tag attached on host1 for user3’s client is set to “no”. Hence, host1 notifies host3 about the existence of the unresolved link. However, after checking the local directory of host3, the system realizes that the link has been already resolved and the message object has been already replicated to the local storage, because the resolve tag on the back link on the local directory is set to “yes” and the URNs of the message it already has and the one that wants to be resolved are the same. Therefore, host3 notifies host1 to update the resolve tag about user3, so that host1 can update it accordingly.

IV. Enhancing Performance with URG

Finding objects from the scattered hosts is expensive. To find objects, we have to either maintain location information for individual objects on every host, or broadcast a query every time an object is requested. The former requires significant amount of storage space and the latter requires quite a deal of network bandwidth.

To improve the performance of the system by reducing the required bandwidth with a small additional storage space, we introduce URG (Uniform Resource Group name). We group related objects and provide a unique name with a URG for each group. Each host maintains a URG table which has the list of hosts that contain objects that have the same URG. A host does not necessarily have all the objects with the same URG. By keeping location information only for groups not for individual objects the system reduces the size of database, and/or the required communication.

Grouping can be done in many ways. In a messaging application, a mailbox and all the messages in it, messages related to a specific project, or messages sent to a mailing list can form a group. Groups can overlap, i.e., an object can have multiple URGs.

URG is just another class of URN. Thus, the syntax is the same as that of the URN we defined earlier, except NID for URG is defined as “x-urng”, (again, urng is an arbitrary
string we assigned, standing for Prospero Message Group name) instead of "x-pmm".

Building a URG table can be done in the following way: After browsing its own local disk, each host broadcasts the URGs of the objects it has. Other hosts on the local subnet fill their own URG tables as they monitor the broadcast from others hosts. A URG table has two columns: the first column has a URG, and the second one has the list of host names that have the URG.

Figure 7 shows an example of building a URG table. There are four mobile hosts: host W, host X, host Y, and host Z. Four objects named Obj1, Obj2, Obj3, and Obj7 belong to two groups that have URGs named URG1 and URG2. The table in the lower left corner of the figure depicts the grouping status. The figure shows the process of building the URG table on host X. After browsing the local directory, host Y broadcasts the URG of Obj3, i.e., URG1. Host Z and host W broadcast URG1 and URG2, respectively. Host X fills the URG table of its own as it listens to the broadcasts from other hosts.

Finding objects using URG table is done in the following way: When an object is to be found, instead of broadcasting the query, a host looks up its URG table and locates the list of hosts that share the same URG of the object. Because the hosts on the list do not necessarily have the specific object, each of them needs to be contacted until the desired object is found.

Figure 8 illustrates an example of finding object using the URG table. If a user on host X wants to send a message named Obj1 to a user whose mailbox is named as Obj2, host X needs to find Obj2, unless host X is connected to the mail server of the recipient. Let's assume that host X is not connected to the server. Without URGs, host X should have to broadcast the query to all the hosts it is mutually connected. In this case, they are hosts Y, Z, and W. However, by looking up the URG table host X has, it only needs to contact hosts Y and Z. After failing to find Obj2 from Host Y, host X finds Obj2 from host Z, so that it can make a link from Obj2 to Obj1 to deliver message Obj1 to a mailbox Obj2.

V. IMPLEMENTATION

Because the performance of the local directory service is critical for the performance of the system, we implemented the system on top of the Prospero Directory Service[7], a directory service that is highly optimized for the quick request/response style of interaction that characterizes directory accesses in this system.

The system is written in C and composed of server and
client parts. The object replication, object discovery, and URG table building modules are plugged into the server side of Prospero code. On the other hand, the client side of programs act as the e-mail client and composed of the modules that handle the usual e-mail operations (e.g., read, send, header scan, reply, delete) for the users.

VI. Evaluation

By implementing the example e-mail application, we successfully demonstrated that the concept is plausible. However, the performance of the system is hard to quantify because there is no other system that can be directly compared with. On the other hand, the performance improvement that can be obtained by applying the grouping mechanism using URG can be measured. Even though a sufficiently large live data set is not available, it is possible to evaluate the efficiency of the grouping mechanism with a reasonable set of data. We came up with a set of data to use by applying a couple of assumptions to a live SMTP e-mail log, so that a simulation result can be obtained.

A. Initial parameters

An SMTP log on a mail server (boreas.isi.edu) for one month (Dec. 2000) that has 133,805 entries has been used as the original data. Then, the following assumptions were applied to the data:

- A user with an e-mail address would have between two and four mobile clients to check his/her e-mail.
- The e-mail log file contains all the hosts that belong to a specific domain. No host exists other than the ones that exist in the log.

Applying the first assumption, each e-mail address that is in the original log file was assigned a random number valued between two and four to produce a table that lists the number of hosts that can be used to check e-mail for each user who owns the e-mail address.

<table>
<thead>
<tr>
<th>Number of hosts</th>
<th>e-mail address</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><a href="mailto:tom@3com.com">tom@3com.com</a></td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:jerry@3com.com">jerry@3com.com</a></td>
</tr>
<tr>
<td>4</td>
<td><a href="mailto:dkim@abc.com">dkim@abc.com</a></td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:bcn@abc.edu">bcn@abc.edu</a></td>
</tr>
</tbody>
</table>

Table I

<table>
<thead>
<tr>
<th>Number of hosts</th>
<th>domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3com.com</td>
</tr>
<tr>
<td>3</td>
<td>a21.com</td>
</tr>
<tr>
<td>11</td>
<td>aciri.org</td>
</tr>
<tr>
<td>15</td>
<td>acm.org</td>
</tr>
<tr>
<td>7</td>
<td>adc.com</td>
</tr>
</tbody>
</table>

Table II

Table I shows an example of the table that lists the number of hosts per e-mail address. For example, the user whose e-mail address is jerry@3com.com has two mobile hosts and the server to check e-mail. Note that fictitious e-mail address are shown in the table to protect privacy.

Using Table I and the second assumption, Table II was created. It lists the number of hosts per domain. For example, if Table I has tom@3com.com and jerry@3com.com in its entry and the sum of the number of hosts for those two entries is five, then the number of hosts for 3com.com is determined as five.
<table>
<thead>
<tr>
<th>Obtained parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average size of URG ($AS_{URG}$)</td>
<td>38</td>
</tr>
<tr>
<td>Average size of host name ($AS_{HN}$)</td>
<td>19</td>
</tr>
<tr>
<td>Average size of URN</td>
<td>58</td>
</tr>
<tr>
<td>Average size of request without URG</td>
<td>58</td>
</tr>
<tr>
<td>Average size of request with URG</td>
<td>96</td>
</tr>
<tr>
<td>Number of domains</td>
<td>1289</td>
</tr>
<tr>
<td>Number of hosts</td>
<td>9287</td>
</tr>
</tbody>
</table>

**TABLE III**

Parameters obtained from the original SMTP log, Table I, and Table II

With the original e-mail log and the two tables acquired from the log (Table I and Table II), the parameters on Table III was built. A simulation and its analysis was performed using these parameters.

The differences in the performance of the system with and without using URGs were compared in terms of storage requirements and network bandwidth requirement, as described below.

### B. Required storage space

The additional storage requirement without URGs is zero, but with URGs, storage space to maintain URG tables on the hosts is required additionally.

The required storage space to maintain URG tables on all the involved hosts is:

Required storage for URG tables =

$$\sum_{\text{domain}} N_{row} \times AS_{row} \times N_{table} =$$

$$\sum_{\text{domain}} N_{URGd} \times AS_{URG} \times AN_{HU} \times AS_{HN} \times NH_d$$

In the above formula, $N_{row}$ is the number of rows, $AS_{row}$ is average size of each row, $N_{table}$ is the number of tables, $N_{URGd}$ is number of URGs in a domain, $AS_{URG}$ is average size of URG, $AN_{HU}$ is average number of hosts for each URG, $AS_{HN}$ is average size of host name, and $NH_d$ is the number of hosts in a domain. After plugging in the actual numbers obtained from the tables and the log, the total required storage space turns out as around 1GB. By dividing 1GB with the total number of hosts (9287), the average size of URG table per host can be determined as approximately 100 KB, in this case.

### C. Required communication

Communication bandwidth requirements to find an object without grouping is the required data that need to be exchanged during the one-month experiment period to find an object. The required bandwidth with grouping is the sum of the required bandwidth for the messages to be exchanged to find an object (smaller than that of the case without grouping) and the bandwidth to build URG tables on each host.

In addition to the required bandwidth for the messages to be exchanged to find an object, the required bandwidth with URGs accompanies the bandwidth required to build URG tables on each host as an overhead.

The detailed equations that calculates the required bandwidth for both cases can be found at the end of Section VI-C.3.

#### C.1 Overhead for building URG Tables

The required data that need to be exchanged to build URG tables on all the involved hosts is calculated as the following:

Required data that need to be exchanged to build URG tables =

$$\sum_{\text{domain}} NH_d(NH_d - 1) \times AS_{URG} \times N_{UH}$$

where, $NH_d$ is number of hosts in each domain, $AS_{URG}$ is average size of URG, and $N_{UH}$ is number of URGs per host. With the data and parameters in this experiment, we
obtain 62 MB as the required data to be exchanged to build all URG tables. By dividing 62 MB with the total number of hosts (9287), the average communication bandwidth required to build URG table per host can be calculated as 6.7 KB, in this case.

C.2 Overhead to find an object

In both cases (with or without URG), the required data to be exchanged to find an object for each delivery is the product of the number of contacted hosts and average size of data that need to be transmitted for each query.

Required data for each delivery = \( T \times AR \)

where, \( T = \# \) of Tries and \( AR = \) Average size of request.

In the case without URG, the average size of query is the average size of URNs, because only the URNs need to be transmitted in the query.

\[
AR_{NURG} = \text{Average length of URNs} \\
T_{NURG} = \frac{NH}{2} \text{ hosts needed to be contacted on average}
\]

where, \( NH \) = Number of hosts in a domain.

In the case with URG, the average size of query is the average length of both URNs and URGs, because both of them need to be included in the query. Also, the cost for building URG table should be added to the total data to be exchanged - after adding up the data size of individual deliveries.

\[
AR_{URG} = \text{Average length of (URGs+URNs)} \\
T_{URG} = \frac{NH}{2} \text{ hosts needed to be contacted on average}
\]

where, \( NH \) = preassigned random number of hosts per e-mail address

C.3 Simulation

The simulation was performed in the following way.
1. Read an entry from the SMTP log.
2. Generate a random number between 0 and 10.
3. If the generated number is less than or equal to the current connection ratio, it is regarded as connected.
4. If it is connected, one will be added to the total number of tries regardless of the grouping.
5. If it is disconnected, the program looks up the randomly generated number of hosts per e-mail table to take the the average value (\( T_{URG} \)) to add to the number of tries with the case of using URG.
6. The number of tries is taken from the number of hosts per domain table and the average number (\( T_{NURG} \)) is added to the accumulated number of tries with the case of no grouping.

At the end of the SMTP log, the total data exchanged is calculated in two different cases: In the case with URG, the average size of request (\( AR_{URG} \)) is multiplied to (\( T_{URG} \)), then the required data that need to be exchanged to build URG tables (62MB) is added to the result. The case without URG, the average size of request (\( AR_{NURG} \)) is multiplied to (\( T_{NURG} \)) yielding the total required data exchanged.

Figure 9 shows the result of the simulation. The simulation was run eleven times with the connection ratio varying from zero to one hundred percent.

As shown in Figure 9, using grouping with URG reduces the required communication bandwidth significantly (up to 93%) when the hosts are more disconnected than connected. This enhancement comes with only slight overhead that is mentioned above. The case with grouping shows worse result when most hosts are connected, because it has the overhead of building URG tables (6.7 KB per host) accounted.

The lines in Figure 9 are decreasing linear functions of the connection ratio (CR). The case with URG has much shallower slope than that of the case without URG.
Required bandwidth without URG =
\[ \sum_{entry} AR_{NURG} \times \left( \frac{NH_d}{2} \times (1 - CR) + CR \right) = \]
\[ \sum_{entry} AR_{NURG} \times \left( \frac{NH_d}{2} + (1 - \frac{NH_d}{2}) \right) \times CR \]
Required bandwidth with URG =
\[ \sum_{entry} AR_{URG} \times \left( \frac{NH_d}{2} \times (1 - CR) + CR \right) = \]
\[ \sum_{entry} AR_{URG} \times \left( \frac{NH_d}{2} + (1 - \frac{NH_d}{2}) \right) \times CR + UB \]
where, CR = connection ratio, UB = cost for building URG tables, and AR = average size of requests.

When \( \frac{NH_d}{2} = 3 \), the performance of the system with using URG will be better than that of without using URG if \( \frac{NH_d}{2} > 21 \), given all the numbers are from the above simulation.

VII. RELATED WORK

Grapevine[9] is a distributed and replicated e-mail delivery system. However, unlike our system, it does not support interaction among clients or disconnected mobile operation.

xFS, The Serverless Network File System[10] from Berkeley supports cooperative caching of data across multiple client machines. xFS requires significantly more coordination between nodes, and although the system can recover when nodes fail, it doesn’t address the situation where nodes regularly disconnect and reconnect in different places. In contrast to xFS, our system assumes an environment where machines routinely connect in varying configurations.

Disconnected operation has been supported in several research file systems, most notably Coda[11]. Such work has focused on the caching of data for availability on clients, and subsequent synchronization with servers upon reconnection. In Coda, clients can resynchronize with multiple servers, and may only be intermittently connected to a subset of the servers. Servers themselves may be disconnected by network partitions. However, clients do not exchange data directly with other clients.

In contrast, with disconnected operation using the self-organizing storage system, clients can share data with other clients and they will discover the data that is available through systems with which they still communicate.

In the Ficus[12], ROAM[13], and Bayou[14], replication and synchronization of replicated data is supported directly between clients without communication through fixed servers. The benefits of this peer-to-peer interaction apply to replicated data, and established replication algorithms are used to maintain the consistency across peers.

This means that the ROAM system will do a better job maintaining the consistency of mutable data that has been replicated on the peer systems. In contrast, the self-organizing storage system is better suited for cooperative sharing of data which was not previously configured for replication across clients. Our approach also supports the cooperative sharing of immutable data and will propagate the relationships between immutable data objects. Unlike other systems that support disconnected peer-to-peer replication, our system provides a flexible grouping mechanism to improve the performance of the system.

A URN (Uniform Resource Name) [2], [3], [4] is a name
that identifies a resource or unit of information independent of its location. URNs are globally unique, persistent, and accessible over the network. URNs are used to provide stable names for resources whose characteristics may vary over time. For instance, a URN may be used as a stable reference to a web page. The web page can then move, be replicated, or change its contents and still remain accessible via the same URN. In contrast to URLs which have wired-in location information, the location information and other characteristics of a URN are provided by external resolution servers. We devised a new class of URNs for groups of objects.

VIII. Conclusion

We presented a new approach for organizing disconnected nodes into stand-alone systems that can operate using cached data without access to the fixed infrastructure. We defined a class of unique identifiers (URNs) for the objects and object groups (URGs) and represented bidirectional relationships among them as links. By exploiting overlapping objects that are cached in disconnected nodes, we provide connectivity to the graph, reconstructing the interconnection.

By grouping objects, we improve the performance of the system with a small overhead.

The system has been implemented on top of Prospero directory service.

References


