Spring 2009 CSCI 551

Final Project

A Peer-to-Peer File Sharing System

Deadlines

Here are the deadlines:

Part	Due	Percentage
(1)	11:45PM 3/31/2009	45
(2)	11:45PM 4/24/2009	35

For part (1), you need to be able to demonstrate that you can handle `join`, `hello`, `keepalive`, `notify`, `check`, and `status` messages, as well as the formation of the SERVANT network and complete logging for the above message types. The only user commands a node must handle are the `shutdown` and the `status neighbors` commands.

For part (2), you need to be able to demonstrate the you have implemented the whole spec. Since part (2) depends on part (1), you should try to get part (1) done as solid as possible.

[BC: Paragraph added 4/1/2009]
To make part (2) less dependent on part (1), it is not required that you have `join` and `check` working. Moreover, if the **`NoCheck=1`** is specified in a node's startup configuration file, the node should **disable** its *join* and *check* mechanisms.

FAQ

The part (1) and part (2) FAQ pages should be considered under constant construction. They will be modified through out the semester. If you have questions about the final project, please check the FAQ pages first because the answer you are looking for may be posted there already.

Introduction

In this project, each student will build a distributed file sharing system using peer-to-peer technology. This system is referred to as the **SERVANT** and it is composed of a collection of nodes. Each node is part server and part client (*conceptually*, we will refer them as the server side and the client side of a node). A user can interact with the client side of a node via a commandline interface in order to perform various tasks within the SERVANT system. For example, a user can:

- store a file into the SERVANT (not necessarily just on the server side of this particular SERVANT node, i.e., the file gets replicated

probabilistically)
- perform various types of searches
- retrieve files based on the result of a search
- delete a file from the SERVANT and destroy all (or most) copies of it

This system is **peer-to-peer** in the sense that if one is using the system, one must be sharing his/her resources.

Peer-to-Peer

The SERVANT distributed system is composed of an unknown number of nodes. The operational status of each node is generally unknown. A node only knows the IP addresses and port numbers of a small subset of all the nodes. These are the *neighbors* of he node. When a node receives a file from one of its *neighbors*, it does *not* know where the file was originated. When a node receives a message from one of its neighbors, it does not know if the neighbor initiated the message or it is simply passing it along.

A set of nodes, know as *beacon* nodes, forms the core of the SERVANT network. Every node in the SERVANT network knows who the beacon nodes are (this knowledge is obtained from a node's start-up configuration file).

When a non-beacon node, say node X, wants to *join* the SERVANT network for the first time, it sends a request to a *beacon* node (which is, by definition, already part of the SERVANT network). ***[BC: fixed 2/14/2009]*** This beacon node then floods the join request to the whole network. Every node in the network will compute its distance to node X and put this distance in the `Distance` field of its join response message. Node X then picks a subset of the nodes from the list (ones with the lowest `Distance` values) to be its *neighbors* and discard its knowledge about the rest of the nodes. Node X then writes the list of neighbors to a `init_neighbor_list` file in its home directory so that the next time node X is started, it doesn't have to go through the join process (unless not enough of its neighbors are up).

When a node comes up, it should attempt to connect to all its neighbors (by reading the `init_neighbor_list` file). The first message a node (say, node A) should send to its neighbor (say, node B) is a `hello` message to advertise its own hostname and port number. Please note that *neighbor* is a bi-directional relationship. Therefore, when node B gets a `hello` message from node A, and node A is not currently node B's neighbor, node B should send a `hello` message to node A.

If there is no traffic from a node to one of its neighbors, the node must send a `keepalive` message so that its neighbor knows that it is still operating properly.

When a node loses connection with one of its neighbors, it is possible that it may be disconnect from the SERVANT network. In order to make sure that it is still connected to the network, the node must intiatiate a connectivity check by flooding a `check` message. If the `check` message cannot reach any beacon node, part of the network must have been disconnected from the

core of the network. This node must therefore rejoin the network (delete its `init_neighbor_list` file, gracefully shut itself down, and start itself again as if it has never been part of the network). The flooding of `check` messages stops at a beacon node where the beacon node sends a `check` response message towards the node from which it received the `check` message.

Message forwarding (for messages of all types) is usually performed using a *flooding* algorithm. A response message must return via the same route as the corresponding request message to provide anonymity (protect the identity of the responding node and the requesting node). Each message has a TTL (time-to-live) value and which is to be decremented when the message is forwarded. If a TTL reaches 0, no forwarding should be performed.

Unlike Napster, there is no centralized catalog of files that is stored in the SERVANT. The location where a file is stored is generally unknown.

As a file travels through the SERVANT network (when it is stored or retrieved), it may be cached at various nodes. Each node is configured with a given cache size. If a node decides to cache a file but it has exhausted its cache space, it should use the LRU method to determine what to purge from its cache. Part of a node's filesystem area is designated as *permanent* storage space. Unlike the cache space, the *permanent* space is *not* subject to any replacement policy. If a file is to be stored in the permanent space of a node and there is not enough space in the filesystem, the file must be rejected.

Some Definitions and Operational Requirements

Node ID:

Each node is identified by a unique *node ID* which is simply the concatenation of its hostname and its port number (with an underscore character inserted between them). For example, **`"merlot.usc.edu_16011"`** is a valid Node ID. You can call `gethostname()` to get the hostname of the machine you are running on.

Node Instance ID:

Every time a node is restarted, it is associated with a *node instance ID*, which is simply the concatenation of its node ID and the time the node got started (with an underscore character inserted between them). For example, **`"merlot.usc.edu_16011_1016320007"`** is a valid node instance ID. To get the time a node is started, please call the `time()` function.

Connections:

For the purpose of this project, you *must* use the same socket for the connection from A to B and from B to A. It is possible that A and B initiates a connection to each other simultaneously. You need to make sure that only one bi-directional connection is setup.

In order to break ties, you must compare the port numbers of nodes A and B. Let `PA` and `PB` be the well-known port numbers of nodes A and B, respectively. If `PA > PB`, you should keep the connection initiated by A. If `PA < PB`, you should keep the connection initiated by B. If `PA == PB`, you should compare the hostnames of nodes A and B. Let `HA` and `HB` be the hostnames (ASCII string) of nodes A and B, respectively. If

```
strcmp(HA, HB) > 0
```

you should keep the connection initiated by A. If

```
strcmp(HA, HB) < 0
```

you should keep the connection initiated by B.

Please note that immediately after node A gets a connection from node B, node A does not know the well-known port number of node B. Node A must wait for the `hello` message from node B in order to find out the well-known port number of node B.

Once a pair of `hello` messages have been exchanged on a connection, this connection is considered to be operational and regular messages can be sent or forwarded on this connection. A node should drop all messages if it has not received a `hello` message over a connection. A node should also drop all message after it has received a `notify` message over a connection.

[BC: Added 2/25/2009]
Auto-shutdown:

When the auto-shutdown timer goes off, the situation should be very similar to auto-shutdown in warmup project #1. Your node should not accept any new connection; your node should ask child threads to close their respective sockets as soon as possible (they do not need to finish sending a message); your node should wait for all child threads to terminate before it can terminate itself. You do **not** need to send **notify** messages in this case (since you may not have finished sending the previous message).

[BC: Added 3/20/2009]
`init_neighbor_list` file:

The `init_neighbor_list` file may be present in the home directory of a non-beacon node.

If this file is present, the corresponding node should *participate* in the SERVANT network when it starts or restarts. If this file is *not* present, the node should *join* the SERVANT network when it starts or restarts.

This file is *write-once*. Once it's created, it should never be modified. It gets created when the node joins the network

successfully.

When the node starts or restarts, if it is participating in the SERVANT network and it cannot connect to `MinNeighbors` number of nodes listed in this file, this file must be *deleted* and the node should perform a soft-restart.

When the node quits, if this file is present, it must not be deleted.

Nonce & FileID:

A file is *created* into the SERVANT network via a `store` user command. At this time, the corresponding file metadata is also created for this file. A random 20-byte long *one-time password* is first generated. The SHA1 hash of the password is then computed and is to be used as the *nonce* for this file. A nonce can be used to distinguish different versions of the same file. (If a file with a certain SHA1 value is created by a user and, at a later time, the same file with the same SHA1 value is created by the same user or another user, these 2 files will have different nonces.) Therefore, we consider that file Y is a **copy** of file X if X and Y have identical FileName, SHA1, and Nonce (everything else about X and Y are assumed to be identical).

How does Y comes to existance? Well, when X was first introduced into the SERVANT network via a `store` user command, copies of X got probabilistically flooded into the rest of the SERVANT network (upto a TTL value). So copies of X can be created. Later on, when some node retrieves X or a copy of X, another copy of X gets probabilistically copied on some nodes in the network, so more copies of X can get created. So, if X is a very popular search target, there can be many many copies of X in the entire network. Now, if you search of X and get many responses, and you only want one copy of X to be sent to you, how do you only ask for one copy of X to be sent to you? This is where FileID comes in.

A FileID distinguishes one copy of X from another copy of X. Since FileID only matters when a user enters a `get` user command, you only need to create a FileID if a corresponding search reply was created.

In additional to distinguishing different versions of a file, a *nonce* is needed to mitigate *replay attacks*. Here is the scenario. File X is stored by the user at node A. Then X is deleted by the user at node A. Then X is again stored by the user at node A. Now anyone who has a copy of the delete message and flood it back to the network and delete all copies of X if a nonce was not used.

Summary of Basic Message Types

Below, we summarize the basic message types.

(1) The **`join`** message should be flooded to the whole network (up to a specified TTL). In response to a `join` message, a node computes its distance (4-byte value) to the requesting node and includes this distance in a join response message. The distance is simply the numeric difference between the location of this node and that of the joining node. A joining node uses the responses to determine who its neighbors shall be. The node then writes the list of neighbors to the `init_neighbor_list` file in its home directory. This message should be sent only once, unless a node cannot connect to enough neighbors.

If a node knows who its neighbors are, when it comes up, it connects to all its neighbors via TCP connections. If not enough of its neighbors are up (according to the value of the `MinNeighbors` setting in its start-up configuration file), the node should delete the `init_neighbor_list` file and rejoin the network.

(2) The **`hello`** message is used to introduce a node (its hostname and its port number) to its immediate neighbor so that the neighbor can talk back to this node. A node should send this message to its neighbors immediately after it comes up (assuming it has done the orignal join/discovery process). The TTL should be set to 1 for this message.

(3) The **`keepalive`** message is used by a node to inform its neighbor that it is still alive. The reason why this message is necessary is that if a node goes down abruptly, TCP will not tell its neighbors that it has gone done. The TTL should be set to 1 for this message.

(4) The **`notify`** message is used by a node to inform its neighbor that it is about to close the connection. No more data should be expected on a connection once this message is received. The TTL should be set to 1 for this message.

(5) The **`check`** message is used by a node to see if it is still connected to the core of the SERVANT network. It should be flooded to the whole network (up to a specified TTL).

A `check` response is initiated by a beacon node to indicate that the core of the network has been reached.

(6) The **`store`** message should be *probabilistically* flooded to the whole network (up to a specified TTL). It takes a file and stores it in the server side of the node to which the user is connected. A file is tagged with various metadata such as file name, file size, and keywords. The type of a file is identified by its file name extension. A node that initiated a `store` for file `F` must store file `F` in the permanent area and *not* in its cache. If there is not enough space in the filesystem, this node should *not* flood **`store`** messages.

(7) The **search** message is to be flooded to the whole network (up to a specified TTL). The arguments to this message depend on the type of search the user wants to perform. The search can be based on keywords, the precise spelling of a file name, or the precise SHA1 hash of a file.

A search response may contain multiple records. Each record contains the metadata of a file plus a FileID which uniquely identifies the file. If a node generates a search response message for a file in its cache, it should update the LRU order of this file.

(8) The **get** message is to be flooded to the whole network (up to a specified TTL) to retrieve a file associated with a specific FileID. The response to a get message contains the file matching with the requested UOID and all its metadata.

If the user at node X attempts to retrieve file F and file F was successfully retrived, node X must serve file F (i.e., respond properly to future search messages). File F should be stored in the permanent area and *not* stored in its cache. Intermediate nodes, i.e., nodes on the return path of the get response message (other than the originating and the final nodes) cache file F *probabilistically*.

(9) The **status** message is to be flooded to the whole network (up to a specified TTL). An argument is passed to this message specifying what type of status needs to be reported.

A status response may contain multiple records.

(10) The **delete** message is to be flooded to the whole network (up to a specified TTL). It is used to delete copies of a specified file from both permanent storage and the cache. A *one-time password* is used to authenticate the originating node of the message. Only the original owner of the file (the one who stored the file in the SERVANT originally) is allowed to delete it.

Message Format

All messages have a 27-byte common header depicted below:

Byte Pos	**Name**	**Description**
0	MessageType	The message type. Please refer to the next table.
1-20	UOID	Message ID. This field is inserted by a message originating node and copied by all forwarding nodes.
21	TTL	How many hops the message has left before it should be dropped.
22	(Reserved)	(always zero)
23-26	Data Length	The length of the function-dependent data.

The first byte of the header is the message type. Bytes 1 through 20 correspond to a unique object ID (UOID) which identifies the message.

If a node is forwarding (via flooding) a message with a particular UOID, it must check to see if a message with the same UOID has been seen before. If it hasn't seen the message, it should not generate a new UOID for the forwarded message but copy the UOID from the received message onto the forwarded messages. Otherwise, the message should be dropped. Please note that this implies that a node must store *all* UOIDs it has *ever* seen until the message expires (please see MsgLifetime regarding message expiration).

Byte 21 is the TTL. For every message you receive, you should decrement the TTL and pass the message on if the TTL is still greater than zero. If you are **forwarding** a message (of **any** type), the only thing you can change in the message is the TTL. You must **not** change anything else in the message!

The last field of the header indicates how large the type-dependent data that follows it is. All numeric fields should be in *network byte order*. All string fields should **not** be terminated by a null character.

Below, we define the all the type-dependent message formats:

join message (MessageType=0xFC)

Byte Pos	Name	Description
27-30	Host Location	Location of the requesting host.
31-32	Host Port	TCP port number of the requesting host.
33+	Hostname	The hostname (not IP address) of the requesting host.

join response message (MessageType=0xFB)

Byte Pos	Name	Description
27-46	UOID	The join UOID that this message is responding to.
47-50	Distance	Distance to the node requesting to join. This is a 4-byte long numeric field.
51-52	Host Port	TCP port number of the listening host.
53+	Hostname	The hostname (not IP address) of the listening host.

hello message (MessageType=0xFA)

Byte Pos	Name	Description

27-28	Host Port	TCP port number of the sending host.
29+	Hostname	The hostname (not IP address) of the sending host.

keepalive message (MessageType=0xF8)

This message has an empty body (Data Length = 0).

notify message (MessageType=0xF7)

Byte Pos	Name	Description
27	Error Code	(see below)

The possible values for Error Code are:

0 : unknown
1 : user shutdown
2 : unexpected kill signal received
3 : self-restart

check message (MessageType=0xF6)

This message has an empty body (Data Length = 0).

check response message (MessageType=0xF5)

Byte Pos	Name	Description
27-46	UOID	This should match the UOID in the corresponding check message header.

search message (MessageType=0xEC)

Byte Pos	Name	Description
27	Search Type	If the search type is 1, the next field contains an exact file name. If the search type is 2, the next field contains an exact SHA1 hash value (not hexstring-encoded). If the search type is 3, the next field contains a list of keywords (separated by space characters).
28+	Query	The content of this field depends on the search type.

search response message (MessageType=0xEB)

Byte Pos	Name	Description
27-46	UOID	This should match the UOID in the corresponding search message header.

47-50	Next Length	The length of the `Metadata` field. If this field is not zero, another length field will follow the next descriptor.
51-70	FileID	A UOID that identifies a file.
71+	`Metadata`	`File Metadata`.

A search response message may contain multiple records. For example, the following search response message contains 3 records:

UOID
NextLength1 (= length of Metadata1)
FileID1
Metadata1
NextLength2 (= length of Metadata2)
FileID2
Metadata2
NextLength3 (= 0)
FileID3
Metadata3

The length of Metadata3 can be calculated from the Data Length field of the common message header.

get message (MessageType=0xDC)

Byte Pos	Name	Description
27-46	FileID	A UOID that identifies a file.
47-66	SHA1	SHA1 hash value of the file with the above FileID.

get response message (MessageType=0xDB)

Byte Pos	Name	Description
27-46	UOID	This should match the UOID in the corresponding `get` message header.
47-50	Metadata Length	Length of the `Metadata` field.
51+	`Metadata`	`File Metadata`.
(?)+	File Data	The actual file. A node receiving this message must compute the SHA1 hash value of the file and compare it against the stored file description. If it does not match, the file should be discarded.

store message (MessageType=0xCC)

Byte Pos	Name	Description

27-30	Metadata Length	Length of the `Metadata` field.
31+	`Metadata`	`File Metadata`.
(?)+	File Data	The actual file. A node receiving this message must compute the SHA1 hash value of the file and compare it against the stored file description. If it does not match, the file should be discarded.

delete message (MessageType=0xBC)

Byte Pos	Name	Description

27+	File Spec	The *File Spec* is a specification for the file to delete. The spec should contain 4 lines:
		`FileName=foo`
		`SHA1=63de...`
		`Nonce=fcca...`
		`Password=bac9...`
		The lines are separated by <CR><LF> (i.e., "\r\n").

status message (MessageType=0xAC)

Byte Pos	Name	Description
27	Status Type	Status type to report. `0x01` means the `neighbors` information should be sent in the reply `0x02` means the `files` information should be sent in the reply.

status response message (MessageType=0xAB)

(It should be clear from the content of the status response message that in a real system, `status` messages should be privileged to preserve anonymity. You are asked to implement them to ease grading. Therefore, it is imperitive that they work properly.)

Byte Pos	Name	Description
27-46	UOID	This should match the UOID in the corresponding `status` message header.
47-48	Host Info Length	The length of the `Host Port` plus `Hostname` fields that immediately follow this field.
49-50	Host Port	TCP port number of the replying host.
51+	Hostname	The hostname (not IP address) of the replying host.
(?)+	Record Length	The length of the `Data` field. If this field is not zero, another length field will follow the `Data`

		field. This is a 4-byte long numeric field. (This field should be omitted if the request type is `files` and no file is stored at the sender node.)
(?)+	Data	This field depends on the type of status being requested. (This field should be omitted if the request type is `files` and no file is stored at the sender node.)

If the status request type is `neighbors`, each `Data` field contains:

(?)+	Host Port	TCP port number of a neighboring host.
(?)+	Hostname	The hostname (not IP address) of a neighboring host.

If the status request type is `files`, each `Data` field contains:

(?)+	Metadata	File Metadata.

A status response message may contain multiple records. For example, the following status response message contains 3 records:

UOID
HostInfoLength
HostPort
Hostname
RecordLength1 (= length of Data1)
Data1
RecordLength2 (= length of Data2)
Data2
RecordLength3 (= 0)
Data3

The length of Data3 can be calculated from the Data Length field of the common message header.

Please note that although action needs to be taken, there is no need to respond to either a **keepalive**, **notify**, **store**, or **delete** message.

Logging

You must log all messages seen at a node and all messages sent by a node into its log file. For each message *received*, *forwarded*, or *sent*, the following log entry (each in one line) must be appended, respectively:

```
r <time> <from> <msgtype> <size> <ttl> <msgid> <data>
f <time> <to> <msgtype> <size> <ttl> <msgid> <data>
s <time> <to> <msgtype> <size> <ttl> <msgid> <data>
```

The `<time>` field has the format `("%10ld.%03d",sec,millisec)`.

The <from> and <to> fields contain a node ID of a neighbor. "Neighbor" here includes the "temporary" neighbor (i.e., a joining node) of a beacon node.

The <msgtype> field is a 4-character long string representing the type of a message. What goes into the <data> field depends on what type the message is. Their relationship is shown in the following table:

<msgtype>	**description**	<data>
JNRQ	join request	<port> <hostname>
JNRS	join response	<uoid> <distance> <port> <hostname>
HLLO	hello	<port> <hostname>
KPAV	keepalive	*(none)*
NTFY	notify	<errorcode>
CKRQ	check request	*(none)*
CKRS	check response	<uoid>
SHRQ	search request	<searchtype> <query>
SHRS	search response	<uoid>
GTRQ	get request	<fileid>
GTRS	get response	<uoid>
STOR	store	*(none)*
DELT	delete	*(none)*
STRQ	status request	<statustype>
STRS	status response	<uoid>

The <size> field should account for the message header and the message body (basically common header length plus data length).

Although a message ID is 20 bytes long, you only need to log the last 4 bytes in hex string format in the <msgid> field. The same goes with <uoid> and <fileid> values in the <data> field.

The <searchtype> can be "filename", "sha1hash", or "keywords".

The <statustype> can be "neighbors" or "files".

If a received message is forwarded to *n* different neighbors, you must enter *n* forwarded log entries (each one with a different <to> field).

Please *flush* the log file output stream as soon as you finish writing a line (or multiple lines) to the log file. This means that you should call `fflush()` or `f.flush()` if you are using C or C++, respectively. You should be able to run the following command and see meaningful output:

```
tail -f /yourhome/servant/12312/servant.log
```

if `/yourhome/servant/12312/servant.log` is a log file.

It is **imperative** that you have logging works correctly becuase the grader needs to observe messages received and sent by each node. If your logging

is not working properly, you stand to lose a lot of points (at least 20%).

It is also **imperative** that you have "status neighbors" (for parts (1) and (2)) and "status files" (for part (2)) user commands work correctly becuase the grader needs to observe the status of all nodes in a reasonable fashion. If your "status" user command is not working properly, you stand to lose a lot of points (at least 20%).

You should write error messages into you log file and you may also output debugging information there. Please begin each debugging lines with `//` (like a comment line in C++) and error messages with `**`.

The log file should be deleted only when a node is started with the `-reset` commandline option.

Unique Object ID

The UOID (or Unique Object ID) is a 20-byte value that uniquely identifies an object. An object can be a message, a file, or anything you want. It should be unique across the whole SERVANT network.

Since SHA1 has the so-called *collision free* (or, more appropriately, *collision unlikely*) property, we can use SHA1 to generate UOIDs. The idea is that if we feed a unique string to SHA1, althought the string can be longer than 20 bytes, we can get a *unique* 20-byte value. Therefore, you can implement a function like the `GetUOID()` function below to generate an UOID. You can easily come up with variations of this function to suit your needs.

```
#ifndef min
#define min(A,B) (((A)>(B)) ? (B) : (A))
#endif /* ~min */

#include <sys/types.h>
#include <openssl/sha.h> /* please read this */

char *GetUOID(
        char *node_inst_id,
        char *obj_type,
        char *uoid_buf,
        int uoid_buf_sz)
{
    static unsigned long seq_no=(unsigned long)1;
    char sha1_buf[SHA_DIGEST_LENGTH], str_buf[104];

    snprintf(str_buf, sizeof(str_buf), "%s_%s_%lld",
            node_inst_id, obj_type, (long)seq_no++);
    SHA1(str_buf, strlen(str_buf), sha1_buf);
    memset(uoid_buf, 0, uoid_buf_sz);
    memcpy(uoid_buf, sha1_buf,
            min(uoid_buf_sz,sizeof(sha1_buf)));
    return uoid_buf;
}
```

The `node_inst_id` is the node instance ID of the calling node. The `SHA1(in,size,out)` function computes a 20-byte SHA1 hash of the `in` argument and writes the result in the `out` argument.

If you need an UOID for a message, a file, or a soda, you can call `GetUOID()` in the following fashion, respectively:

```
unsigned char buf[SHA_DIGEST_LENGTH];

GetUOID(node_inst_id, "msg", buf, sizeof(buf));
GetUOID(node_inst_id, "file", buf, sizeof(buf));
GetUOID(node_inst_id, "soda", buf, sizeof(buf));
```

File Metadata (for Part 2 only)

The file metadata is formatted similarly to an INI file. It must start with a section name of `[metadata]`, followed by <CR><LF>. It is then followed by the following keys and their corresponding values (*in the order specified here*):

- **FileName** - name of the file, followed by <CR><LF>.
- **FileSize** - size of the file (in bytes), followed by <CR><LF>.
- **SHA1** - SHA1 hash of the file (formatted hex string), followed by <CR><LF>.
- **Nonce** - a randomly value associated with the file (formatted hex string), followed by <CR><LF>. (This field has the same length as the SHA1 field.)
- **Keywords** - keywords (separated by space characters), followed by <CR><LF>.
- **Bit-vector** - hexstring-encoded (256 characters long) bit-vector for the keywords, followed by <CR><LF>.

Here is an example (<CR><LF>'s have been left out and the *backslashes* have been inserted for readability):

```
[metadata]
FileName=blondie1.mp3
FileSize=474736
SHA1=ffd3b197e1c0f0c27e7bdc219f553a3e6b139dfb
Nonce=bac9a3247be59ea11a5883df04b80988035430d6
Keywords=categories audio mp3 artist Blondie \
  title Heart of Glass \
  url http://www.blondie.net/ \
  additional_keywords debra harry
Bit-vector= \
  1100001000000004200200000000000000000000000000 \
  1000000000000000002000000000000000010000010004 \
  0000000000048000000008000000000000000000000000 \
  0000000000000000001000088000000000000040048400 \
  0000021000000000000081000000000000200002000200 \
  0000000000000000
```

Routing

On messages that require flooding, you must make sure that a message is not sent over the same link more than once. Let us consider an example using the following graph.

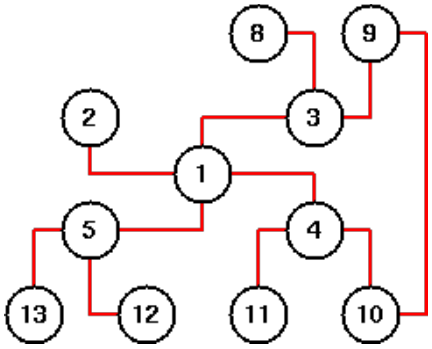

Figure 1: Example network connection.

Imagine yourself as node 1 in the above diagram. You have direct connections to nodes 2, 3, 4, and 5. You have reachable hosts at nodes 2 through 5 and 8 through 13.

1. You get a join message (function 0xFC) from node 2 with a message ID of `x` in the header.
2. Lookup in your message routing table `[UOID x]`.
3. Lookup failed? Save `[UOID x]` in the routing table and record that it was received from node 2.
4. Respond with a join response (0xFB), with the UOID field being `x` to node 2.
5. Send the function 0xFC message to nodes 3, 4, and 5 (**not 2**).
6. Node 3 will respond with a join response (0xFB), with the UOID field being `x`.
7. Forward the message to whoever in the routing table that has `[UOID x]`. Since this message is being *routed* (i.e., this is a response) and not flooded, there is no need to check if it is a duplicate, as routed messages do not make loops.
8. Do the same thing with responses from 4 and 5.
9. Since 3 thru 5 will also pass the message on to 8 thru 13, you'll also get a 0xFB from them.
10. **Problem:** Node 3 is connected to node 9 which is connected to node 10 which is connected to node 4 which is connected to node 1! This is OK. Node 1 looks it up in its routing table `[UOID x]`. Since it's already there, it drops the message and does not respond to node 4, and it does not forward it to anyone.

The above implies that a node should keep message UOIDs in memory and have an efficient way to see if a message UOID has been seen before. Efficient here is defined as O(log(n)), where n is the number of message UOIDs in your data structure. In addition, an UOID entry should timeout approximately MsgLifetime seconds after it has been inserted into the data structure. You should devise a way to handle this expiration efficiently (although it is not a requirement). If you have done something to make these operations efficient, please make sure you document document them in the README file of your submission.

TTL

The basic idea here is that a message (or message in our case) is stamped with a TTL when it is sent out. Then, each host which receives this message decrements the TTL. If the TTL is zero, the message is not forwarded. Your program should work similarly. Hence, in your project, when a NEW message is sent from a node, the TTL is set by that node to whatever is indicated in its configuration file. When the message is received by the next node, the TTL is decremented. Then that TTL is checked against the forwarding node's TTL (i.e., in its configuration file). The lower of the two numbers is placed in the outgoing TTL field. If the outgoing TTL is zero, the message is not forwarded.

SERVANT Node Start-up

The executable to bring up a SERVANT node must be named `sv_node`. The commandline syntax to start a SERVANT node is:

```
sv_node [-reset] startup.ini
```

Square bracketed items are optional. If **`-reset`** is specified, you must first reset this node to the state as if it has never been started before. This includes deleting the `init_neighbor_list` file, the log file, all cached and permanent files, etc.

The **`startup.ini`** is a start-up configuration file. The file name can be different, but the file name extension should be `.ini`. The format of a `.ini` file is described in a separate section.

When a node starts up, it can be in one of two general states: *joining* the network or *participating* in the network. It cannot be in both states simultaneously. The way a node can tell which state it should be in is by looking for the `init_neighbor_list` file in its home directory. If the `init_neighbor_list` file does not exist, it must try to join the SERVANT network. If the `init_neighbor_list` file exists, it must participate in the SERVANT network.

If a node is *participating* in the network, when it makes a connection to another node, the first message it sends must be a `hello` message.

An `init_neighbor_list` file must conform to the format specified here. It must be an ASCII text file where each line in it has the form `hostname:port`. Each line is terminated by a "\n" or "\r\n". The following is an example of a valid `init_neighbor_list` file:

```
foo.usc.edu:12311
bar.usc.edu:12318
```

The following keys are defined for the **`[init]`** section of a start-up configuration file.

Mandatory Fields (these keys **must exist**):

- **Port** - The port number to listen on. Each student will be given a range

of port numbers to use.
- **Location** - The location of this node. (Ideally, this field can be the MD5/SHA1 hash of the node ID. But we restrict ourselves to an unsigned 32-bit integer for this project.)
- **HomeDir** - The home directory for this node. If this directory does not exist, your program should exit. This directory must contain a subdirectory named `files` for storing cached and permanent files. If the `files` subdirectory does not exist, your program should create it programmatically.

Optional Fields (if these keys are not specified, the default values should be assumed):

- **LogFilename** - Name of a log file. This file is assumed to be under the node's home directory. You should write error messages into this log file instead of printing them to stdout or stderr. The default value is "`servant.log`".
- **AutoShutdown** - The number of seconds for the node to auto-shutdown after it starts. The default value is 900 (for 15 minutes).
- **TTL** - The TTL value to be used for all outgoing messages. The default value is 30.
- **MsgLifetime** - The lifetime (in seconds) of a message. Because of the way routing is done in the SERVANT network, the UOID of all messages must be saved in memory so that duplicate messages can be discarded. It is unreasonable to keep these UIOD's forever. Therefore, we assume that a message cannot stay in the network longer than the value specified here. So, if an UOID has been saved in memory longer than the time specified by this value, the UOID can be discarded. The default value is 30.
- **GetMsgLifetime** - The lifetime (in seconds) of a **get message**. Since file transfer may take a very long time, the UOID of a **get message** needs to cached in memory longer than other message types. The default value is 300.
- **InitNeighbors** - The number of neighbors to establish when a node first joins the network. This value should be used to pick the number of neighbors during the `join` process. The default value is 3. (A beacon node should ignore this value.)
- **JoinTimeout** - This key is obsolete.
- **KeepAliveTimeout** - If a node has not seen *any* message from a neighbor after this timeout (in seconds) has occurred, it should close its connection to this neighbor. The default value is 60.
- **MinNeighbors** - ***[BC: updated 3/22/2009]*** The minimum number of neighbors a node needs to establish connections with when it comes up and when it's participating in the network. If it cannot connect to this many neighbors, it must delete the `init_neighbor_list` file and rejoin the network (by contacting a *beacon* node). The default value is 2. (A beacon node should ignore this value.)
- **NoCheck** - If this value is 0, sending of `check` messages is enabled when a non-beacon node loses connection with one of its neighbors. If this value is 1, sending of `cehck` messgaes should be disabled for a

non-beacon node. For a beacon node, it must ignore all `check` messages. The default value is 0. Please note that for grading of part (2), this value will always be 1 for all nodes.

- **CacheProb** - When a node forwards a `get response` message from a neighbor, it flips a coin (with this probability of getting a positive outcome) to decide if it should cache a copy of it. If the result is positive, a copy of the file is stored. The default value is 0.1.
- **StoreProb** - When a node receives a `store` request from a neighbor, it flips a coin (with this probability of getting a positive outcome) to decide if it should cache a copy of it. If the result is positive, a copy of the file is stored. The node where this `store` request was originated ignores this value. The default value is 0.1.
- **NeighborStoreProb** - When a node originates or receives a `store` request, it asks its neighbors to cache a copy of the file. For each neighbor, it flips a coin (with this probability of getting a positive outcome) to decide if it should forward the `store` request to the corresponding neighbor. The default value is 0.2.
- **CacheSize** - ***[BC: updated 4/7/2009]*** An integer value specifying the maximum total storage (in kilobytes) for cached files. The default value is 500. A kilobyte is 1,024 bytes. Please only count file sizes for the `.data` files.
- **PermSize** - This key is obsolete.

The **`[beacons]`** section lists beacon nodes. When a new node wishes to join the SERVANT network, it should go down this list and find a running beacon node to send a join request. Keys in this section have the format of `hostname:port`. The value for each key should be empty. For example (please replace the 12xxx port numbers with the ones that have been assigned to you):

```
[beacons]
foo.usc.edu:12311=
foo.usc.edu:12312=
bar.usc.edu:12311=
bar.usc.edu:12313=
Retry=15
```

The **Retry** key specifies the amount of time to wait (in seconds) for a beacon node before retrying a failed connect attempt with another beacon node (see below). The default value is 30.

Please note that it is perfectly okay to have additional additional keys or sections in a start-up configuration file. Your program must not report them as errors.

When a beacon node comes up (it finds itself in the beacons list), it must make all other nodes in the beacons list its neighbors and need not send out join requests. If it cannot make a connection to another beacon, it should retry after a timeout period specified by the Retry key in the [beacons] section.

One way to think of this is that the beacon nodes form a network that is

fully connected, always.

When a regular node comes up, it should only make one attempt for each of its initial neighbors. If it cannot make a connection, it should not retry.

The following is an example of the `start-12312.ini` file.

```
[init]
Port=12312
Location=3134382376
HomeDir=/yourhome/servant/12312
LogFilename=servant.log
AutoShutdown=60
TTL=255
MsgLifetime=60
GetMsgLifetime=600
InitNeighbors=3
KeepAliveTimeout=7
MinNeighbors=2
CacheProb=0.1
StoreProb=0.1
NeighborStoreProb=0.1
CacheSize=1000
PermSize=20000

[beacons]
Retry=15
foo.usc.edu:12311=
foo.usc.edu:12312=
foo.usc.edu:12313=
foo.usc.edu:12314=
```

Please note that other than the differences noted here, *a beacon node behaves the same as a non-beacon node*.

Start a Bunch of Nodes Quickly

If you want to start a bunch of node quickly, you can use a shell script to start a few nodes in a batch on the same machine. For example, the following shell script should bootstrap into a SERVANT system with 4 nodes on the same machine:

```
#!/bin/csh -f

sv_node start-12311.ini &
sleep 2
sv_node start-12312.ini &
sv_node start-12313.ini &
sleep 3
sv_node start-12314.ini &
```

It could be the case that these 4 nodes are all beacon nodes if the [beacons] section of the .ini files has all of them cross listed and these nodes are started on the correct machine. (It should be clear that multiple nodes will run on the same machine.)

You might want to think about how to kill all these processes to make it easy for yourself to restart your experiment.

Hint: If a `sv_node` is started in the background, it cannot read standard input. So you might want to feed it an empty file as standard input. For example, if you create an empty file and call it `"null"`, you can then do:

```
sv_node start-12311.ini < null &
```

It is **not** a requirement that you are able to start your system this way. Only if you get tired of starting things in different window and want to do thing faster, you can do it this way. Also, you probably can only do this when everything is running smoothly.

Mini File System (for Part 2 only)

In order to manage files and metadata for a node, you must implement a simple mini file system specified here.

Data Files

Each data file must be stored as is. You must use numeric file name locally and the file name extension must be "data". You must store the metadata of a file with the same name (with a file name extension of "meta"). For example, the first time you want to store a file, you must store the file as "`1.data`" and the meta of this file must be stored in "`1.meta`". You must never reuse a file name, until a node has been reset.

You need a way to keep track of what file number to use the next time you want to store a file, and this information must be persistent even if your node crashes or get restarted.

Index Files

In order to speed up search, you must use relatively fast index structures. To speed up keyword searches, you **must** use a linear list of bit-vectors. To speed up name searches, you should use a binary search tree index structure (file names as keys). To speed up SHA1 hash searches, you should also use a binary search tree index structure (SHA1 values as keys). Each node in a index structure references a data file by file name.

All 3 index structures must have a disk image so that after a node restarts (after a crash or a shutdown), it can reconstruct the in-memory index structures in a timely fashion. Therefore, you must provide 3 different *index files* in the home directory of each node. The file names must be "`kwrd_index`", "`name_index`", and "`sha1_index`". These files are disk images of the corresponding memory index structures. The "`name_index`" and "`sha1_index`" must be **sorted**. (Please document the format of these files in your README file.)

There is no efficiency requirement here, but you should think about how to do this in *O(n)*, where *n* is the number of nodes in your data structure. (Please note that inserting a single node into a binary search tree is an *O(log(n))* operation. So, inserting *n* nodes into an initially empty binary search tree costs *O(n log(n))*.) If you have done something to make these operations efficient, please make sure you document document them in the `README` file of your submission.

Bit-vector

You are required to use a simplified Bloom Filter to manage *keywords* for contents that are stored/cached at a node.

In our simplified Bloom Filter, we will use two hash functions and a bit-vector of length *2n* bits. Each hash function maps a keyword to a bit position in the bit-vector. Every file is associated with a bit-vector and a set of keywords. For each keyword, two bits in the bit-vectors will be turned on (set to 1). (The positions of the bits are uniquely determined by the hash functions and the keyword.) So, if a file is associated with *j* keywords, at most *2j* bits of the bit-vector will be turned on. It's possible that multiple keywords can be mapped to the same bit.

The two hash functions we will use for a keyword *k* are:

SHA1(k) **mod** n

and

MD5(k) **mod** n

If *n* is a power of two and equals *2^m* (where the ^ is the exponentiation operator), then the above will be equivalent to taking the trailing *m* bits from SHA1(k) and the trailing *m* bits from MD5(k). In our case, the bit-vector is *2n* bits long. *SHA1(k)* ***mod*** *n* determines which bit in the leading *n* bits (or the *left half* of the bit-vector) should be on while *MD5(k)* ***mod*** *n* determines which bit in the trailing *n* bits (or the *right half* of the bit-vector) should be on.

When you perform a keyword search, you should generate a bit-vector based on the search keywords and perform a *bit-wise AND* with the bit-vector of each file. If the left half of the resulting bit-vector contains all zeros **or** the right half of the resulting bit-vector contains all zeros, then there is no common keywords between the search keywords and the keywords stored for a given file. If the left half of the resulting bit-vector is not a vector of zeros, **and** the right half of the resulting bit-vector is not a vector of zeros, then it is **possible** that there may be keyword match (since it's possible that multiple keywords can be mapped to the

same bit). In this case, you must compare all search keywords with file keywords and see if there is a true keyword match.

For this project, please use 512 as the value of *n*. Also, you must convert all uppercase letters in keywords to lowercase letters before you computer a bit-vector.

Bit-vector example (please refer to the example above):

> Let's take the keyword "`categories`" as an example. The SHA1 value of "`categories`" is:
>
> > 50b9e78177f37e3c747f67abcc8af36a44f218f5
>
> whose trailing 9 bits is "`0f5`", which equals 245 in decimal.
>
> The MD5 value of "`categories`" is:
>
> > b0b5ccb4a195a07fd3eed14affb8695f
>
> whose trailing 9 bits is "`15f`", which equals 351 in decimal.
>
> Since 245 (the SHA1 portion of the hash) indexes the left half of the bit-vector, it should correspond to bit 757 (=245+512) of the bit-vector. Turning on bits 757 and 351 in a 1024-bit bit-vector, we have:
>
> ```
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000020000000000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000000000008000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000
> ```
>
> Please note that 351=87*4+3. Therefore, there are 87 zero hex digits after the "8" above. (Each hex digit can be one of 0, 1, 2, ..., 9, a, b, c, d, e, f.) Please also note that 757=189*4+1; therefore, there are 189 hex digits after the "2" above.
>
> Please note that bit index starts from the right. So, to turn bit 0 on in a 1024-bit bit-vector, we have:
>
> ```
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000000000000000000000000000000000000000000000000000000 \
> 0000000000000001
> ```

Additional bit-vector examples have been added.

Directory Structure

Your node must use the following directory structure:

```
$(HomeDir)
  +- init_neighbor_list
  +- kwrd_index
  +- name_index
  +- sha1_index
  +- ... (other files you want to keep)
  +- files
       +- 1.data
       +- 1.meta
       +- 2.data
       +- 2.meta
       +- ...
```

where `files` is a subdirectory containing the actual data files and the corresponding metadata files.

Commandline Interface

The commandline interface mimics that of the `ftp` program. Here are some details.

- The following command prompt should be used:

  ```
  servant:PORT>
  ```

  where `PORT` is the port number of the node. There should be a space character after the prompt.

- To get status of the SERVANT network and write it to an external file, the user can enter one of the following:

  ```
  status neighbors <ttl> <extfile>
  status files <ttl> <extfile>
  ```

  where `<ttl>` is the TTL value that should go into the common message header, and `<extfile>` is the name of an external file (relative to the current working directory if it does not begin with a `/` character).

  A `<ttl>` value of zero means not to flood the status message (i.e., should only attempts to get status for the node where the command was issued).

  For the `status neighbors` command, the format of the external file should be in a form that can be read by `nam`. A sample of a 2-node network (named `status.out`) is provided here. To display the network, one can simply do:

  ```
  [BC: updated 3/19/2009]
  ~csci551/nam status.out
  ```

  The TA has written some notes about the `nam` file format.

  For the `status files` command, the external file should be a listing of the responses received. For each response, you should list the Hostname

and Hostport of the reporting host, the number of files it has, and the complete metadata for each of the files. For example, you may have (please replace the "..." with the correct information in the status response message):

```
nunki.usc.edu:12345 has 2 files
[metadata]
FileName=blondie1.mp3
FileSize=1874
SHA1=8ae005585be9c44ef1910d25dd6f8da58c432ab5
Nonce=...
Keywords=categories audio mp3 artist Blondie
Bit-vector=...
[metadata]
FileName=usctommy.gif
FileSize=1689
SHA1=11cddf2a0378d6a892cf43198847ce379e85d149
Nonce=...
Keywords=usc tommy trojan gif
Bit-vector=...
```

You do not *have to* output the number of files a node has. Alternatively, you can replace the first line in the above output with:

```
nunki.usc.edu:12345 has the following files
```

If the node has zero file, you should output:

```
nunki.usc.edu:12345 has no file
```

If the node has exactly one file, you should output:

```
nunki.usc.edu:12345 has the following file
```

- To store file `foo` into the SERVANT, the user should enter:

```
store foo <ttl> [tag1="value1" tag2="value2" ...]
```

where `<ttl>` is the TTL value that should go into the common message header. A `<ttl>` value of zero means not to flood the store message (i.e., should only attempts to store the file in the node on which the command was issued).

If `foo` begins with the `/` character, `foo` specifies an absolute path. Otherwise, `foo` is relative to the user's current working directory. (This means that you should call something like `getcwd()` to get the current working directory when a node starts.)

The tags and values are the metadata. They should provide useful information. For example, you can enter:

```
store blondie1.mp3 30 \
    categories="audio mp3" \
    artist="Blondie" \
    title="Heart of Glass" \
    url="http://www.blondie.net/" \
    additional_keywords="debra harry"
```

The tag names are arbitrary. Both tag names and values must be

treated as searchable keywords.

- To search for a file with an exact file name of `foo`, the user should enter:

  ```
  search filename=foo
  ```

  There should be no blank characters around the equal sign. To search for a file with an exact SHA1 hash of `bar`, the user should enter:

  ```
  search sha1hash=bar
  ```

  There should be no blank characters around the equal sign. `bar` must be 40 characters long (since you cannot enter a binary value using the keyboard, you must hexstring-encode the 20-byte hash value and enter the hexstring). To search for a file with keywords `key1`, `key2`, **and** `key3`, the user should enter:

  ```
  search keywords="key1 key2 key3"
  ```

  There should be no blank characters around the equal sign. The double quotation marks are mandatory unless there is only one keyword.

  Please note that all searches are case-insensitive. One way to do this is to convert everything to upper (or lower) case. For this project, please convert everyting to lower case.

- Responses to a search request should be presented as follows. A response should start with a number followed by its metadata. The number is to be used by the user to identify a file. Here is an example:

  ```
  [1] FileID=02adefc1dfc97a082fa18a5ef1e8c487259b7fb4
      FileName=foo
      FileSize=123
      SHA1=b83a758fecbefcd3ea547fbf0f9a97eba0ea984c
      Nonce=01b7a1bd6f169dde22518a865ab2f44c70fcab82
      Keywords=key1 key2 key3
  [2] FileID=45929c03a7c84687a73543cc348484edc3829496
      FileName=bar
      FileSize=4567
      SHA1=6b6c5636c484d47599d20191c3023b8a29b2fe11
      Nonce=fe1834fdf8cd7356ca11e0c77ac38d387e228f94
      Keywords=key4 key5
  [3] ...
  ```

  When the user presses `^C` (`Control-c`) on the keyboard, the displaying of search results is terminated (although the node may be receiving more search results) and a command prompt should be displayed.

- To retrieve the 2nd file listed in the previous search response, a user should enter:

  ```
  get 2 [<extfile>]
  ```

  A copy of the retrieved file should be saved into the current working directory so that the user can access it. If `<extfile>` is not specified, the filename to use should be the same as FileName in the file metadata. If such a file already exists, you should ask the user if the file should be replaced.

Please note that a `get` command should not be allowed if it does not immediately follow a `search` or a `get` command.

- To delete a file named `foo`, a user should enter:

  ```
  delete FileName=foo SHA1=6b6c... Nonce=fe18...
  ```

  with the appropriate SHA1 hash value and nonce for the file. There should be no blank characters around the equal signs.

  Upon receiving this user command, the node should check if it has a *one-time password* for this file. It is does, it should simply create a delete message and flood it to the whole network. If it does **not** have the corresponding password, it should prompt the user to see if it is okay to send a delete message based on the random password as follows:

  ```
  No one-time password found.
  Okay to use a random password [yes/no]?
  ```

  If the user enters anything that starts with the letter "y", it should generate a random 20-byte password, create a delete message and flood it to the whole network.

- To shutdown the node, a user should enter:

  ```
  shutdown
  ```

  Upon receiving this user command, the node should send itself a `SIGTERM` signal by calling `kill()`. The node should subsequently free up all resources and shutdown.

The commandline interface must not allow another command to run if it is waiting for response messages. Messages that have responses are **status**, **search**, and **get**. The user can interrupt (and terminate) these commands by pressing `^C` (`Control-c`) on the keyboard before these commands are completed. `^C` (`Control-c`) should never cause your program to terminate.

INI File Format

The basic format of a `.ini` file is as follows:

```
[section1]
key11=value11
key12=value12

[section2]
key21=value21
key22=value22
```

If there are duplicate sections within the same file, the first one should be honored and the rest should be ignored. If there are duplicate keys within the same section, the first one should be honored and the rest should be ignored.

Leading and trailing blank characters (<SPACE>, <TAB>, <CR>, and

<LF>) in section names, keys, and values must be removed. There is one way to include leading and trailing blanks in a value. This can be done with a pair of matching single or double quotation marks. In this case, the quotation marks should be removed, but what's between the quotation marks should not be touched. For example,

```
foo = " bar "
```

is *not* the same as

```
foo =   bar
```

This is because, in the first case, the value string is of length 5 (*not* 7) and in the second case, the value string is of length 3. Furthermore, the second case is identical to:

```
foo=bar
```

If the first character of a line is a semi-colon, the line is a comment and should be ignored. Blank lines should also be ignored.

In general, section names and keys should be treated as if they are case-insensitive. Also, the orders in which keys appear in a section should not matter, and the order in which section names appear in an INI file should not matter.

Compiling

Please use a `Makefile` so that when the grader simply enters:

```
make
```

an executable named `sv_node` is created.

Please make sure that your submission conforms to other general compilation requirements and README requirements.

Openssl

The `openssl` package can also be used to compute an SHA1 hash value from a memory buffer.

To use openssl on `nunki.usc.edu`, please see the additional notes on `openssl`. *(Please note that this is different from warmup project #1. Please do **not** use `/usr/lsd/openssl/default-0.9.7g` or `/usr/lsd/openssl/default`.)*

Additional information about the SHA1 algorithm can be found at http://www.openssl.org/docs/crypto/sha.html.

The `openssl` program can also be used to generate random numbers. To generate a 1024-byte random number, you can use:

```
openssl rand 1024
```

To generate a 20-byte random number and format it into a hexstring, you can use:

```
openssl rand 256 | openssl sha1
```

Similarly, to generate a 16-byte random number and format it into a hexstring, you can use:

```
openssl rand 256 | openssl md5
```

Please note that invoking `openssl` using `popen()` is a very expensive operation. So, for frequently needed `openssl` operations, you should call the corresponding `openssl/sha1` functions directly and link your code to the `openssl/crypto` library.

Additional Notes on File Retrieval (for Part 2 only)

- Before you flood a GET message, you should use the `FileID` to determine if the current node has the file. If it does, you should not flood a GET message. In this case, if the existing file is in the cache space, its status should be changed so that it is "*moved*" to the permanent space (you should also do whatever adjustments that are necessary).
- When a node successfully gets a file but there is **not** enough space in its filesystem, it should discard the file and print an error message on the console. In this case, a copy of the file should *not* be placed in the node's current working directory. The idea here is that "*you may not get this file if you don't server this file*".
- When a node successfully gets a file and there is enough space in its filesystem and an identical file (i.e., identical filename, SHA1, and nonce) already exists in the mini filesystem, in addition to placing a copy of the file in the node's current working directory, the following should happen.

  If the existing file is in the permanent space, then you are done.

  If the existing file is in the cache space, its status should be changed so that it is "*moved*" to the permanent space (you should also do whatever adjustments that are necessary).

Test Data

The test data used to test part (1) is available.

The test data used to test part (2) is also available.

Please do **not** use the port numbers in the testdata. Please only use ports that are assigned to you.

Project Report

In the `README` file which you will submit, please include a description of your program flow and any major design decisions you have made (e.g., how many threads you have and what do they do). Please do not repeat anything

that is already in this spec.

You are strongly encourage to have good comments in your code.

Grading Guidelines

Part (1) grading guidelines has been made available.

Part (2) grading guidelines has been made available.

Please do **not** use the port numbers in the grading guidelines. Please only use ports that are assigned to you. Your code is suppose to work with any port numbers above 1024.

Please run the scripts in the grading guidelines on `nunki.usc.edu`. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

Miscellaneous

- You are required to use *separate compilation* to compile your source code. You must divide your source code into separate source files in a logical way. You also must **not** put the bulk of your code in header files!
- A SERVANT node must be implemented as a *single process*. If you are not familiar with pthreads, you should read the *Pthreads Programming* book mentioned in the recommanded textbook section of the Course Description.
- Your must not do busy-wait! If you run `"top"` from the commandline and you see that your program is taking up one of the top spots in CPU percentages and show high percentages, this is considered busy-wait. You will lose a lot of points if your program does busy-waiting.

  It's quite easy to find the offending code. If you run your program from the debugger, wait a few seconds, then type <Cntrl+C>. Most likely, your program will break inside your busy-waiting loop! An easy fix is to call `select()` to sleep for 100 millisecond before you loop again.

- Some people are having trouble with `connect()`. Here's some information that may help you.

  Please note that you if you fail to connect to another node for whatever reason (e.g., the node is not up), you must get a new socket (by calling `socket()`) the next time you try to connect.

- Some papers about other peer-to-peer systems are in the class reading list.
- For part (2), the maximum size of a memory buffer is limited to **8,192 bytes**. There is no limit for part (1).
- If you don't have a favorite binary search tree, you might want to checkout GNU's libavl.
- If you don't have a favorite INI file parser or a string based hash table

data structure, you might want to checkout Nicolas Devillard's iniparser (and modify it if necessary).
- If you have `.nfs*` files you cannot remove, please see notes on `.nfs` files.

[*Last updated Sun Apr 19 2009*] [*Please see copyright regarding copying.*]