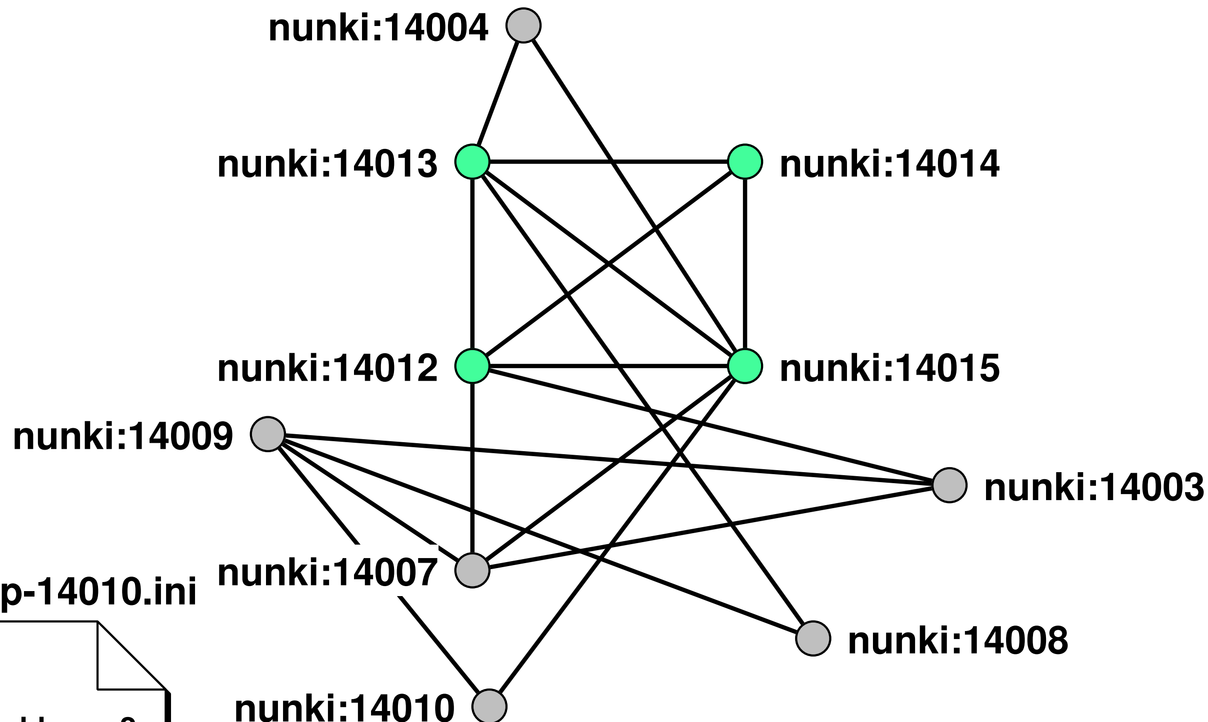
CS551
Final Project Part (2)

Bill Cheng

http://merlot.usc.edu/cs551-f12

SERVANT Network



startup-14003.ini

```
[init]
InitNeighbors=3
MinNeighbors=1

[beacons]
nunki:14012=
nunki:14013=
nunki:14014=
nunki:14015=
```

startup-14010.ini

```
[init]
InitNeighbors=2
MinNeighbors=1

[beacons]
nunki:14012=
nunki:14013=
nunki:14014=
nunki:14015=
```

init_neighbor_list

```
nunki:14009
nunki:14015
```

Part (2) Message Types

- Part (2): think google and napster (35% project grade)
  - Store
    - probabilistic storing of files
      - node that initiates STORE always store the file
      - use NeighborStoreProb to decide if it forwards to a particular neighbor
      - when a node gets a STORE request, use StoreProb to decide if it should cache a copy of the file
  - Search
  - Get
    - probabilistic/opportunistic caching of files
      - node that initiates GET always store the file
      - if forwarding GET response, use CacheProb to decide if it should cache a copy of the file
  - Delete

Part (2) Is Based On Part (1)

- But,
  - no JOIN
    - every regular node will start with a good `init_neighbor_list` file
    - make sure your code can parse it
  - no CHECK
    - do not initiate or forward CHECK messages
    - the startup configuration file has NoCheck=1

Keywords

STORE command

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

```
[metadata]
FileName=blondie1.mp3
FileSize=4885526
SHA1=730764e28a5b66e3f95ceadc976c038d389bd89e
Nonce=b56dba4b2ec8f224de8fc45d6041cdb9f2db9d69
Keywords=categories audio mp3 artist Blondie \
  title Heart of Glass \
  url http://www.blondie.net/ \
  additional_keywords debra harry
Bit-vector= \
  110000100000000420020000000000000000000000000000 \
  100000000000000000200000000000000000010000010004 \
  000000000004800000000800000000000000000000000000 \
  000000000000000000100008800000000000000040048400 \
  000002100000000000000810000000000000200002000200 \
  0000000000000000
```

Keywords (Cont...)

- Content-based addressing
  - mini file system
    - directory and files
      1) think of files as UNIX inodes
      2) directory contains description (metadata) of files
         no need for subdirectories

- Caching is a local behavior
  - every node can have its own implementation

Searching

- **Searching**
  - at commandline, think google.com but slightly different
  - case-insensitive
  - AND searches only
    - e.g., search keywords="glass heart of" will only match a file with metadata containing *all* 3 words
  - example of responses

```
[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] ...
```

Searching (Cont...)

- GET (i.e., retrieving)
  - e.g., get 2 [<extfile>]
  - flood a GET request with a FileID in the message
    - so that only one node will respond
    - you can create a FileID when you create a SEARCH response message
    - keep FileID in memory only

- Opportunistic caching
  - to increase performance (as the expense of extra storeage)
  - for nodes that did not initiate a GET request, cache the file with CacheProb
    - if CacheProb is 0.3, you should cache 30% of the time
    - call `srand48()` during initialization
    - call `drand48()`, if returned value < CacheProb, cache the file

Index Files

- You must implement 3 index structures to support 3 types of searches efficiently
  - one maps a bit-vector to a list of file references
  - one maps a filename to a list of file references
  - one maps a SHA1 value to a list of file references
- Although the spec says that you need to use BSTs for filename and SHA1 indices, using a *sorted linear list* is fine
- When a node goes down, you need to *externalize* these index structures so that when you restart, it can recover the index structures quickly
  - `kwrd_index` maps a bit-vector to a list of file references
  - `name_index` maps a filename to a list of file references
  - `sha1_index` maps a SHA1 value to a list of file references

Delete

- Delete a file
  - only the creator of a file can delete it
    - on file creation (i.e., STORE), generate a random *password* using `GetUOID()`
    - this is a *one-time password*
    - calculate *nonce=SHA1(password)*
    - *nonce* is part of *file metadata*
  - e.g., delete FileName=foo SHA1=6b6c... Nonce=fe18...
    - FileSpec is:

```
FileName=foo
SHA1=6b6c...
Nonce=fe18...
Password=27c3...
```

  - verifying one-time password
    - if SHA1(password) == nonce, delete the file

Bit-Vector

- Bit-vector as a simplest form of a *Bloom Filter*
  - directory entry contains a bit-vector (long, e.g., 1024 bits)
  - map all possible words to the bit-vector
    - for example, use SHA1 mod 1024 to produce a bit index into the bit-vector
    - many words can map to the same bit index
  - take all keywords, compute bit index, set all these bits to one, store bit-vector in directory entry
  - for a single-word query, compute bit index of this word
    - if the corresponding bit in a bit-vector is set, there is a *possible* match; in this case, do string compare
    - if the corresponding bit in a bit-vector is *not* set, there is *no possibility* of a match; try the next directory entry

Bit-Vector (Cont...)

- 2 bit-vectors (n bits on the left and n bits on the right)
  - n = 512 for our project
  - concatenated into one 1024 bit string for storage in *File Metadata*, hexstring encoded
  - for a keyword k:
    corresponding bit in left bit-vector: SHA1(k) mod n
    corresponding bit in right bit-vector: MD5(k) mod n
  - Ex: single keyword, k = "categories"
    - echo -n "categories" | openssl sha1
      - 50b9e78177f37e3c747f67abcc8af36a44f218f5
    - SHA1(k) mod n (same as taking the right-most 9 bits)
      - 0x0f5 (= 245 in decimal)
    - echo -n "categories" | openssl md5
      - b0b5ccb4a195a07fd3eed14affb8695f
    - MD5(k) mod n = 0x15f (= 351 in decimal)

Bit-Vector (Cont...)

- Ex: single keyword, k = "categories" (cont...)
  - need to turn on bits 757 (=245+512) and 351
    - bit index is from the right

      therefore, to turn bit 0 on:

```
000000000000000000000000000000000000000000000000 \
000000000000000000000000000000000000000000000000 \
000000000000000000000000000000000000000000000000 \
000000000000000000000000000000000000000000000000 \
000000000000000000000000000000000000000000000000 \
0000000000000001
```

    - 351 = (87*4+3)

      shift the above bit pattern left 351 bits
    - 757 = (189*4+1)

      shift the above bit pattern left 757 bits

```
000000000000000000000000000000000000000000000000 \
000000000000000000200000000000000000000000000000 \
000000000000000000000000000000000000000000000000 \
000000000000000000000000800000000000000000000000 \
000000000000000000000000000000000000000000000000 \
0000000000000000
```

Node 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
       +- ...
```

- `kwrd_index` is indexed by bit-vector
- `name_index` can be a BST, indexed by file name
  - e.g., "blondie1.mp3" → 5 (if 5.data stores blondie1.mp3 and 5.meta stores the corresponding metadata)
- `sha1_index` can be a BST, indexed by SHA1 hash of files
- you can have additional files
  - e.g., 1.pass to store the one-time password that corresponds to 1.data, 1.extra to store extra information (can't think of anything at this point)

Probabilistic Flooding for STORE Messages

- STORE message is flooded probabilistically
  - for each neighbor, use ***NeighborStoreProb*** to decide if a STORE message should be sent or forwarded
    - call `drand48()`, if returned value < NeighborStoreProb, send/forward the STORE message
  - when a node receives a STORE message, use ***StoreProb*** to decide if the file should be cached
    - call `drand48()`, if returned value < StoreProb, cache the file
    - if the node decides not to cache the file, it should not continue to flood

Permanent vs. Cache Storage and LRU

- Two types of storage areas:
  - *cache* storage space is subject to LRU
    - size is specified by the *CacheSize* key
  - *permanent* storage space is not subject to LRU
    - size is up to filesystem limit (or your disk quota)
    - if a node *initiates* a GET or a STORE, the file goes into its permanent space
    - if a file suppose to go into permanent space and there is not enough space, do not keep the file
- Need to keep track of which file is in cache and which file is in permanent storage
  - if a file is referenced in LRU, then it's in the cache

Cache Storage and LRU

- Cache storage
  - if a file is not suppose to go into permanent space, it should be stored in the cache space
  - if (filesize > CacheSize), do not store it
  - while (filesize + current usage > CacheSize)
    - start deleting files from the head of the LRU list (this would decrease current usage)
- LRU
  - cache storage space is subject to LRU
    - a file is considered accessed if it is selected in a SEARCH response
      - move file reference to the end of the list
  - when a node goes down, you need to ***externalize*** the LRU list so that when you restart, it can recover the LRU list