CS551
Final Project Part (1)
Bill Cheng

http://merlot.usc.edu/cs551-f12
Peer-to-Peer File Sharing System

**Peer-to-peer**: if one is using the system, one must be sharing his/her resources

Application layer network named *SERVANT*
- the words "packet" and "message" are interchangeable
- messages are sent via flooding (no loops)
- reply to messages are sent along the path it was received

Bi-directional connections between neighbors
- $A \rightarrow B$ and $B \rightarrow A$ must use the same connection

Two types of nodes
- *beacon nodes*: well known addresses that every node knows, fully connected to form the core of the network
- regular nodes
- operationally, beacons are just like regular nodes (except a beacon does not need to join the network)
Message Types

Part (1): form and maintain the network (45% project grade)
- Join
- Hello
- Keepalive
- Notify
- Status
- Check (keep things connected)

Part (2): think google and napster (35% project grade)
- Store
- Search
- Get
- Delete
Message Format

Common header

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

- UOID
- Message Type
- Data Length
- TTL
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];

    sprintf(str_buf, "%s_%s_%1ld",
        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;
}
Requests

For example, join request

Requests are *flooded* to the entire SERVANT network
- anonymity of the message senders and message receivers
- must avoid loops
- nodes must *cache* a copy of any flooded message
  1) for *loop detection*, i.e., drop duplicate messages based on UOID
  2) to *route a response message* to the originator of the corresponding request message
- message cache expires after *MsgLifetime* or *GetMsgLifetime*
Responses

For example, join response

Do not know exactly *who* initiated the join (only know the UOID of the join request)

- use the UOID of the join request to do *routing*
- intermediate nodes must *cache* a copy of the join request message and which link it came from in order to send the join response
- join request initiator uses *JoinTimeout*
Network Formation

startup-14012.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

beacon nodes are fully connected
Join

startup-14012.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

nunki:14012

nunki:14013

nunki:14014

nunki:14015

startup-14007.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

nunki:14007

join req

send join request to a beacon

beacon nodes are fully connected
Join

- Beacon nodes are fully connected
- Send join request to a beacon
- Join request is flooded to the whole network, send join reply

**startup-14012.ini**

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

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

**startup-14007.ini**

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

[beacons]
nunki:14012=
nunki:14013=
nunki:14014=
nunki:14015=
```
Flooding stops if packet already seen.

Beacon nodes are fully connected.

Send join request to a beacon.

Join request is flooded to the whole network, send join reply.

Flooding stops if packet already seen.
Join

**startup-14012.ini**

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

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

**startup-14007.ini**

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

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

- Beacon nodes are fully connected
- Send join request to a beacon
- Join request is flooded to the whole network, send join reply
- Flooding stopped if packet already seen
- Sort replies, write bottom ones to init_neighbor_list in HomeDirectory
Hello

start-up-14012.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

start-up-14007.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

init_neighbor_list

nunki:14013
nunki:14014
nunki:14012
nunki:14015

- beacon nodes are fully connected
- send join request to a beacon
- join request is flooded to the whole network, send join reply
- flooding stopped if packet already seen
- sort replies, write bottom ones to init_neighbor_list in HomeDirectory
- restart, no need to join, just say hello
Hello

**startup-14012.ini**

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

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

**startup-14007.ini**

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

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

- Beacon nodes are fully connected
- Send join request to a beacon
- Join request is flooded to the whole network, send join reply
- Flooding stopped if packet already seen
- Sort replies, write bottom ones to init_neighbor_list in HomeDirectory
- Restart, no need to join, just say hello
- Say hello back
Node Goes Down

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

check

check

check
Node Goes Down (Cont...)

![Graph showing network nodes and connections with check rply and init_neighbor_list.

```
startup-14010.ini
[init]
InitNeighbors=2
MinNeighbors=1

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

init_neighbor_list
nunki:14009
nunki:14015

startup-14003.ini
[init]
InitNeighbors=3
MinNeighbors=1

[beacons]
nunki:14012=
nunki:14013=
nunki:14014=
nunki:14015=
```
Node Goes Down (Cont...)

- nunki:14015 goes down
- network is partitioned, nunki:14010 will not get any check response messages (even if there are nodes connected to it from below)

startup-14010.ini

[init]
InitNeighbors=2
MinNeighbors=1

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

init_neighbor_list

nunki:14009
nunki:14015

startup-14003.ini

[init]
InitNeighbors=3
MinNeighbors=1

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

18
Node Goes Down (Cont...)

- nunki:14010 must delete its init_neighbor_list file
- nunki:14010 must join the network (start from scratch), it will reappear somewhere else in the network
Ex: `start-14014.ini`

```
[init]
Port=14014
Location=4294967295
HomeDir=/YOURHOME/servant/14014
LogFilename=servant.log
AutoShutdown=60
TTL=255
MsgLifetime=60
GetMsgLifetime=600
InitNeighbors=3
JoinTimeout=5
KeepAliveTimeout=7
MinNeighbors=2
NoCheck=0
CacheProb=0.1
StoreProb=0.1
NeighborStoreProb=0.1
CacheSize=1000

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

- **Port** is the well-known port that this node listens to
- the black keys in the `[init]` section are optional
- check the spec for their default values
Don’t complain it takes too much effort to parse the file!!

- suggestion: utility file (for the rest of your grad school)
  - char *GetALine(FILE*): read an arbitrary long line
    - use malloc() and realloc()
  - void TrimBlanks(char*): get rid of leading and trailing space and tab characters
  - int GetKeyValue(char *buf, char separator, char **ppsz_key, char **ppsz_value): get key and value from an input buffer

```c
char *psz_value=strchr(buf, separator);
if (psz_value == NULL) return ERR_CANNOT_FIND_SEPARATOR;
*psz_value++ = '\0';
TrimBlanks(buf);
TrimBlanks(psz_value);
if (ppsz_key != NULL) *ppsz_key = buf;
if (ppsz_value != NULL) *ppsz_value = psz_value;
```
typedef struct tagKwInfo {
    int id;
    char *key;
    /* what else? */
} KwInfo;

#define KW_PORT 1001
#define KW_HOMEDIR 1002
...  
#define KW_PERMSIZE 10xx

static KwInfo gkwinfo[] = {
    { KW_PORT, "port" },
    { KW_HOMEDIR, "homedir" },
    ...
    { KW_PERMSIZE, "permsize" },
    { INVALID, NULL }
};

char *psz_key=NULL, *psz_value=NULL;

if (GetKeyValue(line, '=', &psz_key, &psz_value) == 0) {
    KwInfo *pwi=gkwinfo;
    int found_id=(-1);
    for (;; pwi++) {
        if (pwi->key == NULL) break;
        if (strncasecmp(pwi->key,psz_key) == 0) {
            found_id = pwi->id;
            break;
        }
    }
    if (found_id == (-1)) {
        return ERR_UNRECOGNIZED_KEY;
    }
    switch (found_id) {
        case KW_PORT: ...
        case KW_HOMEDIR: ...
    }
}
Data Structures

- List
  - from warmup #2

- Efficient data structure
  - Binary Search Tree (BST)
    - you can use `libavl` if you don’t have something you already like
    - not required
  - Bloom Filter
    - required
    - for part (2), you don’t have to worry about it for now
How to Manage Timers

Networking programming often requires you to manage many timers

- e.g., MsgLifetime
  - you need to implement a message cache, keyed on UOID
    - drop duplicate messages
    - route response messages
  - every time you cache a message, conceptually, you should start a timer
  - when the timer expires, you can remove the message from your message cache data structure
  - need to cache a message for quite a long time
    - you can end up with thousands of timer
- e.g., KeepAliveTimeout
  - not as many timers, but you need to keep track of them
How to Manage Timers (Cont...)

Solution: have a timer that goes off every second
- for all the timeouts that are specified as multiple of seconds
  - if a timer suppose to go off in 9 seconds, does it matter if it goes off 9.7 seconds later?
- if a timeout is suppose to be for 15 seconds, initialize a count of 15
  - every time the timer goes off, *scrub* all timer-related data structures
  - if a count reaches zero, delete the object from the data structure
- you can use a timer thread for this
- if you have events that needs to be timed-out in resolution of multiple hundreds of milliseconds, use another timer that goes off every 100 millisecond
Soft Restart

Here is a simple way to implement soft restart:

```c
int main(int argc, char *argv[]) {
    gnShutdown = FALSE;
    while (!gnShutdown) {
        Init();
        Process();
        CleanUp();
    }
    return 0;
}
```

- only set `gnShutdown` to TRUE if you want the program to exit (such as when the autoshutdown timer goes off)
- otherwise, you are doing a soft restart
- in `CleanUp()`, you can clean up everything
  - kill all threads, free up all memory, reset all variables (except `gnShutdown`)
- keep the state of your program in your node’s HomeDir
A Design, Just A Design

sv_node

Timer

Wait for Shutdown

Event Queue

Event Dispatcher

Node Processing

Files

Request

NR1

NR2

NRn

Request

Keyboard

User Input

other nodes

NW1

NW2

NWn

...
A Design, Just A Design (Cont...)

- Event-driven style

- Identify all your threads
  - draw them as circles
  - on the previous slide, there are 2 threads to handle communication with a neighboring node
    - this is not the only way, you need to decide what you are most comfortable with

- Identify all your shared data structures
  - draw them as queues
  - use shared data structures for thread-to-thread communication and thread synchronization
  - protect each shared data structure with a mutex
  - there may be other shared data structures that needs to be protected by mutexes, e.g., logfile
Keep Track of Neighbors

How do you keep track of neighbors so you can look it up?

For example, if you get a message from a neighbor and want to forward it to all other neighbors:

- should you use socket descriptor number to distinguish different neighbors?
  - probably not a good idea
  - socket descriptors get reused as you lose and gain connections

- should you use hostname and port numbers?
  - may not be a good idea
  - neighbors go up and down
Keep Track of Neighbors (Cont...)

Solution: use a connection data structure/object
- Each "connection" has a unique numeric ID
  - monotonically increase it when you need a new ID
- Store neighbor hostname and port numbers in it
- Store socket descriptors in it
- Write a bunch of utility functions/methods for it

When you want to refer to something related to a neighbor, have it refer to a connection object
- For a network-read thread, have it reference a "connection" (or a connection ID)
- For a message in the message cache, have it reference a connection number

Hopefully, this can remove some ambiguities