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

Application layer network named SERVANT

Messages are flooded to the entire SERVANT network

Requests are sent to the nearest SERVANT

Peer-to-peer file sharing system

Join
Hello
Keepalive
Check (keep things connected)
Status
Get
Delete
Notify

2) For log direction, i.e., drop duplicate messages
3) To make a response message the originating of the request on ID, i.e., drop duplicate messages

TTL
Message Type
UOID

Request

For example, join request

void 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;
}

Message Type Formats

Message Types

Probabilitically unique

Requests

Join
Hello
Keepalive
Check (keep things connected)
Status
Get
Delete
Notify

Part (1): form and maintain the network (45% project grade)

Part (2): think google and napster (35% project grade)
For example, join response

Join Request
- UOID
- Port
- Hostname

Ran-
- Random

use the UOID of the join request to do routing

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

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

beacon nodes are fully connected

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

flooding stopped if packet already seen
Node Goes Down (Cont...)

- nunki:14013
- nunki:14012
- nunki:14014
- nunki:14015
- nunki:14010 must delete its init_neighbor_list file
- nunki:14009
- nunki:14015
- nunki:14010 must join the network (start from scratch), it will reappear somewhere else in the network

Ex:

start-14014.ini

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]

foo.usc.edu:12311=foo.usc.edu:12312=foo.usc.edu:12313=foo.usc.edu:12314=

Table driven

Data Structures

- Binary Search Tree (BST)
- Bloom Filter

Efficient data structure

Bloom Filter

require

not required

for part (2), you don't have to worry about it for now

Networking programming often requires you to manage many timers:

- 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
- also, every time you send a message, conceptually, you do not remove messages you've sent in the last TTL

How to Manage Timers

For part (2), you don't have to worry about it for now
How to Manage Timers (Cont...)

For all the timeouts that are specified as multiples of seconds, a timer that goes off every second will work. If a timeout is specified to occur in 9 seconds, does it matter if the timer goes off in 9.7 seconds instead? If a timeout is specified for 15 seconds, initialize a count of 15. Every time the timer goes off, decrement the count by 1. When the count reaches 0, delete the object from the data structure. You can use a timer thread for this. For events that need to be timed-out in resolution of multiple hundreds of milliseconds, use another timer that goes off every 100 milliseconds.

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). In `CleanUp()`, you can clean up everything: kill all threads, free up all memory, reset all variables (except `gnShutdown`). Otherwise, you are doing a soft restart. Keep the state of your program in your node's HomeDir.

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

Identify all your threads. Draw them as circles. For a network-read thread, have it reference a "connection" (or a connection ID). For a network-write thread, have it reference a "connection" (or a connection ID). Try to inter-thread communication if you need to communicate with a neighboring node. Store neighbor information and port numbers in a "connection". In the previous slide, there are 2 threads to handle communication and thread synchronization. You can use a mutex to protect each shared data structure. There may be other shared data structures that need to be protected by mutexes, for example, the logfile.

Keep Track of Neighbors

Solution: use a connection data structure/object. Each "connection" has a unique numeric ID. Store neighbor hostname and port numbers in it, store socket descriptors in it. When you want to refer to something related to a neighbor, have your thread reference a "connection" (or a connection number). Monotonically increase the connection number when you need a new ID.

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

Solution: use a connection data structure/object, which has a numeric ID. When you want to refer to something related to a neighbor, have your thread reference a "connection" (or a connection number). Monotonically increase the connection number when you need a new ID. You can use a mutex to protect each shared data structure. There may be other shared data structures that need to be protected by mutexes, for example, the logfile.