**CS1292 WorkShop**

**Development of a Simple Name Server**

**Prepared by**

Student Name: Chang Man Wai (91xxxxxx)

Chan Ming-lap (91xxxxxx)

Group: 1

Course: MSc in Computer Science (54802-1)

Tutor: Raymond Chan

Date: 21 February 2023

**Architecture of the Name Server**

Name server provides white pages services to a data network. It provides a mapping of objects onto network addresses. Objects are usually identified by character strings called names. If a user wants to talk to a node, the network address of the node must first be resolved by enquiring the name server using the name of the node. The user program will establish a connection to the name server, sends a message specifying the object's name, and the name server will sends back the network address of the node.

The name <-> address mapping is to be supplied by either the network administrator who create the name <-> address mapping table for the name server, or by having all the objects in the network registering at the name server voluntarily. The second approach is employed in this lab exercise.

Our name server is developed using the iterative server model, that is, the name server will go into an infinite loop servicing client requests queued in a message queue. The communication protocol between the server and the clients will be connection-oriented. As a result, the clients have to know beforehand the host name in which the name server resides. Association between the clients and the server is done by the Berkeley sockets, requiring explicit connection. This setup makes the response time of the name server deterministic.

socket() socket()

bind()

listen()

accept() connect()

read() write()

write() read()

close() close()

The internal database will be kept completely in memory to speed up searching speed and hence reduces client wait time. It is a simple linked list of name <-> address relation entries, sorted in ascending order of names. The sorting is performed during addition of entries into the table. To look for an entry, only a simple linear search over the names is needed. Dynamic memory allocation is used to create a table entry. As a result, the size of the table is virtually unlimited.

**Data Structures**

There are two important data structures in our name server: *message* and *name table*.

The *message* defines the format of information to be exchanged between the server and the client. All structure definitions could be found in the file *NSERVER.H*.

From Client to Server:

Command Object's Name Address

Command = REGISTER | MODIFY | DELETE | ENQUIRE

From Server to Client:

Result

Result = OK | ALREADY EXISTS | UNKNOWN

Address

Address = NULL | (address of an object)

The *name table* is a list of records. This table is maintained at the name server. Each entry in the table contains the name and its location/internet address, plus a pointer pointing at the next entry. New entries are added when the client send a REGISTER message to the server and be removed when the object sends a DELETE message to the name server.

Name[1] Address[1] Next

Name[2] Address[2] Next

**Services**

Only four system services are availble from the name server. They are represented using the following simplified Z schemas:

NAME : set of strings;

ADDRESS : set of network address;

name\_table : NAME -> ADDRESS

1. REGISTER

client:Send a message containing the new name and its node address to the name server.

server:

Register

name? : NAME;

addr? : ADDRESS;

name E dom name\_table

name\_table' = name\_table U (name? -> addr?)

2. DELETE

client:Send a message containing the name of object to the name server.

server:

Delete

name? : NAME;

addr : ADDRESS;

(name? -> addr) E name\_table

name\_table = name\_table' U name? -> addr

3. MODIFY

client:Send a message containing the name of object whose address is to be modified.

server:

Modify

name? : NAME;

addr?, addr : ADDRESS;

(name? -> addr) E dom name\_table

name\_table' = name\_table / (name? -> addr) U (name? -> addr?)

4. ENQUIRE

client:Send a message containing the name of object whose address is to be obtained.

server:

Enquire

name? : NAME;

addr! : ADDRESS;

(name? -> addr!) E name\_table

name\_table' = name\_table

**User Interface**

There are only two programs: *client* and *server*.

To bring up a server:

1. Choose a workstation.

2. Open a Command Tool window in OpenWindows.

3.Execute the name server program by typing "server" at the UNIX command prompt. You will see the following screen:

**venus:** server

server: Socket is <1234>

The server program will display a number which the actual socket number availble for connection with clients. You will need this number when you run the client program.

You can start the client program at any workstation. Execute the client program by simply "client" at the command prompt will bring up a help screen, as shown below:

**venus:** client

CLIENT

Usage: client <host> <port> <cmd> <name> <addr>

<cmd> = 4 for DELETE

<cmd> = 3 for MODIFY

<cmd> = 2 for ENQUIRE

<cmd> = 1 for REGISTER

To register a name, try the command sequence below.

**venus:** client localhost 1234 1 name my\_address

**taurus:** client venus 1234 1 name

The <port> is the number you saw after you executes the server program. <host> is the workstation in which you start the name server.

Similar command sequences could be used to delete a name, enquire the address of a name and modify the address of a name.

**taurus:** client venus 1234 2 name

Object <name> sits at <my\_address>

But then you don't need to supply <addr> since it is not required by the three operations.

**Error Handling**

1.Reading/writing socket stream

A socket stream is not as stable as it is expected. A successful transmission of messages through a socket my required multiple read() or write() operations over the socket. This problem may be caused by the inherent buffer limit of the sockets defined by different implementation of UNIX. As a result, two functions, namely readn() and writen(), are created to ensure that exactly n bytes of data is sent or received completely to or from the source.

2.Duplicated object name

Different objects may register the same name in the name server. This will introduce ambiguities in resolving names into network addresses. A check is thus made during name registration to ensure uniqueness of name <-> address mapping.

3.Network reliability

Murphy's law stated that things that might go wrong would go wrong. As a result, a check is always made after every call the networking primitives. Any error would lead to immediate abort of the server or the client program.

**Further Directions**

**Management of Concurrent Requests**

In our implementation, a client must first connect with the name server before it could receive services from the server. When one client is connected to the name server, the other clients must wait for the server to complete the request. There are at least two ways to improve performance. They both aim to reduce the waiting period owning to the inability of our name server to entertain multiple requests concurrently.

A better model would have structured the name server as one dispatch process and multiple servicing processes. The dispatch process is responsible for capturing all client requests. For each client request, the dispatch process would spawn one servicing process that will perform the actions according to client requests. Parameters are passed to the servicing process through shared memory by the dispatch process.

Integrity of the name table would be enforced by the concurrency control primitives of UNIX. If the name table is kept in the file system, file and record locking schemes would be used. If the name table is to be kept in memory, semaphores would be employed. Each entry in the table would be associated with one semaphore. In both cases, the table entry must be exclusively locked before the handling process could write on it.

No changes is needed for the client programs. The partial pseudo-code of the new server program was described below:

dispatch\_process() {

open socket;

loop {

accept client request;

spawn servicing process;

}

}

servicing\_process() {

lock appropriate table entry;

perform command with parameters;

unlock table entry;

reply client through client port;

close client connection;

}

**Server Reliability**

Being a centralized model, our implementation of name server is highly vulnerable to node failures in which the name server resides. A better approach is to have a redundant name server, or a mirrored name server that runs on a different network node. The primary and the mirrored name servers are actually two processes which exchanges messages constantly through the network.

The two name servers could be brought up independently, which then synchronize their activities. When the primary server fails, the operating system would automatically route service requests to the redundant name server. At the same, the failed name server process would be killed and restarted. This fault-tolerance model demands a reliable communication network and operating system support.

**Distributed Name Server**

An even more reliable and potentially more efficient way of implementing a name server is to distribute the naming server databases to all name server nodes across the network. Updating of information in the replicated databases is to be achieved by distributed algorithms like the well-known Two-Phrase or Three-Phrase Commit protocol. Clients could then query the closest name server for information rapidly. The trade-off is the extra overhead in updating the databases.

**Hashing to Speed Up Searches**

The current linear searching algorithm is less than optimal. A typical name <-> address table may have over a thousand entries. To speed the search, an hash function may be included, or a B-tree should be built instead.