Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

Design Document – ELEC 477– Lab 2

1.0 - Main Problem

The task for this assignment was adding a flat namespace to the Remote Procedure Call (RPC) simulation and making the remote procedure calls more robust. In the previous assignment, the RPC system supported basic client-server communication, however, it was limited in the sense that the clients needed to know the specific server address to make requests. Through the introduction of a flat namespace, services can now be found using logical names, rather than relying on hard-coded addresses, simplifying the process of clients connecting to services.

2.0 - Design Solution

To address these goals, our solution involved modifications and additions to the existing RPC simulation framework, provided in the A1 solutions. The lab was completed by testing two components, the directory service, and re-instantiating the Key-Value Service to include the functionality of a directory service through communication with the directory service's client stub. Additionally, ensuring "at least once" semantics were included throughout.

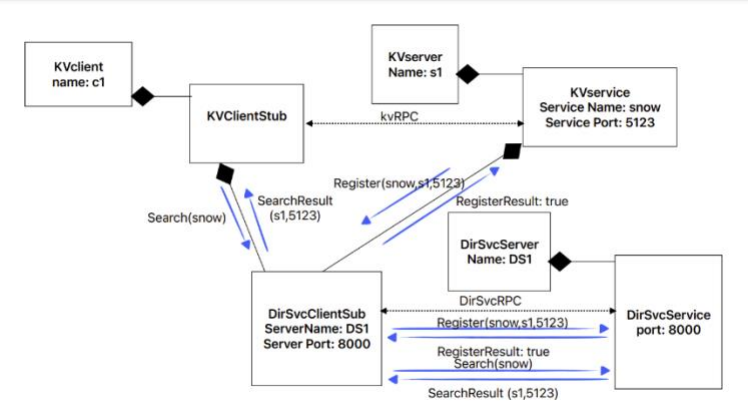

Figure 1: Naming Service System and implementation to kvRPC

2.1 – Directory Service

The directory service acts as a centralized repository where services can register their availability, allowing clients to search for and connect to desired services seamlessly. It maintains a database of service names, associated with their network locations (e.g., server names and ports), where clients can query the directory to find services matching specific criteria. The key components used to create the directory service are explained below.

2.1.1 – Directory Service Protocol Buffer (E477ServiceDirectory.proto)

To setup the directory, the first step was to create a Directory Service protocol buffer file separate from the KV Service file. The *DSRequest* and *DSResponse* functions were extended to handle the tasks for

registering, searching, and deleting a service from the directory map. The *DSReques* was used to handle the different types of requests and *DSResponse* encapsulates the types of responses.

```
message DSRequest {
    uint32 magic = 1;
    uint32 version = 2;
    uint32 serial = 3;
    oneof function {
        registerServiceRequest registerServiceArgs = 4;
        searchServiceRequest searchServiceArgs = 5;
        deleteServiceRequest deleteServiceArgs = 6;
    }
}
```

Figure 2: DSRequest message struct

For registering a service, the protobuf schema generated from the E477DirectoryService namespace takes in *serviceName*, *serverName* and *port* number as the key arguments for a request and a response object returns a Boolean *success* object to confirm it has successfully registered. For deleting a service, the request object takes in *serviceName* and returns the Boolean *success* object to confirm successful deletion from the directory. And finally, for searching an object, since a structure needs to be returned, the request takes in the *serviceName* and returns a Boolean called *found* to confirm whether the item is currently in the directory, and then returns the *serverName* and the *port*.

Once compiled the protobuf schema packages all the messages under the E477DirectoryService namespace and provides and efficient means of communication between a directory service client and server. These are found in the rest of our code following formats as shown below.

```
// searching for server
if(receivedMsg.has_searchserviceargs()) {
    cerr << "searching for server in directory" << endl;

    const E477DirectoryService::searchServiceRequest &searchReq = receivedMsg.searchserviceargs();

    string name = searchReq.servicename();

    struct directoryRecord searchedRecord = searchForService(name);

    bool inDirectory = false;

    if(searchedRecord.server != "" && searchedRecord.port != 0) {
        cerr << "search results good, found server - " << searchedRecord.server << " and port - " << searchedRecord.port << endl;
        inDirectory = true;
    } else {
        cerr << "couldn't find server" << endl;
        inDirectory = false;
    }

    E477DirectoryService::searchServiceResponse *searchResponse = replyMsg.mutable_searchserviceres();
    searchResponse->set_found(inDirectory);
    searchResponse->set_servername(searchedRecord.server);
    searchResponse->set_port(searchedRecord.port);
```

Figure 3: Directory Service protobuf schema search functions - sample use case

This example shows the protobuf functions used within the sdservice.cpp file to identify a search service request. It first filters whether the received message is a search request by using the has_searchserviceargs() proto function, then performing the search of the directory with the searchForDirectory() it creates a directoryRecord object ('searchedRecord'), which we will go into detail with later, and finally sets the results on the reply message.

2.1.2 – Directory Service Custom Structure (DNS_custom.hpp)

To create a directory map and help with retrieval and registration into the directory service, the directoryRecord structure was implemented. The structure was made up of a string type labeled server and the in_port_t type labled port to host a service's server name and port number and then added to the map with a string of the service's nickname for the key.

To create the directory map, the following function was used:

```
unordered_map<string, struct directoryRecord> directory;
```

Figure 4: unordered map

With the main directory structure and protobuf schema created, the rest of the files created the server, service and client stub that would allow external clients, such as KVService and KVClient, to use the directory service to find services by nickname instead of only by network addreses.

2.1.3 – Service Directory Server (SDServer.hpp/ SDServer.cpp)

The Service Directory Server code provides a simple shared pointer and ensure the SD server adds itself to the directory service first. In our code, we developed a service directory server that stores each registered service and its corresponding string nickname.

2.1.4 – Service Directory Service (SDService.hpp/ SDService.cpp)

The SDService (Service Directory Service) is designed to manage registration, search, and deletion of services in a distributed environment. It derives from the network's generic Service class using a weak Node pointer in its constructor and is hosted in port 1515. For this design, a queryDirectory function was also added to help route the different type of messages through the system.

In the implementation file's start function, the SDService dynamically binds a socket with server address, port by getting a socket to receive messages and clearing previously used variables. Then enters a loop to continuously listen for incoming messages while the server is alive. It uses recvfrom() to receive UDP messages, and once received a message prints a message indicating success. To validate the message, it checks for the correct magic number and version, if it matches the expected value it proceeds to process the message using the queryDirectory function to tailor its response. Finally, sending the response using the sendto() function and closing the socket with close(). To stop the SDServer, the function simply sets its alive status to false.

If the received message contains registration arguments ('registerserviceargs'), it extracts the service name, server name, and port from the request in the queryDirectory() function, then attempts to register the server by calling the registerServer() function, which adds the server with the specific service to the directory. The registerServer function returns a Boolean if the service is registered

successfully and if not found will return false along with printing out a statement indicating the failure of the registration.

If the received message contains search arguments ('searchserviceargs'), it extracts the service name from the request in the queryDirectory() function and searches for it in the directory using the searchforService() function. If the server is found, sets appropriate response indication success, otherwise returns an empty record.

Finally, if the received message contains deletion arguments ('deleteserviceargs'), it extracts the service name from the request in the queryDirectory() function and then attempts to delete the server from the directory using the deleteServer() function. Similarly, referring to it as a server since it is a server for a specific service, but not necessarily deleting the server form the computer system. The deleteServer() function returns a Boolean indicating whether the deletion was successful or not. However, due to a bug in the provided network code, the delete method is not functional and was not tested for accuracy.

Overall, the SDService acts as the main routing services for handling the different UDP messages sent to the directory and actions accordingly. Making sure RPC semantics of "at least once" are maintained by always returning a value.

2.1.5 – Service Directory Client Stub (SDStub.hpp/ SDStub.cpp)

The SDStub (Service Directory Client Stub) serves to interact with the directory service for the directory service functions of registering, searching, and deleting a service. The KVClient and KVService both use the stub to find how to communicate with the directory server, and the stub acts as the client interface of the directory for any other services in the network directory as well.

Similar to the SDService, the SDStub sets up UDP communication by setting up the network connection and preparing the socket for communication with the directory service and clients in its init() function. The client must send the service's nickname to stub in order for to stub to make any directory calls.

To register a new service, the SDStub constructs a request message containing the server's nickname, name and port, making use of the directoryRecord structure to facilitate an entry into the directory map. It then sends the message to the network and waits for and parses the response message to determine the success of the registration.

To search for or delete a service, the stub sends a search request to the directory service specifying the service by its nickname, and similarly waits for and parses the response message to retrieve the server's details or parses the response message to determine the success of the deletion.

For any request message, the SDStub also thoroughly checks the validity of the message by checking for the version, serial and magic number. Along with providing errors for messages received incorrectly or parsed incorrectly. Instead of a routing call procedure, the SD stub uses the Boolean *ready* to activate the functions and perform the correct actions on the message to be sent to the directory.

2.2 - Name Service Utilization and DNS Integration

We integrated a Domain Name System (DNS) into our network simulator, which acts as a service directory. This DNS system allows services to register themselves with a human readable name

"nickname", and network information. Then, the clients can query this directory using *getaddrinfo* to resolve the service names to addresses, allowing them to communicate.

To integrate this DNS server with the existing key-value server code from assignment 1, we modified the code to include the name and port number for each instance of a service. When starting up, the KV server instance will register with the service directory using its logical name. This then allows instances of KV servers to be identified by clients using the same logical names.

2.2.1 - KVServer Code

For this assignment, the KVServer code was updated to incorporate a simulated DNS mechanism within the RPC system that was established in Assignment 1. The changes made to KVServer are designed to integrate with the with the updated KVServiceServer code to manage the key-value pairs in the distributed setting.

The KVServer code instantiates itself with a KVServiceServer object, showing the functionality of the server in managing key-value storage and management of RPC requests. The KVServiceServer acts as the server-side implementation of the key-value service, however, it is extended to include service registration, lookup and deletion functions within the DNS environment. Also, when initialized, the KVServer constructor passes the nodeName and a pointer to itself to the KVServiceServer. This ensures that each server instance is uniquely identified within the network and can interact correctly with the DNS system.

```
KVServer::KVServer(string nodeName): Node(nodeName){
        kvService = make_shared<KVServiceServer>(nodeName,weak_from_this());
            addService(kvService);
        }
```

Figure 5: KVServer code showing the functionality of instantiating as a KV ServiceServer object.

For this assignment, new methods have been introduced in the KVServer code. These methods are *setNickname*, which is used to set the human readable name for service identification, and *setPort*, which is used to set the port where the server should listen for RPC calls. These methods allow the system to be more flexible, and to support multiple key-value servers. Additionally, KVServer registers itself with the DNS when starting, by using their nickname and port number. This allows clients to search the service directory and connect with services.

```
void KVServer::setDBMFileName(string name){
    kvService -> setDBMFileName(name);
}


void KVServer::setNickname(string nickname){
    kvService -> setNickname(nickname);
}


void KVServer::setPort(in_port_t PORT) {
    kvService ->setPort(PORT);
}
```

Figure 6: KVServer code showing setDBMFileNime, setNickname, setPort methods.

Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

2.2.2 – KVService

Like KVServer, the KVService code has been changed since Assignment 2 to integrate the functionality of the DNS system.

One key element of this code is the introduction of the *registerService* and *deleteService* methods, which allows the directory to update dynamically and stay up to date. The *registerService* method allows the server to automatically register the server with its human-readable nickname, server node name, and port number, which is important for allowing the client to search for and connect to services, by searching by a human readable name.

```
bool KVServiceServer::registerService() {
    cerr << "KVServiceServer registering new service" << endl;

    directoryRecord aRecord;
    aRecord.server = this->nodeName();
    aRecord.port = this->PORT;

    return sdstub->registerNewServer(this->nickname, aRecord);
}
```

Figure 7: KVService code showing the registerService method definition.

The *deleteService* method ensures that services are removed from the directory when they are no longer available. However, due to issues in the provided network code, the *deleteService* method is not utilized and tested in this assignment.

```
bool KVServiceServer::deleteService() {
    cerr << "KVServiceServer deleting service" << endl;

    return sdstub->deleteServer(nickname);
}
```

Figure 8 KVService code showing the deleteService method definition.

Some other key methods in the KVService code include *setDBMFileName*, which is used to set the database filename, *setNickname*, for setting the human readable name, and *setPort*, which is used to set the listening port. These methods are essential for the service's registration in the directory and its operation.

```
void setDBMFileName(string name) {DBMFileName = "data/" + name;}
void setNickname(string newNickname) {nickname = newNickname;}
void setPort(in_port_t newPORT) {PORT = newPORT;}
```

Figure 9:KVService code showing the setDBMFileName, setNickename, and setPort methods.

Additionally, the *start* method has been modified to include the registration of the service with the directory, in addition to the network and database initialization. By integrating service registration in the starting process, the KVServiceServers is immediately accessible by clients.

Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

```
void KVServiceServer::start(){
    //cerr << "in kvserviceServer::start" << endl;
    struct sockaddr_in servaddr, cliaddr;

    if(registerService()) {
        cerr << "service successfully registered in directory" << endl;
    } else {
        cerr << "service failed to register in directory" << endl;
    }

    // open the GDBM file.
    if (DBMFileName.empty()){
       cerr << "Name of DB file not specified" << endl;
       return;
    }
#ifdef __APPLE__
    dataFile = dbm_open(DBMFileName.c_str(), (O_RDWR | O_CREAT), 0644);
    if (!dataFile){
        cerr << "NDBM Error: could not open database file" << endl;
        return;
    }
#else
    dataFile = gdbm_open(DBMFileName.c_str(), 0, GDBM_WRCREAT, 0644, NULL);
    if (!dataFile){
        cerr << "GDBMM Error: " << gdbm_strerror(gdbm_errno) << endl;
        return;
    }
#endif

    // get a socket to recieve messges
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        return; // this will exit the service thread and stop the server
    }

    // clear variables before initializing
    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    // Port and Interfact information for binding
    // the socket
    servaddr.sin_family    = AF_INET;        // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;   // whatever interface is available
    servaddr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if (::bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0 )
    {
        perror("bind failed");
        return; // this will exit the service thread and stop the server
    }

    // register here -TD

    socklen_t len;
    int n;
    //cerr << "alive = " << alive << endl;
    while(alive){
    //cerr << "waiting for call from client" << endl;

        // wait for a mesage from a client
    len = sizeof(cliaddr);  //len is value/result
    n = recvfrom(sockfd, (uint8_t *)udpMessage, MAXMSG,
            MSG_WAITALL, ( struct sockaddr *) &cliaddr,
            &len);
        //cerr << "server received " << n << " bytes." << endl;
    //std::cerr << HexDump{udpMessage,(uint32_t)n} << endl;
```

Figure 10: KVService code showing the start method part 1

```
E477KV::kvRequest receivedMsg;
E477KV::kvResponse replyMsg;

if (!receivedMsg.ParseFromArray(udpMessage,n)){
    cerr << "Couild not parse message" << endl;
    // ignore
}

//cerr << "message parsed" << endl;

if ((receivedMsg.magic()) != magic){
        cerr << "service unrecognized message" << endl;
    } else {
    // start by copying version and serial to reply
    replyMsg.set_magic(magic);
    replyMsg.set_version(receivedMsg.version());
    replyMsg.set_serial(receivedMsg.serial());

    if ((receivedMsg.version() & 0xFF00)  == version1x){
       // dispatch version 1.x
       callMethodVersion1(receivedMsg, replyMsg);
    } else {
    cerr << "unrecognized version" << endl;
    // For now ignore, message doesn't have a wrong version reply
    }
    // at this point in time the reply is complete
    // send response back
    uint32_t msglen = replyMsg.ByteSizeLong();
    // double check size
    replyMsg.SerializeToArray(udpMessage, msglen);
    //cerr << "reply message" << HexDump{udpMessage,msglen} ;

    int servern = sendto(sockfd, udpMessage, msglen,
        MSG_CONFIRM, (const struct sockaddr *) &cliaddr, len);
    //cerr << "server sent " << servern << " bytes" << endl;
}

}

close(sockfd);
```

Figure 11: KVService code showing the start method part 2

Finally, the KVService code has been modified to improve resource cleanup within the destructor and *stop* method. The *stop* method is used to ensure that network sockets are closed, and database connections are terminated, to prevent issues during service shutdown.

```
void KVServiceServer::stop(){
    alive = false;
}
```

Figure 12: KVService code showing the stop method.

2.2.3 KVClient1

Similarly to the server and service code, the KVClient1 code also makes changes to the code from assignment 1 in order to implement DNS service directory functionality. The overall goal of this code is to allow the client to lookup by using service names, rather than static server addresses.

This was implemented through the addition of the *setServerName* method, which allows the client to specify the name of the service it wants to interact with. This method is critical for integrating with the new DNS service discovery, and enables the client dynamically find service addresses based on service names.

```
// void KVClient1::setServerAddress(string addr) {kvService.setServerAddress(addr);}
void KVClient1::setServerName(string addr) {kvService.setServerName(addr);}
```

Figure 13: KVClient1 code showing the setServerName method.

Additionally, the *start* method has been improved to allow a client to interact with services dynamically. It shows the client making a *kvPut* call to store a value and then a *kvGet* call to retrieve it, which can be used to tests the client's ability to communicate with the key-value service using the DNS service.

```
void KVClient1::start(){

    // store a value;
    string value1="This is \0 a test!!"s;
    bool putRes = kvService.kvPut((int32_t)25,(const uint8_t*)value1.data(), (uint16_t)value1.size());
    cerr << "status is " << putRes << endl;

    // get the same value back.
    kvGetResult gres;
    gres = kvService.kvGet((int32_t)25);
    cerr << "status is " << gres.status << endl;
    if (gres.vlen > 0){
    cerr << HexDump{gres.value, gres.vlen};
    }
    kvService.shutdown();

}
```

Figure 14: : KVClient1 code showing the start method.

2.2.4 KVClientStub

Finally, significant changes were made in the KVClientStub code in order to allow dynamic service lookups using the DNS system. These changes are essential for allowing clients to locate and communicate with services based on their human readable names.

The code now includes the *setServerName* method for setting service names.

```
void setServerName(string name){
    serverName = name;
}
```

Figure 15::KVClient1 code for setServerName.

It also implements the *init* function, which allows the KVServiceStub to dynamically resolve server addresses using the service names. This allows the client to communicate with the server, without initially knowing it's address but knowing its nickname. The *init* function also displays the network connection setup, based on the dynamically resolved address information.

```
bool KVServiceStub::init(){
    // cerr << "client looking for server: " << this->serverName << endl;
    if (this->serverName.empty()){
        cerr << "No server name given -- Does not know server name!! " << endl;
        return false;
    }

    searchRecord = sdstub->searchForService(this->serverName);
    // cerr << "client stub init() port and server from search " << searchRecord.server << " " << searchRecord.port << endl;
    if(searchRecord.port == 0) {
        cerr << "server name not found in directory" << endl;
        return false;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(searchRecord.port);

    // look up address by name (Version 2 of network.cpp)
    struct addrinfo * res;
    int numAddr = getaddrinfo(searchRecord.server.c_str(), nullptr, nullptr, &res);
    servaddr.sin_addr = ((struct sockaddr_in*)res -> ai_addr) -> sin_addr;
    freeaddrinfo(res);

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
    perror("socket creation failed");
        return false;
    }

    struct timeval tv;
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv,sizeof(tv)) < 0) {
        perror("Error");
        return false;
    }
    ready = true;
    return true;
}
```

Figure 16:KVClient1 code for init.

The other important methods in KVClientStub are kvPut and kvGet, which are responsible for handling the serialization, transmission, and acknowledgment of RPC requests and responses.

2.3 - Increased Fault Tolerance

Finally, we changed the RPC semantics to ensure that operations are always completed ‘at least once’. This was completed by ensuring the client stub retries the RPC until it receives an acknowledgement from the server, or until the maximum number of tries is reached. This was implemented mainly using while loops in the sdservice.cpp and sdstub.cpp code files, such that communication of any failure to send messages is persistent. Unfortunately, the ability to retry the message was not yet implemented, though for future implementations would be considered.

```
E477DirectoryService::DSResponse responseMsg;
bool recievedResponse = true;
do {
    len = sizeof(struct sockaddr_in);
    // cerr << "recv " << len << "bytes from address " << inet_ntoa(serveraddrreply.sin_addr) << endl;
    n = recvfrom(sockfd, (char *) buffer, MAXMSG,
                 MSG_WAITALL, (struct sockaddr *) &serveraddrreply, &len);
    // cerr << "recv " << len << "bytes from address " << inet_ntoa(serveraddrreply.sin_addr) << endl;

    if(n == -1) {return false;}

    if(!responseMsg.ParseFromArray(buffer, n)) {
        cerr << "could not parse mnessage" << endl;
    } else {
        if(requestMsg.version() != responseMsg.version()) {
            cerr << "version mismatch" << endl;
            recievedResponse = false;
        } else {
            if(requestMsg.serial() != responseMsg.serial()) {
                cerr << "serial numbers mismatch" << endl;
                recievedResponse = false;
            } else {
                if(responseMsg.has_registerserviceres()) {
                    E477DirectoryService::registerServiceResponse registerResponse = responseMsg.registerserviceres();
                    // cerr << "registerResponse.success();" << registerResponse.success() << endl;
                    return registerResponse.success();
                }
            }
        }
    }
} while(!recievedResponse);
```

Figure 17:While loop implementation

Overall, by implementing a flat namespace and improving the RPC robustness, we were able to improve the overall functionality of the RPC simulation. This is because clients are now able to dynamically look up services, leading to increased flexibility. Additionally, the implementation of ‘at least once’ in the RPC network ensures that operations are successfully completed, allowing the system to be more fault tolerant.

2.0 – Testing and Main

This code was tested using 2 test cases, implemented in the main code, and is detailed in the accompanying testing document.