1 Protocol

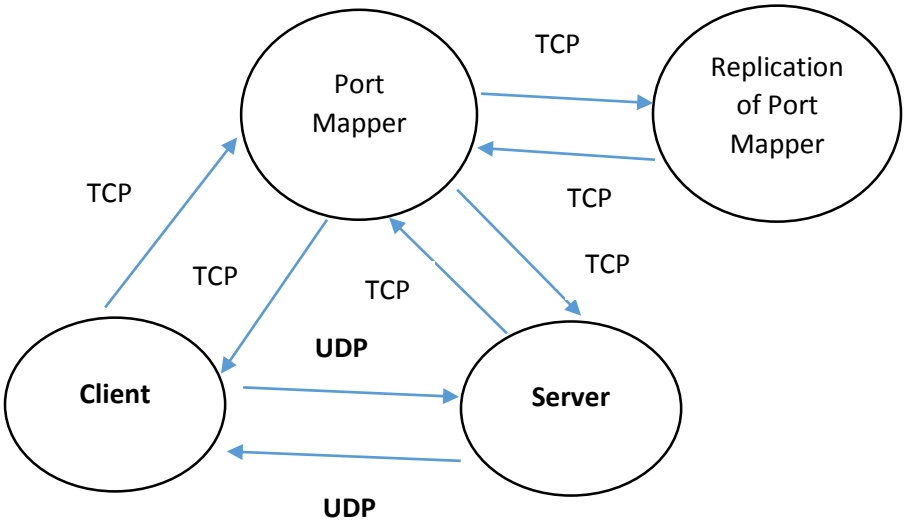


Fig 1.Protocol of the System

In general, we use UDP as the protocol between Client and Server, TCP between Port Mapper and Client/Server, TCP between Port Mapper and Replication of Port Mapper.

The reason we choose UDP as the protocol between Client and Server is because this part is the bottleneck of the system. The communication between Client and Server is much more frequent than the other part. Meanwhile, we will send large size of data (e.g., parameters) between Client and Server. For the other parts, we use TCP for the two reasons. First, communications in these parts are not frequent and the data between them is relatively small. Second, TCP is more reliable and can simplify the system design.

2 The Implementation of Port Mapper

2.1 The Main class

Main Class	Description
Address	Address entity, represents an address
Client2Mapper	Client2Mapper entity, represents a request from client to mapper
PortMapper	Responsible for handling server registration, server refresh, and redirect client requests
PMThread	A PortMapper task, responsible for handling of a new request (search, register, refresh, check, etc.)
HashCodeUtil	Collected methods which allow easy implementation of hashCode.
Mapper2Mapper	Mapper2Mapper entity, represent a dummy message between main mapper and standby mapper

NetTool	For convenience, all the helper methods
Server2Mapper	Server2Mapper entity, represent a registration message from server to mapper
Server2MapperRefresh	Server2MapperRefresh entity, represent a refresh from server to mapper
CopyOfPortMapper	Responsible for checking upon main mapper and taking over its job when it crushes, standby mapper has the identical functionality as the main mapper once it takes over

Table 1. Main class of Port Mapper

2.2 The Main Process Flow

The process flow of PortMapper is as followings. Note because the standby PortMapper has similar process, so we use a dummy process for reuse purpose.

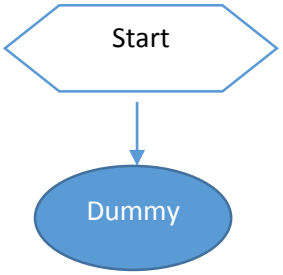


Fig 2. Process of PortMapper

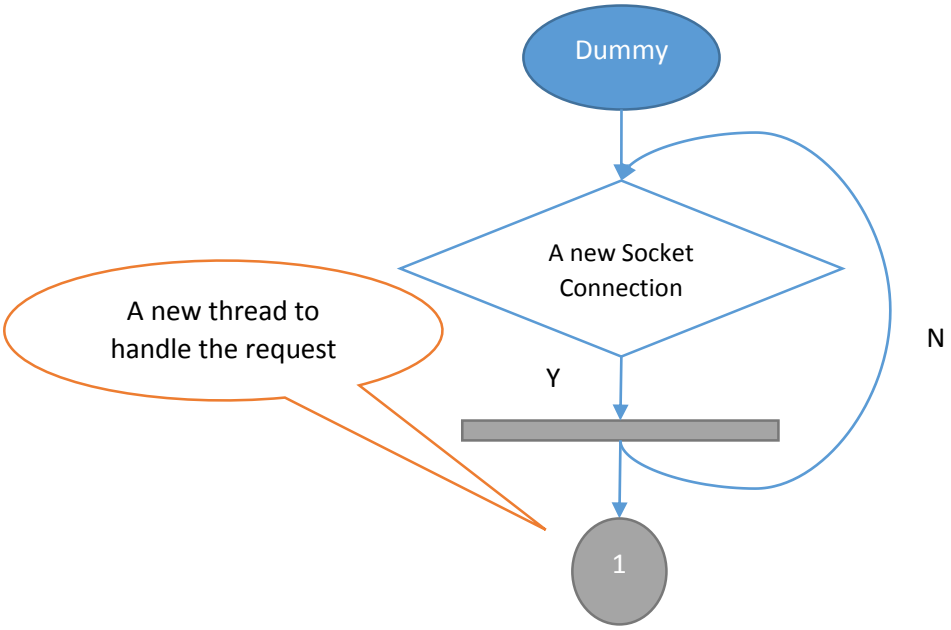


Fig 3. Dummy Process of PortMapper

PortMapper supports multithreading, so when a new socket connection is built, a new thread will be constructed to handle the socket request.

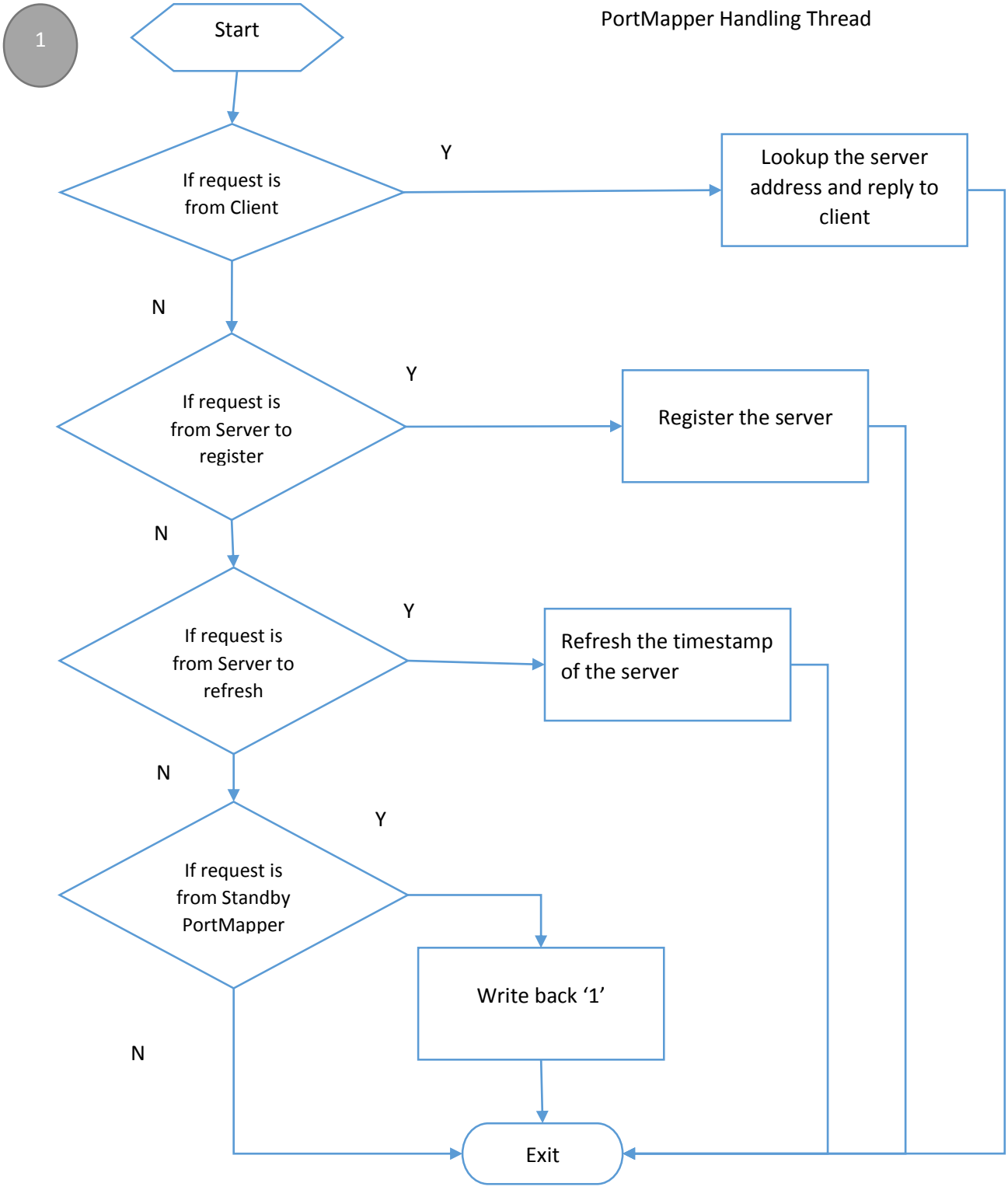


Fig. 4 Process of PortMapper handling Thread

CopyOfPortMapper is the replication of PortMapper. The process flow is as followings:

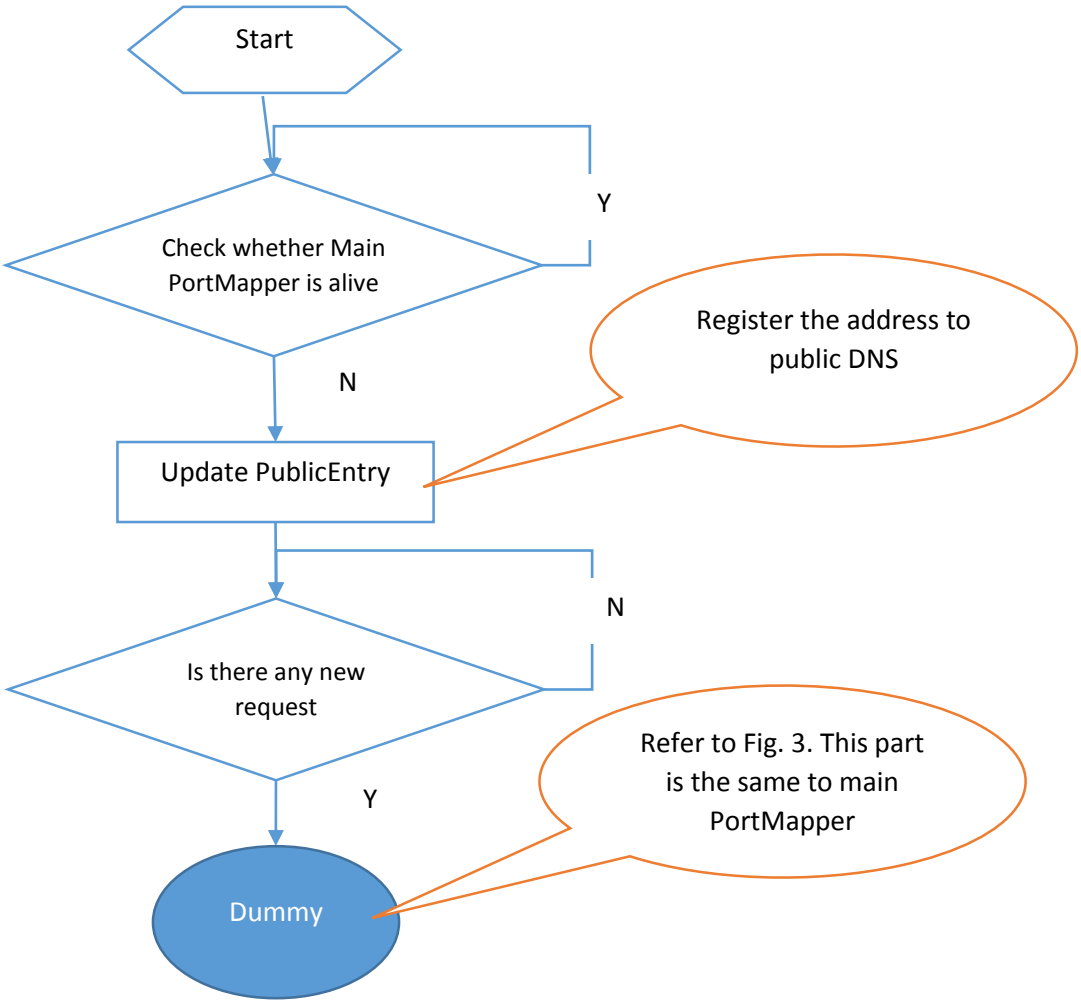


Fig. 5 Process of Standby PortMapper

2.3 Load Balancing

To achieve load balancing, we maintain a list of available server address. When a client request a server address, we always copy the head of the list to the end of the list and return the head of the list. Through this simple policy (Round-Robin), we can achieve load balancing, since we evenly distribute the workload across all available servers for a certain kind of request. The illustration of the method is as followings:

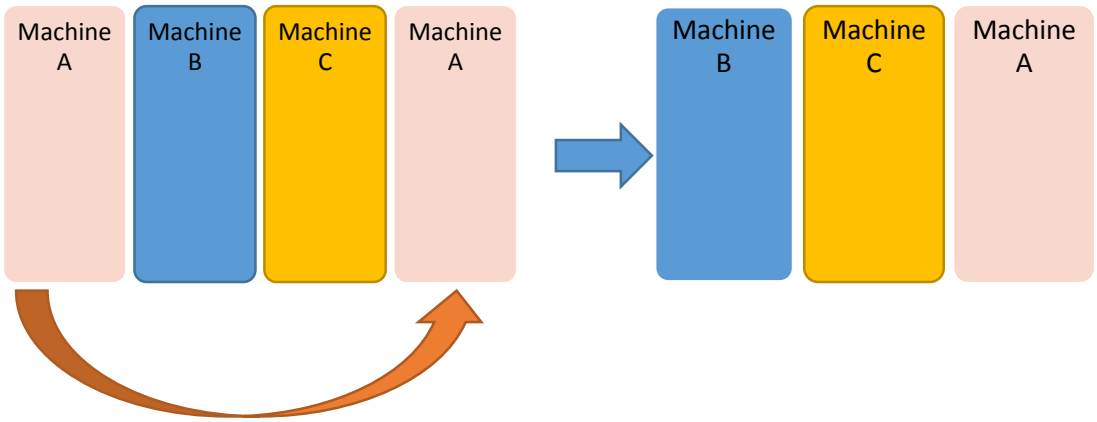


Fig. 6 Demonstration of Load Balancing

3 The Implementation of Server

3.1 The main class of Server

Main Class	Description
Address	Address entity, represents an address
Client2Server	Client2Server entity, represents a request from client to server
HashCodeUtil	Collected methods which allow easy implementation of hashCode
NetTool	For convenience, all the helper methods
P0	P0 entity, represents a procedure 0 message from client to server
PacketGenerator	Takes in any entity, convert to a collection of fixed maximum size UnifiedPacket(s)
Server	The server class that contains all the actual implementations of procedures, it's the only part the developer of the server side should worry about, the request handling part is decoupled and put in separated class. It also implements all the interfaces of all the libraries supported, and in Java, if you implements a interface, you must implements all procedures defined in the interface, so this mechanism can be used as a reminder for the developer of the server side to implement all the procedures supported
Server2ClientNeed	Server2ClientNeed entity, represents a request from server to client for all lost packets
Server2ClientResult	Server2ClientResult entity, represents a response from server to client, containing the result desired by the client

Server2Mapper	Server2Mapper entity, represent a registration message from server to mapper
Server2MapperRefresh	Server2MapperRefresh entity, represent a refresh from server to mapper
ServerRequestHandlerV2	The handler of request
ServerTaskV2	The execution of a task by server
TransactionId	TransactionId entity, represent a transaction Id, which can uniquely identify a transaction between client to server
UnifiedPacket	A entity that can be serialized into a maximum fixed size packet, can be used to store a chunk of data of a certain entity

Table 2. Main class of Server

The Cache

The cache is a hashmap which store the transctionID and corresponding result.

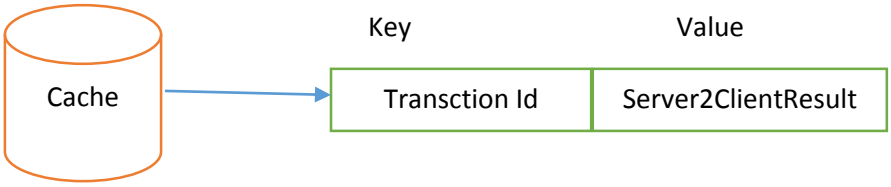


Fig 7. The structure of cache

Server cache, has a maximum capacity, eliminate the least recently used record (LRU-based) when cache is full and a new record is going to be added

The Packets Buffer

Because we use UDP to transmit data between client and server, and we split the message to small packets with **fixed size**, the packets can arrive at **server out of order.** We need to store the packets in a buffer. Here we use a hashmap to store the packets. The key is the UUID which is constructed from TransactionId. The value is a Packet Map. This is the implementation in Java:

```
static Map<UUID, Map<Integer, UnifiedPacket>> packetReceived = new ConcurrentHashMap<>();
```

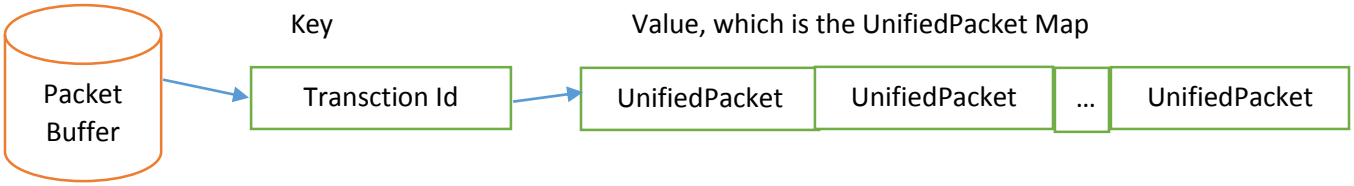


Fig 8. The structure of the Packet Buffer

Unified Label, Unified Packet & Packet Generator

Unified Packet is a customized data structure (basically a byte array with customized layout) designed by us to solve a series of problems when sending data with UDP as the protocol.

For receiver, we want to use a simple data structure as the buffer, like a fixed size byte array. But if that's the case, how big should the buffer be? For sender, how big should one packet be? We're supposed to send data over the network no matter how big the data is.

We can divide the data into small chunks, it should achieve same size for each packet, but what if we want to send data in parallel in multiple threads to server? At the client side we can use separate port for each request, but eventually packets that carry the data for the same type of procedure may get sent to the same server (let's assume there's only one server for each type of procedure now, it's easy to explain the problem). How is the server supposed to do with all these anonymous chunks?

In their original form which is the actual request, we have transaction Id to differentiate 2 Max procedure requests from 2 transactions, so how about we give each chunk an "id"?

Unified Packet // a byte array

Unified Label (UUID id, int current, int total) // region 1

byte[] partial // fixed size 10240, region 2

We came up with this data structure. Each chunk (fixed size 10240) is place in the "partial" byte array. "id" here is a fixed size type 3 universally unique identifier (name-based), which is mapped directly from transaction Id string, and "id" is also a inherently fixed size string. So with "id", we can classify all packets from different transactions in parallel without worrying about mixing them up. "current" indicates the "position" of this packet in the corresponding collection generated from a certain transaction. "total" here indicates the total number of packets generated from a certain transaction.

Packet Generator is used to generate these collections of packets (byte arrays), we basically serialize the data, cut them into same size chunks, put each in region 2 of a Unified Packet structure, record its belonging information at the same time and put it in region 1, and join 2 parts together. And, for serialization, UUID has fixed size, int has fixed size, and we set byte array to a fixed size, thus each UnifedPacket can be a fixed size chunk.

Each UnifiedPacket can be sent in a UDP packet, and we can use a fixed size buffer to receive them at server side.

We have a mapping structure <UUID, <Integer, UnifiedPacket>>, each packet, after it arrives, it'll be placed at the i^{th} position in the certain UUID's corresponding mapping. The server can easily assemble chunks from these packets into their original form: the request message.

Java Reflection

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.

To better explain why we need Java Reflection in our design, we will explain shortly how it works.

For example, say, you have an object of an unknown type in Java, and you would like to call a certain method on it if one exists. Java's static typing system isn't really designed to support this unless the object conforms to a known interface, but using reflection, your code can look at the object and find out if it has a method and then call it if you want to.

So, a code example of this in Java:

```
// dynamically load a class

Class aClass = Class.forName("FULL_PATH_TO_THE_CLASS");

// dynamically load a method in that class, TYPE_ARRAY is an array of all the
parameters' class, so it's like a description of the method's signature

Method aMethod = aClass.getMethod("METHOD_NAME", TYPE_ARRAY);

// dynamically invoke this method with values in VALUE_ARRAY on an anonymous
object of that class

Object returnValue = method.invoke(aClass.newInstance(), VALUE_ARRAY);
```

It's just like we provide the name of the class and what does the target method look like in the textual form, then it'll get executed and we can get the return value.

So we have programs, each program has versions, each version has procedures. Procedures from different versions can have the same signature but totally different implementations. And that's why we decide to design each version as a separate interface. Since they are in different interfaces, it's ok for them to have any signature, even the same. But this is not the most important part.

In a request, we only have the textual representation of the desired program, version, procedure, parameters. How can we execute the actual procedure if we only know the signature in a string? It's silly to implement this in a switch, so that's why we decide to introduce Java Reflection, with this technology, as long as a certain combination of program id, version, procedure, parameter exists in our library, we can dynamically load it, execute it, and get the result. That also means, for a server update, if the developer of the server wants to add new functionality to the library, he only needs to add implementations of new procedures in corresponding versions, and that's it, the mapping is handled dynamically.

Java Reflection of Object, primitive type and arrays are all different in details, but the idea is the same as above. See code comments for reference.

3.2 The Main Process Flow

This is the process flow of server

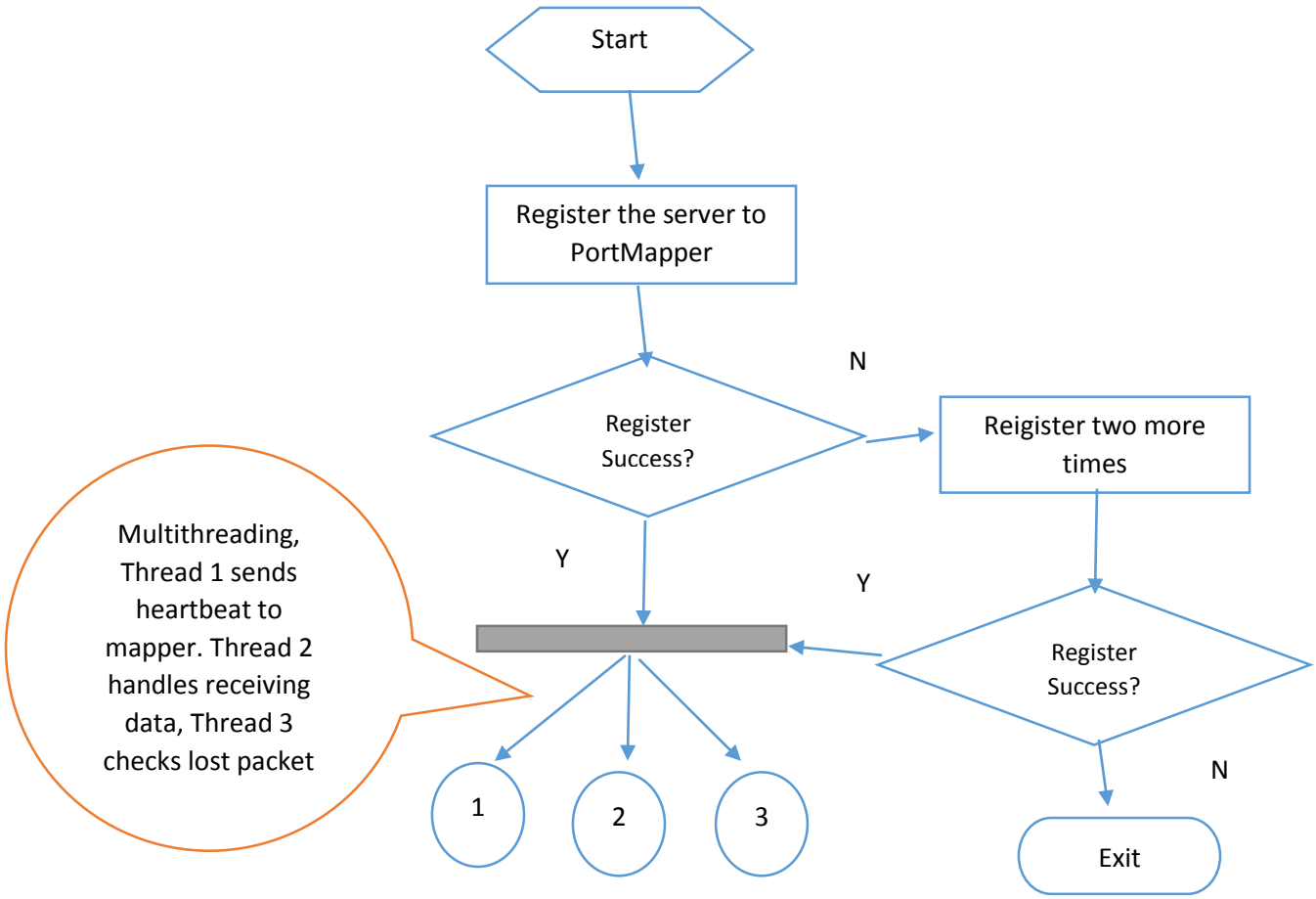


Fig. 9 Process of Server

This is the process of the refresh thread. It will send heartbeats to the PortMapper every 15 seconds to keep alive.

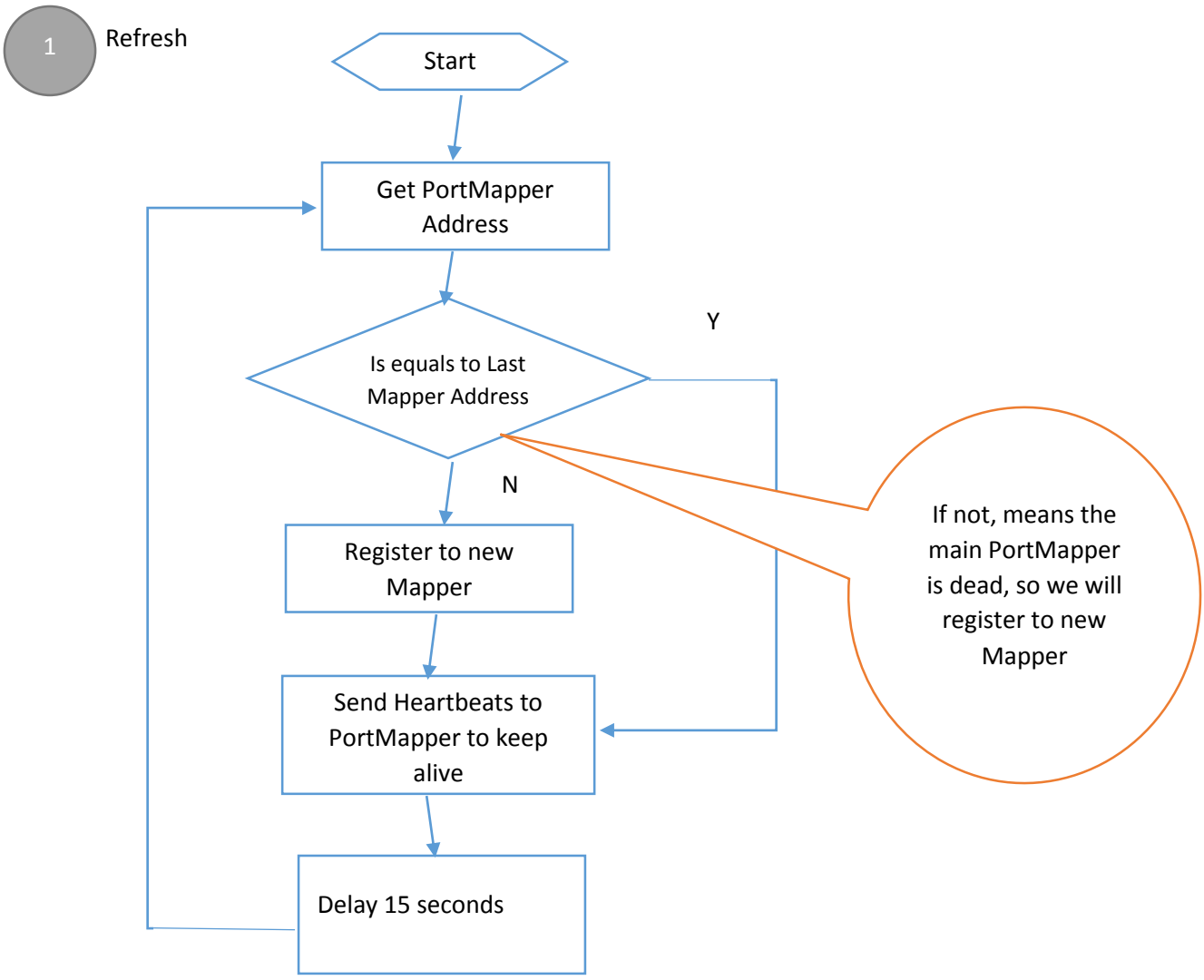


Fig. 10 Process of Refresh

2 Main thread, monitor the data buffer

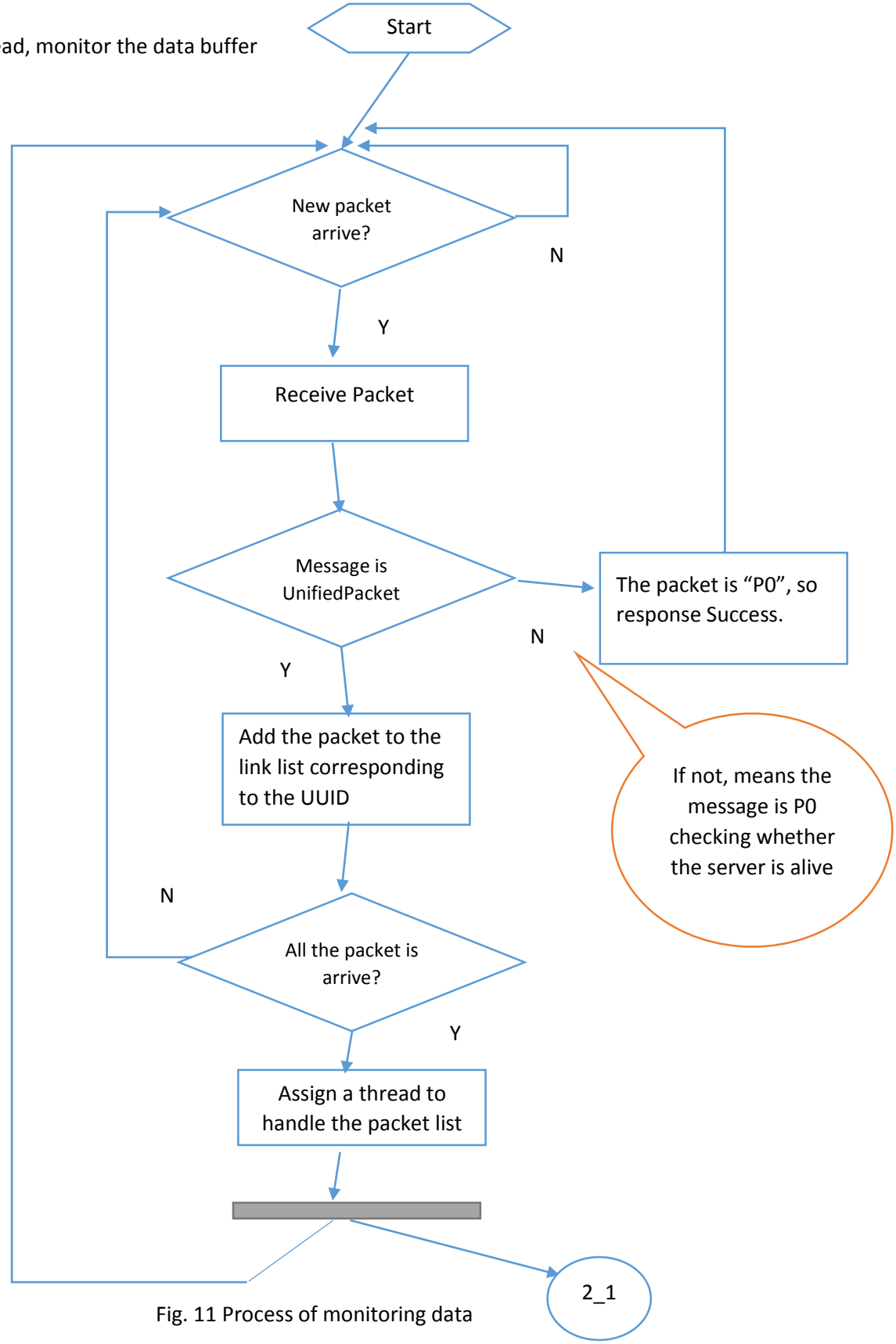


Fig. 11 Process of monitoring data

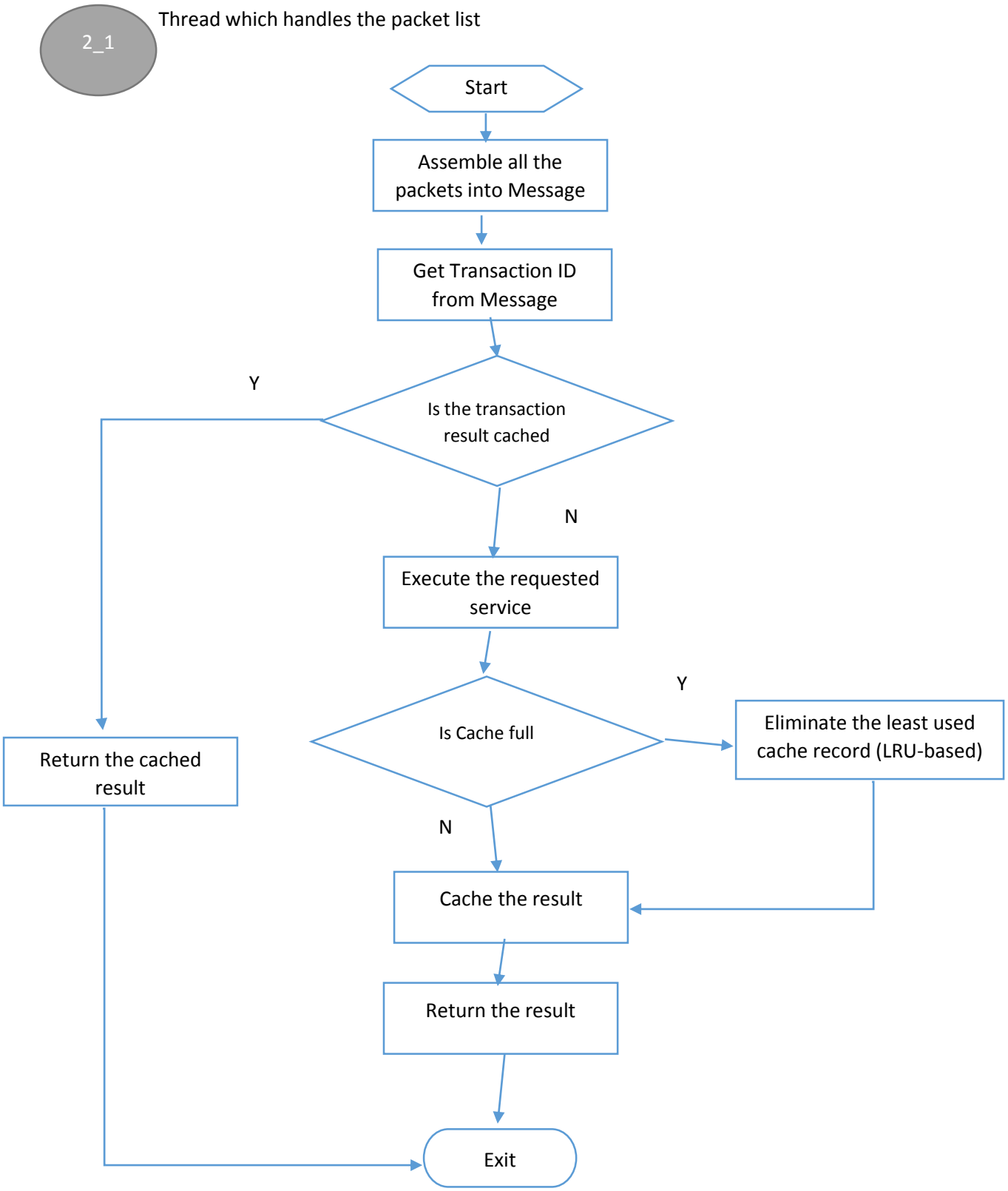


Fig. 12 Process handles the packet list

3 Checking non-received packet Thread

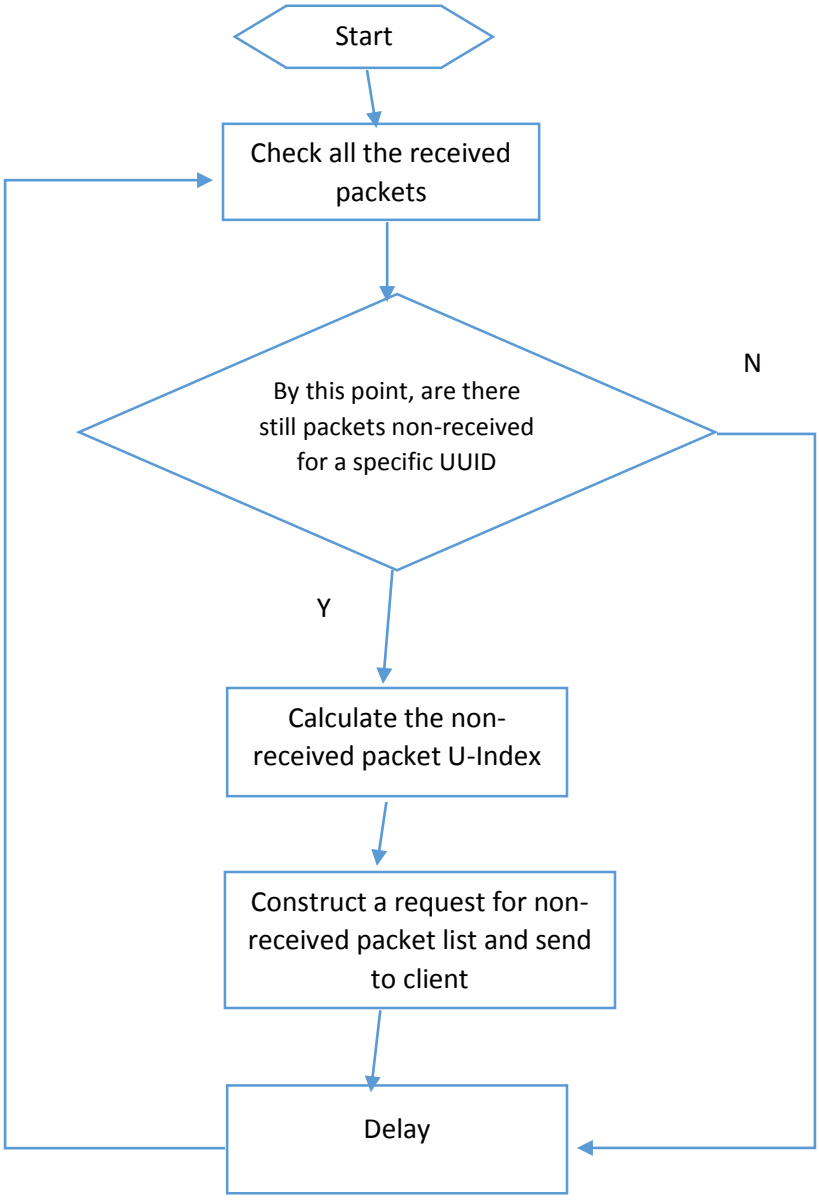


Fig. 13 Process Checking non-received packet Thread

4 The design of Client

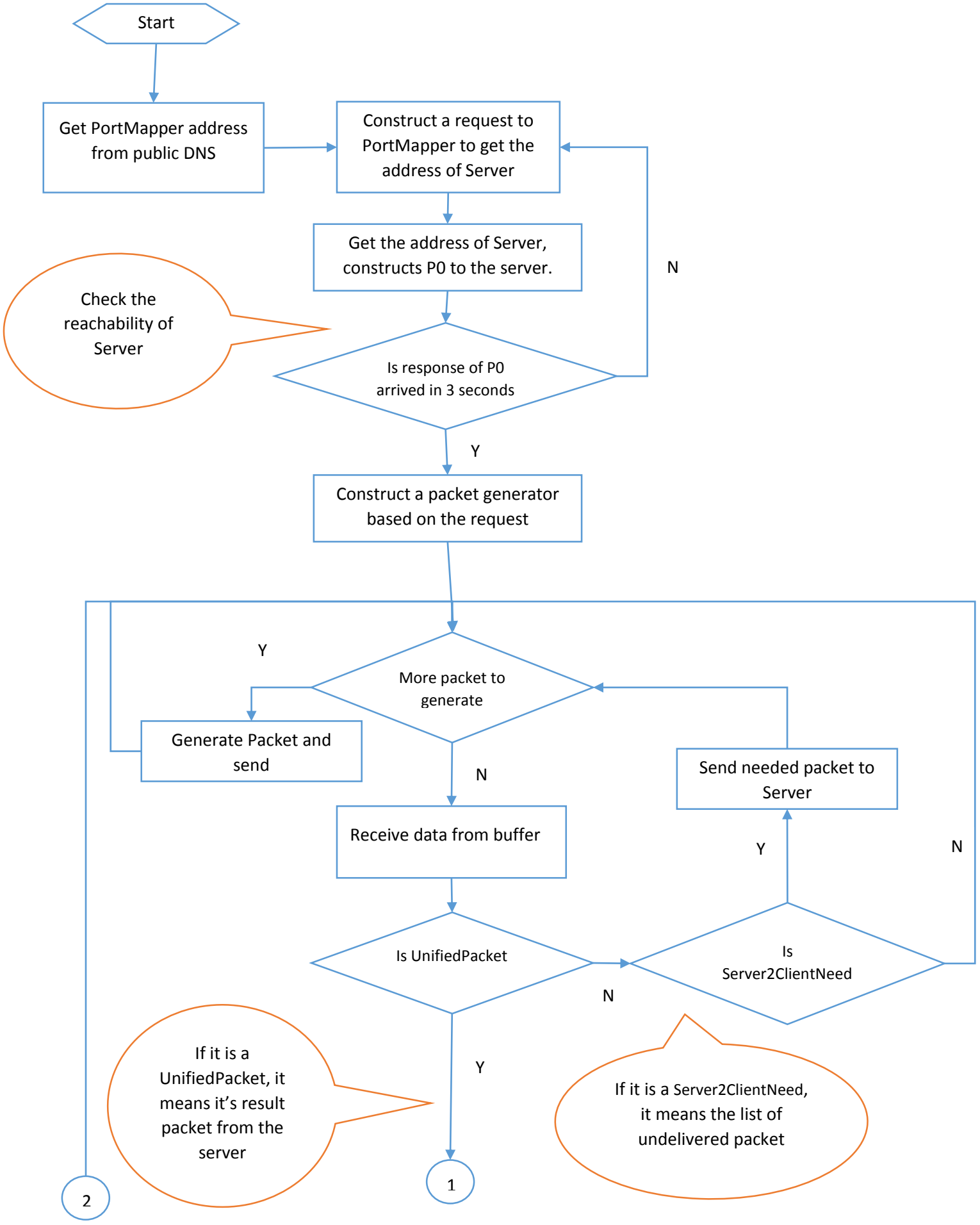
We support multithreading in clients. Each RPC request is handled by a separate thread. Our library can be used in a single thread. For multithreading in client, the developer of the client side can our libraries in a parallel manner.

Here we list the major class in the implementation of Client

Main Class	Description
Address	Address entity, represents an address
P0	*P0 entity, represents a procedure 0 message from client to server*
UnifiedPacket	A entity that can be serialized into a maximum fixed size packet, can be used to store a chunk of data of a certain entity
Client2Mapper	Client2Mapper entity, represents a request from client to mapper
Client2Server	Client2Server entity, represents a request from client to server
ClientRequestHandlerV3	Request handler of the client, responsible for handling the sending and receiving
HashCodeUtil	Collected methods which allow easy implementation of hashCode.
NetTool	For convenience, all the helper methods
PacketGenerator	Takes in any entity, convert to a collection of fixed maximum size UnifiedPacket(s)
Server2ClientNeed	Server2ClientNeed entity, represents a request from server to client for all lost packets
Server2ClientResult	Server2ClientResult entity, represents a response from server to client, containing the result desired by the client
TransactionId	TransactionId entity, represent a transaction Id, which can uniquely identify a transaction between client to server

Table 3. Main class of the client

In the instructions, we get the idea that we should implement procedure 0 in each version. In our implementation, we consider procedure 0 as a way to verify reachability of the server that contains all those versions. So we construct P0 message sending from the client to server before sending the real request, which will simulate the same functionality as the procedure 0 mentioned in the instructions. We assume the other reason that a procedure 0 should be implemented as an actual procedure in each version is that the client may want to verify the reachability of a certain version, but in our implementation, this is not necessary, since a version is fetched, we can find out if a desired procedure is in this version or not, the details are already explained above (Java Reflection).


Start
Get PortMapper address from public DNS
Construct a request to PortMapper to get the address of Server
Get the address of Server, constructs P0 to the server.
Check the reachability of Server
Is response of P0 arrived in 3 seconds
N
Y
Construct a packet generator based on the request
Y
More packet to generate
Generate Packet and send
N
Receive data from buffer
Send needed packet to Server
Y
N
Is UnifiedPacket
Is Server2ClientNeed
N
If it is a UnifiedPacket, it means it's result packet from the server
Y
If it is a Server2ClientNeed, it means the list of undelivered packet
1
2

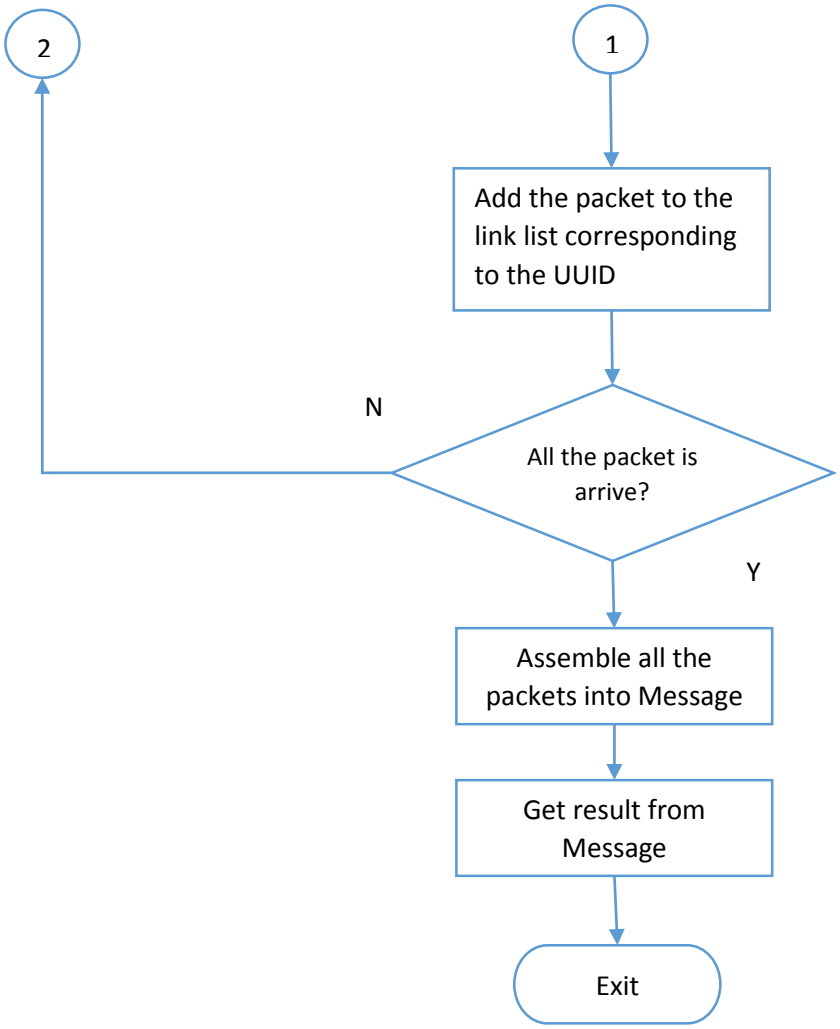


Fig. 14 Process of the Client

5 Exception Handling

5.1 The failure of a Server

We set a timeout for Client request. If the request is timeout because the failure of the server, the client will ask the PortMapper to return the next available server. So the client can send the request to a new server. In another word, the failure of a Server will not cause the failure of the client.

5.2 The failure of main PortMapper

The standby port mapper will check the liveliness of main port mapper constantly by a TCP connection, they will keep sending dummy messages to each other, if main mapper crushes, the standby mapper will immediately catch the exception. If main mapper is dead, the standby

mapper will write its address to the public DNS file right away. For servers, each server will get the current mapper address to perform the refresh, and each server will also cache the mapper address, if a server notice that the mapper address changes, then the main mapper must crushed, so it will register all procedures it supports to the new mapper instead of solely refreshing. Clients then can request the new mapper to get the address of the server. So the failure of main mapper will not cause failure of the system.

5.3 The lost of UDP packet

Due to the characteristics of UDP protocol, a packet can be lost during transmission. The server will check its local data structure periodically and ask the client resent the **lost packet**. This is an advantage of our system. If you ask the client to send all the packets again, the network traffic can be increased dramatically. Our system will check the index of the lost packets and request the client send the corresponding packets, which is very efficient.

6 BenchMark

This is the setup of RPC of our Benchmark, for LPC, we simply initiate terminals only on selenium

Role	Hostname	Processors	Memory	Architecture
Client	aluminum	Dual Quad-Core 2.33GHz Xeons	16GB RAM	64-bit Linux
Client	selenium	Dual Quad-Core 2.33GHz Xeons	16GB RAM	64-bit Linux
Standby Mapper	nickel	Dual Hyper-Threaded Six-Core 3.33GHz Xeons	96GB RAM	64-bit Linux
Mapper	neodymium	Dual Hyper-Threaded Six-Core 3.33GHz Xeons	96GB RAM	64-bit Linux
Server	neptunium	Dual Hyper-Threaded Six-Core 3.33GHz Xeons	96GB RAM	64-bit Linux
Server	germanium	Dual Hyper-Threaded Eight-Core 2.3GHz Xeons	128GB RAM	64-bit Linux

Table 4. Set up of the Benchmark

We run some basic experiments to test the system, all the RPC call can finish successfully. As required by the instruction, we have four procedures:

public double Min(double[] x){}

public double Max(double[] x) {}

public double[][] Multiply(double[][] A, double[][] B) {}

public double[] Sort(double[] v) {}

The following table is the result

Test case #	Details	Time consumed
1	Run each kind of procedure sequentially only once	0.156 secs
2	Run each kind of procedure 1000 times sequentially	10.86 secs
3	Run 3000 mixed procedures	10.34 secs
4	Run each kind of procedure concurrently for 1000 times	2.653 secs
5	Run 3000 mixed procedures concurrently	2.382 secs

Table 5. Result

The RPC vs LPC Matrix Multiply

We design an experiment to show the performance of RPC and LPC. The result is as follows:

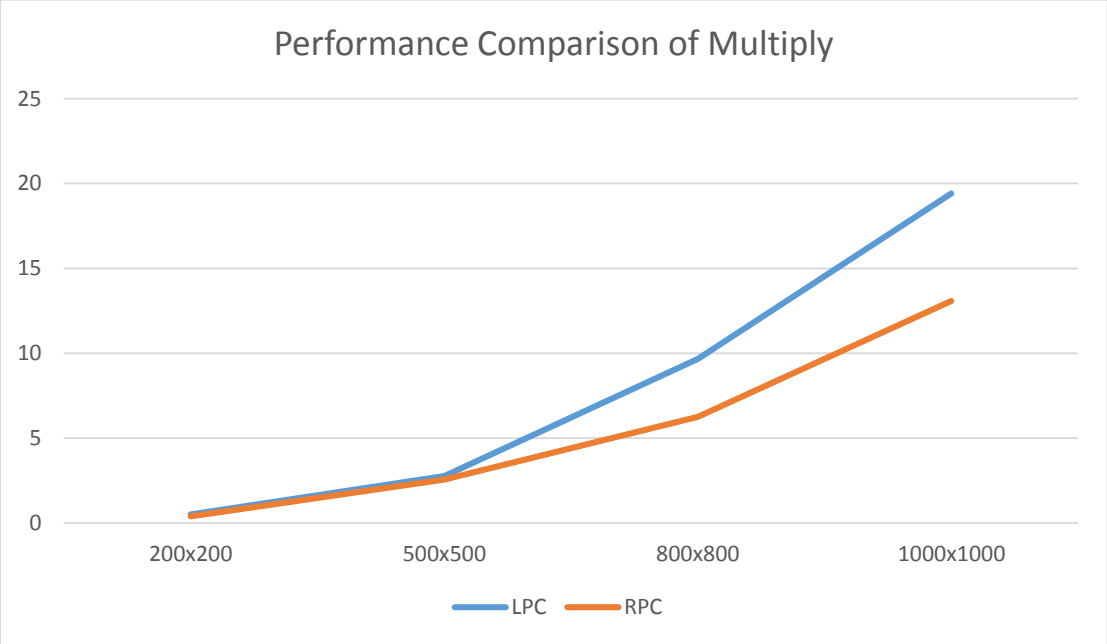


Fig. 15 Performance comparison

The x-axis is the size of matrix, the y-axis is the time in seconds to finish the call. As you can see, when the size of matrix is small, LPC has roughly the same, probably a little bit higher performance as RPC. This is because comparing to LPC, RPC will do the marshaling, split the message into small packets, and send the packets to the server. These operations consume time. However, when the size of matrix become bigger, the bottleneck of the system is the computation capability. The server has a high computation capability. The other overhead of the RPC could be ignored comparting to the time of computation. So RPC will be faster when the size of matrix is bigger.

In conclusion, when the bottleneck of the system is computation capability, using RPC is much more efficient.

For detail, please refer to code.