Distributed Computing using multicast

Table of Contents

1 Problem Definition 3

2 Software Architecture 3

2.1 Client Server Model 3

2.1.1 Server Role 4

2.1.2 Client Role 4

2.1.3 Moderator/ Client Leader 4

2.2 Task Distribution 4

2.3 XDR Communication 6

2.4 Multicast Communication 6

3 Software Requirements 7

3.1 Assumptions 7

4 Memory and Performance Impact 7

5 CLI and Epoll 7

5.1 Server 7

5.2 Client 8

6 Data Structures 8

6.1 Server 8

6.2 Client 9

7 Testing Considerations 10

# Problem Definition

For Phase I, implement simple server and client program using IPC/Sockets which displays messages sent from each client periodically. Clients join a group on server and display of messages would be per group.

Extend phase 1 to execute task scheduled on the server on the client nodes. A parallel program (eg:- max of given large set of numbers), will be executed on server and server will distribute the task into smaller subsets and send it across to clients for execution and then server will collate the results and display the output.

# Software Architecture

## Client Server Model

Client-server is a [software architecture](https://simple.wikipedia.org/wiki/Computer_architecture) model consisting of two parts, [client](https://simple.wikipedia.org/wiki/Client) systems and [server](https://simple.wikipedia.org/wiki/Server_(computing)) systems, both communicating over a [computer network](https://simple.wikipedia.org/wiki/Computer_network) or on the same [computer](https://simple.wikipedia.org/wiki/Computer). A client-server application is a [distributed system](https://simple.wikipedia.org/wiki/Distributed_computing) made up of both client and server software. The client process always initiates a connection to the server, while the server [process](https://simple.wikipedia.org/wiki/Process_(computing)) always waits for requests from any client.

The client server model here is a hybrid one where combination of multicasting and unicasting is done for communication.

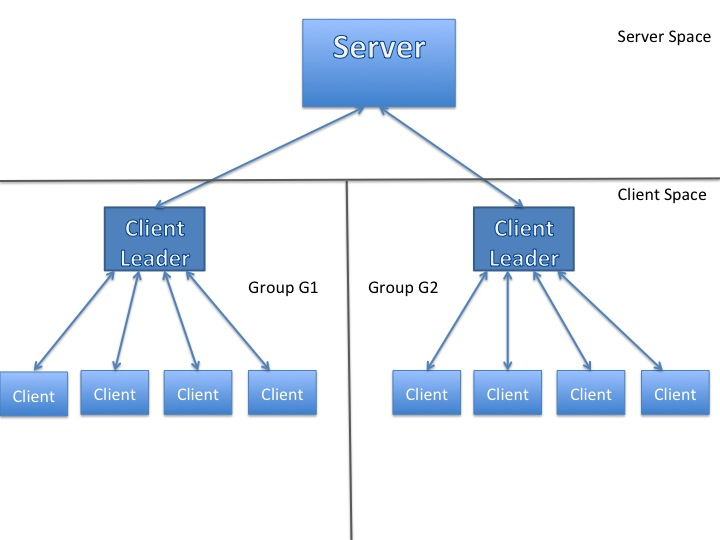


Figure Client Server Model using Multicast

### Server Role

Server always waits for a request from one of the clients. It also provide’s the group IP address on client’s request and is engaged actively in seletecting the moderator/client leader and assigning task’s to the client’s in a group.

Also, epool is used at server side to facilitate CLI execution.

### Client Role

Client always initiates connect request’s to the server using UDP. Also, client can join and leave a new group and is also involved in performing task’s assigned by the server.

### Moderator/ Client Leader

This is a special role performed by one of the client’s of the multicast group. The moderator is involved in following activities:

* Management of the client’s in its group by doing keep alive’s
* Providing the seggregated results of the assigned task from the different clients.

The benefits of this model are as below:

* Better scalability in real time situations
* Clear separation between Server and Client function. (Server is pure management function whereas clients specialize in task execution. )
* Even under fully loaded conditions, server will not turn into the bottleneck for task execution since small part of management function is pushed one layer below.

## Task Distribution

Following is the flow diagram for a client-server communication for task distribution:

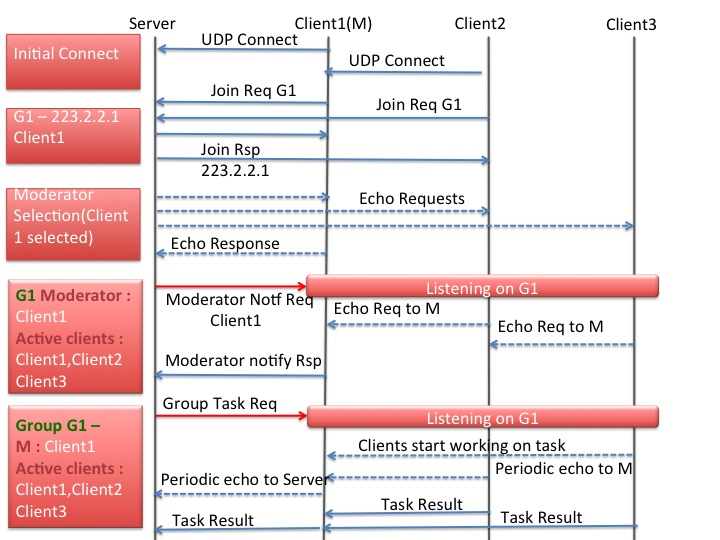


Figure Flow diagram for the client server communication

The sequence of events is as follows:

1. A client connect’s to the server via UDP requesting the multicast Group IP address which it wants to be a part of. Server responds to the join request with the multicast group IP address. The client then join’s the multicast group.
2. The server sends echo requests to maximum of 3 clients and the client which responds first is chosen as the moderator of the group.
3. The server then sends a moderator notification request to the concerned group announcing the moderator, via a multicast message. In response to this, the rest of the client’s send echo requests to the moderator to showcase their availability. Based on these replies, the moderator in turn, sends a moderator notification response to the server, which the list of active clients.
4. Server to assign task to a group using a multicast message then can use the above client list and the task set. The clients fetch the sub-set of data set provided based on relative ID’s in the client list, and perform the task in parallel. Moderator accumulates all the result’s and notifies the server with the result.

## XDR Communication

**External Data Representation** (**XDR**) is a [standard](https://en.wikipedia.org/wiki/Technical_standard) [data serialization](https://en.wikipedia.org/wiki/Data_serialization) format, for uses such as [computer network](https://en.wikipedia.org/wiki/Computer_network) protocols. It allows data to be transferred between different kinds of computer systems. Converting from the local representation to XDR is called encoding. Converting from XDR to the local representation is called decoding. XDR is implemented as a software library of functions which is portable between different [operating systems](https://en.wikipedia.org/wiki/Operating_system) and is also independent of the [transport layer](https://en.wikipedia.org/wiki/Transport_layer).

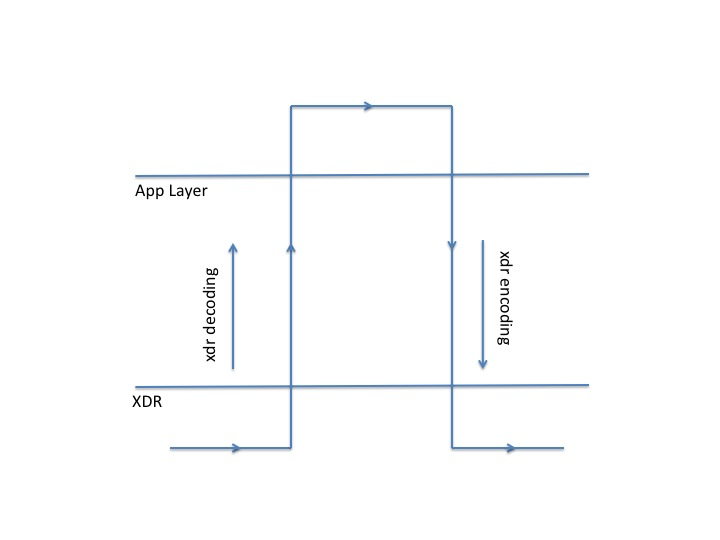


Figure XDR Communication

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1.

XDR is used for all the communication’s between client’s and server, in the current architecture.

## Multicast Communication

In [computer networking](https://en.wikipedia.org/wiki/Computer_networking), multicast is group communication where [information](https://en.wikipedia.org/wiki/Information) is addressed to a group of destination computers simultaneously. IP multi-casting is a communication mechanism in which data is communicated from server to a set of clients who are interested in receiving that data. Any client can dynamically enter or leave the communication. Multicast IP address form the basis of multicast communication. In terms of classes, it’s the Class D IP addresses that are used as multicast IP addresses. Here is the structure of Class D IP addresses :

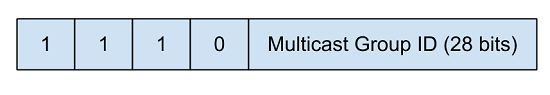


Figure Multicast group IP format

So, it can be easily said that multicast IPs range from 224.0.0.0 to 239.255.255.255. As with the case of ports (where we have well known ports ie 0-1024), there are some reserved multicast IP addresses or well known IP addresses.

In the current architecture, multicast messages are send by server to the group’s while announcing the moderator for the group and while distributing the task. This ensures efficient use of network bandwidth.

# Software Requirements

## Assumptions

1. Max 1000 groups will be supported
2. Scale a single group to 255 clients
3. A single group can perform only one task at a time.

# Memory and Performance Impact

Memory impact is less as memory is dynamically allocated/free as per the requirement.

Also, to optimize the performance Red Black tree’s are used for long list’s where lookup is needed more often.

# CLI and Epoll

## Server

Following are the server side CLI’s:

1. show groups -- Displays list of groups
2. show group info <group\_name|all> -- Displays group - client association
3. enable msg group <group\_name> -- Enables display of messages for a specific group.
4. no msg group <group\_name> -- Disables display of messages for a specific group.
5. send msg <group\_name> -- Sends a multicast message to the specified Group
6. task <task\_type> group <group\_name>-- Assigns a specific task to the specified Group
7. cls -- Clears the screen

## Client

Following are the client side CLI’s:

1. show client groups -- displays list of groups joined by client
2. enable keepalive group <group\_name> -- Sends periodic messages to Server
3. disable keepalive -- Stops periodic messages to Server
4. join group <name> -- Joins a new group
5. leave group <name> -- Leaves a group
6. cls -- Clears the screen
7. show pending clients -- Moderator CLI. Shows list of working clients
8. show done clients -- Moderator CLI. Shows list of done clients
9. test echo -- Echo debug CLI

In order to execute the above CLI’s at user’s command, epoll is being using at both client and server side. Epoll is a I/O event notification facility. In event-driven servers it is important that the server focuses on connections that can be serviced without blocking its main process. The Epoll event mechanism  is designed to scale to larger numbers of connections than select and poll.

Epoll is an **O(1) algorithm** – this means that it scales well as the number of watched file descriptors increase. Epoll is the latest, greatest, newest polling method in Linux. It differs both from poll and select in such a way that it keeps the information about the currently monitored descriptors and associated events inside the kernel, and exports the API to add/remove/modify those.

# Data Structures

## Server

Following data structures are used at server side:

**/\* Declaration of Server Info - Main Data structure on Server\*/**

struct server\_information\_t;

typedef struct {

unsigned int server\_fd; /\*Server FD\*/

unsigned int task\_id; /\*Next task Id\*/

mcast\_group\_t \*server\_list; /\*Server List having group nodes for all the multicast groups.\*/

void \*client\_RBT\_head; /\*Pointer to RBTree head, which maintains global list of all clients.\*/

bool (\* fsm)(struct server\_information\_t \*server\_info, server\_event\_t event, void \*fsm\_msg); /\*Server FSM function pointer\*/

} server\_information\_t;

**/\*Group Node - This node maintains information related to multicast group.\*/**

typedef struct {

char \*group\_name; /\*Multicast group name\*/

int number\_of\_clients; /\*No. of clients under the group\*/

struct sockaddr\_in grp\_mcast\_addr; /\*Group Multicast Address\*/

unsigned int group\_port; /\*Group Multicast Port\*/

mcast\_client\_t \*client\_info; /\*List of clients, joined to multicast group\*/

mcast\_client\_node\_t \*moderator\_client; /\*Points to client which is moderator\*/

server\_state\_t fsm\_state; /\*FSM state of group, to be used by server FSM\*/

sn\_list\_element\_t list\_element;

int task\_type; /\* A group can perform only one task at a time. Maintaining the task type \*/

} mcast\_group\_node\_t;

**/\*Client Node - This node maintains information about the client, which is part of a group\*/**

typedef struct {

unsigned int client\_id; /\*Client ID\*/

struct sockaddr client\_addr; /\*Address of Client\*/

svr\_client\_state av\_status; /\*Availibility status of client - busy or free\*/

sn\_list\_element\_t list\_element;

} mcast\_client\_node\_t;

For efficient look up in the client list, Red black trees are implemented.

typedef struct RBT\_node {

unsigned int key;

struct sockaddr client\_addr;

unsigned int port;

void \*client\_grp\_list;

avail\_state av\_status;

uint8\_t is\_moderator:1;

int red; /\* if red=0 then the node is black \*/

struct RBT\_node\* left;

struct RBT\_node\* right;

struct RBT\_node\* parent;

} RBT\_node;

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

The RB **tree height is always O(log n),** where n is the number of node in the tree. The effect of this is that searching for a node in a balanced tree takes **O(log n)** time. Similarly, adding and removing also take O(log n). This is in contrast to unbalanced trees, where the worst-case complexity for searching/adding/removing is O(n).

## Client

**/\* Declaration of Client Info - Main Data structure on Client\*/**

typedef struct {

uint8\_t is\_moderator:1; /\*Is this is a moderator?\*/

moderator\_information\_t\* moderator\_info; /\*If moderator, then it must have list of clients\*/

unsigned int client\_id; /\*Client ID\*/

unsigned int client\_fd; /\*Client FD\*/

unsigned int epoll\_fd; /\*EPoll FD\*/

unsigned int client\_status; /\*Busy or Free\*/

struct sockaddr\_in server; /\*Address of server\*/

struct sockaddr moderator; /\*Address of moderator\*/

struct epoll\_event \*epoll\_evt; /\*Pointer to epoll event structure, to register the new FD's\*/

client\_grp\_t \*client\_grp\_list; /\*List of client group nodes, each node will have info. abt the joined group.\*/

} client\_information\_t;

**/\* client group node - This node comprises of multicast group information, of which client is member of.\*/**

typedef struct {

char group\_name[10]; /\*Name of multicast group\*/

struct sockaddr\_in group\_addr; /\*Address of multicast group\*/

int group\_port; /\*Multicast group port\*/

unsigned int mcast\_fd; /\*Multicast FD associated with every group\*/

sn\_list\_element\_t list\_element;

} client\_grp\_node\_t;

**/\* Moderator info - This contains all the information about moderator\*/**

typedef struct {

char group\_name[10]; /\*Name of group moderator belongs to.\*/

client\_grp\_t \*pending\_client\_list; /\*List of clients who are working on task and currently being checked for keepalive\*/

client\_grp\_t \*done\_client\_list; /\*List of clients who have completed the task\*/

moderator\_state\_t fsm\_state; /\*Moderator FSM state\*/

uint8\_t task\_id; /\*Task Identifier \*/

uint8\_t active\_client\_count; /\*Number of active clients in the multicast group\*/

bool (\* fsm)(struct client\_information\_t \*client\_info, moderator\_event\_t event, void \*fsm\_msg); /\*Moderator FSM function pointer\*/

void \* moderator\_resp\_msg; /\*Pointer to buffer where moderator will be storing the result of the group\*/

} moderator\_information\_t;

# Testing Considerations

**Test-driven development** (**TDD**) is a [software development process](https://en.wikipedia.org/wiki/Software_development_process) that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated [test case](https://en.wikipedia.org/wiki/Test_case) that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally [refactors](https://en.wikipedia.org/wiki/Code_refactoring) the new code to acceptable standards.

End of Document