# Project Report

## System Architecture:

The system used to design this project relies a lot on the singleton pattern. Specific work is split to different components which communicate with each other. All of these components are implemented as Singleton classes.

The following are the major components of the system.

### RequestIdProvider:

This is a singleton class that lives for the whole lifecyle of the program. This class can be quered to get a unique request id to be sent through messages. This class is used only by peers who are leaders.

### ProcessInfoHelper:

This component keeps track of the information about the process, like its ID and hostname. Moreover, it also keeps a list of all other processes in the system. It is handy helper for the others when required.
This class also keeps a track of the LEADER and its information. Whenever a leader update happens, the peers notify this class.

### MessageSender:

This source file is where all the helper methods to send messages via TCP to other peers, live. It is specific in its functionality that it contains functions to send a message via TCP. Moreover, for simplicity, every send is a fire and close. It opens a socket, sends the message and closes the socket. This can be optimised to keeping a track of the open sockets and re-using the same socket to send messages to the respective peer.

### ReqMessageOkCollecter:

This is a standalone class which aids in tracking OkMessages for a request. This is used by the leader when performing 2pc (ADD and DEL). The class constructs itself with a request id, the size of the membership and a function which will be called when all the OkMessages are received for a particular request. It expects OkMessages equal to the size of the membership. Once it receives all the OkMessages, it calls the handling function that was passed to it through its constructor.

## NewLeaderResponseHandler:

This is a standalone class which aids in handling the pending operations sent by the peers during the reconciliation process. This class ignored all operations of type NOTHING and sends all unique operations(duplicates are ignored, i.e same operation for same peer from different alive peers) to the MessageHandler for processing.

## MessageHandler:

This is where most of the work for this project happens. MessageHandler is where all the tcp messages received are handled. Whenever a message is received, it is parsed and then handle appropriately:

- **JoinRequest**: When a join request is received, the handler ensures that the receiving process is the Leader. If so, as the leader, this peer initiates a 2pc to add the new peer to the membership. Initially, the membership list only consists of the Leader.
  Once a 2pc to add a peer is initiated, the handler makes use of the **ReqMessageOkCollecter** to keep track of the OkMessages. After initiating the 2pc, the handler instantiates a **ReqMessageOkCollecter** with the sent request Id, the expected response count and a function which handles agreement for this request from all peers. Once all the peers respond with an OK, the leader then updates its membership list and sends the updated list to everyone in the membership (inclusive of the new peer) as a NEWVIEW message.
- **RequestMessage**: These are handled by saving the request in the process memory and then sending an OkMessage with the same requestID back to the leader.
- OkMessages: The handler, ensures that the receiver of these messages is a LEADER. Afterwards, it sends this message to the **ReqMessageOkCollecter** for processing.
- NewViewMessage: These messages are handled by updating the MembershipList with the new view id and the new list.
- NewLeaderMessage: These are handled by updating the leader and then sending the pending operations to the new leader. *Note: The only operation supported for this type of message is PENDING.*
- NewLeaderResponse: These are also only supposed to be received by the new leader. The new leader takes in these messages and delegates the work of handling them to the **NewLeaderResponseHandler.**

Every message sent by the leader is sent to everyone in the membership, this includes the leader itself. This can be optimised

## MembershipList:

This is a singleton class which maintains the membership list for the peer along with the view id. Each time the membership updates, the peer call this class to update it. The leader also uses this class. Moreover, this class is thread safe and can handle modifications from multiple threads without inconsistencies.

## FailureDetector:

The failure detector is a singleton class which performs the job of detecting the peers who have failed. It delegates the handling of failure to main() with the help of a callback function. Its primary functions are to send periodic heartbeats to all the peers it thinks are alive and also expects heartbeats from peers. It uses UDP for this functionality with the help of the UdpSender and UdpListener.

# Serialising Membership List:

Sending the membership list was proving to be a little tricky. In this project, a list is represented by a uint32_t type. Every peer id's are bits flipped to 1.
The **bitset** class by c++ is used for this.

**Converting the list to int:**

- Initialise a bitset<32> for a value of 0 (bits of 0)
- Go through the list and for every peerid flip the corresponding bit.
  For eg: if list is [1, 3, 4] then the bitset value would be [0000011010]
- This bitset is then converted to a uint32_t and then sent via the socket.

**Converting uint32_t to list:**

- Convert uint32_t received to bitset<32>
- Go through each bit in the bitset and add the index with 1's into the list of peers.

# The following are utilities that are generic in their functionalities:

## TcpSender

This class encapsulates everything required to send a message via TCP to a host at a specified port. It extends the **SocketImpl** class, which consists of all the boilerplate code for socket connections.

## TcpListener

This class encapsulates everything required to receive messages via TCP. It uses select() and can listen to multiple connections. This class also, extends the **SocketImpl** class.

## UdpSender

This class is used to send messages to a host at a port via UDP. Similar to the **TcpSender** it extends the SocketImpl class.

## UdpListener

Much like the TcpListener, the UdpListener is also a class that extends the **SocketImpl** class. It listens to messages via udp send to its localhost at the specified port.

## Log

Through the system, logs are handled by this class. It has static methods that help keeps logs within the system (print to the console). There is an option to set the log level for the system and track the respective log (error, verbose, info, debug, none, print and fatal). Fatal logs crash the program after printing. And print log prints to the screen regardless of the log level specified.
***Usage**: Log::v("message") //v stands for verbose*

### SockImpl

Is a helper class that makes handling sockets easy.

# Implementation Assumptions:

1. The system currently DOES NOT handle more than one event at a time. If more than one events are encountered, the process is forced to crash to avoid unexpected behaviour.
2. The testing is done for mostly 5 peers, although it should work for more than 5 peers. The only caveat being that when crashing peers, enough intervals need to be provided to avoid a process from detecting an event before completely handling the previous event. The testcases are written assuming 5 peers, change to this number will require change in the SLEEP time for each test case.
3. Tcp failures are also considered Fatal and crashes the system, which is why it is important to clear used addresses before running the program.