Q1: Elixir and Erlang use the BEAM Virtual Machine which is a single Operating System process. BEAM assigns one scheduler per operating system thread by default; which then schedules Erlang processes to that thread. The number of processes created is not constrained to operating system limits.

A process is an instance of a function being executed. Should the need arise, this code can be scaled out by spawning more processes to run the same code The processes will never share state but instead use asynchronous message passing if they need to share data during computations.

A node is a running BEAM Virtual Machine inside which processes can be run. Nodes are run on a physical machine with the possibility of running multiple nodes on a single machine. Like processes, nodes can also communicate with each other through message passing however before message passing can be done between nodes, a node has establish a connection with other nodes by invoking Node.connect() in the code and passing the node and machine name. Once connected, processes between nodes can be invoked by using the Node.spawn() function or looking for the process ID in the global registry if a unique process ID was registered there.

Q3 (a): The bulletin board implementation has two types of background processes; one that represents a user and the second one representing a topic manager. When User.start() is run, this spawns a new user background process and adds its pid to the global registry under the user name. The topic managers will resolve the user pid by searching the global registry with the subscriber name. The user background process also keeps the current shell pid in its state so that it can pass messages back up when User.fetch_news() is run. Running User.subscribe() will spawn a new topic manager process if none is found in the global registry and subscribe the user to that. A message with the format {:subscribe, user_name} is sent to the topic manager.

When a user posts a new message to a topic, the message content is sent to the topic manager process by looking up the pid in the global registry with the topic name with the message format {:publish, user_name, content}. When User.fetch_news() is run, it runs the receive function to pull messages sent to the running iex shell process from the user background process. The receive function looks for messages with the format {:inbox, topic, sender, content}.

A topic manager process can be run with two roles; either as a master or replica which is set as a parameter (:master | :replica) when TopicManager.run() is executed alongside the topic name as the other parameter. The topic manager processes use the Paxos algorithm to update their state when the subscriber lists change or to elect a new master if a master node goes down. If spawned with a master role, the topic manager process will start with a ballot number of 201 and will spawn replica processes on other connected nodes which will have random ballot numbers less than 201. Replica nodes are sent a message to begin monitoring the master when initialised in the form {:monitor_master} and another message to register themselves globally. They register with the name format “#{topic_name}_replica_#{node_name}”.

When subscribing a user, the topic manager adds the user to its own subscriber list then sends a Paxos {:propose, ballot_num, subscribers} (subscriber list being v) to propagate the updated state amongst the replicas. The same process is done for un-subscribing a user. When a publish message is received, a message is send to every user process by looking up their pids in the global registry. Message format is {:new_post, topic_name, sender, content}. Messages are never held by the topic manager and are broadcast to all subscribed users on a best effort basis.

Q4: In an asynchronous message passing system, there are no upper bounds on time taken to deliver a message as a message can take an infinite number of cycles to deliver. Message delivery failure can be due to message delay or a process crashing. With regards to broadcast implementation B, as message delivery can take an infinite amount of steps, the message may come long after other messages have been received meaning the messages on all processes will not be received in the same order. There is also no guarantee that processes will get m1, m2 because of no check on message delivery. Asynchronous execution of message delivery models a reliable system in which it is assumed no message is lost and no process stops working.