**Treballcomplementari SODX**

**Pau Alcázar Perdomo i Francesco Oncins Spedo**

**Index**

Lab1 pàg. 1

Lab2 pàg. 12

Lab3 pàg. 26

Lab4 pàg. 48

## Chatty: a simple chat service

**Jordi Garcia**

Adapted with permission from Jordi Guitart & Xavier Leon (UPC) October 7, 2016

# Introduction

Your task will be to implement a distributed system that will allow you to chat among buddies. The purpose of this seminar is that you think about the main problems in distributed systems as well as learn a little bit of Erlang. We are going to implement two different versions of our chat system: i) a system composed by a single chat server to chat with your buddies; ii) a decentralized system with several chat servers which allows to clients connected to different servers chat with each other.

This document provides almost every piece of code. However, you may need to fill in the gaps (*. . .*) to ensure you understand what is the expected behavior of the system.

# Chatting with buddies

The initial version of the chat will consist of two different types of processes: *clients* (friend to chat with) and a *server*. Clients will connect –and, of course, disconnect– to the server and the server is going to be responsible to maintain the list of clients attached and relay messages sent by a client to the rest of clients. As this design is quite simple from the distributed systems point of view, we are going make things more robust later on.

## The server

We need to implement a server that keeps track of connected users and relays messages sent by one user to the rest of users. Thus, the message interface the server is going to handle is as follows:

* + - *{*client join req, Name, From*}*: a join request from a client containing its username (Name) and a reference of the process identifier (From) that the server should use to contact him/her. The server needs to update the list of connected clients and send to the connected users this new event (join).
    - *{*client leave req, Name, From*}*: a leave request from a client contain- ing its username (Name) and a its process identifier (From). The server needs to remove the process from the list of connected clients and send to the connected users this new event (leave).
    - *{*send, Name, Text*}*: a request to send a message (Text) to the con- nected users and the client’s username (Name).
    - disconnect: server receiving the message disconnects.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open up a new file **server.erl** and declare the module **server**:

To complete this code, you need to:

* Implement the logic for adding a new client to NewClients in the {client\_join\_req, ...} clause.
* Correctly remove the leaving client in the {client\_leave\_req, ...} clause.
* Complete the broadcast call parameters in the {send, ...} clause and update the process\_requests calls with the appropriate client list.
* **-module(server).**: This declares the module named server.
* **-export([start/0]).**: This exports the start/0 function, making it callable from other modules. The /0 indicates that start takes no arguments.
* **start()**: This function starts the server.
* **spawn(fun() -> process\_requests([]) end)**: This spawns a new process that runs process\_requests/1 with an empty list [] (initially no connected clients).
* **register(myserver, ServerPid)**: This registers the newly created process with a name myserver, so it can be easily referenced.
* **process\_requests(Clients)**: This function handles different messages received by the server.
* **{client\_join\_req, Name, From}**: When a client joins, it sends a join request with its name and process identifier.
  + **NewClients = [...|Clients]**: Add the new client to the list of clients.
  + **broadcast(NewClients, {join, Name})**: Notify all clients about the new client joining.
* **{client\_leave\_req, Name, From}**: When a client leaves, it sends a leave request.
  + **NewClients = lists:delete(..., ...)**: Remove the client from the list.
  + **broadcast(Clients, ...)**: Notify all clients about the client leaving.
  + **From ! exit**: Send an exit signal to the leaving client's process.
* **{send, Name, Text}**: When a client sends a message.
  + **broadcast(..., ...)**: Relay the message to all connected clients.
* **disconnect**: If the server receives a disconnect message, it unregisters itself.
* **broadcast(PeerList, Message)**: This function sends a given message to all peers in PeerList.
* **Fun = fun(Peer) -> Peer ! Message end**: A function that sends Message to each Peer.
* **lists:map(Fun, PeerList)**: Applies Fun to each element in PeerList.

-module(server).

%% Exported Functions

-export([start/0]).

%% API Functions start() ->

ServerPid = spawn(fun() -> process\_requests([]) end), register(myserver, ServerPid).

process\_requests(Clients) -> receive

{client\_join\_req, Name, From} ->

NewClients = [...|Clients], %% TODO: COMPLETE broadcast(NewClients, {join, Name}), process\_requests(...); %% TODO: COMPLETE

{client\_leave\_req, Name, From} ->

NewClients = lists:delete(..., ...), %% TODO: COMPLETE broadcast(Clients, ...), %% TODO: COMPLETE

From ! exit,

process\_requests(...); %% TODO: COMPLETE

{send, Name, Text} ->

broadcast(..., ...), %% TODO: COMPLETE

process\_requests(Clients); disconnect ->

unregister(myserver)

end.

%% Local Functions broadcast(PeerList, Message) ->

Fun = fun(Peer) -> Peer ! Message end, lists:map(Fun, PeerList).

## The client

So far, we have seen how the server is implemented. Now, we will specify how the client works. The client process will have two different tasks to perform. We will have a background task responsible for handling replies from the server and the main task will be responsible to read a message from the standard input to be sent to the rest of your buddies. The background task (the one handling server replies) should handle the following message interface:

* + - *{*join, Name*}*: the server informs that a new client is connected. We should write this information through the standard output.
    - *{*leave, Name*}*: the server informs that a connected client is about to dis- connect. We should write this information through the standard output.
    - *{*message, Name, Text*}*: this is a message sent back by the server from one of the connected users (Name). We should print this message (Text) to the standard output so we can actually read it.
    - exit: the client background task is terminated.

Notice that, in this case, we do not need the reference to the client sending a message (From) as a buddy does not contact directly with other buddies but with the server.

The main process should block waiting for the user to write down some text to send to the chatting room. If a user wants to leave, we need to write a *command* (exit) to actually leave the chat. Once the user writes this keyword, the main process will send a request to the server to leave the room.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file **client.erl** and declare the module **client**.

The background task of the client handles messages from the server. This is done by a loop that continuously listens for the following types of messages:

1. **Join Notification**: When a new client connects to the server, the server sends a {join, Name} message to all connected clients. The client's background task receives this message and outputs a notification, such as "[JOIN] [Name] joined the chat", to the standard output.
2. **Leave Notification**: Similarly, when a client disconnects, the server sends a {leave, Name} message. Upon receiving this, the client's background task outputs a message indicating that the client has left the chat.
3. **Chat Message**: When a client sends a message, the server relays this as {message, Name, Text} to all clients. The receiving client's background task then displays this message on the standard output, typically formatted as "[Name] Text".
4. **Exit**: If the background task receives an exit message, it terminates.

### Main Task

The main task of the client is responsible for reading user input and sending messages. This involves:

* **Reading User Input**: The client waits for the user to type a message into the standard input. If the user types "exit" and hits enter, it triggers a disconnection process. For any other input, it is considered a chat message to be sent.
* **Handling 'exit' Command**: When the user types "exit", the main task sends a {client\_leave\_req, MyName, ClientPid} message to the server, indicating that the client wants to leave the chat room. After sending this request, the client process will typically terminate.
* **Sending Messages**: For other inputs, the main task sends a {send, MyName, Text} message to the server. This message includes the client's name and the text they wish to send. The server then relays this message to other connected clients.

### Initialization

When the client starts, it first initializes the connection to the server by sending a {client\_join\_req, MyName, ClientPid} message. This is part of the init\_client function, which is called as soon as the client process starts. This function notifies the server of the new client's presence and starts the background task for handling incoming messages.

-module(client).

%% Exported Functions

-export([start/2]).

%% API Functions start(ServerPid, MyName) ->

ClientPid = spawn(fun() -> init\_client(ServerPid, MyName) end), process\_commands(ServerPid, MyName, ClientPid).

init\_client(ServerPid, MyName) ->

ServerPid ! {client\_join\_req, ..., ...}, %% TODO: COMPLETE process\_requests().

%% Local Functions

%% This is the background task logic process\_requests() ->

receive

{join, Name} ->

io:format("[JOIN] ~s joined the chat~n", [Name]),

%% TODO: ADD SOME CODE

{leave, Name} ->

%% TODO: ADD SOME CODE

process\_requests();

{message, Name, Text} -> io:format("[~s] ~s", [Name, Text]),

%% TODO: ADD SOME CODE

exit ->

ok

end.

%% This is the main task logic process\_commands(ServerPid, MyName, ClientPid) ->

%% Read from standard input and send to server Text = io:get\_line("-> "),

if

end.

Text == "exit\n" ->

ServerPid ! {client\_leave\_req, ..., ...}, %% TODO: COMPLETE ok;

true ->

ServerPid ! {send, ..., ...}, %% TODO: COMPLETE

process\_commands(ServerPid, MyName, ClientPid)

## Testing

Debugging Erlang code may be quite confusing sometimes because of the error messages. Once you get used to them, you will quickly understand what is wrong with your code. In the meanwhile, it is useful to put debugging information to know where your code is failing and the state of your variables.

If you are not using an Erlang IDE which automatically compiles your code, you will need to do it by hand -every time you modify the file- before calling your implemented procedures. A simple test can be done in a single computer. You will execute a server and a client in different terminals. To start a server write the following:

user@host:~/Chatty$ erl -name [server\_node@127.0.0.1](about:blank) -setcookie secret

(server\_node@127.0.0.1)1> c(server). %% Compile server module (server\_node@127.0.0.1)2> server:start().

true

To start a client open up a new terminal and write the following:

user@host:~/Chatty$ erl -name [client\_node@127.0.0.1](about:blank) -setcookie secret

(client\_node@127.0.0.1)1> c(client). %% Compile client module (client\_node@127.0.0.1)2> client:start({myserver, [’server\_node@127.0.0.1’}](about:blank), "John"). [JOIN] John joined the chat

And if you type a message, you should see something like...

-> hi! [John] hi!

**Experiments**. Try opening 2 or 3 clients and test that all of them receive all the messages. You can also try to communicate among different physical machines (remember to change the local IP by the fully-qualified domain name (or the public IP) of the corresponding machines). Read the ’Erlang primer’ to refresh how to contact remote nodes.

**Open Questions**. i) Does this solution scale when the number of users increase? ii) What happens if the server fails? iii) Are the messages from a single client guaranteed to be delivered to any other client in the order they were issued? (hint: search for the ’order of message reception’ in Erlang FAQ) iv) Are the messages sent concurrently by several clients guaranteed to be delivered to any other client in the order they were issued? v) Is it possible that a client receives a response to a message from another client before receiving the original message from a third client? vi) If a user joins or leaves the chat while the server is broadcasting a message, will he/she receive that message?

# Making it robust

The previous design has its disadvantages. Mainly, it is not robust to failures –i.e. if the server fails the whole system will become useless. A solution to this problem is to have more servers (replication). Thus, if a server fails, we will have other servers still running and sending messages. As usually, solving a problem is an open door for other interesting problems.

We will have a set of replicated servers to which users may connect indis- tinctly. This way, if a server fails, only those users connected to the failing server will lose connectivity but the rest will remain connected. Of course we could implement a solution in which clients automatically reconnect to another server when they detect a failure but we are going to keep things simple.

We will only need to change the server implementation to introduce this new functionality. This new implementation needs to know the list of replicated servers. Besides, we need to handle the membership of the set of servers. Users will connect to one of the servers and send messages to it. This server will forward the message to the set of servers which in turn will relay the message to its clients.

The new messages that the server needs to handle are as follows:

* *{*server join req, From*}*: a new server is added to the set of replicated servers. We need to update the list of servers and inform the rest of servers of this new node.
* *{*update servers, NewServers*}*: a server gets informed when a new server has joined the set.
* RelayMessage: if the message received does not match any of the previous clauses, the server relays the message to all of its clients (it may be either a message of type join, leave, or message).

Be aware that now, the messages previously implemented (client join req, client leave req, and send) **are not forwarded directly to the clients but to every member of the set of servers**. They in turn will relay those messages to their connected clients. In addition, when a server disconnects, the rest of servers are informed.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file **server2.erl** and declare the module **server2**.

To enhance the robustness of the chat system by using replicated servers, the Erlang code must be adapted to handle multiple servers and ensure messages are relayed through these servers. Here's an overview of the approach and the changes needed:

### Overview of the Replicated Server Approach

1. **Multiple Servers**: Multiple instances of the chat server run in parallel. Clients can connect to any of these servers.
2. **Server Replication**: Each server is aware of other servers in the network. This is achieved by maintaining a list of servers on each server instance.
3. **Message Forwarding**: When a client sends a message to a server, the server forwards this message to all other servers, which then relay it to their connected clients. This ensures all clients receive the message, regardless of which server they are connected to.
4. **Handling Server Failures**: If a server fails, only the clients connected to that server lose connectivity. Other clients remain unaffected. While automatic reconnection is not implemented in this design, it could be a potential enhancement.

### Implementation Details

* **Initialization of Servers**: Servers are initialized with either no boot server (init\_server()) or with a boot server (init\_server(BootServer)). A boot server is a server that the new server will contact to join the network of replicated servers.
* **Handling New Server Requests**: When a new server joins ({server\_join\_req, From}), the receiving server updates its list of servers and informs all other servers about this new addition.
* **Updating Server Lists**: When a server receives an {update\_servers, NewServers} message, it updates its list of replicated servers.
* **Client-Server Interaction**: Messages from clients ({client\_join\_req, ...}, {client\_leave\_req, ...}, {send, ...}) are handled as before, but now these messages are also forwarded to all servers in the network.
* **Message Relay to Clients**: Any message that doesn't match the specific patterns for server-server communication is considered a client message and is relayed to all connected clients.
* **Server Disconnection**: When a server disconnects, it removes itself from the server list and informs other servers to update their lists.

### Code Gaps and Modifications

To complete the implementation, the following changes and additions are needed:

* **Filling in the Gaps**: The ellipses (...) in the code need to be filled with appropriate logic to handle the addition and removal of clients and servers from their respective lists, and to broadcast messages correctly.
* **Broadcast Function**: The existing broadcast/2 function must be adapted to handle broadcasting messages not just to clients, but also to other servers.

-module(server2).

%% Exported Functions

-export([start/0, start/1]).

%% API Functions start() ->

ServerPid = spawn(fun() -> init\_server() end), register(myserver, ServerPid).

start(BootServer) ->

ServerPid = spawn(fun() -> init\_server(BootServer) end), register(myserver, ServerPid).

init\_server() -> process\_requests([], [self()]).

init\_server(BootServer) ->

BootServer ! {server\_join\_req, self()}, process\_requests([], []).

process\_requests(Clients, Servers) -> receive

%% Messages between client and server

{client\_join\_req, Name, From} ->

NewClients = [...|Clients], %% TODO: COMPLETE broadcast(..., {join, Name}), %% TODO: COMPLETE process\_requests(..., ...); %% TODO: COMPLETE

{client\_leave\_req, Name, From} ->

NewClients = lists:delete(..., ...), %% TODO: COMPLETE broadcast(..., {leave, Name}), %% TODO: COMPLETE

From ! exit,

process\_requests(..., ...); %% TODO: COMPLETE

{send, Name, Text} ->

broadcast(Servers, ...), %% TODO: COMPLETE process\_requests(Clients, Servers);

%% Messages between servers disconnect ->

NewServers = lists:delete(self(), ...), %% TODO: COMPLETE broadcast(..., {update\_servers, ...}), %% TODO: COMPLETE unregister(myserver);

{server\_join\_req, From} ->

NewServers = [...|Servers], %% TODO: COMPLETE

broadcast(..., {update\_servers, NewServers}), %% TODO: COMPLETE process\_requests(Clients, ...); %% TODO: COMPLETE

{update\_servers, NewServers} ->

io:format("[SERVER UPDATE] ~w~n", [NewServers]), process\_requests(Clients, ...); %% TODO: COMPLETE

RelayMessage -> %% Whatever other message is relayed to its clients broadcast(Clients, RelayMessage),

process\_requests(Clients, Servers)

end.

%% Local Functions broadcast(PeerList, Message) ->

Fun = fun(Peer) -> Peer ! Message end, lists:map(Fun, PeerList).

You need to first start a server with the function server2:start() –which creates a server with an empty list of server references– and, then start different servers with the function server2:start({myserver,’server\_node@IP’})–

which will connect this server to the rest of servers. Remember to change the remote hostname by the IP of the server you are connecting to.

**Experiments**. Once you have a set of servers up and running, try connect- ing some clients to each of the server instances and begin to chat. Does it work? You can also try to crash some of the servers and see what happens.

**Open Questions**. i) What happens if a server fails? ii) Do your answers to previous questions iii, iv, and v still hold in this implementation? iii) What happens if there are concurrent requests from servers to join or leave the system?

iv) What are the advantages and disadvantages of this implementation regarding the previous one?

## Muty: a distributed mutual-exclusion lock

**Jordi Garcia**

Adapted with permission from Jordi Guitart & Xavier Leon (UPC)

November 11, 2016

# Introduction

Your task is to implement a distributed mutual-exclusion lock. The lock will use a multicast strategy and work in an asynchronous network where we do not have access to a synchronized clock. You will do the implementation in three versions: the deadlock prone, the unfair, and the Lamport clocked. Before you start you should have good theoretical knowledge of the Ricart & Agrawala mutual exclusion algorithm and how Lamport clocks work.

# The architecture

The scenario is that a set of workers need to synchronize and, they will randomly decide to take a lock and when taken, hold it for a short period before releasing it. The lock is **distributed**, and each worker will operate with a given instance of the lock. Each worker will collect statistics on how long it took them to acquire the lock so that it can present some interesting figures at the end of each test.

Let’s first implement the worker and then do refinement of the lock.

## The worker

When the worker is started it is given its lock instance. It is also given a name for nicer print-out. We also provide information on for up to how long the worker is going to sleep before trying to get the lock and work with the lock taken.

We will have four workers competing for a lock so if they sleep for up to 1000 ms and work for up to 2000 ms, we will have a lock with high chance of congestion. You can easily change these parameters to simulate more or less congestion. The withdrawal constant is how long (8000 ms) we are going to wait for a lock before giving up.

The gui is a process that will give you some feedback on the screen on what the worker is actually doing. The complete code of the gui is given in the appendix.

-module(worker).

-export([start/4])

define(withdrawal, 8000). start(Name, Lock, Sleep, Work) ->

spawn(fun() -> init(Name, Lock, Sleep, Work) end).

init(Name, Lock, Sleep, Work) -> Gui = gui:start(Name),

{A1,A2,A3} = now(),

random:seed(A1, A2, A3),

Taken = worker(Name, Lock, [], Sleep, Work, Gui), Gui ! stop,

Lock ! stop, terminate(Name, Taken).

### Worker Process Logic

The core functionality of each worker process resides in a function (presumably named worker) which was referenced but not detailed in the code snippet. This function would:

* Attempt to acquire the lock.
* Perform some work if the lock is acquired, for a random duration up to the specified Work time.
* Release the lock after the work is done.
* Handle the situation where the lock cannot be acquired within the withdrawal time, in which case it would stop trying and possibly perform some other action.

### Random Sleep and Work Durations

The workers sleep for a random duration up to Sleep milliseconds and work for up to Work milliseconds. This randomness is essential for simulating a realistic scenario where processes do not always request resources simultaneously or for the same duration. The random:seed(A1, A2, A3) call initializes the random number generator, which ensures that each worker process has a different random sequence, making the simulation more accurate.

### Lock Mechanism

The lock mechanism (referred to by Lock in the code) is pivotal but its implementation details are not provided. This lock could be another Erlang process that manages access to a shared resource. Worker processes would send messages to this lock process to request and release the lock.

### Withdrawal Mechanism

The withdrawal constant defines how long a worker waits to acquire the lock before giving up. This mechanism is crucial in preventing deadlock scenarios where a worker might wait indefinitely for a lock that it can never acquire.

### GUI Process

Each worker interacts with a GUI process for visual feedback. This GUI process is likely another Erlang process that receives messages from the worker and updates the GUI accordingly. The specifics of how the GUI is updated based on the messages it receives (such as displaying a worker's status) are not detailed in the snippet.

### Termination and Cleanup

After completing its work (or giving up on acquiring the lock), the worker process sends a stop message to both the GUI and the Lock process. It then calls a terminate function with its Name and a boolean Taken indicating whether it successfully took the lock. This cleanup is crucial for proper resource management and preventing orphan processes.

We will do some book-keeping and save the time it took to get the locks.

In the end we will print some statistics.

A worker sleeps for a while and then decides to move into the critical sec- tion (if worker has not been stopped while sleeping). The call to acquire/3 will return information on if the critical section was entered and how long it took to acquire the lock. For each invocation of the acquire function, the worker stores this information in the Taken list.

### Worker Function

The worker function is the main logic for each worker process. It simulates a worker trying to enter a critical section (i.e., a section of code that should only be accessed by one process at a time) by acquiring a lock.

### Sleep and Work Simulation

* **Sleeptime**: Each worker sleeps for a random amount of time up to Sleep milliseconds. This simulates the delay before attempting to acquire the lock.
* **Worktime**: After acquiring the lock, the worker "works" for a random amount of time up to Work milliseconds.

### Acquiring the Lock

* **acquire/3 Function**: This function tries to acquire the lock. It returns information about whether the lock was acquired and how long it took. The details of this function are not provided in the snippet, but it's crucial for interacting with the lock.
* **Handling the Result of acquire/3**:
  + **stopped**: If the worker is stopped while trying to acquire the lock, it stops and returns the Taken list.
  + **withdrawn**: If the worker withdraws (presumably after waiting too long), it recurses with the updated Taken list.
  + **Other Cases (Lock Acquired)**: If the lock is acquired, the worker enters the critical section and "works".

### Handling Stop Requests

The worker can receive a stop message during its sleep time or while working. In such cases, it immediately stops its operation. If this happens while the lock is held, the worker releases the lock and notifies the GUI.

### Work Completion and Lock Release

* After the Worktime elapses, the worker releases the lock, informs the GUI (by sending a leave message), and recurses with the updated Taken list. This signifies the end of the worker's operation in the critical section.

### Book-keeping and Statistics

* **Tracking Lock Acquisition Time**: The time taken to acquire the lock each time is stored in the Taken list. This allows for statistical analysis after the worker has completed its operations, such as calculating the average time to acquire the lock.

worker(Name, Lock, Taken, Sleep, Work, Gui) -> Sleeptime = random:uniform(Sleep),

receive

stop ->

Taken after Sleeptime ->

T = acquire(Name, Lock, Gui), case T of

stopped ->

Taken; withdrawn ->

worker(Name, Lock, [T|Taken], Sleep, Work, Gui);

\_ ->

Worktime = random:uniform(Work), receive

stop ->

Gui ! leave, Lock ! release,

Taken after Worktime ->

io:format("~s: lock released~n", [Name]), Gui ! leave, end.

end

end

Lock ! release,

worker(Name, Lock, [T|Taken], Sleep, Work, Gui)

The critical section is entered by requesting the lock to the worker’s lock instance. Notice that locks instances are implemented as independent processes. We wait for a reply taken or for a withdrawal timeout. Note that we can get a timeout when we are really in a deadlock, or simply when the lock instance is taking too long to respond. The now diff/2 function should calculate the time in milliseconds from the times T1 and T2. The elapsed time T is then returned to the caller.

The gui is informed as we send the request for the lock and if we acquire the lock or have to abort.

acquire(Name, Lock, Gui) -> T1 = now(),

Gui ! waiting, Ref = make\_ref(),

Lock ! {take, self(), Ref}, receive

{taken, Ref} ->

T2 = now(),

T = timer:now\_diff(T2, T1) div 1000, io:format("~s: lock taken in ~w ms~n", [Name, T]), Gui ! taken,

{taken, T}; stop ->

Gui ! leave, Lock ! release, stopped

after ?withdrawal ->

io:format("~s: giving up~n", [Name]), Gui ! leave,

Lock ! release, withdrawn

end.

The worker terminates when it receives a stop message. It will simply print out some statistics.

terminate(Name, Taken) ->

{Locks, Time, Dead} =

lists:foldl( fun(Entry,{L,T,D}) ->

case Entry of

{taken,I} ->

{L+1,T+I,D};

\_ ->

{L,T,D+1}

end end,

{0,0,0}, Taken),

if

Locks > 0 ->

Average = Time / Locks; true ->

Average = 0

end,

io:format("~s: ~w locks taken, ~w ms (avg) for taking, ~w withdrawals~n", [Name, Locks, Average, Dead]).

## The locks

We will work with three versions of the lock implemented in three modules: lock1, lock2, and lock3. The first lock, lock1, will be very simple and will not fulfill the requirements that we have on a lock. It will prevent several workers from entering the critical section but that is all about it.

When each lock instance is started, it is given a unique identifier and a list of peer-lock processes (i.e. the other lock instances). The identifier will not be used in the lock1 implementation, but we keep it there to make the interface to all locks the same.

The lock instance enters the state open and waits for either a command to take the lock or a request from another lock instance. If it is requested to take the lock, it will multicast a request to all the other lock instances and then enter a waiting state. A request from another lock instance is immediately replied with an ok message. Note how the reference is used to connect the request to the reply.

### The acquire Function

* **Entering the Critical Section**: The worker requests a lock from its lock instance, which is an independent Erlang process.
* **Waiting for a Response**: The worker waits for either:
  + A {taken, Ref} message indicating the lock has been acquired.
  + A timeout, indicating a potential deadlock or delay in lock acquisition.
* **Time Measurement**: The time taken to acquire the lock is calculated using timer:now\_diff(T2, T1).
* **GUI Interaction**: The GUI is updated with the worker's status at various stages - when waiting for the lock, upon lock acquisition, or when aborting due to a timeout.

#### Code Explanation for acquire

* **Gui ! waiting**: Notifies the GUI that the worker is waiting for the lock.
* **Lock ! {take, self(), Ref}**: Sends a request to the lock process to acquire the lock, identifying the request with a unique reference Ref.
* **Handling Responses**:
  + **Lock Acquired**: On receiving {taken, Ref}, calculates the elapsed time, updates the GUI, and returns {taken, T}.
  + **Stop Message**: If a stop message is received, the worker informs the GUI, releases the lock, and returns stopped.
  + **Timeout (?withdrawal)**: If the lock isn't acquired within the defined timeout period, the worker gives up, informs the GUI, releases the lock, and returns withdrawn.

### The terminate Function

This function is called when the worker is terminating. It processes the list of Taken times (accumulated during lock acquisitions) to print out statistics.

#### Code Explanation for terminate

* **Statistics Calculation**: Uses lists:foldl to iterate over the Taken list, counting the number of successful lock acquisitions (Locks), total time spent acquiring locks (Time), and the number of withdrawals (Dead).
* **Average Time Calculation**: If any locks were acquired, calculates the average time to acquire a lock; otherwise, sets the average to 0.
* **Output**: Prints statistics about the number of locks taken, the average time taken to acquire locks, and the number of times the worker withdrew from trying to acquire the lock.

-module(lock1).

-export([start/1]).

start(MyId) ->

spawn(fun() -> init(MyId) end).

init(\_) ->

receive

{peers, Nodes} -> open(Nodes);

stop ->

ok

end.

open(Nodes) -> receive

{take, Master, Ref} ->

Refs = requests(Nodes),

wait(Nodes, Master, Refs, [], Ref);

{request, From, Ref} -> From ! {ok, Ref}, open(Nodes);

stop ->

ok

end.

requests(Nodes) -> lists:map(

fun(P) ->

R = make\_ref(),

P ! {request, self(), R}, R

end, Nodes).

In the waiting state, the lock instance is waiting for ok messages. All requests have been tagged with unique references (using make ref() BIF) so that the lock instance can keep track of which lock instances have replied and which it is still waiting for (Refs). There is a simpler solution where we simply wait for *n* locks to reply, but this version is more flexible if we want to extend it.

While the lock instance is waiting for ok messages, it could also receive request messages from other lock instances that have also decided to take the lock. In this version of the lock we simply add these to a set of lock instances that have to wait (Waiting). When the lock is released we will send them ok messages.

As an escape from deadlock, we also allow the worker to send a release message even though the lock is not yet held. We will then send ok messages to all waiting lock instances and enter the open state.

### Lock Instance States

* **Waiting State**: The lock instance is in this state when it is waiting for ok messages from other lock instances. This state signifies that the lock is tentatively acquired, pending confirmation from all necessary nodes.
* **Held State**: Once all ok messages are received, the lock transitions to the held state, indicating that the lock is fully acquired.
* **Open State**: If a release message is received before the lock is fully acquired, the lock instance transitions back to the open state, indicating that it is available again.

### Handling of Messages in Waiting State

1. **Request Messages**: When the lock instance receives a {request, From, Ref} message from another lock instance, it adds this request to the Waiting set. This set keeps track of other lock instances that need to wait until the current lock is released.
2. **Ok Messages**: The lock instance also waits for ok messages ({ok, Ref}) from other nodes it previously communicated with. Upon receiving an ok message, it removes the corresponding reference from the Refs list.
3. **Transition to Held State**: If all expected ok messages are received (i.e., Refs becomes empty), the lock instance notifies the Master process that the lock has been successfully taken ({taken, TakeRef}) and transitions to the held state.
4. **Release Message Handling**: If a release message is received before the lock is fully acquired, the lock instance sends ok messages to all instances in the Waiting set and then enters the open state. This mechanism acts as an escape from potential deadlocks.

### The ok/1 Function

* **Notifying Waiting Lock Instances**: The ok(Waiting) function is called to notify all waiting lock instances that they can proceed. This is done by mapping over the Waiting list and sending each waiting instance an {ok, R} message.

### Code Explanation

* wait/5 Function: Manages the waiting state of the lock.
  + If Refs is empty, it sends {taken, TakeRef} to Master and calls held/2.
  + Handles incoming {request, From, Ref} and {ok, Ref} messages, updating the state accordingly.
  + On a release message, calls ok/1 to notify all waiting instances and transitions to the open state.
* held/2 Function: Represents the state where the lock is held (not detailed in the snippet).
* open/1 Function: Transitions the lock instance back to an open (available) state.

wait(Nodes, Master, [], Waiting, TakeRef) -> Master ! {taken, TakeRef},

held(Nodes, Waiting);

wait(Nodes, Master, Refs, Waiting, TakeRef) -> receive

{request, From, Ref} ->

wait(Nodes, Master, Refs, [{From, Ref}|Waiting], TakeRef);

{ok, Ref} ->

NewRefs = lists:delete(Ref, Refs),

wait(Nodes, Master, NewRefs, Waiting, TakeRef); release ->

ok(Waiting), open(Nodes)

end.

ok(Waiting) -> lists:map(

fun({F,R}) -> F ! {ok, R}

end, Waiting).

In the held state we keep adding requests from lock instances to the waiting list until we receive a release message from the worker.

For the Erlang hacker there are some things to think about. In Erlang, messages are queued in the mailbox of the processes. If they do match a pattern in a receive statement they are handled, but otherwise they are kept in the queue. In our implementation, we happily accept and handle all messages even though some, such as the request messages when in the held state, are just stored for later. Would it be possible to use the Erlang message queue instead and let request messages be queued until we release the lock? Yes! The reason for not doing so was to make it explicit that request messages are treated even if we are in the held state.

held(Nodes, Waiting) -> receive

{request, From, Ref} ->

held(Nodes, [{From, Ref}|Waiting]); release ->

ok(Waiting), open(Nodes)

end.

## Some testing

Next test procedure creates four locks instances and four workers. Note that we are using the name of the module (i.e. lock1) as a parameter to the start

procedure. We will easily be able to test different locks. We also provide the time (in milliseconds) for up to how long the worker is going to sleep before trying to get the lock (Sleep) and work with the lock taken (Work).

-module(muty).

-export([start/3, stop/0]).

start(Lock, Sleep, Work) ->

register(l1, apply(Lock, start, [1])), register(l2, apply(Lock, start, [2])), register(l3, apply(Lock, start, [3])), register(l4, apply(Lock, start, [4])), l1 ! {peers, [l2, l3, l4]},

l2 ! {peers, [l1, l3, l4]},

l3 ! {peers, [l1, l2, l4]},

l4 ! {peers, [l1, l2, l3]},

register(w1, worker:start("John", l1, Sleep, Work)), register(w2, worker:start("Ringo", l2, Sleep, Work)), register(w3, worker:start("Paul", l3, Sleep, Work)), register(w4, worker:start("George", l4, Sleep, Work)), ok.

stop() ->

w1 ! stop, w2 ! stop, w3 ! stop, w4 ! stop.

**Experiments**. i) Make tests with different Sleep and Work parameters to analyze how this lock implementation responds to different contention degrees. ii) Split the muty module and make the needed adaptations to enable each worker-lock pair to run in different machines (that is, *john* and *l1* should run in a machine, *ringo* and *l2* in another, and so on). Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

**Open Questions**. What is the behavior of the lock when you increase the risk of a conflict?

# Resolving deadlock

The problem with the first solution can be handled if we give each lock instance a unique identifier 1, 2, 3 and 4. The identifier will give a priority to the lock instance. A lock instance in the waiting state will send an ok

message to a requesting lock instance if the requesting lock instance has a higher priority (1 having the highest priority).

Implement this solution in a module called lock2, and show that it works even if we have high contention. There is a situation that you have to be careful with (i.e. a process wants to access the lock and it has already acknowledged another process with lower priority that it is still gathering ok messages). If you do not handle correctly this situation, you run the danger of having two processes in the critical section at the same time.

**Experiments**. Repeat the previous tests to compare the behavior of this lock with respect to the previous one.

**Open Questions**. i) Justify how your code guarantees that only one process is in the critical section at any time. ii) What is the main drawback of lock2 implementation?

# Lamport time

One improvement is to let locks be taken with priority given in time order. The only problem is that we do not (assuming we are running over an asynchronous network) have access to synchronized clocks. The solution is to use logical clocks such as Lamport clocks.

To implement this you must **add a clock variable to the lock in- stance**, which keeps track of the instance logical time. The value is initial- ized to zero and is increased every time the lock instance requests access to the critical section (when it sends the request messages to the other instances). In addition, it is updated when the instance receives a request message from another lock instance to the greater of the own clock and the timestamp received in the message. Thus, the clock keeps track of the highest request we have seen so far. Note that we do not need to add the Lamport timestamp to all the messages in the system but only to the request messages.

When a lock instance is in the waiting state and receives a request it must determine if this request was sent before or after it sent its own request message. To do this, **each request will have an associated timestamp**. Request timestamps have to be compared to determine which was raised first. If this cannot be determined (timestamps are equal), the lock instance identifier is used to resolve the order (as in lock2). Implement the solution in a module called lock3.

**Experiments**. Repeat the previous tests to compare this version with the former ones.

**Open Questions**. Note that the workers are not involved in the Lam- port clock. According to this, would it be possible that a worker is given access to a critical section prior to another worker that issued a request to its lock instance before (assuming real-time order)?

# Appendix

Here is the gui. The worker will start the gui and send messages when it is waiting for a lock (the window of the gui will be YELLOW), when it takes the lock (the gui will be RED), and when the lock is released (or attempt to take the lock is aborted) (the gui will be BLUE).

### GUI Module Overview

* **Module and Function Declaration**: The module is named gui, and it exports the start/1 function.
* **wxWidgets Library**: The code includes the wxWidgets library for Erlang (wx), which provides functionalities for creating and managing GUI elements.

### The start/1 Function

* **Purpose**: This function is responsible for spawning a new process to initialize the GUI.
* **Implementation**: It calls spawn/1 with a function that invokes init/1, passing the Name of the worker process.

### The init/1 Function

* **Setting Up the GUI**: Initializes the GUI with a window (frame) and sets its size.
* **wxServer**: Creates a new wxWidgets server.
* **wxFrame**: Initializes a frame (window) with the specified Name, size, and displays it.

### The loop/1 Function

* **Event Handling**: The GUI enters a loop, waiting for messages indicating the state of the worker process.
* **State Representation**:
  + **Waiting for Lock (YELLOW)**: On receiving a waiting message, the background color of the frame is set to yellow ({255, 255, 0}), representing that the worker is waiting for a lock.
  + **Lock Acquired (RED)**: On receiving a taken message, the color changes to red (?wxRED), indicating that the lock is acquired.
  + **Lock Released/Aborted (BLUE)**: On receiving a leave message, the color changes to blue (?wxBLUE), indicating the lock is released or the attempt to take the lock is aborted.
* **Termination**: On receiving a stop message, the process terminates.
* **Error Handling**: If an unexpected message is received, it prints an error message and continues the loop.

### 

-module(gui).

-export([start/1]).

-include\_lib("wx/include/wx.hrl").

start(Name) ->

spawn(fun() -> init(Name) end).

init(Name) ->

Width = 200,

Height = 200,

Server = wx:new(), %Server will be the parent for the Frame Frame = wxFrame:new(Server, -1, Name, [{size,{Width, Height}}]), wxFrame:show(Frame),

loop(Frame).

loop(Frame)->

receive

waiting ->

%wxYELLOW doesn’t exist in "wx/include/wx.hrl" wxFrame:setBackgroundColour(Frame, {255, 255, 0}), wxFrame:refresh(Frame),

loop(Frame); taken ->

wxFrame:setBackgroundColour(Frame, ?wxRED), wxFrame:refresh(Frame),

loop(Frame); leave ->

wxFrame:setBackgroundColour(Frame, ?wxBLUE), wxFrame:refresh(Frame),

loop(Frame); stop ->

ok;

Error ->

io:format("gui: strange message ~w ~n", [Error]), loop(Frame)

end.

## Totty: a total order multicast

**Jordi Garcia**

Adapted with permission from Jordi Guitart & Johan Montelius (KTH)

November 19, 2018

# Introduction

The task is to implement a total order multicast service using a distributed algorithm. The algorithm is the one used in the ISIS system and is based on requesting proposals from all nodes in a group.

# The architecture

We will have a set of workers that communicate with each other using multi- cast messages. Each worker will have access to a multicast process that will hide the complexity of the system. The multicast processes are connected to each other and will have to agree on an order on how to **deliver** the messages. There is a clear distinction between receiving a message, which is done by the multicast process, and delivering a message, which is when the worker sees the message. A multicast process will receive messages in un-specified order but only deliver the messages to its worker in a **total order** i.e. all workers in the system will deliver the messages in the same order.

## The worker

A worker is in our example a process that at random intervals wish to send a multicast message to the group. The worker will wait until it sees its own message before it sends another one to prevent an overflow of messages in the system.

The worker will be connected to a gui process that is simply a colored window. The window is initially black or in RGB talk *{*0, 0, 0*}*. This is also the initial state of the worker. Each message that is delivered to the worker is an integer N in some interval, say 1 to 20. A worker will change its state by adding N to the R value and rotate the values. If the state of the worker is *{*R, G, B*}* and the worker is delivered message N, the new state is *{*G, B, (R+N) rem 256*}*.

The color of a worker will thus change over time and the order of messages is of course important. The sequence 5, 12, 2 will of course not create the same color as the sequence 12, 5, 2. If all workers start with a black color and we fail to deliver the messages in the same order the colors of the workers will start to diverge.

We will experiment with different implementations and to prepare for the future we make the initialization a bit cumbersome. To start with, we provide some parameters to the worker to make experiments easier to manage. We can change the module of the multicast process and we can experiment with different values of sleep and jitter time. The sleep time is for up to how many milliseconds the workers should wait until the next message is sent and the jitter time is a parameter to the multicast process to simulate different network delays for the individual messages.

When started, the worker should register with a group manager and will be given the initial state and the process identifier of the other members in the group.

Since the purpose of the exercise is not to debug the worker, the code is given in ’Appendix A’. You will also find in ’Appendix B’ the code of a simple gui.

# Basic multicast

Initially, we will use a process that implements basic multicast. We give a set of peer processes to this multicast process and when it is told to multicast a message the message is simply sent to the rest of peers, one by one.

To make the experiments more interesting we include a Jitter param- eter when we start the process. The process will set an asynchronous timer up to these many milliseconds before sending each message. This will allow messages to interleave and possibly cause problems to the workers.

### Module and Function Declarations

* **Module Name**: The module is named basic.
* **Exported Function**: The start/3 function is exported, which means it can be called from other modules.

### The start/3 Function

* **Purpose**: This function initializes the multicast process.
* **Implementation**: It spawns a new process that calls the init/3 function with Id, Master, and Jitter parameters.

### The init/3 Function

* **Random Seed Initialization**: Initializes the random number generator using now() to ensure different sequences for different invocations.
* **Setting Up Peers**: Waits to receive a {peers, Nodes} message, then calls the server/4 function with the list of peer nodes (excluding itself).

### The server/4 Function

* **Handling Messages**: The server function enters a receive loop to handle different types of messages:
  + **Sending a Message ({send, Msg})**: Calls multicast/3 to send the message to all nodes, then informs the Master about the delivery.
  + **Receiving a Multicast Message ({multicast, \_From, Msg})**: Informs the Master about the message delivery.
  + **Stop**: Stops the server process.

### The multicast/3 Function

* **Sending Messages to Nodes**: The multicast function sends a given message to all nodes in the list. It has two variations:
  + **Without Jitter**: Sends the message directly to each node.
  + **With Jitter**: Introduces a random delay (up to Jitter milliseconds) before sending the message to each node using timer:send\_after/3.

### Jitter Implementation

* **Purpose**: Jitter introduces a delay in message sending, simulating a more realistic network environment where messages may arrive out of order.
* **Implementation**: For each node, a random delay T is generated, and the message is sent after this delay.

-module(basic).

-export([start/3]).

start(Id, Master, Jitter) ->

spawn(fun() -> init(Id, Master, Jitter) end).

init(Id, Master, Jitter) ->

{A1,A2,A3} = now(),

random:seed(A1, A2, A3), receive

{peers, Nodes} ->

server(Id, Master, lists:delete(self(), Nodes), Jitter)

end.

server(Id, Master, Nodes, Jitter) -> receive

{send, Msg} ->

multicast(Msg, Nodes, Jitter), Master ! {deliver, Msg},

server(Id, Master, Nodes, Jitter);

{multicast, \_From, Msg} -> Master ! {deliver, Msg},

server(Id, Master, Nodes, Jitter); stop ->

ok

end.

multicast(Msg, Nodes, 0) -> Self = self(), lists:foreach(fun(Node) ->

Node ! {multicast, Self, Msg} end,

Nodes); multicast(Msg, Nodes, Jitter) ->

Self = self(), lists:foreach(fun(Node) ->

T = random:uniform(Jitter),

timer:send\_after(T, Node, {multicast, Self, Msg}) end,

Nodes).

**Experiments**. Set up the basic multicast system, and use the following test program to experiment with different values for Sleep and Jitter. Does it keep workers synchronized? Justify why. Note that we are using the name of the module (i.e. basic) as a parameter to the start procedure. We will easily be able to test different multicast implementations. Sleep stands for the average number of milliseconds the workers should wait until the next message is sent. Jitter stands for the average number of milliseconds of network delay.

-module(toty).

-export([start/3, stop/0]).

start(Module, Sleep, Jitter) ->

register(toty, spawn(fun() -> init(Module, Sleep, Jitter) end)).

stop() ->

toty ! stop.

init(Module, Sleep, Jitter) -> Ctrl = self(),

worker:start("P1", Ctrl, Module, 1, Sleep, Jitter), worker:start("P2", Ctrl, Module, 2, Sleep, Jitter),

worker:start("P3", Ctrl, Module, 3, Sleep, Jitter), worker:start("P4", Ctrl, Module, 4, Sleep, Jitter), collect(4, [], []).

collect(N, Workers, Peers) -> if

N == 0 ->

Color = {0,0,0},

lists:foreach( fun(W) ->

W ! {state, Color, Peers} end,

Workers), run(Workers);

true ->

receive

{join, W, P} ->

collect(N-1, [W|Workers], [P|Peers])

end

end.

run(Workers) -> receive

stop ->

lists:foreach( fun(W) ->

W ! stop end, Workers)

end.

# Total order multicast

To avoid messages to be delivered out of order, we will implement a total order multicaster. We will here go through the code but you will have to do some programing yourself. We are leaving ’...’ at places in the code where you have to fill in the right values.

-module(total).

-export([start/3]).

start(Id, Master, Jitter) ->

spawn(fun() -> init(Id, Master, Jitter) end).

init(Id, Master, Jitter) ->

{A1,A2,A3} = now(),

random:seed(A1, A2, A3), receive

{peers, Nodes} ->

server(Master, seq:new(Id), seq:new(Id), Nodes, [], [], Jitter)

end.

The server procedure has the following state:

* **Master**: the process to which messages are delivered
* **MaxPrp**: the largest sequence number proposed so far
* **MaxAgr**: the largest agreed sequence number seen so far
* **Nodes**: all peers in the network
* **Cast**: a set of references to messages that have been sent out but no final sequence number have yet been assigned
* **Queue**: messages that have been received but not yet delivered
* **Jitter**: this parameter induces some network delay

The sequence numbers are represented by a tuple *{*N, Id*}*, where N is an integer that is incremented every time we make a proposal and Id is our process identifier. In ’Appendix C’, you will find code to create, modify, and compare sequence numbers.

The Cast set is represented as a list of tuples *{*Ref, L, Sofar*}*, where L is the number of proposals that we are still waiting for and Sofar the highest proposal received so far.

The Queue is an ordered list of entries representing messages that we have received but for which no agreed value exist. The list is ordered based in the proposed or agreed sequence number. The proposed entries are en- tries where we have proposed a sequence number. If we have entries with agreed sequence numbers at the front of the queue these can be removed and delivered to the worker.

### Overview of Total Order Multicast

* **Total Order Guarantee**: The system ensures that all messages are delivered in the same sequence across all nodes in the network.
* **Sequence Numbers**: Messages are ordered using sequence numbers, represented as tuples {N, Id}, where N is a unique integer and Id is the process identifier.
* **Cast Set**: Tracks messages that have been sent but not yet assigned a final sequence number.
* **Message Queue**: Stores received messages in order based on their sequence numbers, waiting for an agreed sequence number to deliver them.

### Module and Function Declarations

* **Module Name**: The module is named total.
* **Exported Function**: The start/3 function is exported.

### The start/3 Function

* **Purpose**: Initializes the total order multicast process.
* **Implementation**: Spawns a new process that calls the init/3 function with Id, Master, and Jitter.

### The init/3 Function

* **Initialization**: Sets up the random seed and waits to receive a {peers, Nodes} message.
* **Server Function**: Calls server/7 with the master process, initialized sequence numbers (MaxPrp and MaxAgr), the list of nodes, and an empty Cast and Queue.

### The server/7 Function

* **Parameters**:
  + Master: The process to which messages are delivered.
  + MaxPrp: The largest sequence number proposed so far.
  + MaxAgr: The largest agreed sequence number seen so far.
  + Nodes: All peers in the network.
  + Cast: A set of message references awaiting final sequence numbers.
  + Queue: Messages received but not yet delivered.
  + Jitter: Network delay inducement parameter.

### Cast Set and Queue

* **Cast Set Structure**: Represented as a list of tuples {Ref, L, Sofar}, where L is the number of proposals awaited, and Sofar is the highest proposal received.
* **Queue Structure**: An ordered list of messages based on their sequence numbers. The list is sorted such that messages with agreed sequence numbers at the front can be delivered.

### Handling Messages

* The server/7 function, which is not fully detailed in the snippet, would be responsible for handling the reception of messages, managing the Cast set and Queue, and ensuring the delivery of messages in the correct order based on sequence numbers.

## Sending of a message

A send message is a directive to multicast a message. We first have to agree in which order to deliver the message and therefore send a request for proposals to all peers (using the function request/4).

The request should be sent to all nodes with a unique reference. This reference is also added to the casted set with information on how many nodes have to report back (using the function cast

server(Master, MaxPrp, MaxAgr, Nodes, Cast, Queue, Jitter) -> receive

{send, Msg} ->

Ref = make\_ref(),

request(... , ... , ... , ...),

NewCast = cast(... , ... , ...),

server(... , ... , ... , ... , ... , ... , ...);

Note that we are also sending a request to ourselves. We will handle our own proposal the same way as proposals from everyone else. This might look strange but it makes the code much easier.

### Handling a Send Message

* **Send Directive**: When the server process receives a {send, Msg} directive, it initiates the process of multicasting the message.

### Generating a Unique Reference

* **make\_ref()**: A unique reference (Ref) is created for the message. This reference is used to track responses from other nodes regarding the sequence number proposal.

### Requesting Proposals

* **request/4 Function**: This function is called to send a proposal request to all nodes in the multicast group. It likely includes the unique reference, the message, and other necessary information.
* **Including Self in Proposals**: The server sends a request to itself as well. Handling its own proposal in the same way as others simplifies the code.

### Updating the Cast Set

* **cast/3 Function**: This function updates the Cast set with the new message reference. It tracks how many nodes need to report back with their proposals.
* **Structure of Cast Entry**: Each entry in the Cast set is a tuple that likely includes the reference, the count of nodes that need to reply, and possibly the highest proposal received so far.

### Recursion in server/7

* **Updating Server State**: The server/7 function recurses with the updated state, including the new Cast and potentially modified MaxPrp, MaxAgr, and Queue.
* **Parameters**: The parameters for the recursive call to server/7 would include the Master, updated MaxPrp, MaxAgr, Nodes, NewCast, Queue, and Jitter.

### Code Implementation

The ellipses (...) in the request and server function calls need to be filled with appropriate arguments. For request, this would include the nodes to send the request to, the message, the reference, and any other necessary data. For server, the updated parameters reflecting the new state after handling the send directive are required.

## Receiving a request

When the process receives a request message it should reply with a new proposed sequence number. The sequence number to be proposed is calcu- lated by incrementing by one the maximum of MaxAgr and MaxPrp. What ever happens we must not propose a lower sequence number than the ones we have proposed already. It should also queue the message using the proposed sequence number as key. This is handled by the function insert/4.

{request, From, Ref, Msg} -> NewMaxPrp = ... ,

From ! {proposal, ... , ...},

NewQueue = insert(... , ... , ... , ...), server(... , ... , ... , ... , ... , ... , ...);

## Receiving a proposal

A proposal message is sent as a reply to a request that we have sent earlier. The proposal contains the message reference and the proposed sequence number.

If the proposal is the last proposal that we are waiting for, we have also found an agreed sequence number. We implement this by calling the function proposal/3 that will update the set and either return *{*agreed, MaxSeq, NewCast*}* if an agreement was found or simply the updated Cast list.

{proposal, Ref, Proposal} ->

case proposal(... , ... , ...) of

{agreed, MaxSeq, NewCast} -> agree(... , ... , ...),

server(... , ... , ... , ... , ... , ... , ...);

NewCast ->

server(... , ... , ... , ... , ... , ... , ...)

end;

### Handling a Proposal Message

* **Proposal Directive**: When the server process receives a {proposal, Ref, Proposal} message, it indicates that a node is sending back a proposed sequence number for a message identified by Ref.

### Updating the Cast Set

* **proposal/3 Function**: This function updates the Cast set with the new proposal information. It checks whether this proposal is the last one needed to reach an agreement on the sequence number.
  + **Possible Return Values**:
    - {agreed, MaxSeq, NewCast}: An agreement on the sequence number has been reached.
    - NewCast: The updated Cast set if an agreement has not yet been reached.

### Handling Agreement

* **Reaching Agreement**: If an agreement is reached ({agreed, MaxSeq, NewCast}), the server needs to notify all nodes in the network of the agreed sequence number.
* **agree/3 Procedure**: This function likely handles the process of sending the agreed sequence number to all nodes.

### Recursion in server/7

* **With Agreement**: If an agreement is reached, the server calls agree/3 and then recurses with the updated state.
* **Without Agreement**: If no agreement is reached, the server recurses with the updated Cast set but without calling agree/3.

### Code Implementation

The ellipses (...) in the proposal, agree, and server function calls need to be filled with appropriate arguments. For proposal, this would include the received proposal, the reference, and the current Cast set. For agree and server, the updated parameters reflecting the new state after handling the proposal are required.

If we have an agreement this should be sent to all nodes in the network.

This is handled by the agree/3 procedure.

## Agree at last

An agreed message contains the agreed sequence number of a particular message. The message that is in the queue must be updated and possibly moved back in the queue (the agreed number could be higher than the proposed number). This is handled by the function update/3.

### Handling an Agreed Message

* **Agreed Message**: When an {agreed, Ref, Seq} message is received, it indicates that a final agreement on the sequence number (Seq) for a message (Ref) has been reached.
* **Updating the Queue**: The update/3 function is called to update the message in the queue with the agreed sequence number. This might involve reordering the message in the queue if the agreed sequence number is higher than the proposed number.
* **Delivering Messages**: The agreed/1 function checks the queue's front for messages with an agreed sequence number and prepares them for delivery.
* **Updating MaxAgr**: The MaxAgr (the largest agreed sequence number seen so far) is updated to the maximum of the current MaxAgr and the newly agreed sequence number Seq.

### Delivering Messages

* **deliver/2 Procedure**: This function handles the actual delivery of messages to the Master process. It iterates over the list of messages that can be delivered and sends them to Master.

### Sending Request and Agreed Messages

* **request/4 Function**: This function sends a request message to all nodes. It handles both immediate sending and sending with a jitter (delay).
* **agree/3 Function**: This function broadcasts an agreed message to all nodes, indicating the final sequence number for a message.

### Supporting Functions

* **cast/3 Function**: Adds a new entry to the Cast set with the reference, the number of nodes, and an initial sequence number.
* **proposal/3 Function**: Updates the Cast set with a new proposal, handling both the case where an agreement is reached and where more proposals are awaited.
* **agreed/1 Function**: Removes messages from the queue front that have an agreed sequence number and are ready for delivery.
* **update/3 Function**: Updates the queue with the agreed sequence number for a specific message.
* **insert/4 and queue/5 Functions**: Handle the insertion of new messages into the queue, maintaining the order based on sequence numbers.

{agreed, Ref, Seq} ->

Updated = update(... , ... , ...),

{Agreed, NewQueue} = agreed(...), deliver(... , ...),

NewMaxAgr = ... ,

server(... , ... , ... , ... , ... , ... , ...);

stop ->

ok

end.

If the first message in the queue now has an agreed sequence number it could be delivered. The function agreed/2 will remove the messages that can be delivered and return them in a list. These messages can then be delivered using the deliver/2 procedure. The largest agreed sequence number it has seen so far must be updated as the maximum of MaxAgr and Seq.

The remaining code that support the previous functionality is as follows:

%% Sending a request message to all nodes request(Ref, Msg, Nodes, 0) ->

Self = self(), lists:foreach(fun(Node) ->

%% TODO: ADD SOME CODE

end, Nodes);

request(Ref, Msg, Nodes, Jitter) -> Self = self(), lists:foreach(fun(Node) ->

T = random:uniform(Jitter),

timer:send\_after(T, Node, ... ) %% TODO: COMPLETE end,

Nodes).

%% Sending an agreed message to all nodes agree(Ref, Seq, Nodes)->

lists:foreach(fun(Pid)->

%% TODO: ADD SOME CODE

end, Nodes).

%% Delivering messages to the master deliver(Master, Messages) ->

lists:foreach(fun(Msg)->

Master ! {deliver, Msg} end,

Messages).

%% Adding a new entry to the set of casted messages cast(Ref, Nodes, Cast) ->

L = length(Nodes),

[{Ref, L, seq:null()}|Cast].

%% Update the set of casted messages proposal(Ref, Proposal, [{Ref, 1, Sofar}|Rest])->

{agreed, seq:max(Proposal, Sofar), Rest}; proposal(Ref, Proposal, [{Ref, N, Sofar}|Rest])->

[{Ref, N-1, seq:max(Proposal, Sofar)}|Rest]; proposal(Ref, Proposal, [Entry|Rest])->

case proposal(Ref, Proposal, Rest) of

{agreed, Agreed, Rst} ->

{agreed, Agreed, [Entry|Rst]}; Updated ->

[Entry|Updated]

end.

%% Remove all messages in the front of the queue that have been agreed agreed([{\_Ref, Msg, agrd, \_Agr}|Queue]) ->

{Agreed, Rest} = agreed(Queue),

{[Msg|Agreed], Rest}; agreed(Queue) ->

{[], Queue}.

%% Update the queue with an agreed sequence number update(Ref, Agreed, [{Ref, Msg, propsd, \_}|Rest])->

queue(Ref, Msg, agrd, Agreed, Rest); update(Ref, Agreed, [Entry|Rest])->

[Entry|update(Ref, Agreed, Rest)].

%% Insert a new message into the queue

insert(Ref, Msg, Proposal, Queue) -> queue(Ref, Msg, propsd, Proposal, Queue).

%% Queue a new entry

queue(Ref, Msg, State, Proposal, []) -> [{Ref, Msg, State, Proposal}];

queue(Ref, Msg, State, Proposal, Queue) -> [Entry|Rest] = Queue,

{\_, \_, \_, Next} = Entry,

case seq:lessthan(Proposal, Next) of true ->

[{Ref, Msg, State, Proposal}|Queue]; false ->

[Entry|queue(Ref, Msg, State, Proposal, Rest)]

end.

**Experiments**. i) Set up the total order multicast system, and repeat the previous tests. Does it keep workers synchronized? ii) We have a lot of messages in the system. Derive a theoretical quantification of the number of messages needed to deliver a multicast message as a function of the number of workers and check experimentally that your formulation is correct. iii) Compare with the basic multicast implementation regarding the number of messages needed.

**Appendix A: *worker.erl***

### Module and Function Declarations

* **Module Name**: The module is named worker.
* **Exported Function**: The start/6 function is exported, indicating it can be called from other modules.

### The start/6 Function

* **Purpose**: Initializes the worker process.
* **Implementation**: Spawns a new process that calls the init/6 function with the parameters Name, Grp, Module, Id, Sleep, and Jitter.

### The init/6 Function

* **Initialization**: Sets up the random number generator and starts the GUI process.
* **Starting the Cast Process**: Calls the start function of the specified Module with Id, self(), and Jitter to start a multicast process (referred to as Cast).
* **Joining the Group**: Sends a {join, self(), Cast} message to the group (Grp).
* **Setting Initial State**: Waits to receive {state, Color, Peers} message, then sends the peers to Cast and sets the initial color on the GUI.
* **Worker Process**: Calls cast\_change to initiate message sending and enters the main worker loop.

### The worker/5 Function

* **Main Loop**: Handles incoming messages:
  + **{deliver, {From, N}}**: Changes the color based on the message, updates the GUI, and, if the message is from itself, calls cast\_change to send a new message.
  + **stop**: Terminates the process.
  + **Error**: Handles unexpected messages and continues the loop.

### Helper Functions

* **change\_color/2**: Changes the color based on the received message.
* **cast\_change/3**: Sends a new message after a delay (Sleep), creating a new message with a random component.

-module(worker).

-export([start/6]).

-define(change, 20).

start(Name, Grp, Module, Id, Sleep, Jitter) ->

spawn(fun() -> init(Name, Grp, Module, Id, Sleep, Jitter) end).

init(Name, Grp, Module, Id, Sleep, Jitter) ->

{A1,A2,A3} = now(),

random:seed(A1, A2, A3), Gui = gui:start(Name),

Cast = apply(Module, start, [Id, self(), Jitter]), Grp ! {join, self(), Cast},

receive

{state, Color, Peers} -> Cast ! {peers, Peers}, Gui ! {color, Color},

cast\_change(Id, Cast, Sleep), worker(Id, Cast, Color, Gui, Sleep), Cast ! stop,

Gui ! stop

end.

worker(Id, Cast, Color, Gui, Sleep) -> receive

{deliver, {From, N}} ->

Color2 = change\_color(N, Color), Gui ! {color, Color2},

if

From == Id ->

cast\_change(Id, Cast, Sleep); true ->

ok

end,

worker(Id, Cast, Color2, Gui, Sleep); stop ->

ok;

Error ->

io:format("strange message: ~w~n", [Error]), worker(Id, Cast, Color, Gui, Sleep)

end.

change\_color(N, {R,G,B}) ->

{G, B, ((R+N) rem 256)}.

cast\_change(Id, Cast, Sleep) ->

Msg = {Id, random:uniform(?change)}, timer:send\_after(Sleep, Cast, {send, Msg}).

**Appendix B: *gui.erl***

### Module and Function Declarations

* **Module Name**: The module is named gui.
* **Exported Function**: The start/1 function is exported for external invocation.

### The start/1 Function

* **Purpose**: Initializes the GUI process.
* **Implementation**: Spawns a new process using spawn\_link that calls the init/1 function with the Name parameter. spawn\_link is used to create a new process linked to the spawning process.

### The init/1 Function

* **Creating the Frame**: Calls make\_frame/1 to create a GUI frame with the specified Name.

### The make\_frame/1 Function

* **Frame Creation**: Creates a new wxWidgets server and frame.
  + **Server**: Acts as the parent for the frame.
  + **Frame**: A new window with the title Name, size defined by ?width and ?height, and initially set to a black background.
  + **Show Frame**: Displays the frame on the screen.
  + **Event Monitoring**: Connects to a close\_window event to handle the window closing action.

### The loop/1 Function

* **Event Handling Loop**: Continuously receives and handles messages:
  + **Window Close Event**: Destroys the frame and terminates on a close window event.
  + **Color Change**: Updates the frame's background color on receiving a {color, Color} message.
  + **Stop**: Terminates the process on a stop message.
  + **Error Handling**: Prints an error message for unexpected messages and continues the loop.

### The color/2 Function

* **Color Update**: Changes the background color of the frame and refreshes the frame to reflect the new color.

-module(gui).

-export([start/1]).

-define(width, 200).

-define(height, 200).

-include\_lib("wx/include/wx.hrl").

start(Name) ->

spawn\_link(fun() -> init(Name) end).

init(Name) ->

Frame = make\_frame(Name), loop(Frame).

make\_frame(Name) -> %Name is the window title

Server = wx:new(), %Server will be the parent for the Frame

Frame = wxFrame:new(Server, -1, Name, [{size,{?width, ?height}}]), wxFrame:setBackgroundColour(Frame, ?wxBLACK),

wxFrame:show(Frame),

%monitor closing window event wxFrame:connect(Frame, close\_window), Frame.

loop(Frame)->

receive

%check if the window was closed by the user #wx{event=#wxClose{}} ->

wxWindow:destroy(Frame), ok;

{color, Color} -> color(Frame, Color), loop(Frame);

stop ->

ok;

Error ->

io:format("gui: strange message ~w ~n", [Error]), loop(Frame)

end.

color(Frame, Color) -> wxFrame:setBackgroundColour(Frame, Color), wxFrame:refresh(Frame).

**Appendix C: *seq.erl***

### Module and Function Declarations

* **Module Name**: The module is named seq.
* **Exported Functions**: The module exports several functions - null/0, new/1, increment/1, max/2, and lessthan/2.

### Functions Description

* **null/0 Function**:
  + **Purpose**: Generates a null or initial sequence number.
  + **Implementation**: Returns {0,0}, representing the lowest possible sequence number.
* **new/1 Function**:
  + **Purpose**: Creates a new sequence number for a given identifier.
  + **Implementation**: Takes an Id and returns a sequence number initialized to {0, Id}.
* **increment/1 Function**:
  + **Purpose**: Increments a given sequence number.
  + **Implementation**: Increments the first element of the tuple, leaving the Id unchanged.
* **max/2 Function**:
  + **Purpose**: Determines the maximum of two sequence numbers.
  + **Implementation**: Compares two sequence numbers using lessthan/2 and returns the greater one.
* **lessthan/2 Function**:
  + **Purpose**: Compares two sequence numbers.
  + **Implementation**: Compares two sequence number tuples {Pn, Pi} and {Nn, Ni}. A sequence number is considered less than another if either its first element is smaller, or if the first elements are equal and the second element (identifier) is smaller.

### Usage in Distributed Systems

* Sequence numbers are often used in distributed systems to maintain the order of events or messages. For instance, in a total order multicast system, sequence numbers ensure that all messages are delivered in a globally agreed order.
* The seq module's functionality allows for the creation, comparison, and incrementation of sequence numbers, essential for managing the order of operations in distributed systems.

-module(seq).

-export([null/0, new/1, increment/1, max/2, lessthan/2]).

%%% Functions to handle the sequence numbers. null() ->

{0,0}.

new(Id) ->

{0, Id}.

increment({Pn, Pi}) ->

{Pn+1, Pi}.

max(Proposal, Sofar) ->

case lessthan(Proposal, Sofar) of true ->

Sofar; false ->

Proposal

end.

lessthan({Pn, Pi}, {Nn, Ni}) ->

(Pn < Nn) or ((Pn == Nn) and (Pi < Ni)).

## Groupy: a group membership service

**Jordi Garcia**

Adapted with permission from Jordi Guitart & Johan Montelius (KTH) February 5, 2016

# Introduction

This is an assignment where you will implement a group membership service that provides atomic multicast. The aim is to have several application layer processes with **a coordinated state** i.e. they should all perform the same sequence of state changes. A node that wishes to perform a state change must first multicast the change to the group so that all nodes can execute it. Since the multicast layer provides total order, all nodes will be synchronized. The problem in this assignment is that all nodes need to be synchronized even though nodes may come and go (crash). As you will see it is not as

trivial as one might first think.

# The architecture

We will implement a group membership service that provides atomic mul- ticast in view synchrony. The architecture of this service consists of a set of nodes where one is the elected leader. All nodes that wish to multicast a message will send the message to the leader and the leader will do a basic multicast to all members of the group. If the leader dies a new leader is elected.

A new node that wishes to enter the group will contact any node in the group and request to join the group. The leader will determine when the node is to be included and will deliver **a new group view** to the group.

The application layer processes will use this group membership service to synchronize their states. As commented, this service is implemented by means of a set of group processes. Each application layer process will have its own group process that it communicates with. The application layer process will send the messages to be multicasted to the group process and will receive all multicasted messages from it. The group process will tell the application level process to deliver a message only when the rest of processes have also delivered it. The application layer process must also be prepared to decide if a new node should be allowed to enter the group and also decide the initial state of this node.

Note that we will not deliver any group views to the application layer process. We could adapt the system so that it reports any view changes but for the application that we are targeting this is not needed. We will keep it

as simple as possible and then discuss extensions and how much they would cost.

## View synchrony

Each node in the group should be able to multicast messages to the members of the group. The communication is divided into views and messages will be said to be delivered in a view. For all messages in a view we will guarantee the following:

* + - in FIFO order: in the order that they were sent by a node
    - in total order: all nodes see the same sequence
    - reliably: if a correct node delivers a message, all correct nodes deliver the message

The last statement seems to be a bit weak, what do we mean by a correct node? A node will fail only by crashing and will then never be heard from again. A correct node is a node that does not fail during a view (i.e. it survives to install the next view).

It will not be guaranteed that sent messages are delivered, we will use asynchronous sending without acknowledgment and if we have a failing leader a sent message might disappear.

## The leader

A node will either play the role of a leader (let’s hope there is only one) or a slave. The slaves will forward messages to the leader and the leader will tag each message with a sequence number and multicast it to all nodes. The leader can also accept a message directly from its own master (i.e. the application layer process). The application layer process is unaware of whether its group process is acting as a leader or a slave.

## A slave

A slave will receive messages from its application layer process and forward them to the leader. It will also receive messages from the leader and forward them to the application layer process. If nodes would not fail this would be the easiest job in the world but since we must be able to act if the leader dies we need to do some book keeping.

In our first version of the implementation we will not deal with failures but only with adding new nodes to the system. This is complicated enough to start with.

## The election

The election procedure is very simple. All slaves have the same list of peers and they all elect the first node in the list as the leader. A slave that detects that it is the first node will of course adopt the role as leader.

## The application layer process

An application layer process will create a group process and contact any other application layer process it knows of. It will request to join the group providing the process identifier of its group process, which will then wait for a view delivery, containing the peer processes in the group.

There is no guarantee that the request is delivered to the leader, or rather the leader could be dead and we have not detected this yet. The requesting application layer process is however not told about this so we cannot do anything but wait and hope for the best. We will use a timeout and if we have not been invited in a second we might as well abort the attempt.

Once added to the group, the application layer process has the problem of obtaining the correct state (the color). It does this by using the atomic multicast system in a clever way. It sends a request to obtain the state and waits for this message to be delivered to itself. It now knows that the other processes have seen this message and will respond by sending the state.

The state message might however not be the first message that is deliv- ered. We might have other state changes in the pipeline. Once the state is received these state changes must of course be applied to the state. The im- plementation uses the implicit deferral of Erlang and simply lets any state change messages remain in the message queue and chooses to handle the state message first before the state change messages.

# The first implementation

Our first version, called gms1, will only handle starting of a single node and adding more nodes. Failures will not be handled so some of the states that we need to keep track of are not described. We will then extend this implementation to handle failures later on.

The group process will when started be a slave but might in the future become a leader. The first process that is started will however become a leader directly.

## The leader

The leader keeps the following state:

* + - **Name**: a unique name of the node, only used for debugging
    - **Master**: the process identifier of the application layer process
    - **Slaves**: an ordered list of the process identifiers of all slaves in the group

The list of peers is ordered based on when they were admitted to the group. We will use this order in the election procedure.

The leader should be able to handle the following messages:

* + - *{*mcast, Msg*}*: a message either from its own master or from a peer node. A message *{*msg, Msg*}* is multicasted to all peers and a message

*{*deliver, Msg*}* is sent to the application layer process (i.e. Master). We use a function bcast/3 that will send a message to each of the processes in a list.

* + - *{*join, Peer*}*: a message, from a peer or the master, that is a request from a node to join the group. A message *{*view, Leader, Slaves*}* containing the new set of slaves is multicasted to all peers.

leader(Name, Master, Slaves) -> receive

{mcast, Msg} ->

bcast(Name, ..., ...), %% TODO: COMPLETE

%% TODO: ADD SOME CODE

leader(Name, Master, Slaves);

{join, Peer} ->

NewSlaves = lists:append(Slaves, [Peer]), bcast(Name, ..., ...), %% TODO: COMPLETE

leader(Name, Master, ...); %% TODO: COMPLETE stop ->

ok;

Error ->

io:format("leader ~s: strange message ~w~n", [Name, Error])

end.

bcast(\_, Msg, Nodes) ->

lists:foreach(fun(Node) -> Node ! Msg end, Nodes).

### State Information

* **Name**: A unique name of the leader node (used for debugging).
* **Master**: The process identifier of the application layer process.
* **Slaves**: An ordered list of process identifiers of all slaves in the group. The order reflects when they were admitted to the group.

### Handled Messages

1. **{mcast, Msg}**: This message can be from either the leader's own master or from a peer node. It is handled as follows:
   * {msg, Msg} is multicasted to all peers.
   * {deliver, Msg} is sent to the application layer process (Master).
2. The code for multicasting to all peers and delivering to the Master is incomplete and marked with "TODO." You would need to complete this part of the code to implement the desired behavior.
3. **{join, Peer}**: When a node (Peer) requests to join the group, this message is received. The leader appends the new Peer to the list of Slaves and then multicasts the updated list to all peers, informing them of the new view. Again, the code for multicasting and updating the list of Slaves is marked with "TODO."
4. **stop**: This message signals the leader to terminate.
5. **Error**: If any other unexpected message is received, it is logged as a "strange message."

### bcast/3 Function

The bcast/3 function is used to multicast a given message (Msg) to a list of nodes (Nodes). It iterates over the list of nodes and sends the message to each node using the Node ! Msg operation.

Notice that we add the new node at the end of the list of slaves. This is important, we want the new node to be the last one to see the view message that we send out. More on this later when we look at failing nodes.

## A slave

A slave has an even simpler job, it will not make any complicated decisions. It is simply forwarding messages from its master to the leader and vice versa. The state of a slave is as follows:

Aquest trobo que esta ben explicat, res a afegir.

* + - **Name**: a unique name of the node, only used for debugging
    - **Master**: the process identifier of the application layer process
    - **Leader**: the process identifier of the leader
    - **Slaves**: an ordered list of the process identifiers of all slaves in the group

The messages from the master and the leader are the following:

* + - *{*mcast, Msg*}*: a request from its master to multicast a message, the message is forwarded to the leader.
    - *{*join, Peer*}*: a request from its master to allow a new node to join the group, the message is forwarded to the leader.
    - *{*msg, Msg*}*: a multicasted message from the leader. A message *{*deliver, Msg*}* is sent to the master.
    - *{*view, Leader, NewSlaves*}*: a multicasted view from the leader. For the slave only the new set of peers (NewSlaves) is of interest.

slave(Name, Master, Leader, Slaves) -> receive

{mcast, Msg} ->

%% TODO: ADD SOME CODE

slave(Name, Master, Leader, Slaves);

{join, Peer} ->

%% TODO: ADD SOME CODE

slave(Name, Master, Leader, Slaves);

{msg, Msg} ->

%% TODO: ADD SOME CODE

slave(Name, Master, Leader, Slaves);

{view, Leader, NewSlaves} ->

slave(Name, Master, Leader, ...); %% TODO: COMPLETE stop ->

ok;

Error ->

io:format("slave ~s: strange message ~w~n", [Name, Error])

end.

Since we will not yet deal with failure there is no transition between being a slave and becoming a leader. We will add this later but first let us have this thing up and running.

## Initialization

Initializing a process that is the first node in a group is simple. The only thing we need to do is to give it an empty list of peers. Since it is the only node in the group it will of course be the leader of the group.

### start/1 Function

This function is used to initialize a node that becomes the first node in a group (leader). Here's what it does:

1. start(Name) -> ...: The function is called with the Name parameter, which is a unique name for the node.
2. Self = self(), ...: It stores the process identifier of the current node (the one running this function) in the Self variable.
3. spawn\_link(fun()-> init(Name, Self) end): It spawns a new process that will execute the init/2 function, passing the Name and Self as arguments. The spawn\_link function creates a new process that is linked to the current process, so if one terminates, the other will also terminate.

### init/2 Function

This function is the entry point for a node that becomes the first node in a group (leader). Here's what it does:

1. init(Name, Master) -> ...: It takes the node's Name and the Master process identifier as arguments.
2. leader(Name, Master, []): It initializes the node as a leader by calling the leader/3 function with the provided Name, Master, and an empty list [] of peers (as it's the first node).

### start/2 Function

This function is used to initialize a node that wants to join an existing group as a slave. Here's what it does:

1. start(Name, Grp) -> ...: The function is called with the Name parameter (unique name for the node) and the Grp parameter, which is the process identifier of an existing group member.
2. Self = self(), ...: It stores the process identifier of the current node in the Self variable.
3. spawn\_link(fun()-> init(Name, Grp, Self) end): It spawns a new process that will execute the init/3 function, passing the Name, Grp, and Self as arguments.

### init/3 Function

This function is the entry point for a node that wants to join an existing group as a slave. Here's what it does:

1. init(Name, Grp, Master) -> ...: It takes the node's Name, the Grp (group leader) process identifier, and the Master process identifier (where the node will send a "joined" message) as arguments.
2. Grp ! {join, Self}: It sends a {join, Self} message to the group leader (Grp), indicating that the node wants to join the group.
3. receive ... end: It waits for a response, which should be a {view, Leader, Slaves} message from the leader. This message contains information about the group's leader and the list of slaves in the group.
4. slave(Name, Master, Leader, Slaves): It initializes the node as a slave by calling the slave/4 function with the provided Name, Master, Leader, and Slaves.

In summary, the start/1 and start/2 functions are used to initialize nodes as leaders or slaves in a group, depending on whether they are the first node in the group or joining an existing group. The init functions then set up the appropriate behavior for each type of node (leader or slave).

-module(gms1).

-export([start/1, start/2]).

start(Name) -> Self = self(),

spawn\_link(fun()-> init(Name, Self) end).

init(Name, Master) -> leader(Name, Master, []).

Starting a node that should join an existing group is only slightly more problematic. We need to send a *{*join, self()*}* message to a node in the group and wait for an invitation. The invitation is delivered as a view message containing everything we need to know. The initial state is of course as a slave.

start(Name, Grp) -> Self = self(),

spawn\_link(fun()-> init(Name, Grp, Self) end).

init(Name, Grp, Master) -> Self = self(),

Grp ! {join, Self}, receive

{view, Leader, Slaves} -> Master ! joined,

slave(Name, Master, Leader, Slaves)

end.

## The application layer process

To do some experiments we create an application layer process (so-called worker) that uses a gui, which is simply a colored window, to describe its state. The window is initially black or in RGB talk *{*0, 0, 0*}*. This is also the initial state of the worker. Each message that is delivered to the worker is an integer N in some interval, say 1 to 20. A worker will change its state by adding N to the R value and rotate the values. If the state of the worker is *{*R, G, B*}* and the worker is delivered message N, the new state is *{*G, B, (R+N) rem 256*}*.

The color of a worker will thus change over time and the order of messages is of course important. The sequence 5, 12, 2 will of course not create the same color as the sequence 12, 5, 2. If all workers start with a black color and we fail to deliver the messages in the same order the colors of the workers will start to diverge.

A worker and the gui are given at the appendices.

## Experiments

i) Do some experiments to see that you can create a group, add some peers and keep their state coordinated. You can use the following code to start and stop the whole system. Note that we are using the name of the module (i.e. gms1) as the parameter Module to the start procedure. All the workers but the first one need to know a member of the group in order to join. Sleep stands for up to how many milliseconds the workers should wait until the next message is sent. ii) Split the groupy module and make the needed adaptations to enable each worker to run in different machines. Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

-module(groupy).

-export([start/2, stop/0]).

start(Module, Sleep) ->

P = worker:start("P1", Module, Sleep), register(a, P),

register(b, worker:start("P2", Module, P, Sleep)), register(c, worker:start("P3", Module, P, Sleep)), register(d, worker:start("P4", Module, P, Sleep)), register(e, worker:start("P5", Module, P, Sleep)).

stop() ->

stop(a),

stop(b),

stop(c),

stop(d),

stop(e).

stop(Name) ->

case whereis(Name) of undefined ->

ok;

Pid ->

Pid ! stop

end.

# Handling failures

We will build up our fault tolerance gradually. First we will make sure that we detect crashes, then ensure that a new leader is elected and then make sure that the multicast service preserves the properties of the atomic multicast. Keep gms1 as a reference and call the adapted module gms2.

## Failure detectors

We will use the Erlang built-in support to detect and report that processes have crashed. A process can monitor another process and if that process dies a message will be received. For now, we will assume that the monitors are perfect i.e. they will eventually report the crash of a process and they will never report the death of a process that has not died.

We will also assume that the message that informs a process about a death of another process is the last message that it will see from it. The message will thus be received in FIFO order as any regular message.

The question we first need to answer is, who should monitor who? In our architecture we do not need to report new views when a slave dies and there is nothing to prevent a dead slave to be part of a view so we will keep things simple; the only node that will be monitored is the leader. A slave that detects that the leader has died will move to an election state.

This is implemented by first adding a call to erlang:monitor/2 in the initialization of the slave:

erlang:monitor(process, Leader)

and a new clause in the state of the slave:

{’DOWN’, \_Ref, process, Leader, \_Reason} -> election(Name, Master, Slaves);

Since the leader can crash it could be that a node that wants to join the group will never receive a reply. The message could be forwarded to a dead leader and the joining node is never informed of the fact that its request was lost. We simply add a timeout (for instance, 1000 ms) when waiting for an invitation to join the group in the initialization of the slave.

after ?timeout ->

Master ! {error, "no reply from leader"}

The election procedure will select the first process in the list of peers as the leader. If a process finds itself being the first node in the list, it will thus become the leader of the group and multicast a new view to the rest. If a process finds that it must remain as a slave, it will start monitoring the new elected leader.

election(Name, Master, Slaves) -> Self = self(),

case Slaves of [Self|Rest] ->

%% TODO: ADD SOME CODE

leader(..., ..., ...); %% TODO: COMPLETE

[NewLeader|Rest] ->

%% TODO: ADD SOME CODE

slave(..., ..., ..., ..., ...) %% TODO: COMPLETE

end.

One thing that we have to pay attention to is what we should do if, as a slave, we receive the view message from the new leader before we have noticed that the old leader is dead. In that case, we should accept the new view, enable the monitoring of the new leader and ignore trailing DOWN messages from the old leader. Note that **you need to extend the slave method with the reference of the process that we are monitoring**, which must be updated accordingly when we change the monitored process.

{view, NewLeader, NewSlaves} -> erlang:demonitor(Ref, [flush]),

NewRef = erlang:monitor(process, NewLeader), slave(Name, Master, NewLeader, NewSlaves, NewRef);

**Experiments**. Do some experiments to see if the peers can keep their state coordinated even if nodes crash.

## Missing messages

It seems to be too easy and unfortunately it is not. To show that current implementation is not working we can change the bcast/3 procedure and introduce a random crash. We define a constant arghh that defines the risk of crashing. A value of 100 means that a process will crash in average once in a hundred attempts. The definition of bcast/3 now looks like this:

bcast(Name, Msg, Nodes) -> lists:foreach(fun(Node) ->

Node ! Msg, crash(Name, Msg)

end, Nodes).

crash(Name, Msg) ->

case random:uniform(?arghh) of

?arghh ->

io:format("leader ~s CRASHED: msg ~w~n", [Name, Msg]), exit(no\_luck);

\_ -> ok

end.

We also add seeding of the random number generator when starting a process so that we will not have all processes crashing at the same time. The leader initialization is for example done as follows (the slave MUST be initialized in a similar manner).

init(Name, Master) ->

{A1,A2,A3} = now(),

random:seed(A1, A2, A3), leader(Name, Master, []).

**Experiments**. Repeat the experiments and see if you can have the state of the workers become out of synch.

**Open Questions**. Why is this happening?

## Reliable multicast

To remedy the problem we could replace the basic multicaster with a reliable multicaster. A process that would forward all messages before delivering them to the higher layer.

Assume that we keep a copy of the last message that we have seen from the leader. If we detect the death of the leader it could be that it died during the basic multicast procedure and that some nodes have not seen the message. We will now make an assumption that we will discuss later:

* + - Messages are reliably delivered and thus,
    - if the leader sends a message to A and then B, and B receives the message, then also A will receive the message.

The leader is sending messages to the peers in the order that they occur in the list of peers. If anyone receives a message then the first peer in the list receives the message. This means that only the next leader needs to resend the message.

This will of course introduce the possibility of doublets of messages being received. In order to detect this we will number all messages and only deliver new messages to the application layer process.

Let’s go through the changes that we need to make and create a new module gms3 that implements these changes.

* + - The slave procedure is extended with two arguments: N and Last. N is the expected sequence number of the next message and Last is a copy of the last message (**a regular multicast message or a view**) received from the leader.
    - The election procedure is extended with the same two arguments.
    - The leader procedure is extended with the argument N, the sequence number of the next message (regular multicast message or view) to be sent.

The multicast messages (msg and view) also change and will now contain the sequence number.

We must also add clauses to the slave to accept and ignore duplicate messages from the leader. If we do not remove these from the message queue they will add up and after a year generate a very hard to handle trouble report. When discarding messages we only want to discard messages that we have seen i.e. messages with a sequence number less than *N* . We can do this by using the when construction. For example:

{msg, I, \_} when I < N -> ...

The crucial part is then in the election procedure where **the elected leader will multicast the last received message to all peers** in the group. Hopefully this will be enough to keep slaves synchronized.

**Experiments**. i) Repeat the experiments to see if now the peers can keep their state coordinated even if nodes crash. ii) Try to keep a group rolling by adding more nodes as existing nodes die.

Assuming all tests went well we’re ready to ship the product. There is however one thing we need to mention and that is that our implementa- tion does not work!!! Well, it sort of works depending on what the Erlang environment guarantees and how strong our requirements are.

## What could possibly go wrong

The first thing we have to realize is what guarantees the Erlang system actually gives on message sending. The specifications only guarantee that messages are delivered in FIFO order, not that they actually do arrive. We have built our system relying on reliable delivery of messages, something that is not guaranteed.

The second reason why things will not work is that we rely on that the Erlang failure detector is perfect.

**Open Questions**. i) How would we have to change the implementation to handle the possibly lost messages? ii) How would this impact perfor- mance? iii) What would happen if we wrongly suspect the leader to have crashed?

**Appendix A: *worker.erl***

### Exported Functions:

1. start/3 Function:
   * This function initializes a worker process and sets its initial state.
   * Parameters:
     + Name: A unique name for the worker (used for debugging).
     + Module: The module to apply for starting the worker.
     + Sleep: The sleep interval in milliseconds (controls how often the worker changes its state).
   * Inside the function:
     + It spawns a new process that calls the init/3 function with the provided parameters.
     + The Cast variable is set to the result of applying the Module's start function with the worker's Name.
     + The Color variable is set to the initial color, defined as ?color, which is {0,0,0}.
     + It then calls the init\_cont/4 function to continue the initialization.
2. init/3 Function:
   * This function is the initial setup for a worker process without peer information.
   * Parameters:
     + Name: The worker's name.
     + Cast: The process identifier for message multicasting.
     + Color: The initial color.
     + Sleep: The sleep interval in milliseconds.
   * Inside the function:
     + It sets up random seed values and initializes the GUI using gui:start/2.
     + It calculates the Wait time, which determines how long the worker waits before sending its first state change.
     + It sends a message to cast\_change and starts the worker/5 function to handle subsequent messages.
3. start/4 Function:
   * This function initializes a worker process with peer information (joining an existing group).
   * Parameters:
     + Name: The worker's name.
     + Module: The module to apply for starting the worker.
     + Peer: The process identifier of the peer worker.
     + Sleep: The sleep interval in milliseconds.
   * Inside the function:
     + It spawns a new process that calls the init/4 function with the provided parameters.
     + The Cast variable is set to the result of applying the Module's start function with the worker's Name and Peer.
     + It receives a joined message, indicating that the worker has successfully joined the group and is ready to receive state transfer.
4. init/4 Function:
   * This function is the initial setup for a worker process with peer information (joining an existing group).
   * Parameters:
     + Name: The worker's name.
     + Module: The module to apply for starting the worker.
     + Peer: The process identifier of the peer worker.
     + Sleep: The sleep interval in milliseconds.
   * Inside the function:
     + It initializes the worker's state by calling state\_transfer/2 and awaits a color change.
     + If the Color is not stop, it continues with init\_cont/4, starting the worker process.
     + If the Color is stop, it stops the worker process.

### Worker Process Functions:

* The worker/5 function is the main process function for handling messages. It can receive various messages that control the worker's behavior:
  + change\_state: Changes the worker's color based on a random number.
  + state\_req: Requests the worker's state.
  + set\_state: Sets the worker's state.
  + join: Joins the worker to a group (peer information).
  + cast\_change: Initiates a color change event.
  + stop: Stops the worker process.
  + Any other message is treated as an error message.

### Color Manipulation:

* The change\_color/2 function calculates a new color based on a random number and the current color.

The code is designed to create worker processes that can change colors and communicate with a group or peers. The workers can join existing groups and transfer state information.

-module(worker).

-export([start/3, start/4]).

-define(change, 20).

-define(color, {0,0,0}).

start(Name, Module, Sleep) ->

spawn(fun() -> init(Name, Module, Sleep) end).

init(Name, Module, Sleep) ->

Cast = apply(Module, start, [Name]), Color = ?color,

init\_cont(Name, Cast, Color, Sleep).

start(Name, Module, Peer, Sleep) ->

spawn(fun() -> init(Name, Module, Peer, Sleep) end).

init(Name, Module, Peer, Sleep) ->

Cast = apply(Module, start, [Name, Peer]), receive

joined ->

Ref = make\_ref(),

Cast ! {mcast, {state\_req, Ref}}, Color = state\_transfer(Cast, Ref), if Color /= stop ->

init\_cont(Name, Cast, Color, Sleep), Cast ! stop;

true ->

Cast ! stop

end;

{error, Error} ->

io:format("worker ~s: error: ~s~n", [Name, Error]); stop ->

ok

end.

state\_transfer(Cast, Ref) -> receive

{deliver, {state\_req, Ref}} -> receive

{deliver, {set\_state, Ref, Color}} -> Color

end;

{join, Peer} ->

Cast ! {join, Peer}, state\_transfer(Cast, Ref);

stop ->

stop;

\_Ignore ->

state\_transfer(Cast, Ref)

end.

init\_cont(Name, Cast, Color, Sleep) ->

{A1,A2,A3} = now(),

random:seed(A1, A2, A3),

Gui = gui:start(Name, self()), Gui ! {color, Color},

Wait = if Sleep == 0 -> 0; true -> random:uniform(Sleep) end, timer:send\_after(Wait, cast\_change),

worker(Name, Cast, Color, Gui, Sleep), Gui ! stop.

worker(Name, Cast, Color, Gui, Sleep) -> receive

{deliver, {change\_state, N}} -> NewColor = change\_color(N, Color), Gui ! {color, NewColor},

worker(Name, Cast, NewColor, Gui, Sleep);

{deliver, {state\_req, Ref}} ->

Cast ! {mcast, {set\_state, Ref, Color}}, worker(Name, Cast, Color, Gui, Sleep);

{deliver, {set\_state, \_, \_}} -> worker(Name, Cast, Color, Gui, Sleep);

{join, Peer} ->

Cast ! {join, Peer},

worker(Name, Cast, Color, Gui, Sleep); cast\_change ->

Cast ! {mcast, {change\_state, random:uniform(?change)}}, Wait = if Sleep == 0 -> 0; true -> random:uniform(Sleep) end, timer:send\_after(Wait, cast\_change),

worker(Name, Cast, Color, Gui, Sleep); stop ->

Cast ! stop, ok;

Error ->

io:format("worker ~s: strange message: ~w~n", [Name, Error]),

worker(Name, Cast, Color, Gui, Sleep)

end.

change\_color(N, {R,G,B}) ->

{G, B, ((R+N) rem 256)}.

**Appendix B: *gui.erl***

### Exported Function:

1. start/2 Function:
   * This function initializes a GUI frame and sets up a message loop for handling messages related to the GUI.
   * Parameters:
     + Name: The window title for the GUI frame.
     + Master: The process identifier of the master process that will interact with the GUI.
   * Inside the function:
     + It spawns a new process that calls the init/2 function with the provided parameters.
     + The Frame variable is set to the GUI frame that is created using the make\_frame/1 function.
     + It starts the message loop by calling the loop/2 function to handle various messages and GUI events.
2. init/2 Function:
   * This function initializes the GUI frame and sets up event handling.
   * Parameters:
     + Name: The window title for the GUI frame.
     + Master: The process identifier of the master process that will interact with the GUI.
   * Inside the function:
     + It creates a GUI frame with the specified Name using the make\_frame/1 function.
     + It sets up a message loop by calling the loop/2 function with the frame and master process as parameters.
3. make\_frame/1 Function:
   * This function creates and configures a GUI frame.
   * Parameter:
     + Name: The window title for the GUI frame.
   * Inside the function:
     + It initializes a new server (wx:new()) to manage GUI components.
     + It creates a GUI frame with the specified Name, size, and background color.
     + It shows the frame to make it visible to the user.
     + It sets up an event handler to monitor the closing window event, ensuring that the GUI frame can be closed gracefully.
     + The frame is returned as the result.
4. loop/2 Function:
   * This function handles messages and events related to the GUI frame.
   * Parameters:
     + Frame: The GUI frame.
     + Master: The process identifier of the master process that interacts with the GUI.
   * Inside the function:
     + It enters a receive loop to handle various messages and events.
     + If the GUI frame is closed by the user, it destroys the frame and sends a stop message to the master process.
     + If a color message is received, it updates the background color of the frame using the color/2 function.
     + If a stop message is received, it stops the loop.
     + If an error message is received, it prints a message to the console.
     + The loop continues to listen for messages and events.
5. color/2 Function:
   * This function sets the background color of the GUI frame.
   * Parameters:
     + Frame: The GUI frame.
     + Color: The color to set as the background color.
   * Inside the function:
     + It uses the wxFrame:setBackgroundColour/2 function to set the background color of the frame.
     + It refreshes the frame to apply the color change.

-module(gui).

-export([start/2]).

-define(width, 200).

-define(height, 200).

-include\_lib("wx/include/wx.hrl").

start(Name, Master) ->

spawn\_link(fun() -> init(Name, Master) end).

init(Name, Master) ->

Frame = make\_frame(Name), loop(Frame, Master).

make\_frame(Name) -> %Name is the window title

Server = wx:new(), %Server will be the parent for the Frame

Frame = wxFrame:new(Server, -1, Name, [{size,{?width, ?height}}]), wxFrame:setBackgroundColour(Frame, ?wxBLACK),

wxFrame:show(Frame),

%monitor closing window event wxFrame:connect(Frame, close\_window), Frame.

loop(Frame, Master)-> receive

%check if the window was closed by the user #wx{event=#wxClose{}} ->

wxWindow:destroy(Frame), Master ! stop,

ok;

{color, Color} -> color(Frame, Color), loop(Frame, Master);

stop ->

ok;

Error ->

io:format("gui: strange message ~w ~n", [Error]), loop(Frame, Master)

end.

color(Frame, Color) -> wxFrame:setBackgroundColour(Frame, Color), wxFrame:refresh(Frame).