ARCHITECTURE DESIGN

Nome del programma: CLUSTER !! TEMPORANEO !!

2 PROGRAMMI: main-server (eseguito da daemon vm)
client,
node_creation,
load-balancer (stupido e banale)

POSSIBILI LINGUAGGI: RUST, GO

i programmi operano in locale su una stessa macchina,
I nodi saranno delle VM, inizialment 3 per
rindondanza.

I nodi potranno essere aggiunti tramite
instanze del programma node_creation.

VM:

MAIN SERVER:

```
VAR LOG = Ø

main(){

    input_messages();

}


TIMEOUT_N= COMMON + RANDOM

fun input_messages {

  While (TRUE) do
              recv (APPENDENTRY m).TIMEOUT (TIMEOUT_N)
              • OK(async read_message(m, sender))
              • FAIL (async indici_elezioni());
    done
}
```

```
fn   indire_elezioni()
{

    my_term++;
    send_all   message(REQUEST_VOTE:
                          - MY_TERM
                          - MY_ID
                          - MY_LAST_LOG_INDEX
                          - MY_LAST_LOG_TERM
                       ).send()

}
```

```
Num update_index = 0
var Voted_for = NIL

fn read_message(m, sender)
{
    switch type_message(m){

        case a_e: //APPEND ENTRY
            COROUTINE(append_entry_mex(m,sender));
            break;
        case req_v : //REQUEST VOTE
            COROUTINE(other_node_vote_candidature(m,sender))
            break;
        case new_c : // new client connection
            COROUTINE(input_data_user(m,sender));
            break;
        case acc_c :
            COROUTINE( add_supporter(m,sender));
            break;
        case lb_l : //Load_balancer_leader
            COROUTINE(answer_load_balancer(m, sender);
            break;
        case l_new_conf: //extern to leader //
            COROUTINE( new_conf(m, sender));
            break;

        case f_new_conf: // new_node in cluster //
            COROUTINE ( copy_state(m, sender)); ●
            break;
```

● Rimozione di nodi per downtime del nodo

```
fn  append_entry_msg(m, sender) {
          state= follower
          if(check_consistency(a_e.prev_log_index,
                                                      a_e.prev_log_term))

          then
                    send(leader, {TRUE,
                                          MY_TERM});

                    update_state(a_e.entrys, prev_log_index);
                    update_index(a_e.leader_commit);

          else
                    send(sender, {FALSE, MY_TERM});


}

Leader ——> Bool leader [become_leader]
fn  answer_load_balancer()
{
   if(Leader)
      send(sender, true);
   else
      send(sender, false);

 }
```

```
Bool votante ⟶ add_entry.votante
fn      other_node_vote_candidature(m, sender)
{   if(not(votante))then return endif;
    if(m.term < my_therm) then send(sender, my_term, false) endif
   if( !more_recent_log(m.last_log_index, m.last_log_term ))
    then
           send (sender, my_term , false)

    else if (already_vote=nill  ||  already_vote= sender)
      then
            send( sender, my_term, true); already_vote=sender;
      else
            send( sender, my_term, false)
 }    endif




hode_list={ }
Bool leader=false
fn  become_leader()

{
    send_all   (APPEND_ENTRY);
    leader = true;

    While  true
    do
     send_all    (APPEND_ENTRY);          //
      wait(timeout);
    done
 }
```

```
voted-for ⟶ voted_for[accept_vote]
n-nodes-in-cluster= C
n-supporter = 0
n_non_supporter= 0
fn  add_supporter(m,sender)
 {
    if (m. vote ==TRUE)
    then
        n-supporte ++;

    else
        n-non_supporter ++;

    endif

    var n_victory= (n_nodes_cluster /2)

    if(n_supporter > n_victory)
    then
        become_leader(); voted_for =null;
    endif

    if (n_supporter + n_non_supporter  == n_nodes_cluster)
     then
        voted_for=nill

     endif
  }
```

```
fn input_data_user (m, sender) //mex user instr
  {
                                        {USER_INSTR, R_W, ENTRY_N, DATA}

      switch (m.R_W){
          case R:
              FILE= get_entry (m.ENTRY_N)
              send (sender, read (FILE))
              break;
```

```
Bool leader ⟼ become_leader.leader
fn answer_load_balancer (m, sender)
 {
    send(sender,{LEADER: leader});

 }
```

```
fn added_node()
{

                send ({MY_ID: IP, NEW_CONF: TRUE,
                     TERM: MY_TERM,
                     L_I: MY_LAST_LOG_INDEX,
                     L_T: MY_LAST_LOG_TERM, });

}
```

```
Bool votante = false
fn copy_state(m, sender)
{
    add_entry(m.entry, m.term, m.index);
    votante = m.votante

}
```

```
fn update_new_node(m)
{
  foreach (Log_entry l : Logs)
  {

    mex = {Body: l ; VOTANTE: FALSE};

    send(m, ip_new_node, mex);

  }

}
```

```
list_updated_node = "old_nodes"
List updating = Ø
fn new_conf(m, sender)
{

    if(not(IsEmpty(m.to_add)))
    then
        COROUTINE(      ar_nodes(m.to_add, new));
    endif

    if(not(IsEmpty(m.to-remove)))
    then
        COROUTINE( ar_nodes(m.to_remove, del));
    endif

}

updating ⟼ new_conf.updating
fn            ar_nodes(to_add, op)
{
  Add(updating, to_add);
  foreach(node : to_add)
   {
     if(op = new) then
        COROUTINE(update_node(node, m));
     elseif(op = DEL) then
        COROUTINE(remove_node(node));
  }
}
```

```
fn update_node (node, m)
    Add_log_entry ("joint_conf.", node)
    send_all (APPEND_ENTRY);

    update_new_node(node, m)
    send( node , {BODY: NULL, VOT ANTE = TRUE})
     Add( list_updated_nodes, node );
     Remove ( updating, node );
Add_log_entry ( "nodes updated" , m.ip_new_node);

    if ( not( IsEmpty(updating) ) ) then return;
      send_all ( APPEND_ENTRY );
}
```

```
fn remove_node (node , m)
{
  Add (updating , node)
  Add_log_entry ("removed node:" node)
  Remove(updating, node);
  if ( not(IsEmpty(updating)) ) then return;
     send_all(APPEND_ENTRY);
}
```