ARCHITECTURE DESIGN

Nome del programma: CLUSTER !! TEMPORANEO !!

2 PROGRAMMI: main-server (eseguito da deamon 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, inizialmente 3 per rindondanza.

I nodi potranno essere aggiunti tramite instanze del programma node_creation.

VM:

MAIN SERVER:

```
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()
{

     status.increase_term();
     send_all(REQUEST_VOTE);

}
```

```
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;
        case a_append_entry:
              coroutine(append_entry_answer(m,sender));
              break;
```

```
fn append_entry_answer(m,sender)
{
    if(m.answer == FALSE){
        status.decrease_update_index(sender);
        var node_index = status.get_update_index_node(sender);
        mex append_entry =
        {
            log.get_term();
            log.get_leader_id();
            log.get_prev_log_index(node_index);
            log.get_prev_log_term(node_index);
            log.get_new_entries(node_index);
            log.get_leader_commit(nodex_index);
        };

        send(append_entry,sender);
    }

    if(m.term > status.get_term()){
        status.set_term(m.term);
        status.role = FOLLOWER;
    }
}
```

```
fn append_entry_msg(m, sender) {
            status.set_role(FOLLOWER);
            if(check_consistency(a_e.prev_log_index,
                                 a_e.prev_log_term))
            then
                send(leader, {TRUE,
                              MY_TERM});

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

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

}


fn answer_load_balancer()
{
    if(status.get_role() == LEADER){
      send(sender, true);
    else
      send(sender, false);

 }
```

```
fn    other_node_vote_candidature(m, sender)
{   · if(not(status.can_vote())) {return; }
    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
  }
```

```
fn  become-leader()

 {
    send_all   (APPEND_ENTRY);
    status.role = LEADER;

    While  status.role == LEADER
    do
      send_all   (APPEND_ENTRY);
       wait(timeout);
    done
  }
```

```
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();     status.set_vote_for(NILL);
     endif

     if (n_supporter + n_non_supporter == n_nodes_cluster)
      then
            status.set_vote_for(NILL);

      endif
  }
```

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

   if(status.get_role() != LEADER){
     var leader_id = status.get_leader_ip();
     send(leader_ip, m)
   endif

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

       case W:
           data = m.data;
           status.upload_file(data)
           break;

   }
 }
```

```
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, }) ;

}


fn answer_load_balancer(m, sender)
{
    send(sender,{LEADER: leader});

}


fn copy_state(m, sender)
{
     add_entry(m.entry, m.term, m.index);
     status.set_able_to_vote(m.votante);

}
```

```
fn update_new_node(m)
{
    foreach (Log_entry l : Logs)
    {
        mex = {Body: l ; VOTANTE: FALSE};
        send(m, ip_new_node, mex);
    }
}
```

```
fn new_conf(m, sender)
{

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

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

}


fn        ∂r_nodes(to_add, op)

{

    status.add_updating_node(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)
    status.add_log_entry    . ("join_conf.", node)
    send_all (APPEND_ENTRY);

    update_new_node (node, m)
    send( node    . ,{BODY: NULL; VOTANTE = TRUE})

    status.add_updated_node(node);
    status.remove_updating_node(node);

status.add_log_entry ("nodes updated"    , m.ip_new_node);

    if ( not(status.not_updating_node())    ) then return;

        send_all (APPEND_ENTRY);

}
```

```
fn remove_node (node)
{
    status.remove_updated_node(node);

        status.add_log_entry ("removed node:" node)

    status.remove_updating_node(node);

    if ( not(IsEmpty(updating) ) ) then return;

      send_all ( APPEND_ENTRY );

}


fn      send_all ( message_type T)
 {

      switch (T);
        case   REQ_VOTE:
            COROUTINE( send_all _request_vote() );
            break;
        case APPEND_ENTRY:
            COROUTINE( send_all_append_routine() );
            break;

}
```

```
fn    send_all_request_vote()
 {
  var  mex={REQUEST_VOTE:
                     status.get_term(), // MY_TERM
                     status.get_id(),    // MY_ID
                     status.get_ll_index,// MY_LAST_LOG_INDEX
                     status.get_ll_term}// MY_LAST_LOG_TERM

     foreach ( node_ip   in    status.get_updated_node())
             COUROUTINE(send(node_ip , mex).timeout(TIME)
              .FAIL(remove_node(node_ip));
   } }        .ok() ) ;

  fn   send_all_append_entry()
    {
     var   mex = {APPEND_ENTRY:
                        status.get_term(),
                        status.get_id(),
                        status.get_prev_l_index(),//prev_log_index
                        status.get_prev_l_term(),//prev_log_term
                        status.get_new_entries(), //uncommitted entries
                        status.get_leader_commit()}//index last committed
                                                   entries

     foreach (node_ip   in    status.get_updated_node())
              coroutine(send(mex,node_ip).timeout(TIME)
                    .fail(remove_node(node_ip))
                    .ok();

      }
  }
```