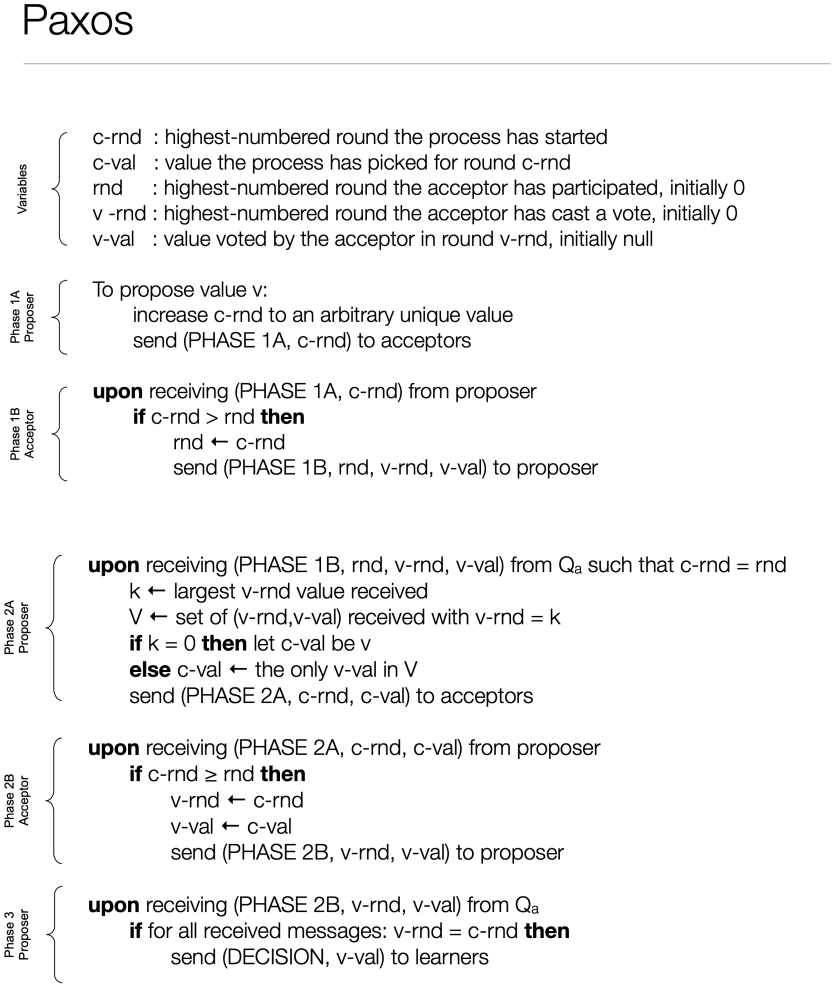
Project Distributed Algorithms I: Paxos

Peter Buttaroni, Mauro Gentile

November 2, 2020

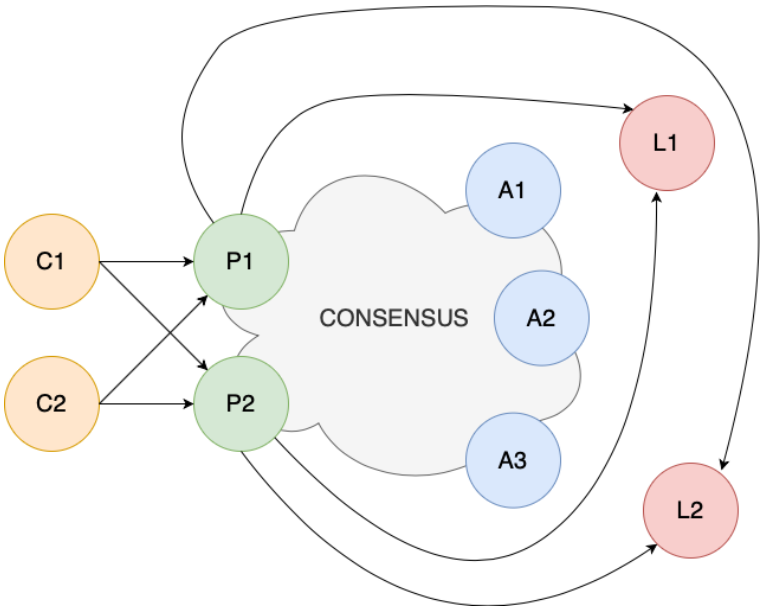
1 The Algorithm

This implementation of Multi-Paxos is made up of 5 phases for every instance.
This phases are represented in the following image (taken from the slides provided during the course).



The only main difference with the pseudocode ahowed above is the fact that, every messages send by the proposer contains also the instance number. The following images is a scheme that represents the functioning of the Paxos.
As we can see, we have two clients (**C1** and **C2**) that sends their values to two proposers (**P1** and **P2**). These proposers, for each value received, start a round of consensus in order to get the approval of the acceptors

(**A1**, **A2**, **A3**). Once a value is approved, the proposers send the decision to the learners (**L1** and **L2**). The goal is that the two learners learn all the values sended by the clients in the same order.



1.1 Clients

In our implementation we have a pre-phase where the clients (indicated as **C1** and **C2** in the image above) send their value to both the proposers (indicated as **P1** and **P2**). The Python code that implements the client is the following:

```
def client(config, id):
  print '-> client ', id
  s = mcast_sender() # Creation socket
  for value in sys.stdin:
    value = value.strip()
    print("client: sending %s to proposers" % (value))
    value_send = create_sending_code(value, 0)
    msg = encode(['client val', value_send])
    s.sendto(msg, config['proposers'])
  print('client done.')
```

As we can easily see, this code just takes the value form the standard input (a text file) and send them to the proposers.

1.2 Proposers

The second element of our model is the proposer. The following images is the code that implements the proposer. As we can see, we created a dictionary **params** that contains all main features of a single proposer.

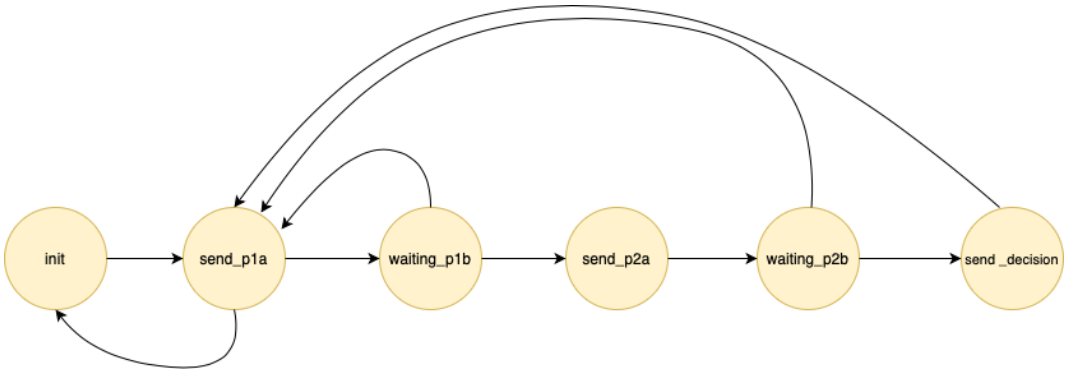
- **state**: is a state variable that assume different values according to the current state of the proposer. The values of this variable are needed to decide which function we need to call during the while Loop.
- **instance**: is the instance round at which our proposer is currently participating.
- **id**: is the id of the proposer.
- **rnd**: is the round number used to propose a value to the acceptors (as explained below)

- **clients_values**: is the list of the values send by the clients.
- **proposed_values**: is the value from the clients that our proposer is currently pushing.
- **my_value**: is a flag that indicates if the value currently proposed comes from the list of the proposer or it was received from the acceptors in phase **Phase 1b**
- **socket_s** and **socket_r**: are the socket used to send and receive the messages.

According to the current value of the variable **state**, there are different actions that our proposer must executes. These actions are represented as function that are called in the while loop according to the value of **state**.

```
def proposer(config, id):
  print '-> proposer', id
  r = mcast_receiver(config['proposers'])
  s = mcast_sender()
  r.settimeout(timeout_message)

  params={
  'state' : "init",
  'instance' : 1,
  'id' : id,
  'rnd' : 0,
  'clients_values' : [],
  "proposed_value" : "None",
  'my_value' : False,
  'socket_s' : s,
  'socket_r' : r
  }

  actions = {"init": listen_clients,
             "send_p1a" : p1a,
             "waiting_p1b" : waiting_p1b,
             "send_p2a": p2a,
             "waiting_p2b" : waiting_p2b,
             "send_decision" : send_decision
            }

   while True:
       params=actions[params["state"]](params)
```

We can describe the states using the following state-transition scheme:



- During the **init state** the proposer has no values from the clients in its buffer. Therefore, the only thing that it can do is to wait of receiving a new value. Once the value is received, we can move to the next state (**send_p1a**).
  The following code is our implmenetation of this state.

```
def listen_clients(params):
  while True:
    r = params["socket_r"]
    try:
      msg = r.recv(msg_length )
      msg_list = decode(msg)
      if msg_list[0] == 'client val':
          params["clients_values"].append(msg_list[1])
          params["state"] = "send_p1a"
          return(params)
    except socket.timeout:
      continue
```

  Please, notice that we have used a timeout for the receiving socket. This is because we need to prevent a situation where our proposer is stucked in a waiting loop. In the **init state** this is not necessary since the proposer need just to wait but there are other situation where the waiting loop is undesiderable.

- In the **send_p1a** the first thing that the proposer does is to check if actually there exists a value to propose in its buffer. This check is just to ensure that, if for some reason the proposer reaches this state without any value, it can go back to wait a message from the clients.
  If it has something in its buffer, the proposer sends a message to all the acceptors specifying the phase, the instance number, the round and its id. After sending the message, the proposer changes its status to **waiting_p1b**.

```
def p1a(params):
  if len(params["clients_values"]) == 0:
    params['state'] = 'init'
    return(params)
  params['rnd']+=1

  _, n_instance, id, rnd,_,_,_,s,_ = unpack_dict(params)
  sending_code = create_sending_code(n_instance, rnd, id)
  send_p1a(s, sending_code)
  params["state"]="waiting_p1b"
  return params
```

- In the state **waiting_p1b** the proposer waits to receive the majority (Qa) of the reply messages from the acceptors. These messages are marked as **p1b**.
  Of course, while it's waiting the acceptors' replies, the proposer can also receive some messages from the client. In this case it has to add these values to its buffer. The proposer waits for a given amount of time to receive the replies. If this does not happen, it just go back to **phase send_p1a**. on the other hand, if it gets enough replies, it decides which value it has to propose and set its own state to **send_p2a**.
  The following image shows the python code that implements this phase.

```
def waiting_p1b(params):
  v_vals = []
  v_rnds = []
  _, n_instance, id, rnd,_,_,_,_,r = unpack_dict(params)
  c_rnd = float(str(rnd) + '.' + str(id))

  start = time.time()
  while (len(v_vals) < Qa) and (time.time() - start < timeout_quorum):
    try:
        msg = r.recv(msg_length)
        msg_list = decode(msg)
        if msg_list[0] == 'client val':
            params['clients_values'].append(msg_list[1])
        elif msg_list[0] == 'p1b':
          if msg_list[1] == n_instance and msg_list[2] == c_rnd:
              v_vals.append(msg_list[4])
              v_rnds.append(msg_list[3])
          elif msg_list[1] > n_instance:
              params['instance'] = msg_list[1]
              params['state'] = "send_p1a"
              return(params)
    except socket.timeout:
            continue

  if len(v_vals) >= Qa:
      params = choose_val(v_rnds, v_vals, params)
      params['state'] = "send_p2a"
  else:
      params['state'] = "send_p1a"
      params['rnd'] += 1
  return(params)
```

We can notice that there is also a supplementary case that we handle in the code. The situation where the proposer receives a messages from the acceptor that specify a higher instance number than its own one. In this case the proposer realizes that it has some delay, updates its instance round and goes back to **phase sennd_p1a**.

- In the state **send_p2a** the proposer just sends its own proposed value and set its state on **waiting_p2b**.

```
def p2a(params):
  _,n_instance,id,rnd,_,c_val,_,s,_ = unpack_dict(params)
  sending_code = create_sending_code(n_instance, rnd, id)
  send_p2a(s, sending_code, c_val)
  params["state"]="waiting_p2b"
  return(params)
```

- In the **phase waiting_p2b** the proposer waits to receive thmajority of acknoledgments from the acceptors. These messages are marked as **p2b**.
  Also in this case we need to handle possible messages from the clients and/or messages that specify a higher instance number that the one of the proposer. In the case where our porposer does not receive enough replies in the limit time of it discover a higher round number, it goes back to **send_p1a**.
  If it succeed in receiving eough acknowledgements, it set its state on **send_decision**.

```
def waiting_p2b(params):
  _, n_instance, id, rnd, clients_values, _, my_value,_,r = unpack_dict(params)
  c_rnd = float(str(rnd) + '.' + str(id))
  sending_code = create_sending_code(n_instance, rnd, id)
  start = time.time()
  Q = 0
  while (Q < Qa) and (time.time() - start < timeout_quorum):
    try:
      msg = r.recv(msg_length)
      msg_list = decode(msg)
    except socket.timeout:
      continue
    if msg_list[0] == 'client val':
      params['clients_values'].append(msg_list[1])
    elif (msg_list[0] == 'p2b'):
      if (msg_list[1] == n_instance) and (msg_list[2] == c_rnd):
        Q += 1
      elif msg_list[1] > n_instance:
        params['instance'] = msg_list[1]
        params['state'] = "send_p1a"
        return(params)

  if Q >= Qa:
    params['state'] = "send_decision"
    if my_value == True:
      params["clients_values"] = params["clients_values"][1:]
      params["my_value"] = False
  else:
    params['state'] = "send_p1a"
  return params
```

- The state **send_decision** is the state where the proposer sends the decided value to the learner and increases the instance number by 1.

```
def send_decision(params):
  _, n_instance, id, rnd, _, v_val, _,s,_ = unpack_dict(params)
  sending_code = create_sending_code(n_instance, rnd, id)
  send_decision_to_learners(s, sending_code, v_val)
  params['state'] = "send_p1a"
  params["instance"] += 1
  return params
```

1.3 Acceptors

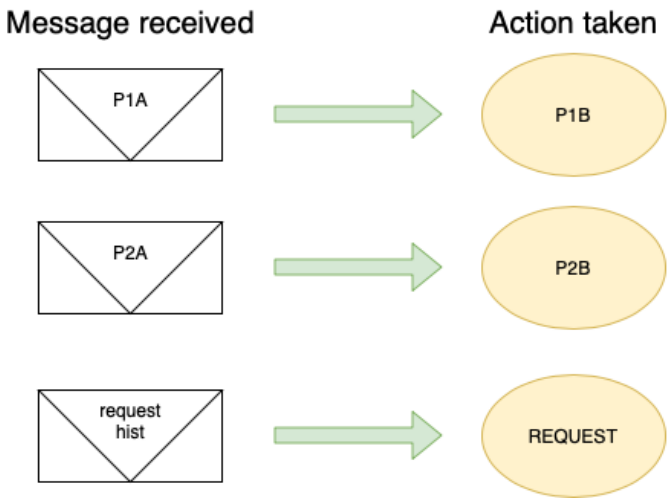
With respect to the proposer, the acceptor is relatively easy to implement.
Indeed, the task of the acceptors is just waiting for some messages from the proposers and send back a reply.
The following images is the implementation we adopt.

```
def acceptor(config, id):
  print '-> acceptor', id

  r = mcast_receiver(config['acceptors'])
  s = mcast_sender()

  hist_list = {}

  actions = {"p1a": p1b,
             "p2a" : p2b,
             "request_hist" : request,
            }

  params={'s' : s,
          'hist_list' : hist_list,
          'msg_list' : [],
          'id' : id,
          'state': 'Alive'
  }
  while params['state'] == 'Alive':
    msg = r.recv(msg_length)
    msg_list = decode(msg)
    params['msg_list'] = msg_list
    if msg_list[0] in actions.keys():
      params = actions[msg_list[0]](params)
    else:
      pass
    if (len(params['hist_list'].keys()) > n_instances_BD * id) and (id <= faulty_acceptors):
        print(id, 'Dead')
        print(len(params['hist_list'].keys()))
        params['state'] = 'Dead'
```

Also in this case we opted for a dictionary (called params) that contains all the main feature of the acceptor.

- **s**: the socket for sending messages.
- **hist_list**: a list where we save all the instances decided so far by the acceptor.
- **msg_list**: a list where we save the last message received by the proposers.
- **id**: the id of the acceptor.
- **state**: a state that we use only if we want to simulate the crash of an acceptor (usually is set on 'Alive').

As we already mention, every acceptor has a **hist_list**: a dictionary where, for every instance (that is the key of the dictionary) the acceptor saves the most recent round he took part, the last round when he chose a value (**v_rnd**), and the value chosen (**v_val**).
The acceptors, according to the message that they receive, behave in different ways.
We can have three possible types of message: **p1a**, **p2a**, **request_hist**. The different behaviours are grouped in a dictionary called actions.
The following scheme represents the relation between received message and action taken.



- The **action p1b** is executed if the acceptor receives a message **p1a** from the proposer.
  In this case, the acceptor checks if it is already a record for the instance number in the message.
  - If it has no record for that instance number, the acceptor creates a new record and send back a message **p1b** of the following type: **("p1b", n_instance, c_rnd, 0, None)**.
  - Viceversa, if already exists a record in the **hist_list** for the given instance number, the acceptor check the **c_rnd** in the message; if it is greater than its own **rnd**, it updates the round and send back to the proposer a message as the following one: **("p1b", n_instance, c_rnd, v_rnd, v_val)**.

```
def p1b(params):
  s = params['s']
  hist_list = params['hist_list']
  msg_list =  params['msg_list']

  prop_instance, c_rnd = msg_list[1], msg_list[2]
  if prop_instance in hist_list.keys():
    rnd,v_rnd,v_val = hist_list[prop_instance]
    if rnd<c_rnd:
        rnd=c_rnd
        hist_list[prop_instance][0]=c_rnd
    else:
      params['hist_list'] = hist_list
      return params
  else:
    rnd = c_rnd
    v_rnd = 0
    v_val = None
    hist_list[prop_instance]=[rnd, v_rnd, v_val]

  sending_code = create_sending_code(prop_instance, c_rnd)
  send_p1b(s, sending_code, v_rnd, v_val)
  params['hist_list'] = hist_list
  return params
```

- The **action p2b** is executed if the acceptor receives a message **p2a** from the proposer.
  Again, the acceptor checks if the instance number received is already in **hist_list**.

- If the instance number is not in the dictionary, the acceptor sends a message **p2b** in the following form: **("p2b", n_instance, c_rnd, v_rnd, v_val)**
- If there already exists an element in **hist_list** with the key equal to the instance number received, the acceptor checks also that the **c_rnd** received is bigger or equal to its **rnd**. If this is the case then he save the new **v_rnd** and **v_val** and sends back a **p2b** message as the following one: **("p2b", n_instance, c_rnd, v_rnd, v_val)**.

The following code implements this algorithm.

```
def p2b(params):
  s = params['s']
  hist_list = params['hist_list']
  msg_list =  params['msg_list']
  prop_instance, c_rnd = msg_list[1], msg_list[2]
  if prop_instance in hist_list.keys():
    rnd, _,_ = hist_list[prop_instance]
    if c_rnd >= rnd:
      v_rnd = c_rnd
      v_val = msg_list[3]
      hist_list[prop_instance] = [c_rnd, v_rnd, v_val]
    else:
      params['hist_list'] = hist_list
      return params
  else:
     v_rnd = c_rnd
     v_val = msg_list[3]
     hist_list[prop_instance]=[c_rnd, v_rnd, v_val]

  sending_code = create_sending_code(prop_instance, c_rnd)
  send_p2b(s, sending_code, v_rnd, v_val)
  params['hist_list'] = hist_list
  return params
```

- The action **request** is executed if the acceptor receives a message **request_hist**. In this case, the acceptor just sent to the learners all its **hist_list**.

```
def request(params):
  s = params['s']
  hist_list = params['hist_list']
  id =  params['id']
  send_hist_list(s,id,hist_list)
  return(params)
```

1.4 Learners

As in the case of acceptor, we implement a learner with a while loop where it just waits to receive a message from the other processes. According to the message recieved, the learner has two possible actions: **reconstruct_decisions** and **decision_to_list**.
Also in this case, all the main features of the learner are embedde in a dictionary.

- **decisions**: is a dictionary where the learner saves all the decisions received from the proposer. The decision are saved using as key the instance number.
- **counter**: this counter points to the last value that the learner has printed from the dictionary.
- **received**: This dictionary contains the **hist_list** messages received from the acceptors for the catch-up.
- **learner_id**: this is just the id number of the learner.

```
def learner(config, id):
  r = mcast_receiver(config['learners'])
  s = mcast_sender()
  r.settimeout(timeout_message)
  send_request(s)
  params = {
           'decisions' : {},
           'counter' : 1,
           'received' : {},
           'learner_id' : id
           }

  actions = {"hist_list": recontruct_decisions,
             "decide" : decision_to_list
             }

  while True:
    try:
      msg = r.recv(msg_length)
      msg_list = decode(msg)

      params = actions[msg_list[0]](params, msg_list)
      sys.stdout.flush()
    except socket.timeout:
      params = printvals(params)
      sys.stdout.flush()
      pass

    params = printvals(params)
    sys.stdout.flush()
```

When we create a new learner the first thing that it does is sending a request to the acceptors in order to receive their **hist_list**. This because, if we want to introduce a new learner in second moment, it needs to recover all the previous decisions (the so-called catch up). After that, the learner just waits to receive messages from the acceptors and/or from the proposer. According to the message received, it acts in different ways.

- The **action decision_to_list** is executed if the learner receives a message **decide** from the proposer. In this case, the learner has to add the new value to the dictionary **decisions**. The following code implements this action.

```
def decision_to_list(params,msg_list):
  msg_type, n_instance, c_rnd, val=msg_list
  params["decisions"][int(n_instance)] = val
  return(params)
```

- The **action reconstruct_decisions** is executed if the messages received is **hist_list** from an acceptor. In this case the learner add the **hist_list** received to the dictionary **received**. Moreover, if it has enough **hist_list** from different acceptors (i.e. the majority), it runs the function **aggregate_decisions** that using the lists received, tries to reconstruct the decisions for the previous instances of Paxos.
  The following images show the two function mentioned.

```
def recontruct_decisions(params, msg_list):
    acceptor_id = msg_list[1]
    acceptor_hist = msg_list[2]
    if not(acceptor_id in params["received"].keys()):
        params["received"][acceptor_id] = acceptor_hist
    if len(params["received"].keys())>=Qa:
        params = aggregate_decisions(params)
    return(params)
```

```
def aggregate_decisions(params):
    acceptors_hist = []
    for decision_dic in params['received'].values():
        acceptors_hist.append(decision_dic)
    decisions = params['decisions']
    value_list = []
    for idx in range(len(acceptors_hist)):
      value_list.extend([str(k) + "-" + str(v[2]) for k,v in acceptors_hist[idx].items() ])
    filtered_values = [k for k, v in Counter(value_list).items() if v >= Qa]
    g = [[int(i) for i in k.split("-")] for k in filtered_values ]
    params['decisions'] = merge_two_dicts(dict(g), decisions)
    return(params)
```

After having executed one of the two actions mentioned above, the learner calls the function **printvals**. This function print the value in the dictionary **decisions** that has a key equal to **counter**. After that, it increases the coiunter. If there is no key corresponding to the current value of **counter**, the function does nothing.
The idea behind it is that the instance numbers (i.e. the keys in the dictionary **decisions**) are progressive and without holes. This means that if there is a missing key, the corresponding instance was not decide yet or the value is not arrived yet. In both cases the learners needs just to wait.

```
def printvals(params):
    decisions=params['decisions']
    counter=params['counter']
    if counter in decisions.keys():
        print(decisions[counter])
        counter += 1
        params['counter'] = counter
    return params
```

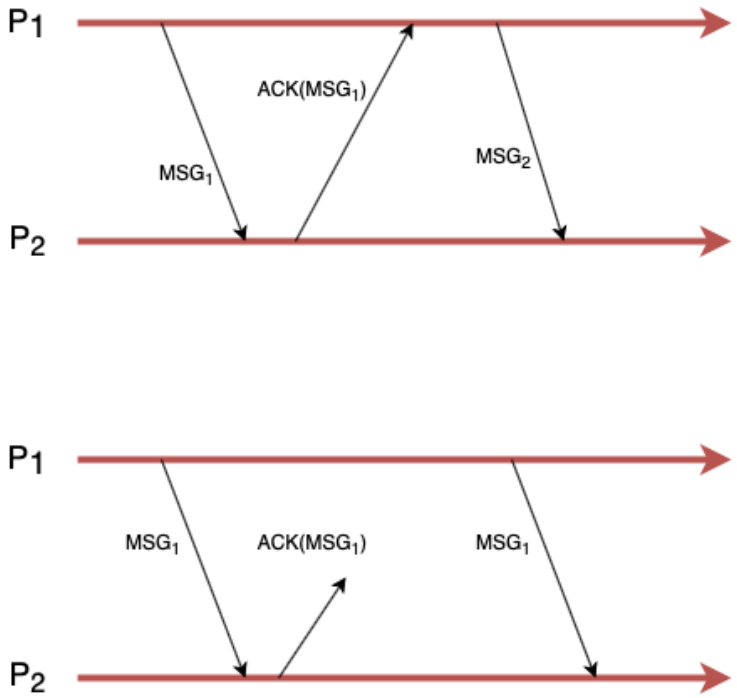
2 Message loss case

If we are in a framework where we can lose the messages exchanged between the processes, our first implementation of Paxos does not guarantee anymore to decide all the values send by the clients. Therefore, in

order to face this issue, we must introduce a new component. Any time a client send a message to a proposer or a proposer send a message to a learner, the receiver must send an **ACK** to the sender.

- If the sender receives the **ACK**, it can the next message.
- If the sender does not receive any **ACK**, it send again the message until when it will receive the **ACK**.

The following image represents this scheme.



In the first case P_1 sends MSG_1 and receives the **ACK** from P_2. Therefore, P_1 can now send the following messages (MSG_2).
In the second case P_1 sends MSG_1 but does not receive the **ACK** from P_2. Hence, it sends again MSG_1.

Of course the receiver, once it received the message for the first time, sends the **ACK** and save the message. In this way, if it receives again the message it will send again **ACK** but it knows that the message received is a copy of am previous message.
This implementation of Paxos requires more time and a bigger number of exchanged messages but is able to learn all the values also non reliable links.

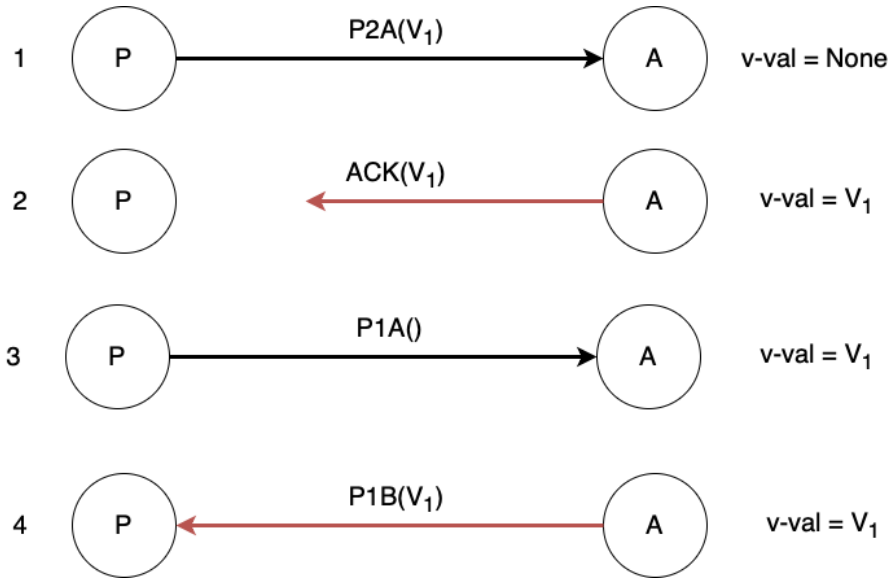
2.1 The replication of the values

In the modified version of the Paxos that we implemented, at the beginning we noticed that sometimes, instead of learning n values, the learners were learning $n+k$ values where k was a small number. In practice some values where learned more than one time. It was not clear to us why this was happening.

The explanation of the issue is the following one.
Every time that a proposer manages to get a value approved, there are two possibilities:

- The value comes from the list of the proposer. Then, we must delete it from that list and pass to the next value. This happen when in phase **P1B** the proposer receives from the acceptors all **v_val = None** and then, the proposer can propose any value it wants (so, it proposes the first value in its list).
- The value comes from an acceptor. Indeed, if at least one acceptor replies in phase **P1B** with a **v_val** $\neq$ **None**, the proposer must propose that value. In this case, since the value was not in the list of the proposer, it has nothing to delete.

The problem arises if a value V_1 is proposed by a proposer in phase **P2A** and the **ACK**(V_1) from the acceptors is lost.
In this case, when in the next phase **P1A**, the proposer sends again a message to the acceptors, they will reply sending back the value V_1. The proposer will receive **v-val** $= V_1$ and it will not realize that it was the value proposed by itself in the previous phase **P2A**. Therefore, it will not delete V_1 from the list after the approval, and at the next instance round, it will propose again V_1 (this time taken from its list).
In this way, some values were proposed and accepted more than one time. The problem is represented in the following images.



This problem is solved by exchanging not only the **v-val** but also the **id** of the proposer that proposed such **v-val** to the acceptors. In this way, when the proposer receive the message **P1B** from the acceptors, it will realize that the **v-val** comes from its list.