1 Introduction

- Process execution and inter process communication takes arbitrary time **(Asynchronous)**, is upper bounded **(Synchronous)** or is mostly synchronous with periods which they aren't **(Partially synchronous)**
- **Complexity**: Number of messages, size of messages, memory used, time taken
- **Fail-stop** crashes can be reliably detected by other processes. **Fail-silent** crashes cannot. **Fail-noisy** if detection takes time. **Fail-noisy** crashes can recover. **Fail-arbitrary** crashes exhibit arbitrary and possibly malicious behaviour.

2 Failure Detector

- **Strong Completeness (L)**: Every crashed process will eventually be permanently suspected by every correct process
- **Strong Accuracy (S)**: No process is suspected before it crashes
- **Eventually Strong Accuracy (L)**: Eventually no correct process is suspected
- **Perfect Failure Detector**: Provides processes with list of suspected processes that have crashed. Makes timing assumptions (no longer synchronous). Never changes its view (suspected processes are suspected forever). Strong completeness + Strong Accuracy.
- **Eventually Perfect Failure Detector**: May make mistakes but will eventually accurately detect a crashed process

```
defmodule PFD do
def start do receive do { :bind, c, pl, delay, ps } -> Process.send_after(self(), :timeout, delay)
    %{c: c, pl: pl, delay: delay, ps: ps, alive: ps, crashed: empty_set() } |> next()
end end
defp next(this) do receive do
{ :pl_deliver, from, :req } -> send this.pl, { :pl_send, from, :rep }; this |> next()
{ :pl_deliver, from, :rep } -> this |> alive_put(from) |> next()
:timeout -> new_c = for p <- this.ps, p not in this.alive and p not in this.crashed, into: empty_set do p end
for p <- this.alive do: send this.pl, { :pl_send, p, :req }; for p <- new_c do: send this.c, { :pfd_crash, p }
Process.send_after(self(), :timeout, this.delay); this |> alive(empty_set) |> crashed_union(new_c) |> next()
end end end
```

3 One Shot Broadcast

- **No Duplication (S):** No message is delivered to a process more than once.
- **No Creation (S):** No message is delivered unless it was sent.
- **Reliable Delivery (L):** If A & B are correct processes, every message sent by A to B is eventually delivered to B.
- **Validity (L)**: If a correct process broadcasts a message then every correct process eventually delivers it.
- **Agreement (L)**: If a correct process delivers message M then every correct process also delivers M.
- **Uniform Agreement (L)**: If a process delivers message M then every correct process also delivers M.
- **Perfect Point-to-Point Links (PL):** Reliable Delivery + No Duplication + No Creation.
- **Best Effort Broadcast (BEB):** Validity + No Duplication + No Creation
- **Regular Reliable Broadcast (RB):** All correct processes will agree on messages they deliver even if broadcasting process crashes while sending. Agreement + Validity + No Duplication + No Creation.
- **Uniform Reliable Broadcast (URB):** Uniform Agreement + Validity + No Duplication + No Creation

```
// BEB:  1 broadcast step, O(N) messages
{ :bind, c, pl, processes } -> %{ c: c, pl: pl, processes: processes} |> next()
{ :beb_broadcast, msg } -> for dest <- this.processes do send this.pl, { :pl_send, dest, msg } end; next(this)
{ :pl_deliver, from, msg } -> send this.c, { :beb_deliver, from, msg }; this |> next()
// Eager RB (fail-silent): O(N) BEB broadcasts, O(N^2) messages
{ :bind, c, beb } -> %{ c: c, beb: beb, delibered: empty_set() } |> next()
{ :rb_broadcast, msg } -> send this.beb, { :beb_broadcast, { :rb_data, nodeID(), msg } }; this |> next()
{ :beb_deliver, from, { :rb_data, sender, msg } = data } -> if msg in this.delivered do this |> next() else
    send this.c, { :rb_deliver, sender, msg }; send this.beb, { :beb_broadcast, data }
    this |> delivered_put(msg) |> next() end
//Lazy RB (fail-stop)
{ :bind, c, beb, ps } -> %{ c: c, beb: beb, correct: ps, delivered: init_map(ps, empty_set)} |> next()
{ :rb_broadcast, msg } -> send this.beb, { :beb_broadcast, { :rb_data, nodeID(), msg } }; this |> next()
{ :pfd_crash, crashed } -> for msg <- this.delivered[crashed] do
    send this.beb, { :beb_broadcast, { :rb_data, crashed, msg } } end; this |> correct_delete(crashed) |> next()
{ :beb_deliver, from, { :rb_data, sender, msg } = data } -> if msg in this.delievered[sender] do next(this) else
    send this.c, { :rb_deliver, sender, msg }; if sender not in this.correct do
        send this.beb, { :beb_broadcast, data } end; this |> delievered_put(sender, msg) |> next() end
// Majority-Ack URB: N + N(N-1) messages
{:bind, c, beb, ps} -> %{c: c,beb: beb,ps: ps,delivered: empty_set(), pending: empty_set(),bebd: %{}} |> next()
{ :urb_broadcast, msg } -> send this.beb, { :beb_broadcast, { :urb_data, nodeID(), msg } }
    send self(), :can_deliver; this |> pending_put({nodeID(), msg}) |> next()
{ :beb_deliver, from, { :urb_data, sender, msg } = urb_m } -> msg_pset = Map.get(this.bebd, msg, empty_set())
```

```
    this = this |> bebd.put(msg, MapSet.put(msg_pset, from)); send self(), :can_deliver
    if { sender, msg } in this.pending do this |> next() else
    send this.beb, { :beb_braodcast, urb_m }; this |> pending_put({sender, msg}) |> next() end
:can_deliver -> new_delivered = for { sender, msg } <- this.pending,
    msg not in this.delivered and MapSet.size(this.bebd[msg]) > this.processes div 2, into: empty_set() do
        send this.c, { :urb_deliver, sender, msg }; msg end
    this |> delivered_union(new_delivered) |> next()
```

4 Multi Shot Broadcast

- **First In, First Out (FIFO)**: If a process broadcasts M1 before M2, all correct processes will deliver M1 before M2
- **Causal Order (CO)**: If any process delivers M2 it must have previously delivered every message M1 where M1 caused M2. M1 $\rightarrow$ M2 if (1) Alice broadcasts M1 and later broadcasts M2 (2) Alice delivers M1 and later broadcasts M2 (3) There is a message M3 where M1 $\rightarrow$ M3 and M3 $\rightarrow$ M2
- **Total Order (TO)**: All correct processes deliver all messages in the same order (no need to respect FIFO or CO). Equivalent to consensus problem
- **Uniform Total Order**: If a process delivers M1 without previously delivering M2, no correct process delivers M2 before M1.

```
// FIFO RB: Same messages as RB + extra memory
{ :bind, c, rb, ps } -> %{ c: c, rb: rb, seq: 0, pseq: init_map(ps, 1), pending: empty_set() } |> next()
{ :frb_broadcast, msg } -> send this.rb, { :rb_broadcast, { :frb_data, { node_ID(), msg, this.seq+1 } } }
    this |> seq_num(this.seq+1) |> next()
{ :rb_deliver, from, { :frb_data, {sender, _,_}=data } } -> this |> pending_put(data) |> check(sender) |> next()
def check(this, sender) do
    case Enum.find(this.pending, fn {from, _msg, _seq} -> from == sender and seq == this.pseq[sender] end) do
        {_from, msg, seq} = data -> send this.c, { :frb_deliver, sender, msg }
            this |> pseq.put(sender, seq+1) |> pending.delete(data) |> check(this)
        _otherwise -> this end end
// CO URB: Same as URB but message size and past grows linearly
{ :bind, c, urb } -> %{ c:c, urb: urb, past: [], delivered: empty_set() } |> next()
{ :crb_broadcast, msg } -> send this.urb, { :urb_broadcast, { :crb_data, this.past, msg } }
    this |> past_append([{ nodeID(), msg }]) |> next()
{ :urb_deliver, from, { :crb_data, past, msg } } -> if msg in this.delivered do this |> next else
    old = for { sender, past_msg } = data <- past, past_msg not in this.delivered, into: empty_set() do
        send this.c, { :crb_deliever, from, msg }; data end;
    send this.c, { :crb_deliver, sender, past_msg }
    this |> delivered_union(old) |> delivered_put(msg) |> past.append(old ++ [{from, msg}]) |> next()
// CRB with vector clock:
{ :bind, c, rb, pnum, ps } ->
    %{ c: c, rb: rb, pnum: pnum, rb_count: 0, pending: empty_set(), vc: init_map(ps, 0)} |> next()
{ :crb_broadcast, msg } -> vc2 = Map.put(this.vc, this.pnum, this.rb_count)
    send this.rb, { :rb_broadcast, { :crb_data, msg, vc2 } }; this |> rbs(this.rb_count+1) |> next()
{ :rb_deliver, sender, { :crb_data, msg, vc } } -> this |> pending_put({ sender, msg, vs }) |> check() |> next()
def check(this) do case Enum.find(this.pending, fn {_,_,vc} -> vc <= this.vc end) do
    { sender, msg, vc } = data -> send this.c, { :crb_deliver, sender, msg }
        this |> vc_elem_put(sender, this.vc[sender]+1) |> pending.delete(data) |> check()
    _otherwise -> this
```

5 Consensus

- **Symmetric**: Leader-less, active replication, all servers equal roles, clients contact any server. **Asymmetric**: Leader-based, passive replication, one server in charge , clients contact leader. Asymmetric is more efficient as it decomposes the problem (normal operation + leader changes) and simplifies normal operation (no conflict).
- **Validity(S)**: If a process decides a value, that value was proposed by some process
- **Integrity (S)**: A process decides one value at most
- **Termination (L)**: Each correct process eventually decides
- **Agreement (L)**: No two correct processes decide different values (Regular Consensus)
- **Uniform Agreement (L)**: No two processes decide different values (Uniform Consensus)
- **FLP Impossibility**: In an asynchronous system with even one possible faulty process, any consensus protocol has the possibility of non-termination even if no process crashes. (Consensus problem cannot be solved in a completely asynchronous system) There exists an initial bivalent configuration. Then it's possible to forever remain in a bivalent configuration.

6 Temporal Logic

- **Equivalence**: $\Box(p \wedge q) \equiv \Box p \wedge \Box q$, $\quad \diamond(p \vee q) \equiv \diamond p \vee \diamond q$, $\quad \bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$, $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$, $\quad p\mathcal{U}(q \vee r) \equiv (p\mathcal{U}q) \vee (p\mathcal{U}r)$, $\quad (p \wedge q)\mathcal{U}r \equiv (p\mathcal{U}r) \wedge (q\mathcal{U}r)$
- **Duals**: $\neg \diamond p \equiv \Box\neg p$, $\quad \neg\Box p \equiv \diamond\neg p$, $\quad \neg \bigcirc p \equiv \bigcirc\neg p$ **Idempotency**: $\Box p \equiv \Box\Box p$, $\quad \diamond p \equiv \diamond \diamond p$, $\quad p\mathcal{U}(q\mathcal{U}r) \equiv (p\mathcal{U}q)\mathcal{U}r \equiv p\mathcal{U}r$
- **Absorption**: $\Box \diamond p \equiv \diamond\Box \diamond p$, $\quad \diamond\Box p \equiv \Box \diamond \Box p$ **Other**: $\texttt{TRUE}\ \mathcal{U}p \equiv \diamond p$

- **Weak Fairness**: $\diamond\Box\underline{A} \Rightarrow \Box\diamond A$ **Strong Fairness**: $\Box\diamond\underline{A} \Rightarrow \Box\diamond A$ **Absolute Fairness**: $\Box\diamond A$

Name	Symbol	Equivalence	Explanation
Next	$\bigcirc p$	$p@(t+1)$	p is true in the next moment
Always	$\Box p$/`[]p`	$\forall t' : (t' \geq t) \Rightarrow p@t'$	p is true now and in all future states
Eventually	$\diamond p$/`<>p`	$\exists t' : (t' \geq t) \wedge p@t'$	p is true now or in some future state
Until	$p\mathcal{U}q$	$\exists t' : ((t' \geq t) \wedge q@t')$ $\wedge(\forall s : (t \leq s < t') \Rightarrow p@s)$	p is true now and continuously until q becomes true in some future state
Leads To	`p~>q`	$\Box(p \Rightarrow \diamond q)$	Always if p is true then eventually q will be true
Always Eventually	$\Box\diamond p$		p will be true infinitely often
Eventually Always	$\diamond\Box p$		There will be a momemnt p will be true and remain true

7 TLA+

- **Boolean**: `BOOLEAN, TRUE, FALSE, ~x, x = y, x # y x => y, x <=> y`. **Integers**: `EXTENDS Integers, a..b, x+y, x-y, x*y, x<y, x<=y, x>y, x>=y`. **Strings**: `STRINGS (*infinite set of all strings*), "abc"`
- **Finite Sets**: `EXTENDS FiniteSets, {a,b}, Cardinality(S), x \in S, x \notin S, S \subseteq T, S \union T, S \intersect T, S \ T (*set difference*), {x \in S: P(x)} (*set filter*), {e: x \in S} (*set map*)`
- **Functions**: `[x \in S |-> e], f[x], DOMAIN f, [f EXCEPT ![x]=e], [S->T] (*set of all functions of S to T*)`
- **Records**: `[x |-> e1, y |-> e2, ..], r.x, [r EXCEPT !.x=e], [x:S,..] (*set of all records with x to set S ..*)`
- **Sequences**: `EXTENDS Sequences, <<a,b>>, t[i], s \o t (*concat*), Len(s), Append(s,x), Head(s)`
- **Tuples**: `<<a,b>>, t[i], S \X T (*set of all tuples S cartesian product T*)`
- **Quantifiers**: `\A x \in S: e, \E x \in S: e, CHOOSE var \in S: e`
- **Control**: `LET x == e1 IN e2, IF p THEN e1 ELSE e2`
- **Actions**: `[A]_v (*stuttering action A or all v unchanged in successor*), <<A>>_v (*non-stuttering action A or some v changed in successor*), ENABLED A (*true if pre conditions true*)`
- **Temporal Logic**: `[]A, <>A, A1 ~> A2, WF_v(A), SF_v(A)`

```
---- MODULE Name ----
EXTENDS Integers, FiniteSets, Sequences // CONSTANTS (*defined in .cfg file*) // VARIABLES ...
Vars == <<...>> // Type == ... // Typed == []Type // Fair == WF_vars(Next)
Spec == Init /\ [][Next]_Vars /\ Fair // NotDeadlock == [](ENABLED Next)
====
.cfg file: SPECIFICATION Spec // PROPERTY Typed // CONSTANTS X = 1
```

8 RAFT

```
Next == \/ \E i \in Server: Restart(i) \/ \E i \in Server: Timeout(i) \/ \E i,j \in Server: RequestVote(i,j)
  \/ \E i \in Server: BecomeLeader(i) \/ \E i \in Server: \E v \in Value: ClientRequest(i,v)
  \/ \E i \in Server: AdvanceCommitIndex(i) \/ \E i,j \in Server: AppendEntries(i,j)
  \/ \E m \in DOMAIN messages : Receive(m) \/ \E m \in DOMAIN messages : DuplicateMessage(m)
  \/ \E m \in DOMAIN messages : DropMessage(m)
DuplicateMessage(m) == Send(m) /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
DropMessage(m) == Discard(m) /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
Restart(i) == /\ state' = [state EXCEPT ![i]=Follower] /\ votesResponded' = [votesResponded EXCEPT ![i]={}]
  /\ votesGranted' = [votesGranted EXCEPT ![i]={}] /\ nextIndex' = [nextIndex EXCEPT ![i]=[j \in Server |-> 1]]
  /\ matchIndex' = [matchIndex EXCEPT ![i]=[j \in Server |-> 0]] /\ commitIndex' = [commitIndex EXCEPT ![i]=0]
  /\ UNCHANGED <<messages, currrentTerm, votedFor, log>>
Timeout(i) == /\ state[i] \in {Follower,Candidate} /\ state' = [state EXCEPT ![i]=Candidate]
  /\ currentTerm' = [currentTerm EXCEPT ![i]=currentTerm[i]+1] /\ votedFor' = [votedFor EXCEPT ![i] = Nil]
  /\ votesResponded' = [voteResponded EXCEPT ![i]={}] /\votesGranted' = [votesGranted EXCEPT ![i]={}]
  /\ UNCHANGED <<messages, leaderVars, logVars>>
RequestVote(i,j) == /\ state[i] = Candidate /\ j \notin votesResponded[i]
  /\ Send([mtype |-> RequestVoteRequest, mterm |-> currentTerm[i], mlastLogTerm |-> LastTerm(log[i]),
   mlastLogIndex |-> Len(log[i]),  msource |-> i, mdest |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
BecomeLeader(i) == /\ state[i] = candidate /\ votesGranted[i] \in Quorum /\ state' = [state EXCEPT ![i]=Leader]
  /\ nextIndex' = [nextIndex EXCEPT ![i]=[j \in Server |-> Len(log(i))+1]]
  /\ matchIndex' = [matchIndex EXCEPT ![i]=[j \in Server |-> 0]]
  /\ UNCHANGED <<messages, currentTerm, votedFor, candidateVars, logVars>>
ClientRequest(i,v) == /\ state[i] = Leader /\ LET entry == [term |-> currentTerm[i], value |-> v] IN
    LET newLog == Append(log[i],entry) IN log' = [log EXCEPT ![i]=newLog]
  /\ UNCHANGED <<messages, serverVars, candidateVars, leaderVars, commitIndex>>
AdvanceCommitIndex(i) == /\ state[i] = Leader /\ LET Agree(id) == {i} \cup {k \in Server: matchIndex[i][k]>=id}
  agreeIndexes == {id \in 1..Len(log[i]): Agree(index) \in Quorum}
  newCommitIndex == IF agreeIndex # {} /\ log[i][Max(agreeIndexes)].term = currentTerm[i] THEN Max(agreeIndexes)
    ELSE commitIndex[i] IN commitIndex' = [commitIndex EXCEPT ![i]=newCommitIndex]
    /\ UNCHANGED <<messages, serverVars, candidateVars, leaderVars, log>>
AppendEntries(i,j) == /\ i # j /\ state[i] = Leader /\ LET prevLogIndex == nextIndex[i][j]-1
```

```
    IN LET prevLogTerm == IF prevLogIndex > 0 THEN log[i][prevLogIndex].term ELSE 0
    IN LET lastEntry == Min({Len(log[i]), nextIndex[i][j]})
    IN LET entries == SubSeq(log[i], nextIndex[i][j], lastEntry)
    IN Send([mtype |-> AppendEntriesRequest, mterm |-> currentTerm[i], mprevLogIndex |-> prevLogIndex,
      mprevLogTerm |-> prevLogTerm, mentries |-> entries, mcommitIndex = min({commitIndex[i], lastEntry}),
      msource |-> i, mdest |-> j])
  /\ UNCHANGED <<serverVars, candiateVars, leaderVars, logVars>>
Receive(m) == LET i == m.mdest IN LET j == m.msource IN
  \/ UpdateTerm(i,j,m) \/ (/\ m.mtype == RequestVoteRequest /\ HandleRequestVoteRequest(i,j,m))
  \/ (/\ m.mtype == RequestVoteResponse /\ (DropStaleResponse(i.j.m) \/ HandleRequestVoteResponse(i,j,m)))
  \/ (/\ m.mtype == AppendEntriesRequest /\ HandleAppendEntriesRequest(i,j,m))
  \/ (/\ m.mtype == AppendEntriesResponse /\ (DropStaleResponse(i,j,m) \/ HandleAppendEntriesResponse(i,j,m)))
UpdateTerm(i,j,m) == /\ m.mterm > currentTerm[i] /\ currentTerm' = [currentTerm EXCEPT ![i]=m.mterm]
  /\ state' = [state EXCEPT ![i]=Follower] /\ votedFor' == [votedFor EXCEPT ![i]=Nil]
  /\ UNCHANGED <<messages, candidateVars, leaderVars, logVars>>
DropStaleResponse(i,j,m) == /\ m.mterm < currentTerm[i] /\ Discard(m)
  /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
HandleRequestVoteRequest(i,j,m) == LET logOk == \/ m.mlastLogTerm > LastTerm(log[i])
      \/ (m.mlastLogTerm = LastTerm(log[i]) /\ m.mlastLogIndex >= Len(log[i]))
  IN LET grant == /\ m.mterm = currentTerm[i] /\ logOk /\ votedFor[i] \in {Nil,j}
  IN /\ m.mterm <= currentTerm[i]
  /\ ((grant /\ votedFor' = [votedFor EXCEPT ![i]=j]) \/ (~grant /\ UNCHANGED votedFor))
  /\ Reply([mtype |-> RequestVoteResponse, mterm |-> currentTerm[i], mvoteGranted |-> grant,
    msource |-> i, mdest |-> j], m)
  /\ UNCHANGED <<state, currentTerm, candidateVars, leaderVars, logVars>>
HandleRequestVoteResponse(i,j,m) == /\ m.mterm = currentTerm[i]
  /\ votesResponded' = [votesResponded EXCEPT ![i]=votesResponse[i] \union {j}]
  /\ ((m.mvoteGranted /\ votesGranted' = [votesGranted EXCEPT ![i]=votesGranted[i] \union {j}])
    \/ (~m.mvoteGranted /\ UNCHANGED <<votesGranted>>))
  /\ Discard(m) /\ UNCHANGED <<serverVars, votedFor, leaderVars, logVars>>
HandleAppendEntriesRequest(i,j,m) == LET logOK == \/ m.prevLog = 0
    \/ /\ m.mprevLogIndex > 0 /\ m.mprevLogIndex <= Len(log[i]) /\ m.mprevLogTerm = log[i][m.mprevLogIndex].term
  IN /\ m.mterm <= currentTerm[i] /\
    \/ /\ (m.mterm < currentTerm[i] \/ (m.mterm = currentTerm[i] /\ state[i] = Follower /\ ~logOk))
       /\ Reply([mtype |-> AppendEntriesResponse, mterm |-> currentTerm[i], msuccess |-> FALSE,
         mmatchIndex |-> 0, msource |-> i, mdest |-> j], m)
       /\ UNCHANGED <<currentTerm, votedFor, logVars, messages>> (*reject*)
    \/ /\ m.mterm = currentTerm[i] /\ state[i] = Candidate /\ state' = [state EXCEPT ![i]=Follower]
       /\ UNCHANGED <<currentTerm, votedFor, logVars, messages>> (*return to follower*)
    \/ /\ m.mterm = currentTerm[i] /\ state[i] = Follower /\ logOk /\ LET index == m.prevLogIndex + 1
       IN \/ /\ m.mentries # <<>> /\ Len(log[i]) = m.mprevLogIndex
             /\ log' = [log EXCEPT ![i]=Append(log[i], m.mentries[1])]
             /\ UNCHANGED <<serverVars, commitIndex, messages>> (**no conflict: append entry)
          \/ /\ m.mentries # <<>> /\ Len(log[i]) >= index /\ log[i][index].term # m.mentries[1].term
             /\ LET new == [index2 \in 1..(Len(log[i])-1) |-> log[i][index2]] IN log' = [log EXCEPT ![i]=new]
             /\ UNCHANGED <<serverVars, commitIndex, messages>> (*conflict: remove 1 entry*)
          \/ /\ \/ m.mentries = <<>>
                \/ /\ m.mentries # <<>> /\ Len(log[i]) >= index /\ log[i][index].term = m.mentries[1].term
             /\ commitIndex' = [commitIndex EXCEPT ![i]=m.mcommitIndex]
             /\ Reply([mtype |-> AppendEntriesResponse, mterm |-> currentTerm[i], msuccess |-> TRUE,
               mmatchIndex |-> m.prevLogIndex + Len(m.mentries), msource |-> i, mdest |-> j], m)
             /\ UNCHANGED <<serverVars, log>> (*processed request send response*)
HandleAppendEntriesResponse(i,j,m) == /\ m.mterm = currentTerm[i]
  /\ \/ /\ m.msuccess /\ nextIndex' = [nextIndex EXCEPT ![i][j]=m.mmatchIndex+1]
        /\ matchIndex' = [matchIndex ![i][j]=m.mmatchIndex] (*successful*)
     \/ /\ ~m.msuccess /\ nextIndex' = [nextIndex EXCEPT ![i][j]=Max({nextIndex[i][j]-1,1})]
        /\ UNCHANGED <<matchIndex>> (*not successful*)
  /\ Discard(m) /\ UNCHANGED <<serverVars, candidateVars, logVars>>
(*Properties*)
currentTermMonotonicallyIncreases == \A i \in Server: currentTerm[i] <= currentTerm'[i]
electionSafety == \A e, f \in elections: e.eterm = f.eterm => e.eleader = f.eleader (*<= 1 leader per term*)
leaderAppendOnly == \A e \in elections: currentTerm[e.leader] = e.term => \A i \in 1..Len(log[e.leader]):
  log'[e.leader][i] = log[e.leader][i] (*leader log grow monotonically during term*)
stateMachineSafety == \A i \in Server: /\ commitIndex <= Len(log[i])
  /\ \A <<index, term>> \in log[i]: index <= commitIndex[i] => <<index, term>> \in committed(currentTerm[i])
(*servers only apply committed entries*)
committed(t) == {<<index, term>>: \A e \in elections: e.eterm > t => <<index, term>> \in e.elog}
(*entry is committed at t if present in every leader's log following t*)
```