MODULE *EDHotStuff*

EXTENDS *Integers*, *Sequences*, *FiniteSets*, *TLC*

VERSION 0.2

This module is a specification based on Algorithms 4, and 5, in the *HotSuff* white paper. In as much as is possible it uses the same notation as the published algorithms. There are several parts that are implementation dependent, namely how the parent chains are implemented, and how leaders are selected. I have chosen the simplest possible implemetations in this version.

No attempt has been made yet to define any invariants

CONSTANT *Value*, *Acceptor*, *FakeAcceptor*

CONSTANTS

ByzQuorum not currently used

A *Byzantine* quorum is set of acceptors that includes a quorum of good ones. In the case that there are $2f+1$ good acceptors and f bad ones, a *Byzantine* quorum is any set of $2f+1$ acceptors.

$ViewNums \triangleq Nat$

$None \triangleq \text{CHOOSE } v : v \notin Value \land v \notin Acceptor \land v \notin FakeAcceptor$

The following operator definitions allow us to form a *TYPE_OK* invariant. Note that *Leaf* needs to be a recursive definition -

RECURSIVE *Leaf*

$QCElement \triangleq [viewNum : ViewNums,\ node : Leaf,\ sig : Acceptor]$

$Leaf \triangleq [parent : Acceptor,\ cmd : Value,\ justify : \{\},\ height : \{\}]$

$Message \triangleq [type : \text{“generic”},\ viewnum : ViewNums,\ node : Leaf,\ justify : \langle\rangle]$

$\cup\ [type : \text{“newview”},\ viewnum : ViewNums,\ node : Leaf,\ justify : \langle\rangle]$ always sent to leader

QC = ???? *FIXME*

$EmptyQC \triangleq \{\}$

$EmptyNode \triangleq [parent : \{\},\ cmd : \{\},\ justify : \{\},\ height : \{\}]$

$EmptyV \triangleq [x \in \{\} \mapsto \text{TRUE}]$ empty function (nothing is TRUE !)

$ByzAcceptor \triangleq Acceptor \cup FakeAcceptor$

$$genesis\_null \triangleq [parent \mapsto \langle \langle\rangle \rangle,\ cmd \mapsto \text{“genesis\_null”},\ justify \mapsto [viewnum \mapsto 0,\ node \mapsto \langle\rangle,\ sig \mapsto 1]]$$

$$
\begin{array}{lll}
 & & \quad height \quad \mapsto 0 \\
 & &] \\
genesis\_bpp & \triangleq & [parent \mapsto \langle \\
 & & \quad \langle genesis\_null \rangle \\
 & & \quad \rangle, \\
 & & \quad cmd \mapsto \text{“genesis\_bpp”}, \\
 & & \quad justify \mapsto [viewnum \mapsto 0, \\
 & & \quad\quad node \mapsto genesis\_null, \\
 & & \quad\quad sig \mapsto 1 \\
 & & \quad], \\
 & & \quad height \mapsto 1 \\
 & &] \\
genesis\_bp & \triangleq & [parent \mapsto \langle \\
 & & \quad \langle genesis\_bpp \rangle \\
 & & \quad \rangle, \\
 & & \quad cmd \mapsto \text{“genesis\_bp”}, \\
 & & \quad justify \mapsto [viewnum \mapsto 0, \\
 & & \quad\quad node \mapsto genesis\_bpp, \\
 & & \quad\quad sig \mapsto 1 \\
 & & \quad], \\
 & & \quad height \mapsto 2 \\
 & &] \\
genesis\_b & \triangleq & [parent \mapsto \langle \\
 & & \quad \langle genesis\_bp \rangle \\
 & & \quad \rangle, \\
 & & \quad cmd \mapsto \text{“genesis\_b”}, \\
 & & \quad justify \mapsto [viewnum \mapsto 0, \\
 & & \quad\quad node \mapsto genesis\_bp, \\
 & & \quad\quad sig \mapsto 1 \\
 & & \quad], \\
 & & \quad height \mapsto 3 \\
 & &] \\
\text{genesis} & & \\
b0 & \triangleq & [parent \mapsto \langle \\
 & & \quad \langle genesis\_b \rangle \\
 & & \quad \rangle, \\
 & & \quad cmd \mapsto \text{“b0”}, \\
 & & \quad justify \mapsto [viewnum \mapsto 0, \\
 & & \quad\quad node \mapsto genesis\_b,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\quad sig \quad\quad \mapsto 1 \\
\qquad\qquad\qquad\quad], \\
\qquad\qquad height \;\mapsto 4 \\
\\
\qquad\qquad]
\end{array}
$$

$$NextValue(sent) \triangleq \text{CHOOSE } x \in Value : x \notin sent$$

TODO

We can make $OnCommit(node)$ recursive if we define it as an operator

RECURSIVE $OnCommit(\_)OnCommit(self) \triangleq \land$ IF $bexec[self].height <$ $node[self].height$ /* define node globally above
THEN $\land node[self] = node[self].parent[1][1]$
$\land OnCommit(self)$
$\backslash * \land EXCECUTE(node.cmd)$
$\land$ UNCHANGED $\langle b0, values, voteChannel,$
$newViewChannel,$
$proposalChannel,$
$lastReadProposal, vheight,$
$block, bexec, qchigh,$
$bleaf, curView, V, b, bp,$
$bpp,$
$emptyQueueFlag, qc, mvote,$
$mprop, mnew, parent,$
$VEntry, bpropose, c, qch,$
$vb, tmp, key, val,$
$bstar\rangle$

***

--algorithm *EDHotStuff* **{**

variables

We first construct the root node $b0$. Just how the parent part of a *Leaf* is constructed is implementation dependent, so for now we define it as follows -

A parent is an ordered list $\langle [node], \ldots \rangle$ of ancestors.

$b0$ is the same genesis node known by all correct replicas.

The *HS* paper suggests using hashes of nodes to implement the parent field, with a lookup table. For this specification, we'll just use the actual node, as we can't easily use any crypto in TLA+. We can place dummy parent records in the list to extend it to the correct height.

$Ledger = [a \in Acceptor \mapsto \langle\rangle],$ A ledger for each *Acceptor*
$sentValues = \{\},$

communication channels
$voteChannel = [a \in Acceptor \mapsto \langle\rangle],$
$newViewChannel = [a \in Acceptor \mapsto \langle\rangle],$
$proposalChannel = \langle\rangle,$ global, as this is for broadcasts

$lastReadProposal \quad = [a \in Acceptor \mapsto 0],$

per process state variables (as per *HS* paper)

$vheight$	$= [a \in Acceptor \mapsto 0],$	height of last voted node.
$block$	$= [a \in Acceptor \mapsto EmptyNode],$	locked node (similar to $lockedQC$).
$bexec$	$= [a \in Acceptor \mapsto EmptyNode],$	last executed node.
$qchigh$	$= [a \in Acceptor \mapsto \langle\rangle],$	highest known QC (similar to $genericQC$) kept by a
$bleaf$	$= [a \in Acceptor \mapsto EmptyNode],$	leaf node kept by *PaceMaker*
$curView$	$= [a \in Acceptor \mapsto 0],$	
V	$= [a \in Acceptor \mapsto EmptyV],$	

the 3-chain

b	$= [a \in Acceptor \mapsto EmptyNode],$
bp	$= [a \in Acceptor \mapsto EmptyNode],$
bpp	$= [a \in Acceptor \mapsto EmptyNode],$

internal variables

$nextEventFlag$	$= [a \in Acceptor \mapsto \text{FALSE}],$
$nextEventLOCK$	$= [a \in Acceptor \mapsto \text{FALSE}],$
qc	$= [a \in Acceptor \mapsto \langle\rangle],$
$mvote$	$= [a \in Acceptor \mapsto \langle\rangle],$
$mprop$	$= [a \in Acceptor \mapsto \langle\rangle],$
$mnew$	$= [a \in Acceptor \mapsto \langle\rangle],$
$VEntry$	$= [a \in Acceptor \mapsto \{\}],$

define {

$CreateLeaf(p,\ v2,\ qc1,\ h) \triangleq [parent \mapsto p,\ cmd \mapsto v2,\ justify \mapsto qc1,\ height \mapsto h]$

$GETLEADER \triangleq$ "a1" *FROB* − should select fro *Acceptor* set

bend extends *bstart* ⇒ TRUE iff bend has an ancestor (parent) in common with *bstart*

$Extends(bstart,\ bend) \triangleq$ TRUE *TODO*

$Range(T) \triangleq \{T[x] : x \in \text{DOMAIN } T\}$

}

The following are all macros which get substituted in-line in the code. This is done to make the code readable but without the overhead of calling a procedure. There lots of commented out print statements that are useful for debuggering.

macro *SendNewView*(*to*, *qc*) **{**

$newViewChannel[to] := Append(newViewChannel[to], \langle[type \mapsto$ "newview",
$viewnum \mapsto curView[self],$
$node \mapsto EmptyNode,$
$justify \mapsto qc],$
$sig \mapsto self\rangle)$ **;**

}

macro *ReceiveNewView*(*m*) **{**

```
    await newViewChannel[self] ≠ ⟨⟩ ;
        m := Head(newViewChannel[self]) ;
        newViewChannel[self] := Tail(newViewChannel[self]) ;
}

macro SendVoteMsg( to, n ) {
    voteChannel[to] := Append(voteChannel[to], ⟨[type ↦ "generic",
                                                viewnum ↦ curView[self],
                                                node ↦ n,
                                                justify ↦ {},   FIXME ???
                                                sig ↦ 1]⟩) ;    FIXME ???
}

macro ReceiveVote( m ) {
    await voteChannel[self] ≠ ⟨⟩ ;
        m := Head(voteChannel[self]) ;
        voteChannel[self] := Tail(voteChannel[self]) ;
}

macro BroadcastProposal( n ) {
    proposalChannel := Append(
                                ⟨[type ↦ "generic",
                                viewnum ↦ curView[self],
                                node ↦ n,
                                justify ↦ {},
                                sig ↦ self]⟩,
                                proposalChannel
                                ) ;
}

macro ReceiveProposal( m, lr ) {
    await Len(proposalChannel) > lr ;
        m := Head(proposalChannel) ;
}

macro PMUpdateQCHigh( qcphigh ) {
    if ( qcphigh.node.height > qchigh[self].node.height ) {
        qchigh[self] := qcphigh ;
        bleaf[self] := qchigh[self].node ;
     } ;
}

macro EXECUTE( b ) {
    Ledger[self] := Append(Ledger[self], ⟨b.cmd⟩) ;
    print ⟨"----------------> Ledger := ", Ledger[self]⟩ ;
}
```

```
macro DoNextEvent( ) {
    await nextEventFlag[self] = TRUE ∧ nextEventLOCK[self] = FALSE ;
        nextEventFlag[self] := FALSE ;
        call PMOnBeat( "loop" ) ;
}

macro JustifyNode( bRet, b ) {
    bRet := b.justify.node ;
}

macro PMOnNextSyncView( ) {      not currently tested
    SendNewView(GETLEADER(), qchigh[self]) ;
}

macro PMOnReceiveNewView( ) {    not currently tested
    ReceiveNewView(mnew[self]) ;
    PMUpdateQCHigh(mnew[self].justify) ;
    curView[self] := curView[self] + 1      ;
}

procedure OnPropose( bpropose, c, qch ) {
    onp1:
        bpropose := CreateLeaf(bpropose, c, qch, (bpropose.height + 1)) ;
        BroadcastProposal(bpropose) ;
    onp_return:
        return ;
}

procedure OnReceiveProposal( ) {
    orp1:
        ReceiveProposal(mprop[self], lastReadProposal[self]) ;
        lastReadProposal[self] := lastReadProposal[self] + 1  ;

        if ( (mprop[self].node.height > vheight[self] ∧ Extends(mprop[self].node, block[self]))
              ∨ mprop[self].node.justify.node.height > block[self].height ) {
                vheight[self] := mprop[self].node.height          ;
                SendVoteMsg(GETLEADER, mprop[self].node) ;
         } ;
        call Update(mprop[self].node) ;
    orp_return:
        return ;
}

procedure PMOnBeat( debug )
variables tmp
{
```

```
    pmob1:
        if (self = GETLEADER){
                    tmp := NextValue(sentValues) ;
                    call OnPropose(bleaf[self], tmp, qchigh[self]) ;
    pmob2:
                    sentValues := sentValues ∪ {tmp} ;
        } ;
    pmob_return:
        return ;
}

procedure MakeQC(vb){
    vb is a set {[votemessage], ...} for node b − FIXME
    mqc1:
        qc[self] := [viewnum ↦ vb.justify.viewnum,    FIXME
                     node    ↦ vb,
                     sig     ↦ vb.justify.sig] ;      FIXME − maybe better as a set of sigs as per paper
    mcq_return:
        return ;  return in qc[self]
}

procedure AddItemToV(key, val){
    aitv1:
        if (key ∈ DOMAIN V[self])
            VEntry[self] := V[self][key] ;
        else
            VEntry[self] := {} ;
    aitv2:
        VEntry[self] := VEntry[self] ∪ {val} ;
        V[self] := [x ∈ DOMAIN V[self] ∪ {key} ↦ VEntry[self]] @@ V[self] ;

        print⟨"V[key] = ", V[self][key]⟩;
        return ;
}

procedure OnReceiveVote(){
    orv1:
        nextEventLOCK[self] := TRUE ;
        ReceiveVote(mvote[self]) ;
        goto orv4 ;  ********** !!
    orv2:
        avoid duplicates
        if (mvote[self][1] ∈ V[self][mvote[self][1].node]){
            goto orv_return ;
        } ;
    orv3:
```

```
        call AddItemToV(mvote[self][1].node, mvote[self][1]) ;
    orv4:
        if (mvote[self][1].sig ≤ Cardinality(Acceptor) − Cardinality(FakeAcceptor)){  (n − f)
    orv4a:
            call MakeQC(mvote[self][1].node) ;
    orv5:
            PMUpdateQCHigh(qc[self]) ;
        } ;
    orv_return:
        nextEventLOCK[self] := FALSE ;
        return ;
}

procedure OnCommit(node){
    oc1:
        In the HotSuff algorithm 4. we would recursively call OnCommit(node.parent)
        here, but unfortunately TLA+ does not allow recursive procedure calls, so we do the
        following -
        if (bexec[self].height < node.height){
            SIMULATED RECURSION with node.parent
            if (bexec[self].height < node.parent[1][1].height){
                recurse again ????
                if (node.parent[1][1].parent[1][1] ≠ genesis_null){
                    if (bexec[self].height < node.parent[1][1].parent[1][1]){
                        print ⟨ "oc: second recursion" ⟩ ;
                    } ;
                } ;
                EXECUTE(node.parent[1][1]) ;
                nextEventFlag[self] := TRUE ;
            } ;
        } ;
    oc_return:
        return ;
}

procedure Update(bstar){
Called whenever a proposal is received, whether it is voted for or not
    u1:
        nextEventLOCK[self] := TRUE ;
    u2:
        JustifyNode(bpp[self], bstar) ;
    u3:
        JustifyNode(bp[self],  bpp[self]) ;
    u4:
        JustifyNode(b[self],    bp[self]) ;
```

```
        u5:
            PMUpdateQCHigh(bstar.justify) ;     pre-commit-phase on bpp
        u6:
            if (bp[self].height > block[self].height){
                block[self] := bp[self] ;       commit-phase on bp
            }else {
               goto u_return ;
            } ;
        u8:
            if (bpp[self].parent[1][1] = bp[self]
                  ∧ bp[self].parent[1][1] = b[self])
            {
                call OnCommit(b[self]) ;
        u9:
                bexec[self] := b[self] ;        decide-phase on b
            } ;
        u_return:
            nextEventLOCK[self] := FALSE ;
            return ;
    }
```

We now start a process for each *Acceptor*.

```
    process (a ∈ ByzAcceptor){
        p1:
            sentValues := {} ;

             would have been set in orp:
            vheight[self] := b0.height ;

             would have been set in pmob:
            bleaf[self] := b0 ;

             would have been set in orv:
        p2:
            qchigh[self] := [viewnum ↦ 0,
                             node ↦ b0,
                             sig ↦ 1
                            ] ;

             these would have been set in u:
            block[self] := genesis_bpp ;
            bexec[self] := genesis_null ;

             genesis_null is implicitly stored on the ledger

            call PMOnBeat("p2") ;   propose the first real cmd
```

```
        loop:
            while (TRUE){
        p3:
                    either call OnReceiveVote()
                    or receiveprop: call OnReceiveProposal()
                    or nextevent: DoNextEvent()
                     or PMOnReceiveNewView() \* not tested
                };
    }
}
 *
```

* Modification History
* Last modified Sat *Mar* 07 15:31:15 *GMT* 2020 by *steve*
* Created *Thu Feb* 20 12:24:16 *GMT* 2020 by *steve*