MODULE *EDHotStuff*

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

VERSION 1.0

This module is a specification based on Algorithms 4, and 5, in the *HotStuff* white paper.

AUTHOR = "Maofan *Yin*, Dahlia *Malkhi*, *Michael K.* Reiter, Guy *Golan Gueta*, *Ittai Abraham*",
TITLE = "*HotStuff*: *BFT* Consensus in the Lens of *Blockchain*", Journal = "*arXiv*: 1803.05069*v*6 [*CS.DC*])", *YEAR* = 2019,

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 implementations in this version.

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

$ViewNums \triangleq Nat$

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

$ByzAcceptor \triangleq Acceptor \cup FakeAcceptor$

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

RECURSIVE *Leaf*

$QCElement \triangleq [viewNum : ViewNums, node : Leaf, sig : ByzAcceptor]$
$Leaf \triangleq [parent : ByzAcceptor, cmd : Value, justify : \{\}, height : \{\}]$

$Message \triangleq [type : \text{"generic"}, viewnum : ViewNums, node : Leaf, justify : \langle\rangle]$
$\cup [type : \text{"newview"}, viewnum : ViewNums, node : Leaf, justify : \langle\rangle]$ always sent to leader

$EmptyQC \triangleq \{\}$
$EmptyNode \triangleq [parent : \{\}, cmd : \{\}, justify : \{\}, height : \{\}]$
$EmptyV \triangleq [x \in \{\} \mapsto \text{TRUE}]$ empty function (nothing is TRUE !)

We construct the genesis node *b*0 as a chain of 4 nodes. When the first real *Value* is proposed the *genesis_bpp* node will be executed, and so on up the chain as more Values are proposed.

$$
\begin{array}{lll}
genesis\_null \triangleq & [parent & \mapsto \langle \\
 & & \quad \langle\rangle \\
 & & \rangle, \\
 & cmd & \mapsto \text{"genesis\_null"}, \\
 & justify & \mapsto [viewnum \mapsto 0, \\
 & & \quad node \mapsto \langle\rangle, \\
 & & \quad sig \mapsto 1 \\
 & &], \\
 & height & \mapsto 0 \\
 &] &
\end{array}
$$

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

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

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

genesis

$$
\begin{array}{lcl}
b0 & \triangleq & [parent \mapsto \langle \\
 & & \quad \langle genesis\_b \rangle \\
 & & \rangle, \\
 & & cmd \mapsto \text{“b0”}, \\
 & & justify \mapsto [viewnum \mapsto 0, \\
 & & \quad node \mapsto genesis\_b, \\
 & & \quad sig \mapsto 1 \\
 & &], \\
 & & height \mapsto 4
\end{array}
$$

]

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

TODO

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

RECURSIVE $OnCommit(\_)OnCommit(self) \triangleq \wedge$ IF $bexec[self].height <$ $node[self].height$ /* define node globally above
THEN $\wedge\ node[self] = node[self].parent[1][1]$
$\wedge\ OnCommit(self)$
* $\wedge\ EXCECUTE(node.cmd)$
$\wedge$ 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.

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

communication channels
$voteChannel = [a \in ByzAcceptor \mapsto \langle\rangle],$
$newViewChannel = [a \in ByzAcceptor \mapsto \langle\rangle],$
$proposalChannel = \langle\rangle,$ global, as this is for broadcasts
$lastReadProposal = [a \in ByzAcceptor \mapsto 0],$

per process state variables (as per *HS* paper)
$vheight = [a \in ByzAcceptor \mapsto 0],$ height of last voted node.
$block = [a \in ByzAcceptor \mapsto EmptyNode],$ locked node (similar to *lockedQC*).

$bexec = [a \in ByzAcceptor \mapsto EmptyNode]$, last executed node.
$qchigh = [a \in ByzAcceptor \mapsto \langle\rangle]$, highest known QC (similar to $genericQC$) kept
$bleaf = [a \in ByzAcceptor \mapsto EmptyNode]$, leaf node kept by $PaceMaker$
$curView = [a \in ByzAcceptor \mapsto 0]$,
$V = [a \in ByzAcceptor \mapsto EmptyV]$,

the 3-chain
$b = [a \in ByzAcceptor \mapsto EmptyNode]$,
$bp = [a \in ByzAcceptor \mapsto EmptyNode]$,
$bpp = [a \in ByzAcceptor \mapsto EmptyNode]$,

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

define {

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

$GETLEADER \triangleq$ "a1" a constant for simplicity

bend extends $bstart \Rightarrow$ TRUE iff bend has an ancestor (parent) in common with $bstart$
$Extends(bstart,\ bend) \triangleq$ TRUE for simplicity we assume that this is always true

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

}

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 $newViewChannel[self] \neq \langle\rangle$ **;**
$m := Head(newViewChannel[self])$ **;**
$newViewChannel[self] := Tail(newViewChannel[self])$ **;**

}

macro *SendVoteMsg*($to,\ n$) **{**
$voteChannel[to] := Append(voteChannel[to], \langle[type \mapsto$ "generic",
$viewnum \mapsto curView[self]$,

```
                                    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){
    mqc1:
        qc[self] := [viewnum ↦ vb.justify.viewnum,
                     node    ↦ vb,
                     sig     ↦ vb.justify.sig] ;
    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]) ;
    orv2:
        call AddItemToV(mvote[self][1].node, mvote[self][1]) ;
    orv3:
        Since we cannot use threshold signatures in TLA+, we just
        count votes.
        if (Cardinality(V[self][mvote[self][1].node]) ≥ Cardinality(Acceptor) − Cardinality(FakeAcceptor))
    orv4:
            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 *Tue Mar* 10 15:16:15 *GMT* 2020 by *steve*
\ * Created *Thu Feb* 20 12:24:16 *GMT* 2020 by *steve*