MODULE *GenericMulticast1*

LOCAL INSTANCE *Commons*
LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *FiniteSets*
LOCAL INSTANCE *TLC*

Number of groups in the algorithm.
CONSTANT *NGROUPS*

Number of processes in the algorithm.
CONSTANT *NPROCESSES*

Set with initial messages the algorithm starts with.
CONSTANT *INITIAL_MESSAGES*

The conflict relation.
CONSTANT *CONFLICTR*(_, _)

ASSUME

Verify that *NGROUPS* is a natural number greater than 0.
$\land\ NGROUPS \in (Nat \setminus \{0\})$
Verify that *NPROCESSES* is a natural number greater than 0.
$\land\ NPROCESSES \in (Nat \setminus \{0\})$

LOCAL $Processes \triangleq \{p : p \in 1 \,..\, NPROCESSES\}$
LOCAL $Groups \triangleq \{g : g \in 1 \,..\, NGROUPS\}$

The module containing the Atomic Broadcast primitive.
VARIABLE *AtomicBroadcastBuffer*
AtomicBroadcast $\triangleq$ INSTANCE *AtomicBroadcast*

The module containing the quasi reliable channel.
VARIABLE *QuasiReliableChannel*
QuasiReliable $\triangleq$ INSTANCE *QuasiReliable* WITH
$INITIAL\_MESSAGES \leftarrow \{\}$

The algorithm's *Mem* structure. We use a separate module.
VARIABLE *MemoryBuffer*
Memory $\triangleq$ INSTANCE *Memory*

VARIABLES

The process local clock.

K,

The set contains previous messages. We use this with the *CONFLICTR* to verify conflicting messages.

$PreviousMsgs$,

The set of delivered messages. This set is not an algorithm requirement. We use this to help check the algorithm's properties.

$Delivered$,

A set contains the processes' votes for the message's timestamp. This structure is implicit in the algorithm.

$Votes$

$vars \triangleq \langle$
K,
$MemoryBuffer$,
$PreviousMsgs$,
$Delivered$,
$Votes$,
$AtomicBroadcastBuffer$,
$QuasiReliableChannel$
$\rangle$

Check if the given message conflict with any other in the *PreviousMsgs*.

$\text{LOCAL } HasConflict(g, p, m1) \triangleq$
$\quad \exists\, m2 \in PreviousMsgs[g][p] : CONFLICTR(m1, m2)$

These are the handlers. The actual algorithm resides here, the lambdas only assert the guarding predicates before calling the handler.

$$
\begin{array}{l}
\text{LOCAL } ComputeGroupSeqNumberHandler(g, p, msg, ts) \triangleq \\
\quad \land \lor \land HasConflict(g, p, msg) \\
\qquad\quad \land K' = [K \text{ EXCEPT } ![g][p] = K[g][p] + 1] \\
\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![g][p] = \{msg\}] \\
\qquad \lor \land \neg HasConflict(g, p, msg) \\
\qquad\quad \land PreviousMsgs' = [PreviousMsgs \text{ EXCEPT } ![g][p] = \\
\qquad\qquad\quad PreviousMsgs[g][p] \cup \{msg\}] \\
\qquad\quad \land \text{UNCHANGED } K \\
\quad \land \lor \land Cardinality(msg.d) > 1 \\
\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{“S1”}, K'[g][p] \rangle) \\
\qquad\quad \land QuasiReliable!Send(\langle msg, g, K'[g][p] \rangle) \\
\qquad \lor \land Cardinality(msg.d) = 1 \\
\qquad\quad \land Memory!Insert(g, p, \langle msg, \text{“S3”}, K'[g][p] \rangle) \\
\qquad\quad \land \text{UNCHANGED } QuasiReliableChannel \\
\quad \land \text{UNCHANGED } \langle Delivered, Votes \rangle
\end{array}
$$

$$
\begin{array}{l}
\textsc{local}\ \mathit{SynchronizeGroupClockHandler}(g,\ p,\ m,\ \mathit{tsf})\ \triangleq \\
\quad \land\ \lor\ \land\ \mathit{tsf} > K[g][p] \\
\qquad\qquad \land\ K' = [K\ \textsc{except}\ ![g][p] = \mathit{tsf}] \\
\qquad\qquad \land\ \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs}\ \textsc{except}\ ![g][p] = \{\}] \\
\qquad\ \ \lor\ \land\ \mathit{tsf} \leq K[g][p] \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle K,\ \mathit{PreviousMsgs} \rangle \\
\quad \land\ \lor\ \land\ \exists \langle n,\ s,\ ts \rangle \in \mathit{MemoryBuffer}[g][p] : s = \text{“S1”} \land m = n \\
\qquad\qquad \land\ \mathit{Memory}!\mathit{Insert}(g,\ p,\ \langle m,\ \text{“S3”},\ \mathit{tsf} \rangle) \\
\qquad\ \ \lor\ \textsc{unchanged}\ \mathit{MemoryBuffer} \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{QuasiReliableChannel},\ \mathit{Delivered},\ \mathit{Votes} \rangle
\end{array}
$$

$$
\begin{array}{l}
\textsc{local}\ \mathit{GatherGroupsTimestampHandler}(g,\ p,\ \mathit{msg},\ \mathit{ts},\ \mathit{tsf})\ \triangleq \\
\quad \land\quad \lor\ \land\ \mathit{ts} < \mathit{tsf} \\
\qquad\qquad\quad \land\ \mathit{AtomicBroadcast}!\mathit{ABroadcast}(g,\ \langle \mathit{msg},\ \text{“S2”},\ \mathit{tsf} \rangle) \\
\qquad\quad \lor\ \textsc{unchanged}\ \mathit{AtomicBroadcastBuffer} \\
\quad \land\quad \mathit{Memory}!\mathit{Insert}(g,\ p,\ \langle \mathit{msg},\ \text{“S3”},\ \mathit{tsf} \rangle) \\
\quad \land\quad \textsc{unchanged}\ \langle K,\ \mathit{PreviousMsgs},\ \mathit{Delivered} \rangle
\end{array}
$$

Executes when process P receives a message M from the Atomic Broadcast primitive and M is in P's memory. This procedure is extensive, with multiple branches based on the message's state and destination. Let's split the explanation.

When M's state is $S0$, we first verify if M conflicts with messages in the *PreviousMsgs* set. If a conflict exists, we increase P's local clock by one and clear the *PreviousMsgs* set.

When message M has a single group as the destination, it is already in its desired destination and is synchronized because we received M from Atomic Broadcast primitive. P stores M in memory with state $S3$ and timestamp with the current clock value.

When M includes multiple groups in the destination, the participants must agree on the final timestamp. When M's state is $S0$, P will send its timestamp proposition to all other participants, which is the current clock value, and update M's state to $S1$ and timestamp. If M's state is $S2$, we are synchronizing the local group, meaning we may need to leap the clock to the M's received timestamp and then set M to state $S3$.

$$
\begin{array}{l}
\mathit{ComputeGroupSeqNumber}(g,\ p)\ \triangleq \\
\quad \land\ \mathit{AtomicBroadcast}!\mathit{ABDeliver}(g,\ p, \\
\qquad\quad \textsc{lambda}\ t : t[2] = \text{“S0”} \land \mathit{ComputeGroupSeqNumberHandler}(g,\ p,\ t[1],\ t[3]))
\end{array}
$$

After exchanging the votes between groups, processes must select the final timestamp. When we have one proposal from each group in message M's destination, the highest vote is the decided timestamp. If P's local clock is smaller than the value, we broadcast the message to the local group with state $S2$ and save it in memory. Otherwise, we update the in-memory to state $S3$.

We only execute the procedure once we have proposals from all participating groups. Since we receive messages from the quasi-reliable channel, we keep the votes in the *Votes* structure. This structure is implicit in the algorithm.

$$
\begin{array}{l}
\textsc{local}\ \mathit{HasNecessaryVotes}(g,\ p,\ \mathit{msg},\ \mathit{ballot})\ \triangleq \\
\quad \land\ \mathit{Cardinality}(\mathit{ballot}) = \mathit{Cardinality}(\mathit{msg}.d) \\
\quad \land\ \mathit{Memory}!\mathit{Contains}(g,\ p,\ \textsc{lambda}\ n : \mathit{msg} = n[1] \land n[2] = \text{“S1”})
\end{array}
$$

$GatherGroupsTimestamp(g, p) \triangleq$

$\land$ $QuasiReliable!ReceiveAndConsume(g, p,$

LAMBDA t :

$\land$ LET

$msg \triangleq t[1]$

$origin \triangleq t[2]$

$vote \triangleq \langle msg.id, origin, t[3]\rangle$

$ballot \triangleq \{v \in (Votes[g][p] \cup \{vote\}) : v[1] = msg.id\}$

$elected \triangleq Max(\{x[3] : x \in ballot\})$

IN

We only execute the procedure when we have proposals from all groups.

$\land \lor \land HasNecessaryVotes(g, p, msg, ballot)$

$\land \exists \langle m, s, ts\rangle \in MemoryBuffer[g][p] : m = msg$

$\land GatherGroupsTimestampHandler(g, p, msg, ts, elected)$

$\land Votes' = [Votes$ EXCEPT $![g][p] = \{$

$x \in Votes[g][p] : x[1] \neq msg.id\}]$

$\lor \land \neg HasNecessaryVotes(g, p, msg, ballot)$

$\land Votes' = [Votes$ EXCEPT $![g][p] = Votes[g][p] \cup \{vote\}]$

$\land$ UNCHANGED $\langle MemoryBuffer, K,$

$PreviousMsgs, AtomicBroadcastBuffer\rangle$

$\land$ UNCHANGED $\langle Delivered\rangle)$

$SynchronizeGroupClock(g, p) \triangleq$

$\land$ $AtomicBroadcast!ABDeliver(g, p,$

LAMBDA $t : t[2] = \text{“S2”} \land SynchronizeGroupClockHandler(g, p, t[1], t[3]))$

When messages are to deliver, we select them and call the delivery primitive. Ready means they are in state $S3$, and the message either does not conflict with any other in the memory structure or is smaller than all others. Once a message is ready, we also collect the messages that do not conflict with any other for delivery in a single batch.

$DoDeliver(g, p) \triangleq$

We are accessing the buffer directly, and not through the *Memory instance*.

We do this because is easier and because we are only reading the values here.

Any changes we do through the instance.

$\exists \langle m\_1, state, ts\_1\rangle \in MemoryBuffer[g][p]$:

$\land state = \text{“S3”}$

$\land \forall \langle m\_2, ignore, ts\_2\rangle \in (MemoryBuffer[g][p] \setminus \{\langle m\_1, state, ts\_1\rangle\})$:

$\land \lor \neg CONFLICTR(m\_1, m\_2)$

$\lor ts\_1 < ts\_2 \lor (m\_1.id < m\_2.id \land ts\_1 = ts\_2)$

$\land$ LET

$G \triangleq Memory!ForAllFilter(g, p,$

LAMBDA $t\_i, t\_j : t\_i[2] = \text{“S3”} \land \neg CONFLICTR(t\_i[1], t\_j[1]))$

$D \triangleq G \cup \{\langle m\_1, \text{“S3”}, ts\_1\rangle\}$

$F \triangleq \{t[1] : t \in D\}$

IN

$\land Memory!Remove(g, p, D)$

$\wedge\ Delivered' = [Delivered\ \text{EXCEPT}\ ![g][p] =$
$\quad Delivered[g][p] \cup Enumerate(Cardinality(Delivered[g][p]),\ F)]$
$\wedge\ \text{UNCHANGED}\ \langle QuasiReliableChannel,\ AtomicBroadcastBuffer,$
$\quad Votes,\ PreviousMsgs,\ K\rangle$

$\text{LOCAL}\ InitProtocol \triangleq$
$\quad \wedge\ K = [i \in Groups \mapsto [p \in Processes \mapsto i]]$
$\quad \wedge\ Memory!Init$
$\quad \wedge\ PreviousMsgs = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\quad \wedge\ Delivered = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$
$\quad \wedge\ Votes = [i \in Groups \mapsto [p \in Processes \mapsto \{\}]]$

$\text{LOCAL}\ InitCommunication \triangleq$
$\quad \wedge\ AtomicBroadcast!Init$
$\quad \wedge\ QuasiReliable!Init$

$Init \triangleq InitProtocol \wedge InitCommunication$

$Step(g,\ p) \triangleq$
$\quad \vee\ ComputeGroupSeqNumber(g,\ p)$
$\quad \vee\ GatherGroupsTimestamp(g,\ p)$
$\quad \vee\ SynchronizeGroupClock(g,\ p)$
$\quad \vee\ DoDeliver(g,\ p)$

$GroupStep(g) \triangleq$
$\quad \exists\, p \in Processes : Step(g,\ p)$

$Next \triangleq$
$\quad \vee\ \exists\, g \in Groups : GroupStep(g)$
$\quad \vee\ \text{UNCHANGED}\ vars$

$Spec \triangleq Init \wedge \Box[Next]_{vars}$

$SpecFair \triangleq Spec \wedge \mathrm{WF}_{vars}(\exists\, g \in Groups : GroupStep(g))$

Helper functions to aid when checking the algorithm properties.

$WasDelivered(g,\ p,\ m) \triangleq$

Verifies if the given process p in group g delivered message m .

$\quad \wedge\ \exists\, \langle idx,\ n\rangle \in Delivered[g][p] : n.id = m.id$

$FilterDeliveredMessages(g,\ p,\ m) \triangleq$

Retrieve the set of messages with the same id as message m delivered by the given process p in group g .

$\{\langle idx,\ n\rangle \in Delivered[g][p] : n.id = m.id\}$

$DeliveredInstant(g,\ p,\ m) \ \triangleq$

Retrieve the instant the process p in group g delivered message m .

$(\text{CHOOSE } \langle t,\ n\rangle \in Delivered[g][p] : n.id = m.id)[1]$