MODULE *GenericMulticast0*

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

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 *NPROCESSES* is a natural number greater than 0.

$\land\ NPROCESSES \in (Nat \setminus \{0\})$

The messages in the protocol must be finite.

$\land\ IsFiniteSet(INITIAL\_MESSAGES)$

LOCAL $Processes \triangleq \{i : i \in 1 \,..\, NPROCESSES\}$

The instance of the quasi-reliable channel for process communication primitive. We use groups with single processes, having *NPROCESSES* groups.

VARIABLE *QuasiReliableChannel*

$QuasiReliable \triangleq$ INSTANCE *QuasiReliable* WITH
$NGROUPS \leftarrow NPROCESSES,$
$NPROCESSES \leftarrow 1$

VARIABLES

Structure that holds the clocks for all processes.

$K,$

Structure that holds all messages that were received but are still pending a final timestamp.

Pending,

Structure that holds all messages that contains a final timestamp but were not delivered yet.

Delivering,

Structure that holds all messages that contains a final timestamp and were already delivered.

Delivered,

Used to verify if previous messages conflict with the message beign processed. Using this approach is possible to deliver messages with a partially ordered delivery.

PreviousMsgs,

Set used to holds the votes that were cast for a message. Since the coordinator needs that all processes cast a vote for the final timestamp, this structure will hold the votes each process cast for each message on the system.

Votes

$$vars \triangleq \langle QuasiReliableChannel, Votes, K, Pending, Delivering, Delivered, PreviousMsgs \rangle$$

Helper to send messages. In a single operation we consume the message from our local network and send a request to the algorithm initiator. Is not possible to execute multiple operations in a single step on the same set. That is, we can not consume and send in different operations.

$$
\begin{aligned}
&\textsc{local}\ SendOriginatorAndRemoveLocal(self, dest, curr, prev, S) \triangleq \\
&\quad \textsc{if}\ self = dest \land prev[2].o = self\ \textsc{then}\ (S \setminus \{prev\}) \cup \{curr\} \\
&\quad\ \textsc{else}\ \ \textsc{if}\ prev[2].o = dest\ \textsc{then}\ S \cup \{curr\} \\
&\quad\ \textsc{else}\ \ \textsc{if}\ self = dest\ \textsc{then}\ S \setminus \{prev\} \\
&\quad\ \textsc{else}\ \ S
\end{aligned}
$$

We have the handlers representing each step of the algorithm. The handlers are the actual algorithm, and the caller is the step guard predicate.

$$
\begin{array}{l}
\textsc{local}\ \mathit{AssignTimestampHandler}(\mathit{self},\ \mathit{msg})\ \triangleq \\
\quad \land\ \lor\ \land \exists\, \mathit{prev} \in \mathit{PreviousMsgs}[\mathit{self}] : \mathit{CONFLICTR}(\mathit{msg},\ \mathit{prev}) \\
\qquad\quad \land\ K' = [K\ \textsc{except}\ ![\mathit{self}] = K[\mathit{self}] + 1] \\
\qquad\quad \land\ \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs}\ \textsc{except}\ ![\mathit{self}] = \{\mathit{msg}\}] \\
\qquad \lor\ \land \forall\, \mathit{prev} \in \mathit{PreviousMsgs}[\mathit{self}] : \neg \mathit{CONFLICTR}(\mathit{msg},\ \mathit{prev}) \\
\qquad\quad \land\ K' = [K\ \textsc{except}\ ![\mathit{self}] = K[\mathit{self}]] \\
\qquad\quad \land\ \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs}\ \textsc{except}\ ![\mathit{self}] = \\
\qquad\qquad\quad \mathit{PreviousMsgs}[\mathit{self}] \cup \{\mathit{msg}\}] \\
\quad \land\ \mathit{Pending}' = [\mathit{Pending}\ \textsc{except}\ ![\mathit{self}] = \\
\qquad\qquad \mathit{Pending}[\mathit{self}] \cup \{\langle K'[\mathit{self}],\ \mathit{msg}\rangle\}] \\
\quad \land\ \mathit{QuasiReliable}!\mathit{SendMap}(\textsc{lambda}\ \mathit{dest},\ S : \\
\qquad \mathit{SendOriginatorAndRemoveLocal}(\mathit{self},\ \mathit{dest}, \\
\qquad\quad \langle \text{“S1”},\ K'[\mathit{self}],\ \mathit{msg},\ \mathit{self}\rangle,\ \langle \text{“S0”},\ \mathit{msg}\rangle,\ S)) \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{Delivering},\ \mathit{Delivered},\ \mathit{Votes}\rangle
\end{array}
$$

$$
\begin{array}{l}
\textsc{local}\ \mathit{ComputeSeqNumberHandler}(\mathit{self},\ \mathit{ts},\ \mathit{msg},\ \mathit{origin})\ \triangleq \\
\quad \land\ \textsc{let} \\
\qquad t\ \triangleq\ \langle \text{“S1”},\ \mathit{ts},\ \mathit{msg},\ \mathit{origin}\rangle \\
\qquad \mathit{vote}\ \triangleq\ \langle \mathit{msg.id},\ \mathit{origin},\ \mathit{ts}\rangle \\
\qquad \mathit{election}\ \triangleq\ \{v \in (\mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}) : v[1] = \mathit{msg.id}\} \\
\qquad \mathit{elected}\ \triangleq\ \mathit{Max}(\{x[3] : x \in \mathit{election}\}) \\
\qquad \textsc{in} \\
\qquad\ \land\ \lor\ \land \mathit{Cardinality}(\mathit{election}) = \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad\ \land\ \mathit{Votes}' = [\mathit{Votes}\ \textsc{except}\ ![\mathit{self}] = \\
\qquad\qquad\qquad \{x \in \mathit{Votes}[\mathit{self}] : x[1] \neq \mathit{msg.id}\}] \\
\qquad\qquad\ \land\ \mathit{QuasiReliable}!\mathit{SendMap}(\textsc{lambda}\ \mathit{dest},\ S : \\
\qquad\qquad\qquad (S \setminus \{\langle \text{“S1”},\ \mathit{ts},\ \mathit{msg}\rangle\}) \cup \{\langle \text{“S2”},\ \mathit{elected},\ \mathit{msg}\rangle\}) \\
\qquad\quad \lor\ \land \mathit{Cardinality}(\mathit{election}) < \mathit{Cardinality}(\mathit{msg.d}) \\
\qquad\qquad\ \land\ \mathit{Votes}' = [\mathit{Votes}\ \textsc{except}\ ![\mathit{self}] = \\
\qquad\qquad\qquad \mathit{Votes}[\mathit{self}] \cup \{\mathit{vote}\}] \\
\qquad\qquad\ \land\ \mathit{QuasiReliable}!\mathit{Consume}(\mathit{self},\ 1,\ t) \\
\qquad\ \land\ \textsc{unchanged}\ \langle K,\ \mathit{PreviousMsgs},\ \mathit{Pending}, \\
\qquad\qquad\quad \mathit{Delivering},\ \mathit{Delivered}\rangle
\end{array}
$$

$$
\begin{array}{l}
\textsc{local}\ \mathit{AssignSeqNumberHandler}(\mathit{self},\ \mathit{ts},\ \mathit{msg})\ \triangleq \\
\quad \land\ \lor\ \land \mathit{ts} > K[\mathit{self}] \\
\qquad\quad \land\ \lor\ \land \exists\, \mathit{prev} \in \mathit{PreviousMsgs}[\mathit{self}] : \mathit{CONFLICTR}(\mathit{msg},\ \mathit{prev}) \\
\qquad\qquad\quad \land\ K' = [K\ \textsc{except}\ ![\mathit{self}] = \mathit{ts} + 1]
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad \land\ \mathit{PreviousMsgs}' = [\mathit{PreviousMsgs}\ \text{EXCEPT}\ ![\mathit{self}] = \{\}] \\
\qquad\qquad \lor\ \land\ \forall\, \mathit{prev} \in \mathit{PreviousMsgs}[\mathit{self}] : \mathit{CONFLICTR}(\mathit{msg}, \mathit{prev}) \\
\qquad\qquad\qquad \land\ K' = [K\ \text{EXCEPT}\ ![\mathit{self}] = \mathit{ts}] \\
\qquad\qquad\qquad \land\ \text{UNCHANGED}\ \mathit{PreviousMsgs} \\
\qquad \lor\ \land\ \mathit{ts} \leq K[\mathit{self}] \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle K, \mathit{PreviousMsgs} \rangle \\
\land\ \mathit{Delivering}' = [\mathit{Delivering}\ \text{EXCEPT}\ ![\mathit{self}] = \\
\qquad\qquad \mathit{Delivering}[\mathit{self}] \cup \{\langle \mathit{ts}, \mathit{msg} \rangle\}] \\
\land\ \text{UNCHANGED}\ \langle \mathit{Votes}, \mathit{Delivered} \rangle
\end{array}
$$

This procedure executes after an initiator GM-Cast a message m to $m.d$. All processes in $m.d$ do the same thing after receiving m, assing the local clock to the message timestamp, inserting the message with the timestamp to the process *Pending* set, and sending it to the initiator to choose the timestamp.

$$
\begin{array}{l}
\mathit{AssignTimestamp}(\mathit{self}) \triangleq
\end{array}
$$

We delegate to the lambda to handle the message while filtering for the correct state.

$$
\begin{array}{l}
\land\ \mathit{QuasiReliable}!\mathit{Receive}(\mathit{self}, 1, \text{LAMBDA}\ t : \\
\qquad\qquad \land\ t[1] = \text{“S0”} \\
\qquad\qquad \land\ \mathit{AssignTimestampHandler}(\mathit{self}, t[2]))
\end{array}
$$

This method is executed only by the initiator. This method processes messages on state $S1$ and can proceed in two ways. If the initiator has votes from all other processes, the message's final timestamp is the maximum received vote, and the initiator sends the message back to all participants in state $S2$. Otherwise, the initiator only store the received message in the *Votes* structure.

$$
\mathit{ComputeSeqNumber}(\mathit{self}) \triangleq
$$

We delegate to the lambda handler to effectively execute the procedure. Here we verify that the message is on state $S1$ and the current process is the initiator.

$$
\begin{array}{l}
\land\ \mathit{QuasiReliable}!\mathit{Receive}(\mathit{self}, 1, \text{LAMBDA}\ t : \\
\qquad\qquad \land\ t[1] = \text{“S1”} \\
\qquad\qquad \land\ t[3].o = \mathit{self} \\
\qquad\qquad \land\ \mathit{ComputeSeqNumberHandler}(\mathit{self}, t[2], t[3], t[4]))
\end{array}
$$

After the coordinator computes the final timestamp for the message m, all processes in $m.d$ will receive the chosen timestamp. Each participant checks the message's timestamp against its local clock. If the value is greater than the process clock, we need to update the process clock with the message's timestamp. If m conflicts with a message in the *PreviousMsgs*, the clock updates to m's timestamp plus one and clears the *PreviousMsgs* set. Without any conflict with m, the clock updates to m's timestamp. The message is removed from *Pending* and added to *Delivering* set.

$AssignSeqNumber(self) \triangleq$

We delegate the procedure execution the the handler, and the message is automatically consumed after the lambda execution. In this one we only filter the messages.

$$
\begin{array}{l}
\land\ QuasiReliable!ReceiveAndConsume(self,\ 1,\ \text{LAMBDA}\ t\_1 : \\
\qquad \land\ t\_1[1] = \text{"S2"} \\
\qquad \land\ \exists\, t\_2 \in Pending[self] : t\_1[3].id = t\_2[2].id \\
\qquad\qquad \land\ AssignSeqNumberHandler(self,\ t\_1[2],\ t\_1[3])
\end{array}
$$

We remove the message here to avoid too many arguments in the procedure invocation.

$$
\qquad\qquad \land\ Pending' = [Pending\ \text{EXCEPT}\ ![self] = @ \setminus \{t\_2\}])
$$

Responsible for delivery of messages. The messages in the *Delivering* set with the smallest timestamp among others in the *Pending* joined with *Delivering* set. We can also deliver messages that commute with all others, the generalized behavior in action.

Delivered messages will be added to the *Delivered* set and removed from the others. To store the instant of delivery, we insert delivered messages with the following format:

```
<<Nat, Message>>
```

Using this model, we know the message delivery order for all processes.

$$
\begin{array}{l}
DoDeliver(self) \triangleq \\
\quad \exists\, \langle ts\_1,\ m\_1 \rangle \in Delivering[self] : \\
\qquad \land\ \forall\, \langle ts\_2,\ m\_2 \rangle \in \\
\qquad\qquad (Delivering[self] \cup Pending[self]) \setminus \{\langle ts\_1,\ m\_1 \rangle\} : \\
\qquad\qquad\quad \lor\ \neg CONFLICTR(m\_1,\ m\_2) \\
\qquad\qquad\quad \lor\ ts\_1 < ts\_2 \\
\qquad\qquad\qquad \lor\ (m\_1.id < m\_2.id \land ts\_1 = ts\_2) \\
\qquad \land\ \text{LET} \\
\qquad\qquad T \triangleq Delivering[self] \cup Pending[self] \\
\qquad\qquad G \triangleq \{t\_i \in Delivering[self] :
\end{array}
$$

$\forall t\_j \in T \setminus \{t\_i\} : \neg CONFLICTR(t\_i[2], t\_j[2])\}$

$F \triangleq \{m\_1\} \cup \{t[2] : t \in G\}$

IN

$\land\ Delivering' = [Delivering\ \text{EXCEPT}\ ![self] = @ \setminus (G \cup \{\langle ts\_1, m\_1\rangle\})]$

$\land\ Delivered' = [Delivered\ \text{EXCEPT}\ ![self] = Delivered[self] \cup Enumerate(Cardinality(Delivered[self]), F)]$

$\land\ \text{UNCHANGED}\ \langle QuasiReliableChannel, Votes, Pending, PreviousMsgs, K\rangle$

Responsible for initializing global variables used on the system. All variables necessary by the protocol are a mapping from the node *id* to the corresponding process set.

The "message" is also a structure, with the following format:

```
[ id |-> Nat, d |-> Nodes, o |-> Node ]
```

We have the properties: *id* is the messages' unique *id*, we use a natural number to represent; *d* is the destination, it may be a subset of the Nodes set; and *o* is the originator, the process that started the execution of the algorithm. These properties are all static and never change.

The mutable values we transport outside the message structure. We do this using the process communication channel, using a tuple to send the message along with the mutable values.

LOCAL $InitProtocol \triangleq$

$\land\ K = [i \in Processes \mapsto 0]$

$\land\ Pending = [i \in Processes \mapsto \{\}]$

$\land\ Delivering = [i \in Processes \mapsto \{\}]$

$\land\ Delivered = [i \in Processes \mapsto \{\}]$

$\land\ PreviousMsgs = [i \in Processes \mapsto \{\}]$

LOCAL $InitHelpers \triangleq$

Initialize the protocol network.

$\land\ QuasiReliable!Init$

This structure is holding the votes the processes cast for each message on the system. Since any process can be the "coordinator", this is a mapping for processes to a set. The set will contain the vote a process has cast for a message.

$\wedge\ Votes = [i \in Processes \mapsto \{\}]$

$Init \triangleq InitProtocol \wedge InitHelpers$

$Step(self) \triangleq$

$\quad \vee\ AssignTimestamp(self)$

$\quad \vee\ ComputeSeqNumber(self)$

$\quad \vee\ AssignSeqNumber(self)$

$\quad \vee\ DoDeliver(self)$

$Next \triangleq$

$\quad \vee\ \exists\, self \in Processes : Step(self)$

$\quad \vee\ \text{UNCHANGED}\ vars$

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

$SpecFair \triangleq Spec \wedge \mathrm{WF}_{vars}(\exists\, self \in Processes : Step(self))$

Helper functions to aid when checking the algorithm properties.

$WasDelivered(p, m) \triangleq$

Verifies if the given process p delivered message m .

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

$DeliveredInstant(p, m) \triangleq$

Retrieve the instant the given process p delivered message m .

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

$FilterDeliveredMessages(p, m) \triangleq$

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

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