MODULE *SWIM*

This module contains a spec for *Atomix′* implementation of the *SWIM* protocol. http://*www.cs.cornell.edu*/projects/Quicksilver/*public_pdfs/SWIM.pdf*

The *SWIM* protocol works by periodically probing peers to detect failures. The *Atomix* implementation of the protocol propagates state changes to peers using a gossip protocol. Members in the implementation can be in one of three states at any given time: *Alive*, *Suspect*, or *Dead*. Time is tracked in this implementation using logical clocks that are managed by each individual member. A member can only increment its own logical clock (known as an *incarnationNumber*), and within any given incarnation the member can only be in a state once. Members always transition from *Alive* $\rightarrow$ *Suspect* $\rightarrow$ *Dead*, and the incarnation must be incremented again to revert back to the *Alive* state. Member states transition back to *Alive* by a *Suspect* or *Dead* member incrementing its incarnation and refuting its state.

While this spec does use probes, it does not request probes of a suspected member from peers. Peer probes are a practical feature that does not add value to the spec for purposes of model checking. A real implementation of the protocol should use peer probes to avoid false positives.

The spec's invariant (*Inv*) asserts that no member can transition to the same state multiple times in the same incarnation, and state transitions always progress from *Alive* to *Suspect* to *Dead*.

To perform model checking on the spec, define a set of numeric Members and define the *Nil*, *Dead*, *Suspect*, and *Alive* constants as numeric values of monotonically increasing values in that order. Additional constants may be defined as desired.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *Bags*, *TLC*

The set of possible members
CONSTANT *Member*

Empty numeric value
CONSTANT *Nil*

Numeric member states
CONSTANTS *Alive*, *Suspect*, *Dead*

The values of member states must be sequential
ASSUME $Alive > Suspect \land Suspect > Dead$

Message types
CONSTANTS *GossipMessage*, *ProbeMessage*, *AckMessage*

Member incarnation numbers
VARIABLE *incarnation*

Member lists
VARIABLE *members*

Pending updates
VARIABLE *updates*

History
VARIABLE *history*

A bag of records representing requests and responses sent from one server to another. *TLAPS* doesn't support the *Bags* module, so this is a function mapping Message to *Nat*.

VARIABLE $messages$

$vars \triangleq \langle incarnation,\ members,\ updates,\ history,\ messages \rangle$

$InitMemberVars \triangleq$
$\quad \land\ incarnation = [i \in Member \mapsto Nil]$
$\quad \land\ members = [i \in Member \mapsto [j \in Member \mapsto [incarnation \mapsto 0,\ state \mapsto Nil]]]$
$\quad \land\ updates\ \ = [i \in Member \mapsto \langle\rangle]$
$\quad \land\ history\ \ = [i \in Member \mapsto [j \in Member \mapsto [k \in \{\} \mapsto \langle\rangle]]]$

$InitMessageVars \triangleq\ messages = [m \in \{\} \mapsto 0]$

Helper for *Send* and Reply. Given a message m and bag of messages, return a new bag of messages with one more m in it.

$WithMessage(m,\ msgs) \triangleq$
$\quad$ IF $m \in$ DOMAIN $msgs$ THEN
$\quad\quad [msgs$ EXCEPT $![m] = msgs[m] + 1]$
$\quad$ ELSE
$\quad\quad msgs \mathbin{@@} (m :> 1)$

Helper for *Discard* and Reply. Given a message m and bag of messages, return a new bag of messages with one less m in it.

$WithoutMessage(m,\ msgs) \triangleq$
$\quad$ IF $m \in$ DOMAIN $msgs$ THEN
$\quad\quad [msgs$ EXCEPT $![m] = msgs[m] - 1]$
$\quad$ ELSE
$\quad\quad msgs$

Add a message to the bag of messages.

$Send(m) \triangleq\ messages' = WithMessage(m,\ messages)$

Remove a message from the bag of messages. Used when a server is done processing a message.

$Discard(m) \triangleq\ messages' = WithoutMessage(m,\ messages)$

The network duplicates a message

$DuplicateMessage(m) \triangleq$
$\quad \land\ messages[m] = 1$
$\quad \land\ Send(m)$

$\land$ UNCHANGED $\langle incarnation, members, updates, history\rangle$

The network drops a message

$DropMessage(m) \triangleq$
$\land\ messages[m] > 0$
$\land\ Discard(m)$
$\land$ UNCHANGED $\langle incarnation, members, updates, history\rangle$

Returns a sequence with the head removed

$Pop(q) \triangleq SubSeq(q, 2, Len(q))$

Records an 'update' to gossipped by the given 'member'

$RecordUpdate(member, update) \triangleq$
$\land\ updates' = [updates$ EXCEPT $![member] = Append(updates[member], update)]$

Removes the first update from the given 'member's updates

$PopUpdate(member) \triangleq$
$\land\ updates' = [updates$ EXCEPT $![member] = Pop(updates[member])]$

Records a member state change on the given 'source' node

$RecordHistory(source, dest, tm, state) \triangleq$
IF $tm \in$ DOMAIN $history[source][dest]$ THEN
$history' = [history$ EXCEPT $![source][dest][tm] = Append(history[source][dest][tm], state)]$
ELSE
$history' = [history$ EXCEPT $![source] = history[source][dest]$ @@ $(tm :> \langle state\rangle)]$

Updates the state of a peer on the given 'source' node
When the state of the 'dest' is updated, an update message is enqueued for gossip
and the state change is recorded in the 'source' node's history for model checking.

$UpdateState(source, dest, inc, state) \triangleq$
$\land\ members' = [members$ EXCEPT $![source][dest] = [incarnation \mapsto inc, state \mapsto state]]$
$\land\ RecordUpdate(source, [id \mapsto dest, incarnation \mapsto inc, state \mapsto state])$
$\land\ RecordHistory(source, dest, inc, state)$

Sends a typed 'message' from the given 'source' to the given 'dest'

$SendMessage(type, source, dest, message) \triangleq$
$Send([type \mapsto type, source \mapsto source, dest \mapsto dest, message \mapsto message])$

Sends a probe 'message' from the given 'source' to the given 'dest'

$SendProbe(source, dest, message) \triangleq SendMessage(ProbeMessage, source, dest, message)$

Sends an ack 'message' from the given 'source' to the given 'dest'

$SendAck(source, dest, message) \triangleq SendMessage(AckMessage, source, dest, message)$

Sends a gossip 'message' from the given 'source' to the given 'dest'

$SendGossip(source, dest, message) \triangleq SendMessage(GossipMessage, source, dest, message)$

Triggers a probe request to a peer
* 'source' is the source of the probe
* 'dest' is the destination to which to send the probe

$$
\begin{array}{l}
Probe(source,\ dest) \triangleq \\
\quad \land\ source \neq dest \\
\quad \land\ incarnation[source] \neq Nil \\
\quad \land\ SendProbe(source,\ dest,\ members[source][dest]) \\
\quad \land\ \text{UNCHANGED}\ \langle incarnation,\ members,\ updates,\ history \rangle
\end{array}
$$

Handles a probe message from a peer
* 'source' is the source of the probe
* 'dest' is the destination receiving the probe
* 'message' is the probe message, containing the highest known destination state and incarnation

If the received incarnation is greater than the destination's incarnation number, update the destination's incarnation number to 1 plus the received number. This can happen after a node leaves and rejoins the cluster. If the destination is suspected by the source, increment the destination's incarnation, enqueue an update to be gossipped, and respond with the updated incarnation. If the destination's incarnation is greater than the source's incarnation, just send an ack.

$$
\begin{array}{l}
HandleProbe(source,\ dest,\ message) \triangleq \\
\quad \land\ incarnation[dest] \neq Nil \\
\quad \land\ \lor\ \land\ message.incarnation > incarnation[dest] \\
\qquad\quad \land\ incarnation' = [incarnation\ \text{EXCEPT}\ ![dest] = message.incarnation + 1] \\
\qquad\quad \land\ SendAck(dest,\ source,\ [incarnation \mapsto incarnation'[dest]]) \\
\qquad \lor\ \land\ message.state = Suspect \\
\qquad\quad \land\ incarnation' = [incarnation\ \text{EXCEPT}\ ![dest] = incarnation[dest] + 1] \\
\qquad\quad \land\ RecordUpdate(dest,\ [id \mapsto dest,\ incarnation \mapsto incarnation'[dest],\ state \mapsto Alive]) \\
\qquad\quad \land\ SendAck(dest,\ source,\ [incarnation \mapsto incarnation'[dest]]) \\
\qquad \lor\ \land\ message.incarnation \leq incarnation[dest] \\
\qquad\quad \land\ SendAck(dest,\ source,\ [incarnation \mapsto incarnation[dest]]) \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle incarnation \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle members,\ updates,\ history \rangle
\end{array}
$$

Handles an ack message from a peer
* 'source' is the source of the ack
* 'dest' is the destination receiving the ack
* 'message' is the ack message

If the acknowledged message is greater than the incarnation for the member on the destination node, update the member's state and enqueue an update for gossip.

$$
\begin{array}{l}
HandleAck(source,\ dest,\ message) \triangleq \\
\quad \land\ \lor\ \land\ message.incarnation > members[dest][source].incarnation \\
\qquad\quad \land\ UpdateState(dest,\ source,\ message.incarnation,\ Alive) \\
\qquad \lor\ \land\ message.incarnation \leq members[dest][source].incarnation \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle members,\ updates,\ history \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle incarnation,\ messages \rangle
\end{array}
$$

Handles a failed probe
* 'source' is the source of the probe
* 'dest' is the destination to which the probe was sent
* 'message' is the probe message

If the probe request matches the local incarnation for the probe destination and the local state for the destination is *Alive*, update the state to *Suspect*.

$$
\begin{aligned}
&\mathit{HandleFail}(\mathit{source},\ \mathit{dest},\ \mathit{message}) \triangleq \\
&\quad \land\ \lor\ \land\ \mathit{message.incarnation} > 0 \\
&\qquad\quad \land\ \mathit{message.incarnation} = \mathit{members}[\mathit{source}][\mathit{dest}].\mathit{incarnation} \\
&\qquad\quad \land\ \mathit{members}[\mathit{source}][\mathit{dest}].\mathit{state} = \mathit{Alive} \\
&\qquad\quad \land\ \mathit{UpdateState}(\mathit{source},\ \mathit{dest},\ \mathit{message.incarnation},\ \mathit{Suspect}) \\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{incarnation},\ \mathit{members},\ \mathit{updates} \rangle
\end{aligned}
$$

Expires a suspected peer
* 'source' is the node on which to expire the peer
* 'dest' is the peer to expire

If the destination's state is *Suspect*, change its state to *Dead* and enqueue a gossip update to notify peers of the state change.

$$
\begin{aligned}
&\mathit{Expire}(\mathit{source},\ \mathit{dest}) \triangleq \\
&\quad \land\ \mathit{source} \neq \mathit{dest} \\
&\quad \land\ \mathit{members}[\mathit{source}][\mathit{dest}].\mathit{state} = \mathit{Suspect} \\
&\quad \land\ \mathit{UpdateState}(\mathit{source},\ \mathit{dest},\ \mathit{members}[\mathit{source}][\mathit{dest}].\mathit{incarnation},\ \mathit{Dead}) \\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{incarnation} \rangle
\end{aligned}
$$

Sends a gossip update to a peer
* 'source' is the source of the update
* 'dest' is the destination to which to send the update

$$
\begin{aligned}
&\mathit{Gossip}(\mathit{source},\ \mathit{dest}) \triangleq \\
&\quad \land\ \mathit{source} \neq \mathit{dest} \\
&\quad \land\ \mathit{members}[\mathit{source}][\mathit{dest}].\mathit{state} \neq \mathit{Nil} \\
&\quad \land\ \mathit{Len}(\mathit{updates}[\mathit{source}]) > 0 \\
&\quad \land\ \mathit{SendGossip}(\mathit{source},\ \mathit{dest},\ \mathit{updates}[1]) \\
&\quad \land\ \mathit{PopUpdate}(\mathit{source}) \\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{incarnation},\ \mathit{members},\ \mathit{history} \rangle
\end{aligned}
$$

Handles a gossip update
* 'source' is the source of the update
* 'dest' is the destination handling the update
* 'message' is the update message in the format with the updated member *ID*, incarnation, and state

If the member is not present in the destination's members, add it to the members set. If the incarnation is greater than the destination's incarnation for the gossipped member, update the member's incarnation and state on the destination node and enqueue the change for gossip. If the incarnation is equal to the destination's incarnation for the member and the state is less than the destination's state for the member, update the member's state on the destination node and enqueue the change for gossip. Record state changes in the history variable for model checking.

$$
\mathit{HandleGossipUpdate}(\mathit{source},\ \mathit{dest},\ \mathit{message}) \triangleq
$$

$\land \lor \land message.incarnation > members[dest][message.id].incarnation$
$\land UpdateState(dest, message.id, message.incarnation, message.state)$
$\lor \land message.incarnation = members[dest][message.id].incarnation$
$\land message.state < members[dest][message.id].state$
$\land UpdateState(dest, message.id, message.incarnation, message.state)$
$\lor \land message.incarnation < members[dest][message.id].incarnation$
$\land$ UNCHANGED $\langle members, updates, history \rangle$
$\land$ UNCHANGED $\langle incarnation, messages \rangle$

Adds a member to the cluster * 'id' is the identifier of the member to add

If the member is not present in the state history:
* Initialize the member's incarnation to 1
* Initialize the member's states for all known members to incarnation: 0, state: *Dead* to allow heartbeats
* Enqueue an update to notify peers of the member's existence
* Initialize the member's history

$AddMember(id) \triangleq$
$\land incarnation[id] = Nil$
$\land incarnation' = [incarnation$ EXCEPT $![id] = 1]$
$\land members' = [members$ EXCEPT $![id] = [i \in$ DOMAIN $members \mapsto [incarnation \mapsto 0, state \mapsto Dead]]]$
$\land history' = [history$ EXCEPT $![id] = [i \in \{\} \mapsto \langle\rangle]]$
$\land$ UNCHANGED $\langle updates, messages \rangle$

Removes a member from the cluster * 'id' is the identifier of the member to remove

Alter the domain of 'incarnation', 'members', and 'updates' to remove the member's volatile state. We retain only the in-flight messages and history for model checking.

$RemoveMember(id) \triangleq$
$\land incarnation[id] \neq Nil$
$\land incarnation' = [incarnation$ EXCEPT $![id] = Nil]$
$\land members' = [members$ EXCEPT $![id] = [j \in Member \mapsto [incarnation \mapsto 0, state \mapsto Nil]]]$
$\land updates' = [updates$ EXCEPT $![id] = \langle\rangle]$
$\land$ UNCHANGED $\langle history, messages \rangle$

Receives a message from the bag of messages

$ReceiveMessage(m) \triangleq$
$\lor \land m.type = GossipMessage$
$\land HandleGossipUpdate(m.source, m.dest, m.message)$
$\land Discard(m)$
$\lor \land m.type = ProbeMessage$
$\land HandleProbe(m.source, m.dest, m.message)$
$\land Discard(m)$
$\lor \land m.type = AckMessage$
$\land HandleAck(m.source, m.dest, m.message)$
$\land Discard(m)$
$\lor \land m.type = ProbeMessage$

$\land$ $HandleFail(m.source,\ m.dest,\ m.message)$
$\land$ $Discard(m)$

Initial state

$Init \triangleq$
$\land$ $InitMessageVars$
$\land$ $InitMemberVars$

Next state predicate

$Next \triangleq$
$\lor \exists\, i, j \in Member : Probe(i, j)$
$\lor \exists\, i, j \in Member : Expire(i, j)$
$\lor \exists\, i, j \in Member : Gossip(i, j)$
$\lor \exists\, i \in Member : AddMember(i)$
$\lor \exists\, i \in Member : RemoveMember(i)$
$\lor \exists\, m \in \text{DOMAIN } messages : ReceiveMessage(m)$
$\lor \exists\, m \in \text{DOMAIN } messages : DuplicateMessage(m)$
$\lor \exists\, m \in \text{DOMAIN } messages : DropMessage(m)$

Type invariant

$Inv \triangleq \forall\, i \in \text{DOMAIN } history :$
$\forall\, j \in \text{DOMAIN } history[i] :$
$\land \neg\exists\, k \in \text{DOMAIN } history[i][j] :$
$history[i][j][k+1] \geq history[i][j][k]$
$\land$ $Len(history[i][j]) \leq 3$

Spec

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

---

* Modification History
* Last modified *Thu Oct* 18 12:45:40 *PDT* 2018 by *jordanhalterman*
* Created *Mon Oct* 08 00:36:03 *PDT* 2018 by *jordanhalterman*