
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 a term), and within any given term the member can only be in a state once. Members always transition from *Alive* \rightarrow *Suspect* \rightarrow *Dead*, and the term 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 term 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 term, 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* \wedge *Suspect* > *Dead*

Message types

CONSTANTS *GossipMessage*, *ProbeMessage*, *AckMessage*

Member terms

VARIABLE *term*

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*

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

InitMemberVars \triangleq
 $\wedge term = [i \in Member \mapsto Nil]$
 $\wedge members = [i \in Member \mapsto [j \in Member \mapsto [term \mapsto 0, state \mapsto Nil]]]$
 $\wedge updates = [i \in Member \mapsto \langle \rangle]$
 $\wedge history = [i \in Member \mapsto [j \in Member \mapsto [k \in \{\} \mapsto \langle \rangle]]]$
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
 IF *m* \in DOMAIN *msgs* THEN
 $[msgs \text{ EXCEPT } ![m] = msgs[m] + 1]$
 ELSE
 $msgs @@ (m :> 1)$

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
 IF *m* \in DOMAIN *msgs* THEN
 $[msgs \text{ EXCEPT } ![m] = msgs[m] - 1]$
 ELSE
 $msgs$

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

Combination of *Send* and *Discard*

Reply(*response*, *request*) \triangleq
 $messages' = WithoutMessage(request, WithMessage(response, messages))$

The network duplicates a message

DuplicateMessage(*m*) \triangleq
 $\wedge \text{messages}[m] = 1$
 $\wedge \text{Send}(m)$
 $\wedge \text{UNCHANGED } \langle \text{term}, \text{members}, \text{updates}, \text{history} \rangle$

The network drops a message
DropMessage(*m*) \triangleq
 $\wedge \text{messages}[m] > 0$
 $\wedge \text{Discard}(m)$
 $\wedge \text{UNCHANGED } \langle \text{term}, \text{members}, \text{updates}, \text{history} \rangle$

Returns a sequence with the head removed
Pop(*q*) $\triangleq \text{SubSeq}(q, 2, \text{Len}(q))$

Records an 'update' to gossiped by the given 'member'
RecordUpdate(*member*, *update*) \triangleq
 $\wedge \text{updates}' = [\text{updates} \text{ EXCEPT } ![\text{member}] = \text{Append}(\text{updates}[\text{member}], \text{update})]$

Removes the first update from the given 'member's updates
PopUpdate(*member*) \triangleq
 $\wedge \text{updates}' = [\text{updates} \text{ EXCEPT } ![\text{member}] = \text{Pop}(\text{updates}[\text{member}])]$

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* :> $\langle \text{state} \rangle$)]

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*, *tm*, *state*) \triangleq
 \wedge IF *dest* \in DOMAIN *members*[*source*] THEN
 members' = [*members* EXCEPT ![*source*][*dest*] = [*term* \mapsto *tm*, *state* \mapsto *state*]]
 ELSE
 members' = [*members* EXCEPT ![*source*] = *members*[*source*] @@ (*dest* :> [*term* \mapsto *tm*, *state* \mapsto *state*])]
 $\wedge \text{RecordUpdate}(\text{source}, [\text{id} \mapsto \text{dest}, \text{term} \mapsto \text{tm}, \text{state} \mapsto \text{state}])$
 $\wedge \text{RecordHistory}(\text{source}, \text{dest}, \text{tm}, \text{state})$

Sends a typed 'message' from the given 'source' to the given 'dest'
SendMessage(*type*, *source*, *dest*, *message*) \triangleq
 $\text{Send}([\text{type} \mapsto \text{type}, \text{source} \mapsto \text{source}, \text{dest} \mapsto \text{dest}, \text{message} \mapsto \text{message}])$

Sends a probe 'message' from the given 'source' to the given 'dest'
SendProbe(*source*, *dest*, *message*) $\triangleq \text{SendMessage}(\text{ProbeMessage}, \text{source}, \text{dest}, \text{message})$

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

$Probe(source, dest) \triangleq$
 $\wedge source \neq dest$
 $\wedge term[source] \neq Nil$
 $\wedge SendProbe(source, dest, members[source][dest])$
 $\wedge UNCHANGED \langle term, members, updates, history \rangle$

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 term

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

$HandleProbe(source, dest, message) \triangleq$
 $\wedge term[dest] \neq Nil$
 $\wedge \vee \wedge message.term > term[dest]$
 $\wedge term' = [term \text{ EXCEPT } ![dest] = message.term + 1]$
 $\wedge SendAck(dest, source, [term \mapsto term'[dest]])$
 $\vee \wedge message.state = Suspect$
 $\wedge term' = [term \text{ EXCEPT } ![dest] = term[dest] + 1]$
 $\wedge RecordUpdate(dest, [id \mapsto dest, term \mapsto term'[dest], state \mapsto Alive])$
 $\wedge SendAck(dest, source, [term \mapsto term'[dest]])$
 $\vee \wedge message.term \leq term[dest]$
 $\wedge SendAck(dest, source, [term \mapsto term[dest]])$
 $\wedge UNCHANGED \langle term \rangle$
 $\wedge UNCHANGED \langle members, updates, history \rangle$

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 term for the member on the destination node, update the member's state and enqueue an update for gossip.

$HandleAck(source, dest, message) \triangleq$
 $\wedge \vee \wedge message.term > members[dest][source].term$

$$\begin{aligned}
& \wedge \text{UpdateState}(\text{dest}, \text{source}, \text{message.term}, \text{Alive}) \\
& \vee \wedge \text{message.term} \leq \text{members}[\text{dest}][\text{source}].\text{term} \\
& \wedge \text{UNCHANGED } \langle \text{members}, \text{updates}, \text{history} \rangle \\
& \wedge \text{UNCHANGED } \langle \text{term}, \text{messages} \rangle
\end{aligned}$$

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 term for the probe destination and the local state for the destination is *Alive*, update the state to *Suspect*.

$$\begin{aligned}
\text{HandleFail}(\text{source}, \text{dest}, \text{message}) & \triangleq \\
& \wedge \vee \wedge \text{dest} \in \text{DOMAIN } \text{members}[\text{source}] \\
& \wedge \text{message.term} = \text{members}[\text{source}][\text{dest}].\text{term} \\
& \wedge \text{members}[\text{source}][\text{dest}].\text{state} = \text{Alive} \\
& \wedge \text{UpdateState}(\text{source}, \text{dest}, \text{message.term}, \text{Suspect}) \\
& \wedge \text{UNCHANGED } \langle \text{term}, \text{members}, \text{updates} \rangle
\end{aligned}$$

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.

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

Sends a gossip update to a peer

* 'source' is the source of the update
 * 'dest' is the destination to which to send the update

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

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*, term, and state

If the member is not present in the destination's members, add it to the members set. If the term is greater than the destination's term for the gossiped member, update the member's term and state on the destination node and enqueue the change for gossip. If the term is equal to the destination's term 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.

$$\begin{aligned} \text{HandleGossipUpdate}(\text{source}, \text{dest}, \text{message}) &\triangleq \\ &\wedge \vee \wedge \text{message.id} \notin \text{DOMAIN members}[\text{dest}] \\ &\quad \wedge \text{UpdateState}(\text{dest}, \text{message.id}, \text{message.term}, \text{message.state}) \\ &\vee \wedge \text{message.id} \in \text{DOMAIN members}[\text{dest}] \\ &\quad \wedge \vee \wedge \text{message.term} > \text{members}[\text{dest}][\text{message.id}].\text{term} \\ &\quad \vee \wedge \text{message.term} = \text{members}[\text{dest}][\text{message.id}].\text{term} \\ &\quad \quad \wedge \text{message.state} < \text{members}[\text{dest}][\text{message.id}].\text{state} \\ &\quad \wedge \text{UpdateState}(\text{dest}, \text{message.id}, \text{message.term}, \text{message.state}) \\ &\vee \text{UNCHANGED } \langle \text{members}, \text{updates}, \text{history} \rangle \\ &\wedge \text{UNCHANGED } \langle \text{term}, \text{messages} \rangle \end{aligned}$$

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 term to 1
- * Initialize the member's states for all known members to term: 0, state: *Dead* to allow heartbeats
- * Enqueue an update to notify peers of the member's existence
- * Initialize the member's history

$$\begin{aligned} \text{AddMember}(\text{id}) &\triangleq \\ &\wedge \text{term}[\text{id}] = \text{Nil} \\ &\wedge \text{term}' = [\text{term} \text{ EXCEPT } ![\text{id}] = 1] \\ &\wedge \text{members}' = [\text{members} \text{ EXCEPT } ![\text{id}] = [i \in \text{DOMAIN members} \mapsto [\text{term} \mapsto 0, \text{state} \mapsto \text{Dead}]]] \\ &\wedge \text{history}' = [\text{history} \text{ EXCEPT } ![\text{id}] = [i \in \{\} \mapsto \langle \rangle]] \\ &\wedge \text{UNCHANGED } \langle \text{updates}, \text{messages} \rangle \end{aligned}$$

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

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

$$\begin{aligned} \text{RemoveMember}(\text{id}) &\triangleq \\ &\wedge \text{term}[\text{id}] \neq \text{Nil} \\ &\wedge \text{term}' = [\text{term} \text{ EXCEPT } ![\text{id}] = \text{Nil}] \\ &\wedge \text{members}' = [\text{members} \text{ EXCEPT } ![\text{id}] = [j \in \text{Member} \mapsto [\text{term} \mapsto 0, \text{state} \mapsto \text{Nil}]]] \\ &\wedge \text{updates}' = [\text{updates} \text{ EXCEPT } ![\text{id}] = \langle \rangle] \\ &\wedge \text{UNCHANGED } \langle \text{history}, \text{messages} \rangle \end{aligned}$$

Receives a message from the bag of messages

$$\begin{aligned} \text{ReceiveMessage}(m) &\triangleq \\ &\vee \wedge m.\text{type} = \text{GossipMessage} \\ &\quad \wedge \text{HandleGossipUpdate}(m.\text{source}, m.\text{dest}, m.\text{message}) \\ &\quad \wedge \text{Discard}(m) \\ &\vee \wedge m.\text{type} = \text{ProbeMessage} \end{aligned}$$

$$\begin{aligned}
& \wedge \text{HandleProbe}(m.\text{source}, m.\text{dest}, m.\text{message}) \\
& \wedge \text{Discard}(m) \\
\vee & \wedge m.\text{type} = \text{AckMessage} \\
& \wedge \text{HandleAck}(m.\text{source}, m.\text{dest}, m.\text{message}) \\
& \wedge \text{Discard}(m) \\
\vee & \wedge m.\text{type} = \text{ProbeMessage} \\
& \wedge \text{HandleFail}(m.\text{source}, m.\text{dest}, m.\text{message}) \\
& \wedge \text{Discard}(m)
\end{aligned}$$

Initial state

$$\begin{aligned}
\text{Init} & \triangleq \\
& \wedge \text{InitMessageVars} \\
& \wedge \text{InitMemberVars}
\end{aligned}$$

Next state predicate

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

Type invariant

$$\begin{aligned}
\text{Inv} & \triangleq \forall i \in \text{DOMAIN history} : \\
& \quad \forall j \in \text{DOMAIN history}[i] : \\
& \quad \quad \wedge \neg \exists k \in \text{DOMAIN history}[i][j] : \\
& \quad \quad \quad \text{history}[i][j][k+1] \geq \text{history}[i][j][k] \\
& \quad \quad \wedge \text{Len}(\text{history}[i][j]) \leq 3
\end{aligned}$$

Spec

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

\ * Modification History
\ * Last modified Mon Oct 08 17:17:17 PDT 2018 by jordanhalterman
\ * Created Mon Oct 08 00:36:03 PDT 2018 by jordanhalterman