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 $Alive > Suspect \land Suspect > Dead$

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 $messages$
```

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

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

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

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.

```
$WithMessage(m, msgs) \triangleq$
    IF $m \in$ DOMAIN $msgs$ THEN
        $[msgs$ 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
        $[msgs$ 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.

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

The network duplicates a message

```
$DuplicateMessage(m) \triangleq$
    $\land messages[m] = 1$
    $\land Send(m)$
    $\land$ UNCHANGED $\langle term, members, updates, history \rangle$

The network drops a message
$DropMessage(m) \triangleq$
    $\land messages[m] > 0$
    $\land Discard(m)$
    $\land$ UNCHANGED $\langle term, members, updates, history \rangle$
```

```
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.
$UpdateState(source, dest, tm, state) \triangleq$
    $\land members' = [members$ EXCEPT $![source][dest] = [term \mapsto tm, state \mapsto state]]$
    $\land RecordUpdate(source, [id \mapsto dest, term \mapsto tm, state \mapsto state])$
    $\land RecordHistory(source, dest, tm, state)$

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

```
$Probe(source, dest) \triangleq$
    $\land source \neq dest$
    $\land term[source] \neq Nil$
    $\land SendProbe(source, dest, members[source][dest])$
    $\land \text{UNCHANGED } \langle term, members, updates, history \rangle$
```

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 gossipped, and respond with the updated term. If the destination's term is greater than the source's term, just send an ack.

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

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.

```
$HandleAck(source, dest, message) \triangleq$
    $\land \lor \land message.term > members[dest][source].term$
            $\land UpdateState(dest, source, message.term, Alive)$
       $\lor \land message.term \leq members[dest][source].term$
            $\land \text{UNCHANGED } \langle members, updates, history \rangle$
    $\land \text{UNCHANGED } \langle term, messages \rangle$
```

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*.

```
$HandleFail(source, dest, message) \triangleq$
    $\land \lor \land message.term > 0$
            $\land message.term = members[source][dest].term$
            $\land members[source][dest].state = Alive$
            $\land UpdateState(source, dest, message.term, Suspect)$
    $\land$ UNCHANGED $\langle term, members, updates \rangle$
```

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.

```
$Expire(source, dest) \triangleq$
    $\land source \neq dest$
    $\land members[source][dest].state = Suspect$
    $\land UpdateState(source, dest, members[source][dest].term, Dead)$
    $\land$ UNCHANGED $\langle term \rangle$
```

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

```
$Gossip(source, dest) \triangleq$
    $\land source \neq dest$
    $\land members[source][dest].state \neq Nil$
    $\land Len(updates[source]) > 0$
    $\land SendGossip(source, dest, updates[1])$
    $\land PopUpdate(source)$
    $\land$ UNCHANGED $\langle term, members, history \rangle$
```

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 gossipped 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.

```
$HandleGossipUpdate(source, dest, message) \triangleq$
```

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

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 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

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

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.

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

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

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

Type invariant

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

Spec

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

---

* Modification History
* Last modified *Mon Oct* 08 18:06:26 *PDT* 2018 by *jordanhalterman*
* Created *Mon Oct* 08 00:36:03 *PDT* 2018 by *jordanhalterman*