MODULE *WSAtomicTransaction*

EXTENDS *TLC*

This is a preliminary version of a formal specification of the Web Services Atomic Transaction protocol, described in the document by Cabrera et al. We are specifying the safety property of the protocol (what is allowed to happen), not its liveness property (what must eventually happen).

The protocol involves an initiator, a transaction coordinator (*TC*), and a set of participants. The *TC* exchanges messages with the participants. For convenience, we assume that the initiator and *TC* are actually executed on the same processor, so they can be considered to be a single process.

The protocol allows messages to be lost, duplicated, or received out of order. A process is therefore free to resend a message at any time. An implementation will resend a message if it times out without receiving a reply. Such resending is not described explicitly in the specification. Instead, we represent the communication infrastructure by a set *msgs* of all messages that have ever been sent. Since resending a sent message does not change the specification's state, it is allowed by our specification.

An action that, in an implementation, would be enabled by the receipt of certain messages is here enabled by the existence of those messages in *msgs*. Loss of a message is represented by simply not executing that action, even though it is enabled. This works because we are specifying only safety properties, so there is no requirement that an enabled action is ever executed.

Since messages are never removed from *msgs*, receipt of the same message twice is allowed. This can happen in an implementation either because the communication infrastructure delivers a duplicate copy, or because the sender mistakenly believed the original copy had been lost and resent the message. In most cases, the specification says that such duplicate copies are ignored because the response has already been sent, so *msgs* already contains the response. However, in an implementation, receipt of a duplicate message may indicate that the sender resent the message because it never received the response. Hence, an implementation would resend the response. The specification sometimes asserts that such responses are in *msgs*, indicating that they would probably be resent in an implementation.

The CONSTANT and VARIABLES sections define constant parameters and variables.

CONSTANT *Participant* The set of all participants.

VARIABLES *iState*, The state of the initiator.
tcData, The data maintained by the coordinator.
pData, *pData*[*p*] is the data maintained by participate *p*.
msgs The set of all sent messages.

Message $\triangleq$

The set of all possible messages. A message sent from the *TC* to a participant has a *dest* field indicating its destination. A message from a participant to the *TC* has a *src* field indicating its sender. A participant's "Register" message also has a *reg* field indicating if it's registering as volatile or durable.

$$
\begin{array}{ll}
 & [type : \{\text{"RegisterResponse"}, \text{"Prepare"}, \text{"Commit"}, \text{"Rollback"}\}, \\
 & \;\; dest : Participant] \\
\cup & [type : \{\text{"Prepared"}, \text{"ReadOnly"}, \text{"Committed"}, \text{"Aborted"}\}, \\
 & \;\; src \;\; : Participant] \\
\cup & [type : \{\text{"Register"}\}, \\
 & \;\; reg \;\; : \{\text{"volatile"}, \text{"durable"}\}, \\
 & \;\; src \;\; : Participant]
\end{array}
$$

$TypeOK \triangleq$

The type-correctness invariant, indicating the possible values that can be assumed by the variables.

$$\land\ iState \in \{\text{"active"}, \text{"committed"}, \text{"aborted"}, \text{"completing"}\}$$

Because we are assuming that the initiator and the TC are the same process, there is no need for the initiator to have a separate "aborting" state. Once the decision to abort has been made by the TC, the then initiator knows that the transaction is aborted.

For convenience, we ignore the action in which the initiator registers the transaction with the TC. Instead, we assume that the initiator and TC begin in the "active" state. We also assume that the initiator never forgets the outcome. (In an implementation, the initiator can forget about the transaction as soon as it knows the outcome. It is forgetting by the TC that's tricky.)

$$\begin{aligned}\land\ tcData \in\ & [st : \{\text{"active"}, \text{"preparingVolatile"}, \text{"preparingDurable"}, \\ & \quad\quad \text{"aborting"}, \text{"committing"}\}, \\ & \ reg : [Participant \rightarrow \{\text{"unregistered"}, \text{"volatile"}, \text{"durable"}, \\ & \quad\quad \text{"prepared"}, \text{"readOnly"}, \text{"committed"}\}]] \\ \cup\ & [st : \{\text{"ended"}\}, \\ & \ res : \{\text{"committed"}, \text{"aborted"}\}]\end{aligned}$$

$tcData$ has an st component that indicates the TC's state. While the TC is performing the transaction, $tcData$ also has a reg field such that $tcData.reg[p]$ indicates the TC's knowledge of the state of participant p. The TC enters the "ended" state when it forgets about the transaction. For convenience in understanding the protocol, when $tcData =$ "ended", we let $tcData$ have a res component that indicates the outcome (whether the transaction committed or aborted). No TC actions depend on the value of $tcData.res$.

Note: The document "Web Services Atomic Transaction *Commit*" lists a "Preparing" and a "*PreparedSuccess*" state. We have split the "Preparing" state into "preparingVolatile", entered initially, and "preparingDurable", entered after all volatile participants have prepared. We have eliminated the "*PreparedSuccess*" state because that is an internal state, not visible to other processes. We model the protocol by having the TC go directly from the "preparingDurable" state into either the "Committing" or "Aborting" state. (An implementation is free to split any of our specification's states into substates.)

$$\begin{aligned}\land\ pData \in [Participant \rightarrow\ & [st : \{\text{"unregistered"}, \text{"prepared"}\}] \\ \cup\ & [st : \{\text{"registering"}, \text{"active"}, \text{"preparing"}\}, \\ & \ reg : \{\text{"volatile"}, \text{"durable"}\}] \\ \cup\ & [st : \{\text{"ended"}\}, \\ & \ res : \{\text{"?"}, \text{"committed"}, \text{"aborted"}\}]]\end{aligned}$$

$pData[p]$ is the data maintained by participant p. It contains an st field indicating the participant's state. When in the "registering" or "active" state, there is also a reg field indicating if the participant is volatile or durable. When a registered participant forgets about the transaction, it enters the "ended" state.

To help us understand the protocol, when $pData[p].st =$ "ended", there is a field $pData[p].res$ that indicates the participant's knowledge of the outcome when it forgot about the transaction. The value "?" indicates that the participant was read-only, so it didn't learn the outcome.

Note that the document "Web Services Atomic Transaction *Commit*" does not distinguish between the "unregistered" and "registering" state, which it lumps with the "ended" state into a single "None" state. The document also has "Prepared" and "*PreparedSuccess*" states that we have combined into the single "prepared" state. Those states are not visible externally (that is, by the TC) as different states. (An implementation is free to split any of our specification's states into substates.) We might have lumped the "preparing" and "prepared" states together as well, but it seems convenient to keep them separated because of the interaction of preparing and registration of durable participants.

We have also eliminated the "committing" and "aborting" states, having the participant immediately forget the transaction by going to the "ended" state. In a similar way, we could have eliminated the "aborting" state of the TC, but didn't for no good reason. Perhaps we will in the next version.

$\land\ msgs \subseteq Message$

msgs equals a set of messages.

$Consistency\ \triangleq$

A predicate that implies that the protocol is not in an inconsistent final or finishing state–that is, where one process thinks the protocol committed and another thinks it has aborted. There are two separate conjuncts, one asserting what's true if the initiator has reached the "committed" state, and the other asserting what's true if a participant has reached the "committed" state. These two conjuncts are not logically independent, but we have not eliminated the redundancy in order to make it clear what is being asserted. The invariance of this predicate is the correctness property that we check.

$$
\begin{array}{l}
\land\ (iState = \text{“committed”}) \\
\quad \Rightarrow\ \lor\ \land\ tcData.st = \text{“ended”} \\
\qquad\qquad \land\ tcData.res = \text{“committed”} \\
\qquad\qquad \land\ \forall\, p \in Participant : \\
\qquad\qquad\quad \lor\ pData[p].st = \text{“unregistered”} \\
\qquad\qquad\quad \lor\ \land\ pData[p].st = \text{“ended”} \\
\qquad\qquad\qquad \land\ pData[p].res \in \{\text{“?”}, \text{“committed”}\} \\
\qquad\quad \lor\ \land\ tcData.st = \text{“committing”} \\
\qquad\qquad \land\ \forall\, p \in Participant : \\
\qquad\qquad\quad \lor\ pData[p].st \in \{\text{“unregistered”}, \text{“prepared”}\} \\
\qquad\qquad\quad \lor\ \land\ pData[p].st = \text{“ended”} \\
\qquad\qquad\qquad \land\ pData[p].res \in \{\text{“?”}, \text{“committed”}\}
\end{array}
$$

$$
\begin{array}{l}
\land\ \forall\, p \in Participant : \\
\quad \land\ pData[p].st = \text{“ended”} \\
\quad \land\ pData[p].res = \text{“committed”} \\
\quad \Rightarrow\ \land\ iState = \text{“committed”} \\
\qquad \land\ \lor\ \land\ tcData.st = \text{“ended”} \\
\qquad\qquad\quad \land\ tcData.res = \text{“committed”} \\
\qquad\qquad\quad \land\ iState = \text{“committed”} \\
\qquad\quad\ \lor\ tcData.st = \text{“committing”} \\
\qquad \land\ \forall\, pp \in Participant : \\
\qquad\quad \lor\ pData[pp].st \in \{\text{“unregistered”}, \text{“prepared”}\} \\
\qquad\quad \lor\ \land\ pData[pp].st = \text{“ended”} \\
\qquad\qquad \land\ pData[pp].res \in \{\text{“?”}, \text{“committed”}\}
\end{array}
$$

$Init \triangleq$

The initial predicate.

$$
\begin{aligned}
&\land\ iState\ = \text{"active"}\\
&\land\ tcData = [st \mapsto \text{"active"},\\
&\qquad\qquad reg \mapsto [p \in Participant \mapsto \text{"unregistered"}]]\\
&\land\ pData\ = [p \in Participant \mapsto [st \mapsto \text{"unregistered"}]]\\
&\land\ msgs\ = \{\}
\end{aligned}
$$

THE ACTIONS

The next-state action is the disjunction of the four actions *TCInternal*, *TCRcvMsg*, *PInternal*, and *PRcvMsg*. The major part of the specification consists of the definitions of these four actions, which follow.

$TCInternal \triangleq$

This action describes the actions of the initiator and the internal actions of the *TC*–that is, the actions of the initiator prompted by timeouts or by a spontaneous action of the initiator. It also describes actions enabled by the *TC* having received enough messages. (Those actions could be described as occurring when the the *TC* receives the last message needed to enable it, but it's more convenient to let it be done as a separate internal action.)

$\land\ \lor$ The initiator decides to commit the transaction. It sets its state to "completing". At the same time, the *TC* sets its state to "preparingVolatile" and sends "Prepare" messages to every participant that has registered as volatile.

$$
\begin{aligned}
&\land\ iState = \text{"active"}\\
&\land\ \forall p \in Participant :\\
&\qquad (pData[p].st = \text{"registering"}) \Rightarrow\\
&\qquad\quad \land\ pData[p].reg = \text{"durable"}\\
&\qquad\quad \land\ \exists pp \in Participant :\\
&\qquad\qquad\quad \land\ pData[pp].st \in \{\text{"active"}, \text{"preparing"}\}\\
&\qquad\qquad\quad \land\ pData[pp].reg = \text{"volatile"}
\end{aligned}
$$

The only participants that may be registering are durable ones that are being installed by a volatile participant that is not yet prepared. It is up to the application to ensure that this condition is met.

$$
\begin{aligned}
&\land\ iState' = \text{"completing"}\\
&\land\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{"preparingVolatile"}]\\
&\land\ msgs' = msgs \cup [type : \{\text{"Prepare"}\},\\
&\qquad\qquad\qquad\qquad dest : \{p \in Participant :\\
&\qquad\qquad\qquad\qquad\qquad tcData.reg[p] = \text{"volatile"}\}]
\end{aligned}
$$

$\lor$ Either the initiator decides to abort the transaction while in its "active" state, or else the *TC* decides to abort it while in a "preparing" state–presumably because of some timeout. The initiator's state is set to "aborted", the *TC* state is set to "aborting", and "Rollback" messages are sent to every registered participant from which the *TC* did not already receive a "ReadOnly" message.

$$
\begin{aligned}
&\land\ \lor\ iState = \text{"active"}\\
&\quad\ \lor\ tcData.st \in \{\text{"preparingVolatile"}, \text{"preparingDurable"}\}\\
&\land\ iState' = \text{"aborted"}\\
&\land\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{"aborting"}]
\end{aligned}
$$

$\land\ msgs' = msgs \cup [type : \{\text{"Rollback"}\},$
$\quad dest : \{p \in Participant :$
$\quad\quad tcData.reg[p] \notin \{\text{"unregistered"}, \text{"readOnly"}\}\}]$

$\lor$ The TC ends the volatile prepare and begins the durable prepare. It does this when it has received "Prepared" or "ReadOnly" messages from every participant that registered as volatile, and it sends "Prepare" message to every participant that registered as durable.

$\land\ tcData.st = \text{"preparingVolatile"}$
$\land\ \forall p \in Participant : tcData.reg[p] \neq \text{"volatile"}$
$\land\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{"preparingDurable"}]$
$\land\ msgs' = msgs \cup [type : \{\text{"Prepare"}\},$
$\quad dest : \{p \in Participant :$
$\quad\quad tcData.reg[p] = \text{"durable"}\}]$
$\land\ \text{UNCHANGED}\ iState$

$\lor$ The TC finishes the durable prepare and decides to commit the transaction. It can do this when it has received a "Prepared" or "ReadOnly" message from every durable particpant. It sets its state to "committing", notifies the initiator that the transaction has committed, and sends "Commit" messages to every participant that replied with a "Prepared" message (instead of a "ReadOnly" message).

$\land\ tcData.st = \text{"preparingDurable"}$
$\land\ \forall p \in Participant : tcData.reg[p] \neq \text{"durable"}$
$\land\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{"committing"}]$
$\land\ msgs' = msgs \cup [type : \{\text{"Commit"}\},$
$\quad dest : \{p \in Participant :$
$\quad\quad tcData.reg[p] = \text{"prepared"}\}]$
$\land\ iState' = \text{"committed"}$

$\lor$ The action by which the TC forgets the transaction, entering the "ended" state. It can do this if it is in the "aborting" state, or if it is in the "committing" state and has received a "ReadOnly" or "Committed" message from every registered participant.

$\land\ \lor\ tcData.st = \text{"aborting"}$
$\quad \lor\ \land\ tcData.st = \text{"committing"}$
$\quad\quad \land\ \forall p \in Participant :$
$\quad\quad\quad tcData.reg[p] \in \{\text{"unregistered"}, \text{"readOnly"}, \text{"committed"}\}$
$\land\ tcData' = [st \mapsto \text{"ended"},$
$\quad res \mapsto \text{IF}\ tcData.st = \text{"aborting"}\ \text{THEN}\ \text{"aborted"}$
$\quad\quad \text{ELSE}\ \text{"committed"}]$
$\land\ \text{UNCHANGED}\ \langle iState, msgs \rangle$

$\land\ \text{UNCHANGED}\ pData$

The participants' states are left unchanged.

The following action definition uses the TLA+ construct

$\text{CASE}\ B1 \rightarrow P1\ \Box\ \ldots\ \Box Bn \rightarrow Pn$

This construct is used here (and in most places) when the state predicates Bi, which are called the *guards*, are mutually disjoint–that is, no two of them can be true in the same state. When the guards are mutually disjoint, the CASE expression equals Pi if guard Bi is true. If none of the guards are true, then the value of the expression is undefined. If TLC ever evaluates such an undefined expression, it reports an error. Thus, this CASE statement is equivalent to the formula

$$(B1 \land P1) \lor \ \ldots \ \lor (Bn \land Pn)$$

except that when none of the guards are true, the formula equals FALSE while the value of the CASE statement is undefined.

In this following action, the guards of each CASE formula describe all the possible cases in which the TC can receive a particular message.

$TCRcvMsg \triangleq$

The action in which the TC receives a message from a participant.

$\exists\, m \in msgs :$

m is the message being received.

LET $Reply(tp) \triangleq msgs' = msgs \cup \{[type \mapsto tp,\ dest \mapsto m.src]\}$

Locally defines $Reply(tp)$ to be the action of sending a message of type tp to the sender of message m.

$HaveSent(tp) \triangleq \exists\, mm \in msgs : (mm.type = tp) \land (mm.dest = m.src)$

Locally defines $HaveSent(tp)$ to be true iff the TC has sent a message of type tp to the sender of m.

IN $\lor$ m is a "Register" message.

$\land\ m.type =$ "Register"

$\land$ CASE The normal case, in which the TC state is either "active", or else this is a durable participant registering while the TC is performing the volatile prepare.

$\lor\ tcData.st =$ "active"

$\lor\ \land\ tcData.st =$ "preparingVolatile"

$\quad \land\ m.reg =$ "durable"

$\rightarrow$ The TC sends a "RegisterResponse" reply to the sender, and sets the appropriate $tcData.reg$ component (the one corresponding to the sender) to the contents of the reg field of m.

$\land\ Reply($"RegisterResponse"$)$

$\land\ tcData' = [tcData$ EXCEPT $!.reg[m.src] = m.reg]$

$\Box$ If the TC is in a "preparing" or "committing" state or has forgotten the transaction, then m is a duplicate message to which the TC has already responded and so is ignored.

$\land\ \lor\ \land\ tcData.st =$ "preparingVolatile"

$\quad \land\ m.reg =$ "volatile"

$\lor\ tcData.st \in \{$"preparingDurable", "committing"$\}$

$\land\ HaveSent($"RegisterResponse"$)$

$\rightarrow$

UNCHANGED $\langle tcData,\ msgs\rangle$

$\Box$ If the TC is in the "aborting" state, then if the sender is not already registered, then the decision to abort was made before the sender could register, in which case a "Rollback" message is sent. (Is this correct?) Otherwise, this is a duplicate message to which the TC has already responded, and it has already sent a "Rollback" message to the sender (unless the participant responded "ReadOnly" to a "Prepare" message).

$\land\ tcData.st = \text{"aborting"}$
$\land\ \lor\ tcData.reg[m.src] \in \{\text{"unregistered"}, \text{"readOnly"}\}$
$\quad\ \lor\ \land\ tcData.reg[m.src] \in \{\text{"volatile"}, \text{"durable"}, \text{"prepared"}\}$
$\qquad\ \land\ HaveSent(\text{"Rollback"})$
$\rightarrow$
$\land$ IF $tcData.reg[m.src] = \text{"unregistered"}$
THEN $Reply(\text{"Rollback"})$
ELSE UNCHANGED $msgs$
$\land$ UNCHANGED $tcData$

$\Box$ If the TC is in the "ended" state, then either the transaction has been aborted before the sender had a chance to register, or else the "Register" message is old and, if it was committed, then the sender has already forgotten the transaction and hence will ignore any message it receives. Therefore, it's safe to send a "Rollback" message.

$\land\ tcData.st = \text{"ended"}$
$\land\ \lor\ tcData.res = \text{"aborted"}$
$\quad\ \lor\ pData[m.src].st = \text{"ended"}$
$\rightarrow$
$\land\ Reply(\text{"Rollback"})$
$\land$ UNCHANGED $tcData$

$\land$ UNCHANGED $\langle iState, pData \rangle$

The initiator's state and the participants' data are not changed.

$\lor$ m is a "Prepared" message.

$\land\ m.type = \text{"Prepared"}$
$\land$ CASE The normal case, in which the TC has sent a "Prepare" message and is waiting for the sender's reply.

$\lor\ \land\ tcData.st = \text{"preparingVolatile"}$
$\quad\ \land\ tcData.reg[m.src] = \text{"volatile"}$
$\lor\ \land\ tcData.st = \text{"preparingDurable"}$
$\quad\ \land\ tcData.reg[m.src] = \text{"durable"}$
$\rightarrow$
$\land\ tcData' = [tcData$ EXCEPT $!.reg[m.src] = \text{"prepared"}]$
$\land$ UNCHANGED $msgs$

$\Box$ The TC has forgotten the transaction.

$tcData.st = \text{"ended"}$
$\rightarrow$

The TC sends a "Rollback" reply to the sender. The transaction could either have aborted or committed. However, if the transaction has committed, then the sender will know that it has and will ignore the "Rollback" message.

$\wedge$ $Reply$("Rollback")
$\wedge$ UNCHANGED $tcData$

$\Box$

If the TC is either in a "preparing" state, or in the "aborting" or "committing" state, then it has already received and acted on a copy of m, so it does nothing.

$\vee$ $\wedge$ $tcData.st \in$ {"preparingVolatile", "preparingDurable"}
$\wedge$ $tcData.reg[m.src] =$ "prepared"
$\vee$ $\wedge$ $tcData.st =$ "aborting"
$\wedge$ $HaveSent$("Rollback")
$\vee$ $\wedge$ $tcData.st =$ "committing"
$\wedge$ $HaveSent$("Commit")
$\rightarrow$
UNCHANGED $\langle tcData, msgs \rangle$
$\wedge$ UNCHANGED $\langle iState, pData \rangle$

The initiator's state is not changed.

$\vee$ m is a "ReadOnly" message.

$\wedge$ $m.type =$ "ReadOnly"
$\wedge$ CASE The normal case, in which the TC has sent a "Prepare" message and is waiting for the sender's reply.

$\vee$ $\wedge$ $tcData.st =$ "preparingVolatile"
$\wedge$ $tcData.reg[m.src] =$ "volatile"
$\vee$ $\wedge$ $tcData.st =$ "preparingDurable"
$\wedge$ $tcData.reg[m.src] =$ "durable"
$\rightarrow$

The TC sets its $tcData.reg$ component corresponding to the sender to "readOnly".

$\wedge$ $tcData' = [tcData$ EXCEPT $!.reg[m.src] =$ "readOnly"]
$\wedge$ UNCHANGED $msgs$

$\Box$ If the TC has forgotten the transaction, then either m is a duplicate message, or else it was decided to abort the transaction before the TC received the response to the "Prepare" message it sent to the sender of m. In either case, the message is ignored.

$tcData.st =$ "ended"
$\rightarrow$
UNCHANGED $\langle tcData, msgs \rangle$

$\Box$ In the following cases, m is a duplicate of a message that the TC has already received and it is ignored.

$\lor\ \land\ tcData.st \in \{\text{"preparingVolatile"}, \text{"preparingDurable"}\}$
$\quad \land\ tcData.reg[m.src] = \text{"readOnly"}$
$\lor\ \land\ tcData.st = \text{"aborting"}$
$\quad \land\ \lor\ tcData.reg[m.src] = \text{"readOnly"}$
$\quad\quad \lor\ \land\ tcData.reg[m.src] \in \{\text{"volatile"}, \text{"durable"}\}$
$\quad\quad\quad \land\ HaveSent(\text{"Rollback"})$
$\lor\ tcData.st = \text{"committing"}$
$\longrightarrow$
$\text{UNCHANGED}\ \langle tcData, msgs \rangle$

$\land\ \text{UNCHANGED}\ \langle iState, pData \rangle$

The initiator's state and the participants' data are not changed.

$\lor$ m is an "Aborted" message.

$\land\ m.type = \text{"Aborted"}$

$\land$ CASE The normal case, in which the TC receives the message when it is "active" or in a "preparing" state and the sender has not replied to a "Prepare" message. In this case, the transaction is aborted, the TC sends "Rollback" messages to all registered participants from which it has not already received a "ReadOnly" message, and the initiator is notified that the transaction has been aborted.

$\land\ tcData.st \in \{\text{"active"}, \text{"preparingVolatile"}, \text{"preparingDurable"}\}$
$\land\ tcData.reg[m.src] \in \{\text{"unregistered"}, \text{"volatile"}, \text{"durable"}\}$
$\longrightarrow$
$\land\ iState' = \text{"aborted"}$
$\land\ tcData' = [tcData\ \text{EXCEPT}\ !.st = \text{"aborting"}]$
$\land\ msgs' = msgs \cup$
$\quad [type : \{\text{"Rollback"}\},$
$\quad\ dest : \{p \in Participant :$
$\quad\quad tcData.reg[p] \notin$
$\quad\quad\quad \{\text{"unregistered"}, \text{"readOnly"}\}\}]$

11 *Dec* 2003, changed "ReadOnly" $\rightarrow$ "readOnly"
typo found by *Colin Campbell*

$\land\ \text{UNCHANGED}\ pData$

$\Box$ If the TC is already in the "aborting" state or it has forgotten an aborted transaction, then the TC has already sent a "Rollback" message to the sender (perhaps because m is a duplicate of a message the TC already received). If the TC has forgotten a committed transaction, then this "Aborted" message was sent because the sender received an obsolete "Prepare" message after it had forgotten the transaction. In either case, m is ignored.

$\land\ \lor\ tcData.st = \text{"aborting"}$
$\quad \lor\ \land\ tcData.st = \text{"ended"}$
$\quad\quad \land\ \lor\ \land\ tcData.res = \text{"aborted"}$
$\quad\quad\quad\quad \land\ HaveSent(\text{"Rollback"})$
$\quad\quad\quad \lor\ \land\ tcData.res = \text{"committed"}$

$\land\ pData[m.src].st\ \ = \text{"ended"}$
$\land\ pData[m.src].res = \text{"committed"}$

$\rightarrow$

UNCHANGED $\langle tcData,\ pData,\ iState,\ msgs\rangle$

$\Box$ If the TC is in the "committing" or "ended" state, or it is in a preparing state and the sender has already responded to the "Prepared" message, then the message is ignored. (It could have been sent in response to a duplicate "Prepared" message after the participant had reached the "ended" state.)

$\land\ \lor\ tcData.st \in \{\text{"committing"}, \text{"ended"}\}$
$\quad \lor\ \land\ tcData.st \in \{\text{"active"}, \text{"preparingVolatile"}, \text{"preparingDurable"}\}$
$\quad\quad \land\ tcData.reg[m.src] \in \{\text{"prepared"}, \text{"readOnly"}, \text{"committed"}\}$

$\rightarrow$

UNCHANGED $\langle tcData,\ pData,\ iState,\ msgs\rangle$

$\lor$ m is a "Committed" message.

$\land\ m.type = \text{"Committed"}$

$\land$ CASE The normal case, in which the TC is in the "committing" state. In this case, it sets the element of $tcData.reg$ corresponding to the sender to "committed".

$tcData.st = \text{"committing"}$

$\rightarrow$

$tcData' = [tcData\ \text{EXCEPT}\ !.reg[m.src] = \text{"committed"}]$

$\Box$ If the TC has forgotten the transaction, then the transaction has been committed and m is ignored.

$\land\ tcData.st\ \ = \text{"ended"}$
$\land\ tcData.res = \text{"committed"}$

$\rightarrow$

UNCHANGED $tcData$

$\land$ UNCHANGED $\langle iState,\ pData,\ msgs\rangle$

The initiator's state and the participants' data are unchanged, and no messages are sent.

$PInternal \triangleq$

This action describes the internal actions of the participants–actions that occur "spontaneously", either prompted by timeouts or, as in the case of the register action, by some communication external to the protocol.

$\exists\, p \in Participant :$

p is the participant performing the action.

LET $SendMsg(tp) \triangleq msgs' = msgs \cup \{[type \mapsto tp,\ src \mapsto p]\}$
$\quad SendRegisterMsg(rg) \triangleq msgs' = msgs \cup \{[type \mapsto \text{"Register"},\ src \mapsto p,\ reg \mapsto rg]\}$

Locally defined action expressions. $SendMsg(tp)$ sends a message of type tp from participant p to the TC. $SendRegisterMsg(rg)$ sends a $Register$ rg message, where rg is either "durable" or "volatile".

IN $\lor$ p registers as a volatile participant. It can do this only if it is *unregistered* and the initiator is in the "active" state.

$$\begin{aligned}
&\land pData[p].st = \text{"unregistered"}\\
&\land iState\ = \text{"active"}\\
&\land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{"registering"},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad reg \mapsto \text{"volatile"}]]\\
&\land SendRegisterMsg(\text{"volatile"})\\
&\land \text{UNCHANGED } \langle iState,\ tcData\rangle
\end{aligned}$$

$\lor$ p registers as a durable participant. It can do this only if it is *unregistered* and, if either the initiator is in the "active" state, or there is some volatile participant that is willing to wait for p to register before preparing. Since we don't model "willingness to wait", we allow the participant to register as long as there is some volatile participant that can wait for it.

$$\begin{aligned}
&\land pData[p].st = \text{"unregistered"}\\
&\land\ \lor iState = \text{"active"}\\
&\quad\ \lor \exists\, pp \in Participant :\\
&\qquad\qquad \land pData[pp].st \in \{\text{"active"},\ \text{"preparing"}\}\\
&\qquad\qquad \land pData[pp].reg = \text{"volatile"}\\
&\land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{"registering"},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad reg \mapsto \text{"durable"}]]\\
&\land SendRegisterMsg(\text{"durable"})\\
&\land \text{UNCHANGED } \langle iState,\ tcData\rangle
\end{aligned}$$

$\lor$ p spontaneously aborts and forgets about the transaction. We do not allow it to abort in the "registering" state. If we allowed this, then we could wind up with a situation in which a participant aborted before it registered, and the transaction committed anyway. In practice, there will have to be some way for a participant to give up when it hasn't received a *RegisterResponse* message. However, to do this, it must learn from the initiator or a volatile participant that the transaction aborted so it can forget about it. Since we are not modeling this kind of inter-participant communication, we do not model this procedure.

$$\begin{aligned}
&\land pData[p].st \in \{\text{"active"},\ \text{"preparing"}\}\\
&\land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{"ended"},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad res \mapsto \text{"aborted"}]]\\
&\land SendMsg(\text{"Aborted"})\\
&\land \text{UNCHANGED } \langle iState,\ tcData\rangle
\end{aligned}$$

$\lor$ p either prepares or becomes read-only. If p is volatile, then it cannot do this if there is a durable participant that is in the "registering" state, and there is no other volatile participant to wait for it to register.

$$\begin{aligned}
&\land pData[p].st = \text{"preparing"}\\
&\land\ \lor pData[p].reg = \text{"durable"}\\
&\quad\ \lor \neg\exists\, dp \in Participant :\ \land pData[dp].st\ = \text{"registering"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \land pData[dp].reg = \text{"durable"}
\end{aligned}$$

$\lor \exists vp \in Participant \setminus \{p\} : \land pData[vp].st = \text{"active"}$
$\qquad\qquad\qquad\qquad\qquad\qquad \land pData[vp].reg = \text{"volatile"}$

11 *Dec* 03 : *Colin Campbell* observes that the "$\setminus \{p\}$" is unnecessary. It should probably be removed.

$\land \lor \land pData' = [pData$ EXCEPT $![p] = [st \mapsto \text{"prepared"}]]$
$\quad \land SendMsg(\text{"Prepared"})$
$\lor \land pData' = [pData$ EXCEPT $![p] = [st \mapsto \text{"ended"},$
$\qquad res \mapsto \text{"?"}]]$
$\quad \land SendMsg(\text{"ReadOnly"})$
$\land$ UNCHANGED $\langle iState, tcData \rangle$

$PRcvMsg \triangleq$

The action in which a participant receives a message from the TC.

$\exists m \in msgs :$

m is the message being received.

LET $Reply(tp) \triangleq msgs' = msgs \cup \{[type \mapsto tp, src \mapsto m.dest]\}$

Locally defines $Reply(tp)$ to be the action of sending a message of type tp from the sender of message m.

$HaveSent(tp) \triangleq \exists mm \in msgs : (mm.type = tp) \land (mm.src = m.dest)$

Locally defines $HaveSent(tp)$ to be true iff the participant receiving m has sent a message of type tp to the TC.

IN $\land \lor$ m is a "RegisterResponse" message

$\land m.type = \text{"RegisterResponse"}$
$\land$ CASE The normal case.

$pData[m.dest].st = \text{"registering"}$
$\rightarrow$
$\land pData' = [pData$ EXCEPT $![m.dest].st = \text{"active"}]$
$\land$ UNCHANGED $msgs$

$\Box$ This is a duplicate of an already-received message, or else it is ignorable because another message to the participant arrived ahead of it.

$pData[m.dest].st \in$
$\{\text{"active"}, \text{"preparing"}, \text{"prepared"}, \text{"ended"}\}$
$\rightarrow$
UNCHANGED $\langle pData, msgs \rangle$

$\lor$ m is a "Prepare" message.

$\land m.type = \text{"Prepare"}$
$\land$ This is either the normal case (the participant is in the "active" state), or else this "Prepare" message has arrived before the "RegisterResponse" message.

CASE $pData[m.dest].st \in \{\text{"registering"}, \text{"active"}\}$
$\rightarrow \land pData' = [pData$ EXCEPT $![m.dest].st = \text{"preparing"}]$
$\quad \land$ UNCHANGED $msgs$

$\Box$ This is a duplicate of a message already received.

$pData[m.dest].st \in \{\text{“preparing”}, \text{“prepared”}\}$

$\rightarrow$

$\text{UNCHANGED } \langle pData, msgs \rangle$

$\Box$ The transaction has been forgotten. The participant responds with an “Aborted” message–even though the transaction might have committed or be in the process of committing. In the latter case, the “Aborted” message will be ignored by the TC.

$\land\ pData[m.dest].st = \text{“ended”}$

$\land\ \lor\ \land\ pData[m.dest].res = \text{“committed”}$

$\quad\quad\ \land\ HaveSent(\text{“Committed”})$

$\quad\ \lor\ \land\ pData[m.dest].res = \text{“aborted”}$

$\quad\quad\ \land\ \lor\ HaveSent(\text{“Aborted”})$

$\quad\quad\quad\ \lor\ \land\ tcData.st = \text{“ended”}$

$\quad\quad\quad\quad\ \land\ tcData.res = \text{“committed”}$

$\quad\ \lor\ \land\ pData[m.dest].res = \text{“?”}$

$\quad\quad\ \land\ HaveSent(\text{“ReadOnly”})$

$\rightarrow$

$\land\ Reply(\text{“Aborted”})$

$\land\ \text{UNCHANGED } pData$

$\lor$ m is a “Commit” message.

$\land\ m.type = \text{“Commit”}$

$\land$ The normal case.

$\text{CASE } pData[m.dest].st = \text{“prepared”}$

$\rightarrow$

$\land\ pData' =$

$\quad [pData \text{ EXCEPT } ![m.dest] = [st \mapsto \text{“ended”},$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad res \mapsto \text{“committed”}]]$

$\land\ Reply(\text{“Committed”})$

$\Box$ The transaction has ended, so this must be a message that was already received.

$\land\ pData[m.dest].st = \text{“ended”}$

$\land\ pData[m.dest].res \in \{\text{“?”}, \text{“committed”}\}$

$\rightarrow$

$\text{UNCHANGED } \langle pData, msgs \rangle$

$\lor$ m is a “Rollback” message.

$\land\ m.type = \text{“Rollback”}$

$\land$ The participant can be in any registered state. If it hasn’t ended, then this causes it to abort. It also sends an “Aborted” message to the TC. This message isn’t needed, but it is apparently used as an acknowledgement.

$\text{CASE } pData[m.dest].st \in$

$\quad \{\text{“registering”}, \text{“active”}, \text{“preparing”}, \text{“prepared”}\}$

$\rightarrow$

$\land\ pData' =$

$[pData\ \textsc{except}\ ![m.dest] = [st \mapsto \text{"ended"}, res \mapsto \text{"aborted"}]]$

$\land\ Reply(\text{"Aborted"})$

$\Box$ If the participant has already finished, then this is ignored. It is possible for this message to arrive even though the participant has ended by committing the transaction. In this case, the "Rollback" message was generated by a duplicate "Register" message arriving at the *TC* after it had forgotten the transaction.

$pData[m.dest].st = \text{"ended"}$

$\rightarrow$

$\textsc{unchanged}\ \langle pData, msgs \rangle$

$\land\ \textsc{unchanged}\ \langle iState, tcData \rangle$

$Next \triangleq TCInternal \lor TCRcvMsg \lor PInternal \lor PRcvMsg$

The specification's next-state action.

$vars \triangleq \langle iState, tcData, pData, msgs \rangle$

The tuple of all variables.

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

The complete spec of the two-phase *Commit* protocol.

$\textsc{theorem}\ Spec \Rightarrow \Box(TypeOK \land Consistency)$

This theorem asserts that the predicates *TypeOK* and *Consistency* are invariants of the specification. *TLC* checked this with 4 participants. It generated 10269919 states, 504306 of them were distinct. The longest non-repeating behavior had 45 states. It took *TLC* about 4-1/4 minutes on a 2-processor, 2.4*GHz* machine.

Last modified on *Thu Dec* 11 13:27:41 *PT* 2003 by lamport