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, tcData, pData, $msgs$

- **iState**, The state of the initiator.
- tcData, The data maintained by the coordinator.
- pData, $pData[p]$ is the data maintained by participant $p$.
- 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.

- $[\text{type} : \{ \text{“RegisterResponse”}, \text{“Prepare”}, \text{“Commit”}, \text{“Rollback”} \},$
  - dest : Participant$]$
- $\cup [\text{type} : \{ \text{“Prepared”}, \text{“ReadOnly”}, \text{“Committed”}, \text{“Aborted”} \},$
  - src : Participant$]$
- $\cup [\text{type} : \{ \text{“Register”} \},$
  - reg : \{ \text{“volatile”}, \text{“durable”} \},$
  - src : Participant$]$. 

1
\( \text{TypeOK} \triangleq \)

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

\( \wedge \text{iState} \in \{ "active", "committed", "aborted", "completing" \} \)

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.)

\( \wedge \text{tcData} \in \)
\[
\begin{align*}
&\{ \text{st} : \{ "active", "preparingVolatile", "preparingDurable", "aborting", "committing" \}, \\
&\phantom{\{ \text{st} : } \text{reg} : \{ \text{Participant} \rightarrow \{ "unregistered", "volatile", "durable", "prepared", "readOnly", "committed" \} \}\}
\end{align*}
\]

\( \cup \)
\[
\begin{align*}
&\{ \text{st} : \{ "ended" \}, \\
&\phantom{\{ \text{st} : } \text{res} : \{ "committed", "aborted" \} \}
\end{align*}
\]

\( \text{tcData} \) has an \( \text{st} \) component that indicates the TC’s state. While the TC is performing the transaction, \( \text{tcData} \) also has a \( \text{reg} \) field such that \( \text{tcData}.\text{reg}[p] \) 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 \( \text{tcData} = "ended" \), we let \( \text{tcData} \) have a \( \text{res} \) component that indicates the outcome (whether the transaction committed or aborted). No TC actions depend on the value of \( \text{tcData}.\text{res} \).

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.)

\( \wedge \text{pData} \in \{ \text{Participant} \rightarrow \)
\[
\begin{align*}
&\{ \text{st} : \{ "unregistered", "prepared" \} \} \\
&\cup \{ \text{st} : \{ "registering", "active", "preparing" \}, \\
&\phantom{\{ \text{st} : } \text{reg} : \{ "volatile", "durable" \} \}
\end{align*}
\]

\( \cup \)
\[
\begin{align*}
&\{ \text{st} : \{ "ended" \}, \\
&\phantom{\{ \text{st} : } \text{res} : \{ "?", "committed", "aborted" \} \}
\end{align*}
\]

\( \text{pData}[p] \) is the data maintained by participant \( p \). It contains an \( \text{st} \) field indicating the participant’s state. When in the “registering” or “active” state, there is also a \( \text{reg} \) 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 \( \text{pData}[p].\text{st} = "ended" \), there is a field \( \text{pData}[p].\text{res} \) 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.

\( \wedge \text{msgs} \subseteq \text{Message} \)

msgs equals a set of messages.

**Consistency \( \hat{=} \)**

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.

\( \wedge (\text{iState} = \text{“committed”}) \)

\( \Rightarrow \wedge \text{tcData}.st = \text{“ended”} \)

\( \wedge \text{tcData}.res = \text{“committed”} \)

\( \wedge \forall p \in \text{Participant} : \)

\( \wedge \forall p \in \text{Participant} : \)

\( \forall p \in \text{Participant} : \)

\( \wedge \forall p \in \text{Participant} : \)

\( \Rightarrow \wedge \text{tcData}.st = \text{“ended”} \)

\( \wedge \text{tcData}.res = \text{“committed”} \)

\( \wedge \text{iState} = \text{“committed”} \)

\( \Rightarrow \wedge \forall p \in \text{Participant} : \)

\( \forall p \in \text{Participant} : \)

\( \Rightarrow \wedge \text{tcData}.st = \text{“ended”} \)

\( \wedge \text{tcData}.res = \text{“committed”} \)

\( \wedge \text{iState} = \text{“committed”} \)

\( \wedge \forall pp \in \text{Participant} : \)

\( \forall p \in \text{Participant} : \)

\( \Rightarrow \wedge \forall pp \in \text{Participant} : \)

\( \wedge \text{tcData}.st = \text{“ended”} \)

\( \wedge \text{tcData}.res = \text{“committed”} \)

\( \wedge \text{iState} = \text{“committed”} \)

\( \forall \text{msgs} \subseteq \text{Message} \)

\( \text{msgs} \) equals a set of messages.
**Init**

The initial predicate.

\[ iState = \text{"active"}, \]
\[ tcData = \{st \mapsto \text{"active"}, \]
\[ \text{reg} \mapsto \{p \in \text{Participant} \mapsto \text{"unregistered"}\}\} \]
\[ pData = \{p \in \text{Participant} \mapsto \{st \mapsto \text{"unregistered"}\}\} \]
\[ msgs = \{\} \]

---

**THE ACTIONS**

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

**TCInternal**

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.)

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.

\[ iState' = \text{"completing"}, \]
\[ tcData' = [tcData \text{ EXCEPT } !.st = \text{"preparingVolatile"}] \]
\[ msgs' = msgs \cup \{\text{dest} : \{p \in \text{Participant} : \]
\[ tcData_.reg[p] = \text{"volatile"}\}\} \]

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.

\[ iState' = \text{"aborted"}, \]
\[ tcData' = [tcData \text{ EXCEPT } !.st = \text{"aborting"}] \]
\[ \text{msgs}' = \text{msgs} \cup \{ \text{type} : \{ \text{"Rollback"} \}, \text{dest} : \{ p \in \text{Participant} : \text{tcData}.reg[p] \notin \{ \text{"unregistered"}, \text{"readOnly"} \} \} \]
This construct is used here (and in most places) when the state predicates $B_i$, 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 $P_i$ if guard $B_i$ 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

$$(B_1 \land P_1) \lor \ldots \lor (B_n \land P_n)$$

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.

$\text{TCCrvMsg} \triangleq$

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

$\exists m \in msgs :$

$m$ is the message being received.

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

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

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

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

IN $\lor \lor \text{CASE}$

$\land m.\text{type} = \text{“Register”}$

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.

$\lor tcData.st = \text{“active”}$

$\lor tcData.st = \text{“preparingVolatile”}$

$\land m.\text{reg} = \text{“durable”}$

$\rightarrow$ 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 $m$.

$\land \text{Reply(“RegisterResponse”)}$

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

$\square$ 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.

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

$\land m.\text{reg} = \text{“volatile”}$

$\lor tcData.st \in \{ \text{“preparingDurable”, “committing”} \}$

$\land \text{HaveSent(“RegisterResponse”)}$

$\rightarrow$

UNCHANGED $\langle tcData, msgs \rangle$
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).

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

The normal case, in which the TC has sent a "Prepare" message and is waiting for the sender’s reply.

\[ m \text{ is a "Prepared" message.} \]
\[ m.type = \text{"Prepared"} \]
\[ \land \text{CASE } \]
\[ \land tcData.st = \text{"preparingVolatile"} \]
\[ \land tcData.reg[m.src] = \text{"volatile"} \]
\[ \land tcData.st = \text{"preparingDurable"} \]
\[ \land tcData.reg[m.src] = \text{"durable"} \]
\[ \implies \]
\[ tcData' = [tcData \text{ EXCEPT !.reg[m.src] = "prepared"}] \]
\[ \land \text{UNCHANGED } msgs \]

The TC has forgotten the transaction.

\[ tcData.st = \text{"ended"} \]
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.

∧ Reply(“Rollback”)  
∧ UNCHANGED tcData

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.

\[
\begin{align*}
\lor \\
\land tcData.st \in \{ “preparingVolatile”, “preparingDurable” \} \\
\land tcData.reg[m.src] = “prepared” \\
\lor \\
\land tcData.st = “aborting” \\
\land HaveSent(“Rollback”) \\
\lor \\
\land tcData.st = “committing” \\
\land HaveSent(“Commit”) \\
\rightarrow \\
\land tcData′ = [tcData \text{ except } !.reg[m.src] = “readOnly”] \\
\land \text{UNCHANGED } msgs
\end{align*}
\]

\( m \) is a “ReadOnly” message.

\[ \land m.type = “ReadOnly” \land \text{CASE} \]

The normal case, in which the TC has sent a “Prepare” message and is waiting for the sender’s reply.

\[
\begin{align*}
\lor \\
\land tcData.st = “preparingVolatile” \\
\land tcData.reg[m.src] = “volatile” \\
\lor \\
\land tcData.st = “preparingDurable” \\
\land tcData.reg[m.src] = “durable” \\
\rightarrow \\
\land tcData′ = [tcData \text{ except } !.reg[m.src] = “readOnly”] \\
\land \text{UNCHANGED } msgs
\end{align*}
\]

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.

\[
\land tcData.st = “ended” \\
\rightarrow \\
\land \text{UNCHANGED } tcData, msgs
\]

In the following cases, \( m \) is a duplicate of a message that the TC has already received and it is ignored.
∨ \(\forall tcData.st \in \{\text{“preparingVolatile”, “preparingDurable”}\}\)
∧ \(tcData.reg[m.src] = \text{“readOnly”}\)
∨ \(\forall tcData.st = \text{“aborting”}\)
∧ \(\forall tcData.reg[m.src] = \text{“ readOnly”}\)
∨ \(\forall tcData.reg[m.src] \in \{\text{“ volatile”, “durable”}\}\)
∧ \(\text{HaveSent(“Rollback”)}\)
∨ \(tcData.st = \text{“committing”}\)
→
\(\text{UNCHANGED} \langle tcData, msgs \rangle\)
∧ \(\text{UNCHANGED} \langle iState, pData \rangle\)

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

∨ \( m \) is an “Aborted” message.
∧ \( m\.type = \text{“Aborted”}\)
∧ 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 noti-
tified that the transaction has been aborted.
∧ \(tcData.st \in \{\text{“active”, “preparingVolatile”, “preparingDurable”}\}\)
∧ \(tcData.reg[m.src] \in \{\text{“unregistered”, “volatile”, “durable”}\}\)
→
∧ \(iState’ = \text{“aborted”}\)
∧ \(tcData’ = [tcData\ except \ !.st = \text{“aborting”}]\)
∧ \(msgs’ = msgs \cup \{
\begin{align*}
&\{type : \{\text{“Rollback”}\}, \\
&\text{dest : } \{p \in \text{Participant : } tcData.reg[p] \notin \\
&\begin{align*}
&\{\text{“unregistered”, “readOnly”}\}\}\}
\end{align*}
\end{align*}
\}
\}
\)

∧ \(\text{UNCHANGED} pData\)

□ 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.
∧ \(\forall tcData.st = \text{“aborting”}\)
∨ \(\forall tcData.st = \text{“ended”}\)
∧ \(\forall tcData.res = \text{“aborted”}\)
∧ \(\text{HaveSent(“Rollback”)}\)
∨ \(\forall tcData.res = \text{“committed”}\)
\(\land pData[m.src].st = "ended"\)
\(\land pData[m.src].res = "committed"\)

\[\rightarrow \]
UNCHANGED \(\langle tcData, pData, iState, msgs \rangle\)

\[\text{□ If the TC is in the "committing" or "ended" state, or it is in a preparing state and the sender has already responded to the "Prepared" message, then the message is ignored. (It could have been sent in response to a duplicate "Prepared" message after the participant had reached the "ended" state.)}\]
\[\land \\lor tcData.st \in \{"committing", "ended"\} \land tcData.st \in \{"active", "preparingVolatile", "preparingDurable"\} \land tcData.reg[m.src] \in \{"prepared", "readOnly", "committed"\}\]

\[\rightarrow \]
UNCHANGED \(\langle tcData, pData, iState, msgs \rangle\)

\[\lor m\text{ is a "Committed" message.}\]
\[\land m.type = "Committed"\]
\[\land \text{CASE}\]

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".

\[tcData.st = "committing"\]

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

\[\text{□ If the TC has forgotten the transaction, then the transaction has been committed and } m \text{ is ignored.}\]
\[\land tcData.st = "ended"\]
\[\land tcData.res = "committed"\]

\[\rightarrow \]
UNCHANGED \(tcData\)
UNCHANGED \(\langle iState, pData, msgs \rangle\)

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

\(P_{\text{Internal}} \triangleq\)

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.

\(\exists p \in \text{Participant} :\)

\(p\) is the participant performing the action.

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

\[ p \text{ registers as a volatile participant. It can do this only if it is } \text{unregistered} \text{ and the initiator is in the “active” state.} \]

\[ \land pData[p].st = \text{“unregistered”} \]
\[ \land iState = \text{“active”} \]
\[ \land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{“registering”}, reg \mapsto \text{“volatile”}]] \]
\[ \land SendRegisterMsg(\text{“volatile”}) \]
\[ \land \text{UNCHANGED } (iState, tcData) \]

\[ p \text{ registers as a durable participant. It can do this only if it is } \text{unregistered} \text{ and, if either the initiator is in the “active” state, or there is some volatile participant that is willing to wait for } p \text{ to register before preparing. Since we don’t model “willingness to wait”, we allow the participant to register as long as there is some volatile participant that can wait for it.} \]

\[ \land pData[p].st = \text{“unregistered”} \]
\[ \land \lor iState = \text{“active”} \]
\[ \lor \exists pp \in Participant : \land pData[pp].st \in \{\text{“active"}, \text{“preparing"}\} \]
\[ \land pData[pp].reg = \text{“volatile”} \]
\[ \land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{“registering”}, \text{reg} \mapsto \text{“durable”}]] \]
\[ \land SendRegisterMsg(\text{“durable”}) \]
\[ \land \text{UNCHANGED } (iState, tcData) \]

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

\[ \land pData[p].st \in \{\text{“active"}, \text{“preparing"}\} \]
\[ \land pData' = [pData \text{ EXCEPT } ![p] = [st \mapsto \text{“ended”}, \text{res} \mapsto \text{“aborted”}]] \]
\[ \land SendMsg(\text{“Aborted”}) \]
\[ \land \text{UNCHANGED } (iState, tcData) \]

\[ p \text{ either prepares or becomes read-only. If } p \text{ is volatile, then it cannot do this if there is a durable participant that is in the “registering” state, and there is no other volatile participant to wait for it to register.} \]

\[ \land pData[p].st = \text{“preparing”} \]
\[ \land \forall pData[dp].reg = \text{“durable”} \]
\[ \land \neg \exists dp \in Participant : \land pData[dp].st = \text{“registering”} \]
\[ \land pData[dp].reg = \text{“durable”} \]
∀ ∃ vp ∈ Participant \{p\} : ∀ pData[vp].st = “active”
∧ pData[vp].reg = “volatile”

11 Dec 03: Colin Campbell observes that the “\{p\}” is unnecessary. It should probably be removed.

∀ ∀ pData′ = [pData EXCEPT ![p] = [st ↦ “prepared”]]
∧ SendMsg(“Prepared”)
∀ ∀ pData′ = [pData EXCEPT ![p] = [st ↦ “ended”,
res ↦ ?”]]
∧ SendMsg(“ReadOnly”)
∧ UNCHANGED ⟨iState, tcData⟩

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

∃ m ∈ msgs :
  m is the message being received.

LET Reply(tp) ≜ msgs′ = msgs ∪ \{[type ↦ tp, src ↦ 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) ≜ ∃ mm ∈ msgs : (mm.type = tp) ∧ (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
∀ ∀ m is a “RegisterResponse” message
∧ m.type = “RegisterResponse”
∧ CASE
  The normal case.
    → ∧ pData[m.dest].st = “registering”
    → ∧ pData′ = [pData EXCEPT ![m.dest].st = “active”]
    ∧ UNCHANGED msgs

□ 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 ∈
\{“active”, “preparing”, “prepared”, “ended”\}

→ UNCHANGED ⟨pData, msgs⟩

∀ m is a “Prepare” message.
∧ m.type = “Prepare”
∧ 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 ∈ \{“registering”, “active”\}
→ ∧ pData′ = [pData EXCEPT ![m.dest].st = “preparing”]
∧ UNCHANGED msgs

□ This is a duplicate of a message already received.
\( pData[m.\text{dest}].\text{st} \in \{ \text{"preparing"}, \text{"prepared"} \} \)

\[ \rightarrow \text{UNCHANGED} \langle pData, msgs \rangle \]

\( \square \) 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:

\[ \land pData[m.\text{dest}].\text{st} = \text{"ended"} \]
\[ \land \lor \land pData[m.\text{dest}].\text{res} = \text{"committed"} \]
\[ \land \text{HaveSent("Committed")} \]
\[ \lor \land pData[m.\text{dest}].\text{res} = \text{"aborted"} \]
\[ \land \lor \text{HaveSent("Aborted")} \]
\[ \lor \land tcData.st = \text{"ended"} \]
\[ \land tcData.res = \text{"committed"} \]
\[ \lor \land pData[m.\text{dest}].\text{res} = \text{"?"} \]
\[ \land \text{HaveSent("ReadOnly")} \]

\[ \rightarrow \]
\[ \land \text{Reply("Aborted")} \]
\[ \land \text{UNCHANGED} \ pData\]

\( \lor \) \( m \) is a “Commit” message.
\[ \land m.\text{type} = \text{"Commit"} \]
\[ \land \text{The normal case.} \]

\( \text{CASE} \ pData[m.\text{dest}].\text{st} = \text{"prepared"} \)

\[ \rightarrow \]
\[ \land pData' = \]
\[ [ pData \text{ EXCEPT } ![m.\text{dest}] = [ st \mapsto \text{"ended"} , \]
\[ \text{res} \mapsto \text{"committed"} ]] \]
\[ \land \text{Reply("Committed")} \]

\( \square \) The transaction has ended, so this must be a message that was already received.
\[ \land pData[m.\text{dest}].\text{st} = \text{"ended"} \]
\[ \land pData[m.\text{dest}].\text{res} \in \{ \text{"?"}, \text{"committed"} \} \]

\[ \rightarrow \text{UNCHANGED} \langle pData, msgs \rangle \]

\( \lor \) \( m \) is a “Rollback” message.
\[ \land m.\text{type} = \text{"Rollback"} \]
\[ \land \text{The participant can be in any registered state. If it hasn’t ended, then this causes it to abort. It also sends an "Aborted" message to the TC. This message isn’t needed, but it is apparently used as an acknowledgement.} \]

\( \text{CASE} \ pData[m.\text{dest}].\text{st} \in \)
\[ \{ \text{"registering"}, \text{"active"}, \text{"preparing"}, \text{"prepared"} \} \]

\[ \rightarrow \]

13
\[ pData' = [pData \text{ EXCEPT !}[m.\text{dest}] = [st \mapsto \text{"ended"}, \text{res} \mapsto \text{"aborted"}]\]

\[ \text{Reply(\"Aborted\")} \]

- 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.

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

\[ \rightarrow \text{UNCHANGED } (pData, msgs) \]

\[ \wedge \text{UNCHANGED } (iState, tcData) \]

\[ \text{Next} \triangleq TC\text{Internal} \lor TSRcvMsg \lor P\text{Internal} \lor PRcvMsg \]

The specification’s next-state action.

\[ \text{vars} \triangleq (iState, tcData, pData, msgs) \]

The tuple of all variables.

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

The complete spec of the two-phase Commit protocol.

**THEOREM** \( \text{Spec} \Rightarrow \Box(\text{TypeOK} \land \text{Consistency}) \)

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, 2.4GHz machine.

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