Validation of the Sessionless Mode of the HTTPR Protocol

Paolo Romano, Milton Romero, Bruno Ciciani, and Francesco Quaglia

DIS, Università “La Sapienza”, Roma, Italy

Abstract. Reliable delivery of messages using open and product-neutral protocols has been identified as a needed technology in enterprise computing and a fundamental middleware component in several E-Business systems. The HTTPR protocol aims at guaranteeing reliable message delivery, even in the presence of failures, by providing the sender with the ability to deliver a message once, and only once, to its intended receiver(s). This work reports the experience in the formalization and validation of the sessionless mode of the HTTPR protocol through the use of the SPIN model checker. To overcome the state space explosion problem that arose while validating the protocol, a decompositional approach was used which could be of general interest in the validation of complex systems.

1 Introduction

Reliable delivery of messages using open and product-neutral protocols has been identified as a needed technology in enterprise computing and a fundamental middleware component in several E-Business systems [5]. IBM has released the specifications of the HTTPR protocol [7], in which a set of rules is proposed in order to let messaging agents with persistent storage capabilities provide reliable messaging, with exactly once delivery semantic, even in the presence of host or network failures that are eventually recovered. Actually, reliable messaging is not a new technology. Middleware products for messaging, such as the IBM MQSeries [6], Oracle Message Broker [9], Microsoft Message Queuing [8] and Java Messaging Systems [14] have supported it for years, within product specific protocols, and are widely deployed in enterprise computing environments. The peculiarities of the HTTPR proposal with respect of the above mentioned solutions are several. Essentially HTTPR is both platform and product neutral. Moreover HTTPR is explicitly designed as a reliable messaging facility to be used outside of the scope of a single enterprise. In addition, being layered on top of HTTP, HTTPR has the additional benefit that it can be used for reliable messaging between enterprises whose only presence on the Internet is a Web server behind a firewall admitting only Web-related traffic.

This work addresses the formalization and proof of correctness of the sessionless mode of the HTTPR protocol by means of: (1) the definition of Finite State Machines (FSMs) that reflect the protocol specifications and (2) the proof of safety and liveness of the protocol. To achieve this goal, the FSMs describing the protocol behavior and the correctness claims are coded in PROMELA (PROcess MEta LAnguage) [3], the input language for the SPIN model checker [4]. Then the formal verification of safety and liveness has been performed by making the SPIN model checker exhaustively examine all the behaviors allowed by HTTPR according to the FSM representation.

HTTPR specifications provided by IBM are expressed in natural language, thus not being a completely unambiguous representation for the protocol behavior. Actually, the specifications provide accurate examples, under the form of sequence diagrams, only for failure-free runs, while they are far away from exhaustively covering all the possible failure scenarios that should be tackled by the protocol. Anyway it is crucial, in order to ensure interoperability of implementations, that developers are given a common, clear and complete description of the protocol behavior. Several times, especially in the first phase of our validation, we derived state diagrams and PROMELA models, which turned out to behave incorrectly, suffering from deadlocks or showing unexpected behaviors. The causes of such problems could often be attributed to misunderstandings of either ambiguous or under-specified fragments of the protocol specifications.

From this point of view, the results of this model-checking case study are not limited to providing a correctness proof of the protocol, in terms of safety and liveness properties. The FSM descriptions and their translations into the PROMELA language ([1]) are, actually, a refinement of the current specifications, which restricts the possible interpretations of the IBM documents to a proper subset which has the noteworthy property of being correct (according to the correctness properties defined in Section 5.1). They might help software developers in building inter-operable and reliable implementations of HTTPR taking the formal description as a reference point.

While validating the protocol through the SPIN model checker we also had to overcome the state space explosion problem. This hurdle is quite common in model checking and, if not tackled adequately, it can strongly limit the value of the validation results. Such a problem has been tackled through a decompositional approach based on the identification of a set of relations which allowed us to extend the results of the validation of a single message exchange to a whole sequence of message exchanges of arbitrary length.

The remainder of this paper is structured as follows. In Section 2 we provide an overview of HTTPR. In Section 3 we provide an overview of the approach to validation we have used. The FSM representation is presented in Section 4. The description of the PROMELA model used for the validation of the protocol, as well as the results obtained from the Spin model checker, are reported in Section 5. In the same section we justify why it is possible to exploit the results of the validation of the execution of a single command, to claim the correctness of a whole sequence of HTTPR commands. Hints on the possibility of extending our analysis to more complex protocol modes (e.g. session) are briefly discussed in the concluding section.

[1] The PROMELA code is available at: `www.dis.uniroma1.it/~quaglia`

2 Overview of HTTPR

HTTPR Version 1.1, presented by the IBM proposal in [7], is a protocol for reliable delivery of messages over the Internet even in the presence of host and/or network failures that are eventually recovered. In this paper we consider the *conditionally compliant* level that is, a protocol behavior that satisfies all the "must" level requirements but not all the "should" level requirements of the specifications [2].

HTTPR provides rules which make it possible to ensure that each message is delivered to its destination exactly once. The protocol is layered on top of HTTP. Specifically, it defines how metadata and application messages are encapsulated within the payload of HTTP requests and responses. HTTPR supports message exchange through a set of *client initiated commands*, (i.e. commands are issued at the client side) by exploiting the POST method of HTTP [2]. The payload of POST requests and responses carry HTTPR state information and, for certain commands, a batch of messages [7]. (A single message is handled as the special case of a batch with only one member.)

HTTPR can operate in three different modes. In the *sessionless mode*, message exchanges are independent of each other, thus messages can be delivered through different HTTPR channels. If the same channel is used for a set of messages, then they are delivered in the same order they have been sent. In the *simple session mode*, a session is established between the HTTPR client and server, and all messages exchanged within the session are delivered through the same HTTPR channel, according to the message send order. The *pipeline* mode allows pipelining of messages within the same session, i.e. a message can be sent through a proper command while a previously issued command is still being handled. In this paper we are interested in verifying the sessionless mode, that supports the following four commands:

PUSH. This command allows the client to send a batch of one or more messages to an HTTPR server, and to eventually get back a response indicating that the batch has been received and reliably recorded on *persistent storage*.

PULL. This command allows the client to ask an HTTPR server to send a batch of pending messages, if any, to be delivered at the client side.

EXCHANGE. This command is the combination of a PUSH and a PULL.

REPORT. This command enables the client to report to the HTTPR server which batch of messages, identified by a so called transaction identifier (TID), it has received (and reliably recorded) from that server. As a response to this command, the client gets back a similar report from the server.
This command also allows the two counterparts to notify each other about the TID of the last batch of messages sent to each other.

HTTPR has been designed to ensure the exactly once message delivery semantic in spite of crash failures at both the client and server sides, and network failures (such as partitioning), provided that the failures are eventually recovered. With respect to the interaction with the persistent storage system, three possible outcomes are defined upon trying to record a message at both client and server sides. These outcomes are COMMIT, ROLLBACK and INDOUBT [7]. COMMIT means that the message payload has been received and persistently recorded. ROLLBACK means that the message payload could not be recorded on persistent storage (for example due to space unavailability). An INDOUBT means that the handling of the message payload is uncertain, i.e. the recipient is not sure whether the payload has been actually recorded on persistent storage (for example because the storage system is remote and no acknowledgment came back within a given timeout, possibly due to network failures).

3 Approach to Validation

There were two main hurdles we had to overcome, while formalizing the HTTPR specifications to build the PROMELA model and to verify its correctness. The first, as mentioned in Section 1, was due to the absence of detailed specifications of the messaging agents behavior. The second problem was raised by the state space explosion issue. Specifically, validating the protocol model as a whole unfortunately turned out to be impossible because of the prohibitive RAM requirements ([2]), even in the simplest protocol mode, namely sessionless.

Abstraction and decomposition are the approaches suggested in literature [11, 12] to reduce the amount of memory required for model validation. These methods are complementary, in the sense that they can be used in combination, and this is just what we had to do to successfully carry out the validation.

A first effort to tackle this problem was made towards the direction of abstraction, by means of simplifying as much as possible the state diagram derived from the IBM specifications. Additionally, we have removed from the PROMELA model, any information not strictly necessary for the proof of safety and liveness. In this task we were sensibly helped by the SPIN tool Slicing Algorithm feature, which automatically detects and outlines any variable in the code not involved in the validation phase. However, the approach which most rewarded our efforts, by definitely reducing the RAM required for the model validation (of at least 3 orders of magnitude) was the decomposition of the protocol model into 3 submodels, one for each command associated with message exchange allowed by the protocol (i.e. PUSH, PULL and EXCHANGE). Actually validating the protocol through different submodels is trivially correct in case of sessionless mode if each command is issued over a distinct HTTPR channel. This is because the lifetime of the interaction supported by the protocol coincides with the lifetime of each single command. Instead, when we consider multiple message exchanges flowing over the same channel (as it is also allowed by the protocol sessionless mode), claiming the correctness of the protocol by means of the correctness of submodels for distinct commands is not so trivial. This is because the starting state of each command in the sequence is affected by the execution of the previous commands in the sequence. How to cope with this issue will be discussed in detail in Section 5, where the PROMELA implementation is presented.

[2] In the early phase of development of our PROMELA model, the validator ran out of 64GB of memory on an IBM SP4.

4 FSM Representation

Our first step in the validation of HTTPR was to produce a formal representation of the protocol behavior by means of Finite State Machines (FSMs). For each command associated with message exchange (i.e. PUSH, PULL, EXCHANGE), we introduce FSMs for both the client and the server side protocol behaviors. In our FSMs the transitions from one state to another are labeled as follows:

$$\frac{input,\ condition}{output,\ action} \tag{1}$$

where

(i) *input* is an HTTPR command at the server side, it is the response to an HTTPR command at the client side.
(ii) *condition* is a logic condition that has to be verified to enable the transition. The predicates involved in this condition might be expressed also as a function of the outcome of the interaction with the persistent storage system.
(iii) *output* is the outcome associated with the state transition, which is delivered to the counterpart. It is an HTTPR command at the client side, it is the response to an HTTPR command at the server side.
(iv) *action* is an action performing the update of local state information. Changes of local state information could be actually represented as a state machine to be coupled to the FSM associated with the protocol behavior representation. For space constraints we omit the representation of such a state machine. However, its evolution can be trivially deduced by state transitions proper of FSMs associated with the HTTPR protocol.

According to the guidelines given in the HTTPR specifications ([3]), state information maintained by both the client and the server consists of the following set of variables:

1. `last_source_id`, which stores the identifier, namely the TID, for the last batch of messages sent to the counterpart.
2. `indoubt_ids`, which stores the identifiers, i.e. the TIDs, of the batches of messages not known to have been received by the counterpart.
3. `indoubt_msgs`, which stores the batches of messages not known to have been received by the counterpart.
4. `last_received_id`, which stores the identifier, namely the TID, of the last batch of messages received from the counterpart.
5. `last_received_outcome`, which stores how the last received batch of messages was disposed of here at the recipient.
6. `last_sent_id`, which stores the identifier of the last batch of messages known to have been sent by the counterpart.

[3] Actually, HTTPR specifications do not provide a definition of what really must be done but, rather, an illustrative suggestion of one way to represent the local state information. In terms of such a representation, they indicate when and how the information must be updated [7].

The first three variables are updated when playing the role of sender of a message. Instead, the last three variables are updated when playing the role of receiver of a message. The sender is free to use whatever TID values it chooses, as long as they are strictly increasing [7], but initially the sender should set the `last_source_id` value to 0. `last_received_id` and `last_sent_id` are initialized to 0, while `last_received_outcome` is initialized to COMMIT. Also, by their semantics, the relation `last_sent_id`$\geq$`last_received_id` always holds.

HTTPR requires that client and server involved in the communication save local state information persistently, and update the information as an atomic action. This can be achieved by implementing state transitions as transactions. It is also worthy emphasizing that state transition atomicity excludes the possibility for crash failures to occur in the middle of a state transition. Note that the outcome of an interaction with the persistent storage system (i.e. COMMIT, ROLLBACK or INDOUBT) is non-deterministic. Examples of non-determinism will be discussed while presenting the model for the protocol behavior. Finally, as already discussed in Section 3, the approach we take initially is to validate HTTPR through separate FSMs, one for each command, implicitly assuming that process restart is able to determine which command, if any, was being executed upon crash ([4]). Addressing the validation of sequences of HTTPR commands that exchange messages on a same HTTPR channel is delayed to Section 5.2.

4.1 PUSH Command

The FSMs for the client and server behaviors related to the PUSH command are depicted in Figure 1. The client machine has the following five states:

WORKING. This is the state in which the client issues the PUSH command to the server.

WAIT PUSH REPLY. In this state the client waits for the response from the server to an issued PUSH command.

WAIT REPORT REPLY. In this state the client waits for the response from the server to an issued REPORT. The client issues a REPORT, i.e. moves from the WAIT PUSH REPLY state to this state, either because of a timeout (this captures situations of network failures or server crashes) or because the server response reports an INDOUBT outcome with respect to the recording of the batch of messages sent by the client.

RESTART. This is the state the client passes through in case of crash failure. For simplicity of representation, no direct arrow is plotted from the other states to the RESTART state. However, a crash failure can occur within each state, therefore the RESTART state is actually reachable from any other state in the FSM.

FINAL. When this state is reached, the management of the PUSH command gets completed. The client moves to this state upon the receipt of a COMMIT

[4] This can be implemented by maintaining an additional state variable indicating the HTTPR command currently being executed.

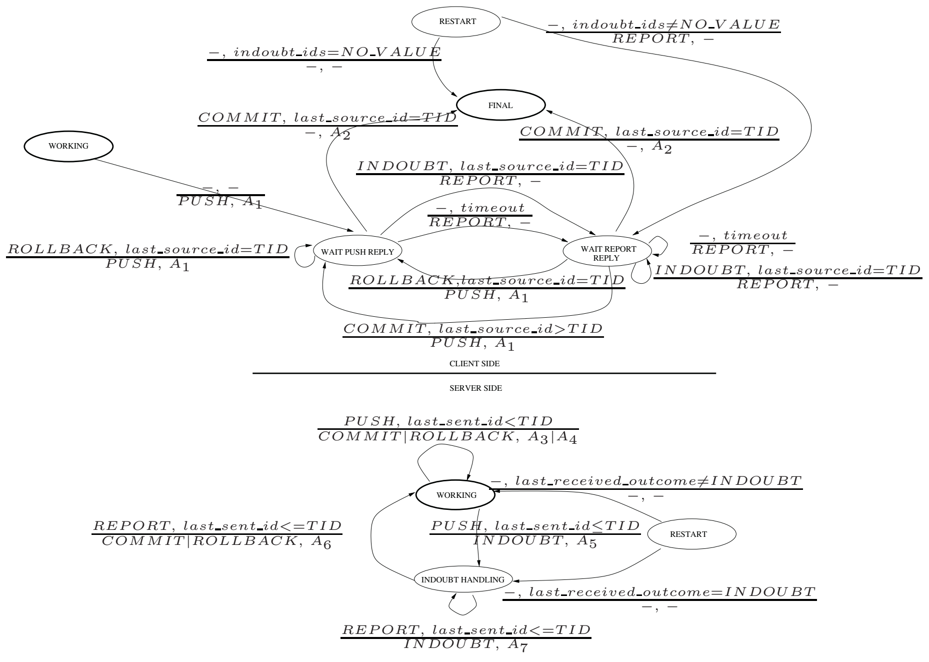

Fig. 1. FSMs for the PUSH command.

response from the server, associated with the last batch of messages sent by the client.

The following set of actions updating state variables characterizes FSM transitions associated with the client:

A_1 TID++; `last_source_id`=TID; `indoubt_ids`=TID; `indoubt_msgs`=msgs.
A_2 `indoubt_ids`=NO_VALUE; `indoubt_msgs`=NO_VALUE.

A_1 is executed while passing from the WORKING state to the WAIT PUSH REPLY state. It updates the TID maintained in the `last_source_id` in order to make the value of this variable to keep track that a new batch of messages is being sent from the client to the server. The `indoubt_ids` variable is updated accordingly. Also, the messages in the batch (i.e. msgs) are stored in the `indoubt_msgs` variable. A_1 is executed also in case of re-transmission of the batch to the server (see transitions from WAIT REPORT REPLY to WAIT PUSH REPLY, and from this latter state to itself). Note that the messages in the batch (i.e. msgs) might change upon the re-transmission of the batch, for example because an additional message is included in the batch. A_2 is executed when moving to the FINAL state. It only resets the variables `indoubt_ids` and `indoubt_msgs` since the PUSH command gets completed upon that transition.

With respect to the conditions in the FSM representations, we have the following set:

1. `indoubt_ids`=NO_VALUE. Upon restarting after a failure, the client is allowed to move to the FINAL state since there is no batch of messages whose receipt at the server side has not yet been acknowledged.
2. `indoubt_ids`≠NO_VALUE. Upon restarting after a failure, the client must move to the WAIT REPORT REPLY state since there is at least one batch of messages whose receipt at the server side has not yet been acknowledged.
3. `last_source_id`=TID. This condition indicates that the response received from the server for PUSH or REPORT, refers to a TID coinciding with the most recent one used by the client while sending a batch of messages.
4. `last_source_id`>TID. This condition allows the client to become aware that a previously issued PUSH command has not been received by the server (since the TID carried by the response shows that the server is not aware of the most recent batch identifier used by the client), and that the batch of messages has to be re-transmitted.

The server machine has the following three states:

WORKING. In this state the server accepts PUSH commands from the clients. The output messages produced by transitions from this state are not deterministic, thus modeling non-determinism in the outcome of the interaction with the persistent storage system which updates local state information.

INDOUBT HANDLING. In this state the server handles an INDOUBT outcome with respect to the interaction with the persistent storage system (the server is not sure whether the batch of messages received by the client has been actually recorded on persistent storage). Solution of the INDOUBT outcome into ROLLBACK or COMMIT is mandatory for the liveness of the protocol.

RESTART. This is the state the server passes through in case of crash failure. Similarly to the client FSM, for simplicity of representation, no direct arrow is plotted from the other states to the RESTART state. However, a crash failure can occur within each state.

The following set of actions updating state variables characterizes FSM transitions associated with the server:

A_3 `last_received_id`=TID; `last_sent_id`=TID;
`last_received_outcome`=COMMIT.
A_4 `last_received_id`=TID; `last_sent_id`=TID;
`last_received_outcome`=ROLLBACK.
A_5 `last_received_id`=TID; `last_sent_id`=TID;
`last_received_outcome`= INDOUBT.
A_6 `last_sent_id`=TID; `last_received_outcome`=COMMIT|ROLLBACK.
A_7 `last_sent_id`=TID; `last_received_outcome`=INDOUBT.

A_3 is executed while passing from the WORKING state to itself. It updates the TID maintained in `last_received_id` in order to make the value of this variable to keep track that the server has received a batch of messages associated with a new TID used by the client. `last_sent_id` is updated accordingly.

The `last_received_outcome` variable is then updated on the basis of the outcome (COMMIT or ROLLBACK) of the interaction with the persistent storage system. A_4 and A_5 are similar to A_3, with the difference that they refer to a ROLLBACK or INDOUBT outcome with respect to the interaction with the persistent storage system. A_6 models the solution of an INDOUBT into a ROLLBACK or COMMIT outcome. This eventual change in the outcome is, as usual, determined in a non-deterministic way. The server updates `last_sent_id` on the basis of the value of TID, and `last_received_outcome` on the basis of the interaction with the persistent storage system. A_7 is similar to A_6, except that it handles the case of a new INDOUBT with respect to the interaction with the persistent storage system.

With respect to the conditions in the FSM, we have the following set:

1. `last_sent_id`<TID. The server becomes aware of a new TID used by the client, i.e. a new batch has been sent by the client.
2. `last_sent_id`≤TID. The server becomes aware of a new REPORT invocation at the client side, which refers either to a new TID used by the client, or to a TID already known by the server.
3. `last_received_outcome`=INDOUBT or `last_received_outcome`≠INDOUBT. The last interaction with the persistent storage system produced, or not, an INDOUBT outcome. Upon restarting after a failure, the server simply verifies this condition to determine whether the WORKING state can be entered or handling of the INDOUBT outcome must be performed.

4.2 PULL Command

The FSMs for the client and server behaviors related to the PULL command are depicted in Figure 2. The client machine has the following five states:

WORKING. This is the state in which the client issues the PULL command to the server.

WAIT PULL REPLY. In this state the client waits for the reply from the server to an issued PULL command. Like in the server's WORKING state for the PUSH FSM, the output messages produced by transitions from this state are not deterministic since they depend on the outcome of the interaction with the persistent storage system.

INDOUBT HANDLING. In this state the client waits for the reply to a REPORT that notifies to the server an INDOUBT outcome with respect to the interaction with the persistent storage system. Upon the receipt of the response from the server, the client moves out this state in case the INDOUBT is resolved into a ROLLBACK or COMMIT outcome. The client issues a new REPORT upon timeout expiration (i.e. no response gets in from the server within the timeout).

RESTART. This is the state the client passes through in case of crash failure. This state is reachable from any other state in the FSM even though, for simplicity, no directed arrow is depicted towards this state.

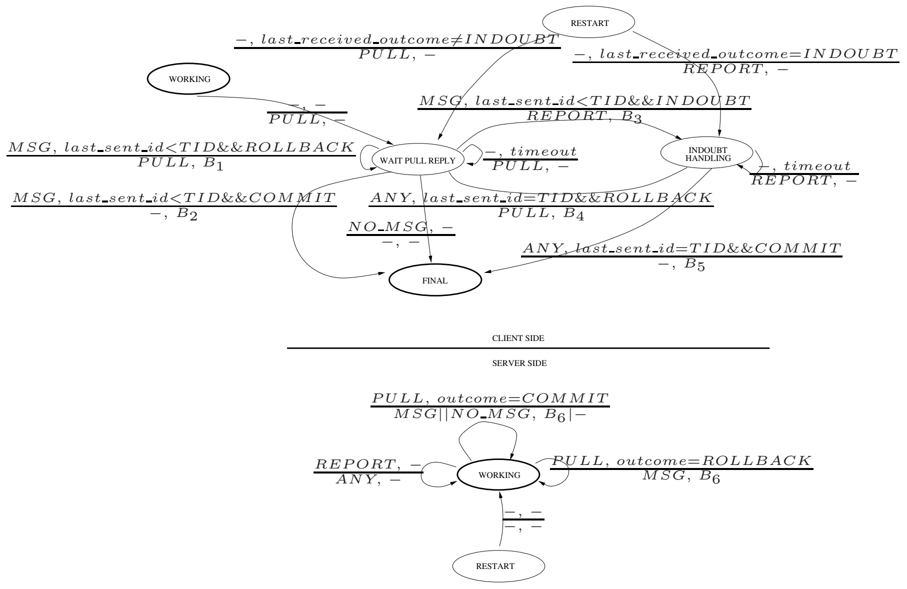
RESTART
−, last_received_outcome≠INDOUBT / PULL, −
−, last_received_outcome=INDOUBT / REPORT, −
WORKING
−, − / PULL, −
MSG, last_sent_id<TID&&INDOUBT / REPORT, B3
MSG, last_sent_id<TID&&ROLLBACK / PULL, B1
WAIT PULL REPLY
−, timeout / PULL, −
INDOUBT HANDLING
−, timeout / REPORT, −
MSG, last_sent_id<TID&&COMMIT / −, B2
ANY, last_sent_id=TID&&ROLLBACK / PULL, B4
NO_MSG, − / −, −
ANY, last_sent_id=TID&&COMMIT / −, B5
FINAL
CLIENT SIDE
SERVER SIDE
PULL, outcome=COMMIT / MSG||NO_MSG, B6|−
REPORT, − / ANY, −
WORKING
PULL, outcome=ROLLBACK / MSG, B6
−, − / −, −
RESTART

Fig. 2. FSMs for the PULL command.

FINAL. When this state is reached, the management of the PULL command gets completed.

The following set of actions updating state variables characterizes FSM transitions associated with the client:

B_1 `last_received_id`=TID; `last_sent_id`=TID; `last_received_outcome`=ROLLBACK.
B_2 `last_received_id`=TID; `last_sent_id`=TID; `last_received_outcome`=COMMIT.
B_3 `last_received_id`=TID; `last_sent_id`=TID; `last_received_outcome`= INDOUBT.
B_4 `last_sent_id`=TID; `last_received_outcome`=ROLLBACK.
B_5 `last_sent_id`=TID; `last_received_outcome`=COMMIT.

B_1 is executed when MSG (i.e. a batch of messages) arrives as the response of the PULL command and the interaction with the persistent storage system at the client side produces a ROLLBACK (i.e. the client is unable to persistently record the message). B_2 is the same as B_1, except for that it refers to an interaction with the persistent storage system that produces a COMMIT outcome. Analogously, B_3 refers to an INDOUBT outcome with respect to that interaction. B_4 characterizes the client behavior upon the receipt of a response to a REPORT; this report is issued by the client in case of an INDOUBT outcome with respect to the interaction with the persistent storage system. Upon the response receipt, the client resolves the pending INDOUBT outcome into

ROLLBACK. B_5 is similar, but refers to a COMMIT final outcome with respect to the interaction with the persistent storage system. Note that these actions are similar to the actions from A_3 to A_7 characterizing the server behavior in the FSM related to the PUSH command. This is quite natural when thinking that this time the client is acting as the recipient of the messages, just like the server does in the PUSH command.

With respect to the conditions in the FSM, we have the following set:

1. `last_received_outcome`=INDOUBT. Upon restarting after a failure, the client issues a REPORT to the server and then moves to the INDOUBT HANDLING state. As already said while explaining the client behavior within such a state, upon the receipt of the REPORT response from the server, the client will eventually resolve the recording of the batch of messages as a ROLLBACK or a COMMIT.
2. `last_received_outcome`≠INDOUBT. Upon restarting after a failure, the client issues again the PULL command. This is required to tackle cases in which the client crashed after issuing the PULL command, but before any response (possibly carrying messages) came back from the server. In this case, no INDOUBT outcome can be stored by `last_received_outcome` since no attempt to record messages on the persistent storage system was performed by the client.
3. `last_sent_id`<TID. This condition indicates that the last batch of messages sent by the server refers to a TID not known by the client. In other words, it refers to a new batch of messages.
4. `last_sent_id`=TID. This condition indicates that the server response to a REPORT refers to the identifier of the batch of messages currently INDOUBT at the client side.

These conditions are sometimes combined with predicates involving the outcome of the interactions with the persistent storage system at the client side.

The server machine has the following two states:

WORKING. This is the state in which the server accepts PULL commands from the clients.
RESTART. This is the state the server passes through in case of crash failure.

The following action updating state variables characterizes FSM transitions associated with the server:

B_6 TID++; `last_source_id`=TID; `indoubt_ids`= TID; `indoubt_msgs`=msgs.

B_6 is identical to A_1, which characterizes the behavior of the client in the PUSH command. Here the server is acting as the sender of the batch of messages. Note that each time the server re-sends a batch of messages previously sent, but not correctly received by the client, the server updates the TID to notify the client that we are in the presence of a re-transmission.

Finally, the conditions *outcome*=COMMIT and *outcome*=ROLLBACK simply refer to the outcome notified by the client to the server with respect to the last interaction with the persistent storage system. These outcomes are piggybacked on the commands issued by the client.

4.3 EXCHANGE Command

EXCHANGE is the combination of PUSH and PULL. Therefore, the FSM representation consists of an adequate composition of the FSMs related to PUSH and PULL. For space constraint we cannot report the complete description of such a composition. The interested reader can refer to [10] for the details.

5 PROMELA Implementation

We have used three PROMELA processes. The client and server processes are directly derived from the FSMs representation previously described. To model network failures, we have introduced an apposite *network message stealing process*, which causes only message losses, but it is not responsible for any form of reordering in the delivery of messages. This choice is justified by the following observation. As HTTPR is layered on top of HTTP 1.1, in absence of network or host failures, HTTPR commands and responses flow on a single TCP connection which ensures reliable FIFO delivery. On the other hand, in case of (i) timeout expiration at the client while waiting for messages from the network, i.e. expected replies do not come in within the timeout due to network failures or crash failure of the server, or in case of (ii) client crash and restart, a new TCP connection is established and both the counterparts discard any data getting in from earlier connections ([5]). As a consequence, no out of order data delivery can ever occur at both the counterparts.

We have modeled re-opening of TCP connections in PROMELA by using an "incarnation number" associated with the current TCP connection, which is maintained into a global variable. Each time a network failure or a client/server crash occurs, the incarnation number is increased ([6]). The client is notified of network failure or server crash by means of a PROMELA channel used for this specific task. The receipt of a message from this channel triggers the timeout event on the client side FSM. This ultimately results in a state transition at the client side, which possibly updates part of its HTTPR protocol state. The messages travelling on this channel carry the incarnation number associated with the closed TCP connection. On the other hand, in case of client crash, there is no need for notification since the client itself will increase the incarnation number while passing in the RESTART state. In case of client/server crash failure, all the messages already in the PROMELA input channel are discarded as a common way to model crash failure of the end point of a TCP connection.

[5] Recall that timeout events are relevant at the client side only, therefore a non-crashed client is responsible for closing the connection upon timeout, and opening a new connection towards the server.

[6] The growth of the incarnation number is bounded and depends on the maximum number of injected network failures.

5.1 Correctness Claims and Results

The correctness claims must be formalized in such a way to capture reliability of message transfer, with exactly once delivery semantic, which the protocol aims at providing. Thereby, the correctness claims to be proven are the following: (A) a batch of messages sent is eventually received and persistently stored by the recipient (liveness), (B) without any duplication (safety). Proving claims in points (A) and (B) simultaneously, corresponds to ensuring both the at-least-once and the at-most-once message delivery semantics. This actually guarantees the exactly once delivery semantic. We have coded the correctness claims through a PROMELA *never claim* that translates the following LTL [1] formula:
`[] ((p -> <> q)&&r)`.

The `[](p -> <> q)` part refers to the claim in point (A), and can be expressed as: "It is always true that, if `p` becomes true, then `q` eventually becomes true". For the PUSH command, this translates into: "It is always true that, if the client issues a PUSH, then the server eventually receives the batch of messages and reliably stores it". For the PULL command we get: "It is always true that, if the client issues a PULL, it will eventually receive a response with no message, or a batch of messages. In the latter case, the messages are persistently recorded at the client side". For the EXCHANGE we obtain the combination of the claims used for PUSH and PULL.

The `[] r` part refers to the claim in point (B). `r` is the following predicate "number of COMMIT outcomes from persistent storage ≤ 1". This means that any batch of messages is recorded at the recipient at most once.

We note that in the FSMs representation, there are cycles that might theoretically prevent the client to reach the FINAL state, and the server to re-enter the WORKING state. State transitions occurring within a cycle are caused by the occurrences of failures (as an example, the client might cyclically enter the WAIT PULL REPLY state in the FSM of the PULL command due to subsequent timeout expirations, which model a down-time period of the network or of the server). If failures are eventually recovered, then the state transition sequence is allowed to eventually get out of the cycle. This, in its turn, allows the liveness of the protocol. In our validation we do not consider pathological situations with unbounded down-time periods. Therefore, we assume that a maximum amount of failures can occur during the processing of whichever command.

In principle, allowing an unbounded amount of failures would be a better validation approach since it would require weaker assumptions on the model. In practice, this would mean adopting fairness assumptions with respect to the behavior of the network message stealing process of our PROMELA implementation. A problem with this approach is that the counters used to store TIDs would grow without bound, but HTTPR specifications say nothing about how to tackle TID counters overflow. In other words, we cannot provide a formal description of the protocol behavior in respect to this problem since HTTPR specifications do not define (even informally) how to face this problem. (In practice the overflow should not be a problem since counters are implemented with relatively large amount of bits, say 64 bits. However this does not help while performing model checking on the protocol behavior.) Additionally, a practical problem arises in case we let the number of failures grow in unbounded way. Specifically, we get an explosion of the state space to be explored by the validator, which precludes any possibility to complete the validation with commonly available computational (memory) resources. The verification results show that the above LTL formula is verified for the PROMELA coded FSMs related to all the three commands. The below table reports the number of states visited by the validator, the reached depth and the PROMELA state vector size. We note that no unreachable state has been detected, which shows that the presented FSMs are not over-specified. With the verification results, we also report the maximum number of failures we have considered for each single failure type (recall that ROLLBACK or INDOUBT outcomes model failures with respect to the interaction with the persistent storage system) and for each specific command. The corresponding RAM occupancy is listed. The validation has been performed on a Pentium III 866 MHz - 256 MB RAM, running LINUX, and hash compact 4 [13, 15] has been used as the compile time directive for SPIN.

HTTPR Command	Failures Network - ROLLB. - IND. - crash	Visited States	Reached Depth	State Vector Size	RAM Occupancy (MB)
PUSH	6 - 4 - 6 - 1	1.09e+07	447	212	221
PULL	6 - 4 - 6 - 1	8.99e+06	444	212	181
EXCHANGE	5 - 2 - 4 - 1	9.91e+06	356	244	200
EXCHANGE	6 - 4 - 6 - 1	-	-	244	RAM exceeded
PUSH	3 - 4 - 4 - 2	7.40e+06	354	212	149
PULL	3 - 4 - 4 - 2	8.85e+06	384	212	178
EXCHANGE	3 - 4 - 4 - 2	8.65e+06	499	244	174
EXCHANGE	4 - 4 - 4 - 2	-	-	244	RAM exceeded

5.2 Coping with Sequences of Commands

Proving the correctness of the protocol in the case of a sequence of commands on the same channel requires some additional steps. In fact, despite the fact that a new message exchange (i.e. a new command execution) can not be requested by the client unless the previous one has correctly terminated, one should note that the history of the previous commands, in terms of values of the HTTPR protocol state variables, do affect future commands. Actually, state variables affected by previous commands execution are those maintaining counters (e.g. `last_source_id`). Strictly speaking HTTPR commands on the same channel can be no longer considered as independent of each other. However, direct validation of a sequence of commands (even of relatively small length) would cause the explosion of the state space to be explored by the validator, which might make the problem intractable. In this section we justify why the correctness of a sequence of commands can be deduced by the results of the validation of a single command execution. The basic observation is that all the conditions enabling state transitions on the FSMs, and determining the protocol behavior, are independent of initialization values for counter state variables, assuming that the following relation between sender and receiver counter state variables holds:
$\mathcal{SYNCH}$: $\texttt{last\_source\_id}_{sender} = \texttt{last\_received\_id}_{receiver} = \texttt{last\_sent\_id}_{receiver}$

This is because each condition in the FSMs can be easily translated into an equivalent condition which still holds in case we add the same offset (i.e. the initialization value) to each counter state variable. In other words, if $\mathcal{SYNCH}$ holds, the protocol behavior can be considered as independent of the initialization values for counter state variables. Consequently, the proof of liveness and safety properties is not affected by the choice of the initialization values. Note that requiring the relation $\mathcal{SYNCH}$ to hold on final values of counter state variables is actually an additional correctness requirement for HTTPR. In practice, if $\mathcal{SYNCH}$ is verified, then client and server have updated local states in a consistent way, allowing the protocol to evolve correctly. This property is obviously relevant just when dealing with sequences of commands. It was easy to extend the HTTPR PROMELA model used in the previous section to prove that if relation $\mathcal{SYNCH}$ holds at the start (and it trivially holds for the initialization values suggested by the protocol specification) of a single command execution, then it always holds at the end of that command execution. This result, in combination with our previous observation about the independence of the protocol behavior from the initialization values of the counter state variables, allowed us to prove the following Theorem.

Theorem. Let $S = \{c_1, c_2, \ldots, c_n\}$ be a sequence of HTTPR session-less mode commands. If $\mathcal{SYNCH}$ is verified at the start of the first command c_1 of S, then safety and liveness properties hold for each command in S. In addition the relation $\mathcal{SYNCH}$ holds at the end of the execution of each command in S.

Proof (by induction)
Base Step. $i = 1$. It is easy to verify that the relation $\mathcal{SYNCH}$ holds for the HTTPR initialization values of counter state variables. SPIN proved not only that our model satisfies safety and liveness properties for c_1, but also that the relation $\mathcal{SYNCH}$ always holds at the end of a single command execution, i.e. at the end of the execution of c_1.
Induction Step. We suppose that for each command c_i in the sequence, with $i > 1$, safety and liveness are verified, and that $\mathcal{SYNCH}$ holds for values of counter state variables at the end of the execution of c_i. We want to verify the same for c_{i+1}. We know that at the start of c_{i+1}, the relation $\mathcal{SYNCH}$ holds since the values of counter state variables at the start of c_{i+1} coincide with those obtained at the end of c_i. As observed above, independence of the model from counter state variable initialization values at the start of a command execution ensures that model properties (liveness, safety and $\mathcal{SYNCH}$ on final counter state variable values) are verified for whichever initialization values under the condition that $\mathcal{SYNCH}$ holds at the start of the command execution. As a consequence, since these properties are verified for c_1, then we can state safety and liveness of c_{i+1}, and also that $\mathcal{SYNCH}$ still holds at the end of c_{i+1}.

By this result, we can claim that safety and liveness also hold for the case of sequences of messages exchanges on the same HTTPR channel.

6 Conclusions

We have presented FSMs as a formal representation of the set of rules describing the behavior of the HTTPR protocol in sessionless mode. The correctness claims, here defined with respect to the HTTPR specifications, were proved considering both network failures and process crash failures. These claims prove the ability for the sender to deliver a message once, and only once, to its intended receiver, even under the presence of such failures. While performing protocol verification we made use of PROMELA and the SPIN model checker. In order not to cause excessive growth of the state space size, we have adopted a decompositional approach that determines submodels on which model checking is performed. How to use the results achieved in the submodels for the protocol validation has been also discussed. We are planning to use these results as the basis for the verification of more complex protocol modes like sessions or pipelines. With respect to the session mode, we argue that it could be possible to adopt the decompositional approach used here, once submodels for the establishment of the session are defined.

Acknowledgments

The authors thank Daniel Dias and Richard P. King (IBM Thomas J. Watson Research Center) for clarifications on the HTTPR specifications. A special thanks goes to Enrico Tronci for helpful suggestions on earlier draft versions of the paper.

References

1. E.A. Emerson, “Temporal and Modal Logic”, Handbook of Theoretical Computer Science, 1990.
2. “Hypertext Transfer Protocol – HTTP/1.1 (RFC)”, http://www.ietf.org/rfc/rfc2616.txt?number=2616
3. G.J. Holzmann, “Design and Validation of Computer Protocols”, PRENTICE HALL, 1991.
4. G.J. Holzmann, “The Model Checker SPIN”, *IEEE Trans. on Software Engineering*, May 1997.
5. IBM, “A Primer for HTTPR”, http://www-106.ibm.com/developerworks/webservices/library/ws-phtt
6. IBM, “WebSphere MQ Family”, http://www-3.ibm.com/software/ts/mqseries
7. IBM, “HTTPR Specification v1.1”, http://www-106.ibm.com/developerworks/webservices/library/ws-httprspec
8. Microsoft, “Microsoft Message Queuing (MSMQ)”, http://www.microsoft.com/msmq
9. Oracle, “Oracle Message Broker 1.0”, Data Sheet, October 1999.
10. P. Romano, M. Romero, B. Ciciani and F. Quaglia, “Validation of the Sessionless Mode of the HTTPR Protocol”, *Tech. Rep. 12-03*, DIS, University of Rome “La Sapienza”, http://ftp.dis.uniroma1.it/pub/quaglia/tr12-03.ps

11. T. Ruys and R. Langerak, “Validation of Bosch’ Mobile Communication Network Architecture with Spin”, *Proceedings of the 9th SPIN Workshop*, Grenoble, 2002.
12. T. Ruys, “SPIN Tutorial: how to become a SPIN Doctor”, *Proceedings of the 3th SPIN Workshop*, Enschede, 1997.
13. “Spin Online References, Directives to Reduce Memory Use”, http://spinroot.com/spin/Man/Pan.html#D4
14. Sun, “Java Messaging System”, http://java.sun.com/products/jms/
15. P. Wolper and D. Leroy, “ Reliable hashing without collision detection”, *Proc. 5th Int. Conference on Computer Aided Verification*, Elounda, 1993.