The Specification in Z of the REX Protocol

J. Derrick & R. Sultana
The Computing Laboratory,
University of Kent,
Canterbury,
CT2 7NF

15 November 1993

Abstract

REX is a protocol supporting a client/server style of interaction between a number of entities in a distributed system. Within this interaction paradigm, **client entities** may request services supplied by **server entities**, by interacting with intermediate **protocol entities**. This paper presents a Z specification of part of the REX protocol.

Contents

1 Introduction

REX is a protocol supporting a client/server style of interaction between a number of entities in a distributed system. Within this interaction paradigm, **client entities** may request services supplied by **server entities**, by interacting with intermediate **protocol entities**.

A significant part of the protocol deals with interactions that take place between a pair of client and server entities. Such a sequence of interactions shall be referred to as a **session**.

In this paper we will make use of the Z notation to describe how a session is supported by two REX protocol entities. Familiarity with the Z notation [Spiv 88, Pott 91] is assumed.

2 The REX Protocol

2.1 ANSA and the ODP Reference Model

The origins of REX lie within the ANSA project. The **Advanced Networked Systems Architecture (ANSA)** was an Alvey project jointly sponsored by eight major IT companies. Staff from these industrial partners were amongst the members of the team that worked together on this project.

ANSA is a major contributor to the work being carried out by ISO to set-up a Reference Model of **Open Distributed Processing (ODP)**. The objectives of this model is to provide a framework for the standardization of distributed systems. Publicly agreed standards are needed to support the design, implementation, operation and evolution of distributed systems where the various components that make up the systems come from different vendors.

Rather than attempt to directly model the full complexity of a distributed system, ANSA and ODP adopt the idea of taking viewpoints, each of which represents a different abstraction of the original system with emphasis on a specific design concern. Each viewpoint has the following properties:

- comparatively simpler to model;
- self-contained and complete;
- bears constraints that are imposed by the fact that all projections relate to some hypothetical model of the complete system.

Five viewpoints are identified. The **Enterprise** viewpoints captures the role that the distributed computer system has within the organization - the objectives; activities; policy constraints, all at the organization level. The **Information** viewpoint establishes an overall view of all the items of information and of the information processing activities in the system. The **Computational** viewpoint is a view of the distributed system as a set of linked program modules and hence gives a breakdown of the functionality of the system. The **Engineering** viewpoint is a description of the actual mechanisms used to support the required functionality. Finally, the **Technology** viewpoint is used to describe the actual hardware and software components of the distributed system.

2.2 The ANSA Testbench

The ANSA **Testbench** is a reference implementation of the engineering model, over a variety of operating systems each of which constitutes a technology model. It has served as a vehicle for technology transfer back to the sponsors and for demonstrating the use of ANSA principles.

The Testbench provides a suite of mechanisms and tools to simplify the task of writing distributed applications. The computational model supported has the constraint that all interactions between components are confined to a client/server interaction paradigm; a client requests a specific function to be performed by a server which performs it at some later time, possibly returning a result.

The form of client/server style of interaction implemented in the Testbench is based on remote procedure calls - remote access to functions is provided via a local procedural interface. Two forms of remote invocation are supported:

Interrogation:- The client is blocked until the server performs the operation and returns any results.

Announcement:- The client does not wait for the server to perform the requested action.

Communication between nodes in the engineering model is implemented in terms of three protocol layers:

- The bottom layer provides a **message-passing service**. This manages connection and disconnection, and the transmission and receipt of messages between individual nodes.
- The top layer is a **session service**, providing end-to-end synchronization of the dialogue.
- The middle layer is the **execution protocols** layer. This maps computational model interactions onto message exchanges, by making use of the message-passing service.

2.3 REX

REX (**R**emote **EX**ecution) is one of the protocols currently included in the Testbench. It is a protocol for single-endpoint to single-endpoint communication. GEX or Group EXecution which is a protocol for multi-endpoint to multi-endpoint communication, has been implemented in the ANSA Testbench version 4.2.

The REX protocol is a set of rules which governs how interactions in the engineering model realize the end-to-end interaction apparent within the computational model. An interaction, as seen in the computational viewpoint, involves merely two entities - the client and the server entities. In the engineering viewpoint however, two other intermediate entities come into the picture - these are the REX-client and the REX-server entities. The client entity interacts only with the REX-client entity and the server entity only with the REX-server entity. The REX-client and the REX-server entities communicate with each other on behalf of the client and the server entities, making use of a lower-layer message-passing service.

One can classify the set of rules constituting the REX protocol by separating rules for mapping interrogation style of computational model interaction - **CALL**s, from mapping rules for announcement style of computational model interaction - **CAST**s.

A detailed description of the rules that define REX is beyond the scope of this paper. The categorization made above will merely be used to outline the sequence of operations that take place under ideal conditions:

1. a client entity submits a CALL to a REX-client entity and is then blocked waiting for a reply;
2. the REX-client entity sends the CALL to the REX-server entity;
3. when the REX-server entity receives the CALL, this is passed on to the server entity;
4. the server entity performs the required operation and eventually returns a reply;
5. the REX-server entity sends the reply back to the REX-client entity;

6. when the REX-client entity receives the reply, it passes it on the client entity which can now continue to operate.

CASTs are dealt with similarly. However, in this case the client entity does not elicit a response from the server entity.

In practice the REX protocol has also to deal with cases such as client requests and server replies getting lost or delayed. Another layer of complexity is due to the fact that requests and replies may be too large to be sent as single messages over the network. For this reason the REX protocol also defines a **fragmentation** strategy by which large requests and replies may be conveyed as a number of smaller component parts.

3 Preliminaries

The set *DataType* whose elements represent all possible data that may need to be communicated between the client entity and the server entity.

$$[\, DataType \,]$$

Any data of type *DataType* will have a size, for which we define a type:

$$DataSize ::= \mathbb{N}$$

The variable *max_data_size* will denote, for the particular network on which the distributed system is installed, the maximum size of data that can be transmitted as a single unit. If this limit is exceeded, the data would have to be sent in fragments.

$$\mid \; max\_data\_size : \mathbb{N}_1$$

4 Entities

Two REX protocol entities are involved in maintaining a session of communication between the client entity and the server entity. The client entity interacts with the **REX-client entity**; the server entity interacts with the **REX-server entity**.

$$Entity ::= REX\_CLIENT \mid REX\_SERVER \mid CLIENT \mid SERVER$$

$$ProtocolEntity == \{REX\_CLIENT, REX\_SERVER\}$$

$$NonProtocolEntity == \{CLIENT, SERVER\}$$

We shall assume that the protocol entities' task is confined to supporting one session.

5 Requests and Responses

The client entity makes **requests** to the REX-client entity. When the REX-server entity receives a request from the REX-client entity, it will pass it on to the server entity which will process the request. A request may be a *CALL* or a *CAST*. Occasionally, we will need to indicate an absence of a request and so we also introduce a null request.

$$RequestType ::= CALL \mid CAST \mid NO\_REQUEST$$

The client entity submits a *CALL* request when the invocation required is an **interrogation**, i.e. the client entity is to be blocked until the server entity performs the operation asked for and its **response** is received. Note that throughout this paper the words response and reply shall be used interchangeably.

The client entity submits a *CAST* request when the invocation required is an **announcement**, i.e. the client entity does not wait for the server entity to perform the requested action.

The server entity is therefore only expected to send a response in the case of an interrogation. We shall also need to represent an absence of a response.

$$ResponseType ::= RESPONSE \mid NO\_RESPONSE$$

We can now define request and response schemas as follows:

Request
$type : RequestType$
$dest : Entity$
$data : DataType$
$data\_size : DataSize$

Response
$type : ResponseType$
$dest : Entity$
$data : DataType$
$data\_size : DataSize$

The components are:

- a *type* identifying the request or response type;
- a *dest* identifying the destination entity;
- *data* which is the actual data-content of the request or response;
- *data_size* which is the size of this data-content.

6 Messages

To communicate with each other, the REX-client and the REX-server entities utilize a **message-passing service**. Various types of messages are defined:

$$\begin{array}{rcl} MessageType & ::= & CALL \\ & \mid & CALL\_FRAG \\ & \mid & CALL\_ACK \\ & \mid & PROBE \\ & \mid & REPLY \\ & \mid & REPLY\_FRAG \\ & \mid & REPLY\_ACK \\ & \mid & CAST \\ & \mid & CAST\_FRAG \\ & \mid & FRAG\_NACK \\ & \mid & NO\_MESSAGE \end{array}$$

CALL, *CAST*:- messages sent by the REX-client entity and convey client entity requests of the respective type.

CALL_FRAG, *CAST_FRAG*, *REPLY_FRAG*:- similar messages but these only convey a fragment of a request or response.

REPLY **:-** messages sent by the REX-server entity and convey server entity responses.

*PROBE***:-** probe messages sent by the REX-client entity to check that the REX-server entity is still active.

*CALL_ACK***:-** messages used to explicitly acknowledge the receipt of a *CALL* or a *PROBE* message.

*REPLY_ACK***:-** explicit acknowledgment messages for *REPLY* messages.

*FRAG_NACK***:-** explicit negative acknowledgment messages for a fragmented transmission of a request or response.

*NO_MESSAGE***:-** we will use this type of message to indicate an absence of a message.

Messages which convey some data-content of a request or a response, will be distinguished from those which contain no data. The former will be called **data messages** and the latter, **control messages**. We will conveniently assume the *NO_MESSAGE* type to fall under both of these categories.

We are not concerned with how the message-passing service operates. However, we need to be aware of the limitations of this service. In particular, messages can get delayed, or lost, and there is no guarantee that they will be received in the order sent. Hence all messages bear a **sequence number**.

We define the set *SeqNo* from which all message sequence numbers are drawn.

$$SeqNo ::= \mathbb{N}$$

Sequence numbers are assigned to messages by the protocol entities from which they originate. The protocol entities keep a record of the highest data message sequence number that has been sent or received. When a protocol entity is to send a new request or response message, the increment of this number is used as the sequence number. When a data message is received, it is identified as referring to a new request, or response, if the sequence number of the message is greater than that held by the protocol entity.

Control messages convey control information about data messages. The sequence number of a control message is set to that of the data message which it refers to.

Messages conveying a fragment of a particular request or response will have the same sequence number. However, each fragment contains an *offset* identifying its relative position within the complete request or response. The total number of fragments that compose the request or reply is also stored in each fragment message. The *offset* and the *total_size* of the request or response will be drawn from the set *FragIndex*.

$$FragIndex ::= \mathbb{N}_1$$

We now give the schemas for data and control messages.

MessageHeader

$$\begin{array}{l} type : MessageType \\ dest : ProtocolEntity \\ seqno : SeqNo \end{array}$$

$$
\begin{array}{|l}
\hline
\textit{DataMessage} \\
MessageHeader \\
data : DataType \\
offset : FragIndex \\
total\_size : FragIndex \\
\hline
type \in \\
\quad \{ \quad CALL, \ CALL\_FRAG, \ CAST, \ CAST\_FRAG, \\
\qquad\quad REPLY, \ REPLY\_FRAG, \ NO\_MESSAGE \\
\quad \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ControlMessage} \\
MessageHeader \\
frags\_map : \mathbb{P}\ FragIndex \\
\hline
type \in \\
\quad \{ \quad CALL\_ACK, \ REPLY\_ACK, \ PROBE, \\
\qquad\quad FRAG\_NACK, \ NO\_MESSAGE \\
\quad \} \\
\hline
\end{array}
$$

We will only be considering message transfers between the two protocol entities. Therefore, we need only specify the destination of the message to be either the REX-client or the REX-server.

The *offset* and *total_size* components of data messages are only relevant in the case of messages which convey fragments of some complete request or response.

The *frags_map* component of control messages is only relevant for *FRAG_NACK* message types. The protocol entity sending a *FRAG_NACK* message uses this component to provide information as to which fragments have been successfully received.

7 Timers

The protocol entities each make use of a **timer-service**. A timer can be *START*ed or *STOP*ped. We shall assume that if two consecutive *START*s are performed on a timer, the effect of the second *START* shall be that of a *STOP* followed by a *START*. We shall use the *NO_UPDATE* operation to represent the absence of an update of the timer, by a protocol entity.

$$TimerUpdateType ::= START \mid STOP \mid NO\_UPDATE$$

The timers used by the REX-client and the REX-server entities are independent of each other, i.e. an update of one timer will not affect the other timer.

When a timer is *START*ed a timeout period is set.

$$TimeoutPeriod ::= REPLY \mid PROBE \mid FLOW \mid CHECK$$

REPLY:- timeout period is used to establish a limit on the time a protocol entity waits for an expected acknowledgment. If this time expires, the protocol entity would suspect that something has gone wrong, and would take some appropriate action.

PROBE:- timeout period is the interval at which the REX-client entity probes the REX-server entity. The concept of probing is explained in a later section.

When a request or reply is too large to be transmitted as a single unit, it is fragmented and the fragments are transmitted separately. Transmission of these fragments is rate-controlled, i.e. transmission is throttled so as to maintain a comfortable average arrival rate.

FLOW:- timeout period is the time a protocol entity waits between transmission of fragments.

CHECK:- timeout period is the the time a protocol entity waits for a next fragment, before suspecting that something has gone wrong.

We now give schemas for the update of timers by the protocol entities, and for the occurrence of timeouts.

$$\begin{array}{|l}
\hline
\textit{TimerUpdate} \\
\textit{type} : \textit{TimerUpdateType} \\
\textit{source} : \textit{ProtocolEntity} \\
\textit{period} : \textit{TimeoutPeriod} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Timeout} \\
\textit{dest} : \textit{ProtocolEntity} \\
\hline
\end{array}$$

We are only concerned with a single session of interaction between two protocol entities. Therefore, identification of initiators of timer updates and of destination of timeouts, is only a matter of stating whether the protocol entity is the REX-client or the REX-server.

8 Protocol entity states

The REX-client entity may only exist in one of the following states:

$$\begin{array}{lcl}
\textit{RexClientState} & ::= & \textit{IDLE} \\
 & | & \textit{CALLING} \\
 & | & \textit{PROBING} \\
 & | & \textit{CALL_SENDING} \\
 & | & \textit{CAST_SENDING} \\
 & | & \textit{RECEIVING}
\end{array}$$

IDLE:- No outstanding requests to be sent and no outstanding responses or acknowledgments to be received. New client entity *CALL* and *CAST* requests may only be accepted while the REX-client entity is in this state.

CALLING:- A non-fragmented *CALL* request has been sent and no response has been received for it. Furthermore, it is not yet known whether the REX-server entity has received the *CALL* request.

PROBING:- A fragmented or non-fragmented *CALL* request has been sent and its successful and complete receipt by the REX-server entity has been acknowledged. However, the response is still outstanding. Whilst in this state, the REX-client entity will keep on probing the REX-server entity by sending *PROBE* messages. This assures the REX-client entity that the REX-server entity is still active and that no serious communication failure has occurred. Provided these probes continue to be acknowledged, the REX-client will remain in this state indefinitely.

CALL_SENDING:- The REX-client entity is in this state if, either a fragmented *CALL* request is being transmitted, or, a fragmented *CALL* request has been transmitted and no response or acknowledgment has been received for it. The REX-client entity remains in this state until the response or an acknowledgment is received.

CAST_SENDING:- A fragmented *CAST* request is being transmitted.

RECEIVING:- A fragmented response, to an outstanding *CALL* request, is being received.

Similarly, the REX-server entity may be in one of the following states:

$$
\begin{array}{lll}
RexServerState & ::= & IDLE \\
& | & ASKED \\
& | & REPLYING \\
& | & CALL\_RECEIVING \\
& | & CAST\_RECEIVING \\
& | & SENDING
\end{array}
$$

IDLE:- No outstanding responses to be sent and no outstanding acknowledgments or request fragments to be received.

ASKED:- A non-fragmented *CALL* request or a complete fragmented *CALL* request has been received, but, the response is still outstanding.

REPLYING:- There is no outstanding response to be sent, but, the last response that was sent has not been acknowledged.

CALL_RECEIVING:- A new fragmented *CALL* request is being received; there are still more fragments to be received.

CAST_RECEIVING:- A new fragmented *CAST* request is being received; there are still more fragments to be received.

SENDING:- The REX-server entity is in this state if either, a fragmented response is being transmitted, or, a fragmented response has been transmitted and no acknowledgment has been received for it. The REX-server entity remains in this state until an acknowledgment is received. The acknowledgment can be an explicit *REPLY_ACK* message; it can also be implicit on the arrival of a new request.

9 Maintaining session information

In this section we describe the information that needs to be maintained by each of the protocol entities. But first another definition is needed:

$$MapFlag ::= OK \mid NOT\_OK$$

The information maintained by the REX-client entity:

$$
\begin{array}{|l}
\hline
\textit{RexClientInfo} \\
client\_state : RexClientState \\
client\_seqno : SeqNo \\
client\_buffer : \mathbb{P}\ DataMessage \\
request\_frags\_map : \mathrm{seq}[MapFlag] \\
\hline
\end{array}
$$

client_state:- The state of the protocol entity, as one of the distinct states mentioned above, in which it can be.

client_seqno:- A record of the largest data message sequence number that has been sent or received.

client_buffer:- If the protocol entity is in one of the states *CALLING*, *PROBING*, *CALL_SENDING*, or *CAST_SENDING*, then a copy of the last request sent, or which is being sent, is maintained. If the protocol entity is in the state *RECEIVING*, a copy of the last response being received is maintained. Note that if the request or response concerned is fragmented then the buffer will consist of a set of message fragments, whereas if the request is non-fragmented then the the buffer will consist of a singleton set containing the corresponding message for the complete request.

request_frags_map:- If the protocol entity is in state *CALL_SENDING* or in state *CAST_SENDING*, it needs to know which of the message fragments still need to be transmitted. For the abstract representation of this information, we have chosen to use a sequence. The sequence will consist of a term for every fragment message of the request being sent. The domain of this sequence corresponds to the fragments' offsets. If a fragment still needs to be transmitted, the term corresponding to this fragment will be *NOT_OK*; otherwise the term will be *OK*.

The information maintained by the REX-server entity:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RexServerInfo}} \\
\hline
\textit{server_state} : \textit{RexServerState} \\
\textit{server_seqno} : \textit{SeqNo} \\
\textit{server_buffer} : \mathbb{P}\,\textit{DataMessage} \\
\textit{response_frags_map} : \mathrm{seq}[\textit{MapFlag}] \\
\hline
\end{array}$$

server_state:- The state of the protocol entity, as one of the distinct states mentioned above, in which it can be.

server_seqno:- A record of the largest data message sequence number that has been sent or received.

server_buffer:- If the protocol entity is in state *REPLYING* or in state *SENDING*, then a copy of the last response sent, or which is being sent, is maintained. If the protocol entity is in state *CALL_RECEIVING* or in state *CAST_RECEIVING*, then a copy of the last request being received is maintained. Note that if the request or response concerned is fragmented then the buffer will consist of a set of message fragments, whereas if the response is non-fragmented then the the buffer will consist of a singleton set containing the corresponding message for the complete response.

response_frags_map:- If the protocol entity is in state *SENDING*, it needs to know which of the message fragments still need to be transmitted. A sequence is used to represent this information, as in the case of the REX-client entity.

We also consider the information that needs to be maintained on the session as a whole:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SessionInfo}} \\
\hline
\textit{RexClientInfo} \\
\textit{RexServerInfo} \\
\hline
\end{array}$$

10 The initial state of the session

Initially, the session will be in some defined state. We will assume that nothing has happened yet. Therefore, both protocol entities will be in state *IDLE*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitialiseSession}} \\
\hline
\Delta \textit{SessionInfo} \\
\hline
\textit{client_state}' = \textit{IDLE} \\
\textit{server_state}' = \textit{IDLE} \\
\\
\textit{server_seqno}' \leqslant \textit{client_seqno}' \\
\hline
\end{array}$$

We can choose to start with any values for the sequence numbers, so long as the value for the REX-client entity is not smaller than the one for the REX-server entity. The reason for this will become apparent in the next section.

11 A simple interrogation

We will first consider an interrogation interaction, involving no timeouts and no fragmentation. An interrogation of this form will involve the following four stages:

1. The REX-client entity accepts a $CALL$ request and sends it as a single message to the REX-server entity.
2. The REX-server entity receives this message and submits it as a request to the server entity.
3. When the server entity returns a response, the REX-server entity sends it as a single message to the REX-client entity.
4. The REX-client entity receives this message and passes it on to the waiting client entity.

We now take a look at each of these stages in turn.

11.1 A simple interrogation - stage 1

The acceptance of a $CALL$ request by the REX-client entity, as part of a simple interrogation, is described by the following schema:

$$
\begin{array}{l}
\textit{ClientCallRequest} \\
\hline
\Xi RexServerInfo \\
\Delta RexClientInfo \\
request? : Request \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = IDLE \\
request?.type = CALL \land request?.dest = REX\_CLIENT \\
request?.data\_size \leqslant max\_data\_size \\
client\_state' = CALLING \\
client\_seqno' = new\_seqno \\
message!.type = CALL \land message!.dest = REX\_SERVER \\
message!.seqno = new\_seqno \\
message!.data = request?.data \\
client\_buffer' = \{message!\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = REPLY \\
\textbf{where} \\
\quad new\_seqno = client\_seqno + 1 \\
\hline
\end{array}
$$

New $CALL$ requests from the client entity will only be accepted if the REX-client entity is in the $IDLE$ state. Furthermore, since for the time-being we are assuming that no fragmentation is required, only requests which are not greater than max_data_size are considered here. On accepting a request, the REX-client entity will:

- change to the $CALLING$ state, thus ensuring that no further client entity requests are accepted before a response has been issued;
- transmit the $CALL$ request as a message to the REX-server entity;
- keep a copy of this message;
- start the timer with a $REPLY$ timeout period.

The REX-client will also increment the value of the sequence number held. This value is then used for the sequence number of the outgoing message.

11.2 A simple interrogation - stage 2

For this stage of the interrogation we describe the conditions under which the REX-server entity should receive the message, and the actions that it should take.

$$
\begin{array}{|l}
\hline
\textit{ServerReceiveCallMessage} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : DataMessage \\
request! : Request \\
timer\_update! : TimerUpate \\
\hline
message?.type = CALL \land message?.dest = REX\_SERVER \\
message?.seqno > server\_seqno \\
server\_seqno' = message?.seqno \\
server\_state \notin \{ASKED, CALL\_RECEIVING\} \\
server\_state = SENDING \Rightarrow \\
\quad NOT\_OK \notin \operatorname{ran} request\_frags\_map \\
server\_state \in \{REPLYING, SENDING, CAST\_RECEIVING\} \Rightarrow \\
\quad timer\_update!.type = STOP \land \\
\quad timer\_update!.source = REX\_SERVER \\
server\_state = IDLE \Rightarrow \\
\quad timer\_update!.type = NO\_UPDATE \\
server\_state' = ASKED \\
request!.type = CALL \land request!.dest = SERVER \\
request!.data = message?.data \\
\hline
\end{array}
$$

On receiving the message, the sequence number of the incoming data message will be greater than that currently held by the REX-server entity. The protocol entity will therefore update its sequence number to that of the incoming message.

A protocol error will exist if the REX-server entity is in an $ASKED$ or $CALL\_RECEIVING$ state, or, if it is in state $SENDING$ and there are still some fragments of a response (to a previous request) to be sent. This is because such a condition would imply that the REX-client entity is waiting for more than one response message; violating our definition of an interrogation.

Prior to the arrival of this data message, the REX-server might still not know whether a previously sent response has arrived at the REX-client entity. If this is the case, the REX-server would not be in the $IDLE$ state and a timer would have been previously $START$ed.

The arrival of this message implicitly acknowledges the receipt of any response sent by the REX-server entity. (The reason for this again being that the REX-client entity is only allowed to have one outstanding response.) The REX-server entity might therefore need to issue a *STOP* to its timer.

Finally, we also point out that the *data_size* component need not be specified when a request is made to the server entity.

11.3 A simple interrogation - stage 3

The server entity may take an indefinite amount of time to process the request and to return a response. On the eventuality of an arrival of a response, the REX-server entity will then transmit this as a message to the REX-client entity.

$$
\begin{array}{l}
\textit{ServerCallReply} \\
\hline
\Xi \textit{RexClientInfo} \\
\Delta \textit{RexServerInfo} \\
\textit{response}? : \textit{Response} \\
\textit{message}! : \textit{DataMessage} \\
\textit{timer\_update}! : \textit{TimerUpdate} \\
\hline
\textit{server\_state} = \textit{ASKED} \\
\textit{response}?.\textit{type} = \textit{RESPONSE} \wedge \textit{response}?.\textit{dest} = \textit{REX\_SERVER} \\
\textit{response}?.\textit{data\_size} \leqslant \textit{max\_data\_size} \\
\textit{server\_state}' = \textit{REPLYING} \\
\textit{server\_seqno}' = \textit{new\_seqno} \\
\textit{server\_buffer}' = \{\textit{message}!\} \\
\textit{message}!.\textit{type} = \textit{REPLY} \wedge \textit{message}!.\textit{dest} = \textit{REX\_CLIENT} \\
\textit{message}!.\textit{seqno} = \textit{new\_seqno} \\
\textit{message}!.\textit{data} = \textit{response}?.\textit{data} \\
\textit{timer\_update}!.\textit{type} = \textit{START} \\
\textit{timer\_update}!.\textit{source} = \textit{REX\_SERVER} \\
\textit{timer\_update}!.\textit{period} = \textit{REPLY} \\
\textbf{where} \\
\quad \textit{new\_seqno} = \textit{server\_seqno} + 1 \\
\hline
\end{array}
$$

The situation is analogous to the one encountered in stage 1 of the interrogation; hence we do not comment any further.

11.4 A simple interrogation - stage 4

The final stage of the interrogation deals with the receipt of the response message by the REX-client entity, and its delivery to the waiting client entity.

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveReplyMessage} \\
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : DataMessage \\
response! : Response \\
timer\_update! : TimerUpdate \\
\hline
message?.type = REPLY \wedge message?.dest = REX\_CLIENT \\
message?.seqno > client\_seqno \\
client\_seqno' = message?.seqno \\
client\_state \notin \{IDLE, RECEIVING, CAST\_SENDING\} \\
client\_state' = IDLE \\
response!.type = RESPONSE \wedge response!.dest = CLIENT \\
response!.data = message?.data \\
timer\_update!.type = STOP \\
timer\_update!.source = REX\_CLIENT \\
\hline
\end{array}
$$

This stage is analogous to stage 2. In this case, however, the REX-client entity will always need to issue a $STOP$ to its timer.

We again make it clear that the protocol entity may have only one outstanding response by restricting the set of states in which it may be in.

12 Interrogations involving timeouts

In this section we start looking at how the REX protocol tackles the problem of delays in message transfers. Delays can in fact be infinite, as in the case of messages getting lost.

12.1 Timeouts

We saw in the first stage of an interrogation that on transmitting a $CALL$ message, the REX-client would activate the timer. Now suppose that the message does not reach the REX-server entity at all. After a time $REPLY$, the timeout will occur. The REX-client entity will assume that the message got lost. It will therefore re-transmit the request message and re-activate the timer.

$$
\begin{array}{|l}
\hline
\textit{ClientCallingTimeout} \\
\Xi SessionInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = CALLING \\
timeout?.dest = REX\_CLIENT \\
message! \in client\_buffer \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = REPLY \\
\hline
\end{array}
$$

The situation is virtually identical in the case of a response message getting lost.

$$
\begin{array}{|l}
\hline
\textit{ServerReplyingTimeout} \\
\Xi SessionInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
server\_state = REPLYING \\
timeout?.dest = REX\_SERVER \\
message! \in server\_buffer \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
timer\_update!.period = REPLY \\
\hline
\end{array}
$$

12.2 Repeat response messages

Even if a response message is received by the REX-client entity, the REX-server entity may still time-out and perform the operations represented in the schema *ServerReplyingTimeout* given above. This might happen because the next request message (sent by the REX-client entity) that would implicitly acknowledge the response, might get delayed or lost.

We therefore have the possibility of the REX-client entity receiving a repeat of a response message.

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveRepeatReplyMessage} \\
\Xi SessionInfo \\
message? : DataMessage \\
message! : ControlMessage \\
\hline
message?.type = REPLY \wedge message?.dest = REX\_CLIENT \\
message?.seqno \leqslant client\_seqno \\
message?.seqno = client\_seqno \Rightarrow \\
\quad client\_state = IDLE \wedge \\
\quad message!.type = REPLY\_ACK \wedge \\
\quad message!.dest = REX\_SERVER \wedge \\
\quad message!.seqno = message?.seqno \\
message?.seqno < client\_seqno \Rightarrow \\
\quad message!.type = NO\_MESSAGE \\
\hline
\end{array}
$$

When a repeat response message is received, the REX-client entity would decide whether or not to send an explicit acknowledgment on the following basis:

- if the sequence number of the repeat response message is equal to that currently held by the protocol entity, then the REX-client entity should be in the *IDLE* state and an explicit acknowledgment is sent.
- if however the sequence number of the message is less, the protocol entity would have issued a new request message; therefore, it may assume that the response would be implicitly acknowledged by the new request.

12.3 Repeat request messages

Similarly, even if a request message is received by the REX-server entity, the REX-client entity may still timeout and perform the operations described by the schema *ClientCallingTimeout* given in section 12.1.

This might be due to:

- the response messages being delayed or lost, or
- the REX-server entity taking longer than expected to process the request and issue a reply.

Therefore, the REX-server entity may also receive repeated $CALL$ requests.

$$
\begin{array}{|l}
\hline
\textit{ServerReceiveRepeatCallMessage} \\
\Xi SessionInfo \\
message? : DataMessage \\
message! : ControlMessage \\
\hline
message?.type = CALL \wedge message?.dest = REX\_SERVER \\
message?.seqno \leqslant server\_seqno \\
message?.seqno = server\_seqno \Rightarrow \\
\quad server\_state = ASKED \wedge \\
\quad message!.type = CALL\_ACK \wedge \\
\quad message!.dest = REX\_CLIENT \wedge \\
\quad message!.seqno = message?.seqno \\
message?.seqno < server\_seqno \Rightarrow \\
\quad message!.type = NO\_MESSAGE \\
\hline
\end{array}
$$

In this case, the explicit acknowledgment might be sent to inform the REX-client entity that the request had in fact been received and that it is being processed.

12.4 Explicit response acknowledgments - $REPLY_ACK$s

Control messages are subject to the same delay problems that we considered earlier for data messages. In particular, explicit acknowledgments may take any period of time to reach their destination protocol entity. This may therefore lead to a situation where the REX-client entity has sent more than one explicit acknowledgment for a particular response.

Bearing the above consideration in mind, we now give a schema which deals with the arrival of an explicit acknowledgment at the REX-server entity.

$$
\begin{array}{l}
\textit{ServerReceiveReplyAckMessage\_A} \\
\hline
\Xi \textit{RexClientInfo} \\
\Delta \textit{RexServerInfo} \\
\textit{message}? : \textit{ControlMessage} \\
\textit{timer\_update}! : \textit{TimerUpdate} \\
\hline
\textit{message}?.\textit{type} = \textit{REPLY\_ACK} \\
\textit{message}?.\textit{dest} = \textit{REX\_SERVER} \\
\textit{message}?.\textit{seqno} \leqslant \textit{server\_seqno} \\
\textit{message}?.\textit{seqno} < \textit{server\_seqno} \Rightarrow \textit{ignore} \\
\textit{message}?.\textit{seqno} = \textit{server\_seqno} \Rightarrow \\
\quad (\textit{server\_state} = \textit{REPLYING} \Rightarrow \textit{awaited\_ack\_received}) \wedge \\
\quad (\textit{server\_state} = \textit{IDLE} \Rightarrow \textit{ignore}) \wedge \\
\quad \textit{server\_state} \notin \\
\qquad \{\textit{ASKED}, \textit{CALL\_RECEIVING}, \textit{CAST\_RECEIVING}\} \\
\textit{server\_seqno}' = \textit{server\_seqno} \\
\textit{server\_buffer}' = \textit{server\_buffer} \\
\textit{response\_frags\_map}' = \textit{response\_frags\_map} \\
\textbf{where} \\
\quad \textit{ignore} = (\\
\qquad \textit{server\_state}' = \textit{server\_state} \wedge \\
\qquad \textit{timer\_update}!.\textit{type} = \textit{NO\_UPDATE}) \\
\quad \textit{awaited\_ack\_received} = (\\
\qquad \textit{server\_state}' = \textit{IDLE} \wedge \\
\qquad \textit{timer\_update}!.\textit{type} = \textit{STOP} \wedge \\
\qquad \textit{timer\_update}!.\textit{source} = \textit{REX\_SERVER}) \\
\hline
\end{array}
$$

The sequence number of the acknowledgment will be equal to the sequence number of the response message which it refers to. Hence, it may not be greater than the sequence number currently held by the REX-server entity.

If the sequence number of the acknowledgment is less than that held by the REX-server entity, this shall indicate that the response in question had already been acknowledged at some prior stage. So, in this case the acknowledgment can be ignored.

If the sequence numbers are equal then:

- the acknowledgment can only be ignored if the protocol entity is in the *IDLE* state.
- if not in the *IDLE* state, the protocol entity would be anxiously awaiting the acknowledgment, i.e. the response in question would not have been acknowledged and the timer would still be active.
- we restrict the set of states in which the protocol entity may be in to reaffirm that the REX-client entity may only have one outstanding *CALL* request.

12.5 Explicit request acknowledgments - *CALL_ACK*s

The receiving of *CALL_ACK*s by the REX-client entity is analogous to the receiving of *REPLY_ACK*s by the REX-server entity discussed above. The significant difference is that when the REX-client entity receives the first explicit acknowledgment for an outstanding *CALL* request, it will then start probing the REX-server entity. Probing is discussed in the next section.

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveCallAckMessage\_A} \\
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CALL\_ACK \\
message?.dest = REX\_CLIENT \\
message?.seqno \leqslant client\_seqno \\
message?.seqno < client\_seqno \Rightarrow ignore \\
message?.seqno = client\_seqno \Rightarrow \\
\quad (client\_state = CALLING \Rightarrow start\_probing) \land \\
\quad client\_state \notin \\
\qquad \{IDLE, RECEIVING, CAST\_SENDING, PROBING\} \\
client\_seqno' = client\_seqno \\
client\_buffer' = client\_buffer \\
request\_frags\_map' = request\_frags\_map \\
\textbf{where} \\
\quad ignore = (\\
\qquad client\_state' = client\_state \land \\
\qquad timer\_update!.type = NO\_UPDATE) \\
\quad start\_probing = (\\
\qquad client\_state' = PROBING \land \\
\qquad timer\_update!.type = START \land \\
\qquad timer\_update!.source = REX\_CLIENT \land \\
\qquad timer\_update!.period = PROBE) \\
\hline
\end{array}
$$

12.6 The *PROBING* state

The REX-client entity will be in the *PROBING* state when:

- it is known that a transmitted *CALL* request has been received by the REX-server entity, and
- the response has not yet been received.

While in the *PROBING* state, the REX-client entity periodically sends a *PROBE* message. The message will carry the sequence number of the outstanding request.

$$
\begin{array}{|l}
\hline
\textit{ClientProbeTimeout} \\
\Xi SessionInfo \\
timeout? : Timeout \\
message! : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = PROBING \\
timeout?.dest = REX\_CLIENT \\
message!.type = PROBE \land message!.dest = REX\_SERVER \\
message!.seqno = client\_seqno \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = PROBE \\
\hline
\end{array}
$$

When a *PROBE* message is received by the REX-server entity:

- the $PROBE$ is ignored if the sequence number of the message refers to a $CALL$ request for which the response has already been sent.
- the protocol entity (which should be in the $ASKED$ state) will send a $CALL\_ACK$ message explicitly acknowledging the $PROBE$ message, if the request has not been replied to.

$$
\begin{array}{|l}
\hline
\textit{ServerReceiveProbeMessage} \\
\Xi SessionInfo \\
message? : ControlMessage \\
message! : ControlMessage \\
\hline
message?.type = PROBE \land message?.dest = REX\_SERVER \\
message?.seqno \leqslant server\_seqno \\
message?.seqno < server\_seqno \Rightarrow \\
\quad message!.type = NO\_MESSAGE \\
message?.seqno = server\_seqno \Rightarrow \\
\quad server\_state = ASKED \land \\
\quad message!.type = CALL\_ACK \land \\
\quad message!.dest = REX\_CLIENT \land \\
\quad message!.seqno = message?.seqno \\
\hline
\end{array}
$$

When a $PROBING$ REX-client entity receives a $CALL\_ACK$ message it is assured that the REX-server entity is still active and that no serious communication failure has occurred. Provided these probes continue to be acknowledged and no response arrives, the REX-client will remain in the $PROBING$ state indefinitely.

We therefore extend our description of the receiving of explicit request acknowledgments (given in Section 12.5) as follows:

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveCallAckMessage\_}\mathcal{B} \\
\Xi SessionInfo \\
message? : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CALL\_ACK \\
message?.dest = REX\_CLIENT \\
message?.seqno = client\_seqno \\
client\_state = PROBING \\
continue\_probing \\
\textbf{where} \\
\quad continue\_probing = (\\
\quad\quad timer\_update!.type = START \land \\
\quad\quad timer\_update!.source = REX\_CLIENT \land \\
\quad\quad timer\_update!.period = PROBE) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
ClientReceiveCallAckMessage\_\mathcal{C} \mathrel{\widehat=} \\
\qquad ClientReceiveCallAckMessage\_\mathcal{A} \\
\qquad \lor \\
\qquad ClientReceiveCallAckMessage\_\mathcal{B}
\end{array}
$$

13 Announcement interactions

In this section we turn our attention to the announcement style of interaction. We again start by looking at announcements that do not involve fragmentation.

Announcements do not elicit any sort of response. Therefore, a simple announcement will only involve the following two stages:

1. The REX-client accepts a *CAST* request and sends it as a single message to the REX-server entity.
2. The REX-server entity receives this message and delivers it as a request to the server entity.

13.1 A simple announcement - stage 1

The receipt of a *CAST* request by the REX-client entity, as part of a simple announcement, is described by the following schema:

$$
\begin{array}{|l}
\textit{ClientCastRequest} \\
\hline
\Xi RexServerInfo \\
\Delta RexClientInfo \\
request? : Request \\
message! : DataMessage \\
\hline
client\_state = IDLE \\
request?.type = CAST \land request?.dest = REX\_CLIENT \\
request?.data\_size \leqslant max\_data\_size \\
client\_state' = client\_state \\
client\_seqno' = new\_seqno \\
message!.type = CAST \land message!.dest = REX\_SERVER \\
message!.seqno = new\_seqno \\
message!.data = request?.data \\
\textbf{where} \\
\quad new\_seqno = client\_seqno + 1 \\
\hline
\end{array}
$$

This stage is similar to the first stage of a simple interrogation. However, the following differences are worth pointing out:

- The protocol entity does not need to keep a copy of the request message that is sent. For announcements the protocol does not require the REX-client entity to confirm that a request message has reached the REX-server entity. So, it will never need to re-transmit a request message.
- Since no response is expected, the protocol entity does not need to activate the timer, and no state change is involved.
- Hence the protocol entity will still be in the *IDLE* state after the *CAST* request has been sent, and the client entity may make a new request if it so wishes.

13.2 A simple announcement - stage 2

This stage is analogous to stage 2 of a simple interrogation. The only significant difference is that when a *CAST* message is received, the protocol entity will return to the *IDLE* state - since no response is to be sent back to the REX-client entity.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerReceiveCastMessage\_A}} \\
\hline
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : DataMessage \\
request! : Request \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CAST \land message?.dest = REX\_SERVER \\
message?.seqno > server\_seqno \\
server\_state \notin \{ASKED, CALL\_RECEIVING\} \\
server\_state = SENDING \Rightarrow \\
\quad NOT\_OK \notin \operatorname{ran} request\_frags\_map \\
server\_state \in \{REPLYING, SENDING, CAST\_RECEIVING\} \Rightarrow \\
\quad timer\_update!.type = STOP \land \\
\quad timer\_update!.source = REX\_SERVER \\
server\_state \in \{IDLE\} \Rightarrow \\
\quad timer\_update!.type = NO\_UPDATE \\
server\_state' = IDLE \\
server\_seqno' = message?.seqno \\
request!.type = CAST \\
request!.dest = SERVER \\
request!.data = message?.data \\
\hline
\end{array}
$$

13.3 Delayed $CAST$ request messages

We have seen in section 13.1 that after sending a $CAST$ message, the REX-client entity does not have to wait before starting another interaction by sending a new request message. This might lead to the following sequence of events happening:

1. a $CAST$ message is sent and gets delayed;
2. the REX-client entity sends another request message;
3. the latter request message arrives at the REX-server entity;
4. the $CAST$ message then arrives at the REX-server entity;

In these situations, where a $CAST$ message is overtaken by another request message, the $CAST$ will be ignored when it eventually arrives at the REX-server entity.

To take this into account we extend the schema given in section 13.2 as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerReceiveCastMessage\_B}} \\
\hline
\Xi SessionInfo \\
message? : DataMessage \\
request! : Request \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CAST \land message?.dest = REX\_SERVER \\
message?.seqno < server\_seqno \\
request!.type = NO\_REQUEST \\
timer\_update!.type = NO\_UPDATE \\
\hline
\end{array}
$$

$$ServerReceiveCastMessage \mathrel{\widehat=} ServerReceiveCastMessage\_A \lor ServerReceiveCastMessage\_B$$

14 Fragmentation in interrogation interactions

In this section we introduce the fragmentation aspects of the REX-protocol. This is done in the context of interrogation interactions.

An interrogation interaction may involve fragmentation in two ways:

- The $CALL$ request may be too large for the REX-client entity to send as a single message. So it is broken down into a number of fragments which are sent as separate fragments.
- Similarly, the response may be too large to be transmitted in the opposite direction.

We will now take the former case and describe it in detail.

14.1 Fragmented $CALL$ requests

14.1.1 Accepting the $CALL$ and sending the first fragment

$$
\begin{array}{l}
\textbf{schema } ClientLargeCallRequest \\
\quad \Xi RexServerInfo \\
\quad \Delta RexClientInfo \\
\quad request? : Request \\
\quad message! : DataMessage \\
\quad timer\_update! : TimerUpdate \\
\hline
\quad client\_state = IDLE \\
\quad request?.type = CALL \land request?.dest = REX\_CLIENT \\
\quad request?.data\_size > max\_data\_size \\
\quad client\_state' = CALL\_SENDING \\
\quad client\_seqno' = new\_seqno \\
\quad client\_buffer' = \\
\qquad fragment(request?.data, new\_seqno, CALL\_FRAG) \\
\quad message! \in client\_buffer' \\
\quad message!.offset = 1 \\
\quad request\_frags\_map' = initialized\_map \oplus \{(1, OK)\} \\
\quad timer\_update!.type = START \\
\quad timer\_update!.source = REX\_CLIENT \\
\quad timer\_update!.period = FLOW \\
\quad \textbf{where} \\
\qquad new\_seqno = client\_seqno + 1 \\
\qquad initialized\_map = \\
\qquad\quad \{ \quad i : FragIndex; \ f : MapFlag \mid \\
\qquad\qquad\quad f = NOT\_OK \land \\
\qquad\qquad\quad \exists_1 \, m : DataMessage \bullet \\
\qquad\qquad\qquad m \in client\_buffer' \land i = m.offset \\
\qquad\quad \}
\end{array}
$$

The Rex-client entity will store all fragment messages returned by the *fragment*[1] function.

The protocol entity will send the first of the fragment messages. It will record that the first fragment has been sent and also that all the remaining ones are still to be transmitted. The timer is activated with a *FLOW* timeout period.

14.1.2 Transmission of the other *CALL* fragments

After a time *FLOW* from the transmission of the first fragment, a timeout will occur and the next fragment is transmitted. The REX-client entity will also re-activate the timer on the following basis:

- If still more fragments remain to be transmitted, the timer is activated with the *FLOW* timeout period again; so that on the next timeout another fragment is transmitted.
- But, if all fragments are now marked as *OK*, then the protocol entity would have sent the complete request; in this case the timer is activated with the *REPLY* timeout period.

This represented by the following schema:

$$
\begin{array}{l}
\textit{ClientCallSendingRemainingFragmentsTimeout} \\
\hline
\Xi RexServerInfo \\
\Delta RexClientInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = CALL\_SENDING \\
timeout?.dest = REX\_CLIENT \\
NOT\_OK \in \operatorname{ran} request\_frags\_map \\
message! \in client\_buffer \\
message!.offset = next\_frag\_to\_send \\
request\_frags\_map' = \\
\quad request\_frags\_map \oplus \{(next\_frag\_to\_send, OK)\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
NOT\_OK \notin \operatorname{ran} request\_frags\_map' \Rightarrow \\
\quad timer\_update!.period = REPLY \\
NOT\_OK \in \operatorname{ran} request\_frags\_map' \Rightarrow \\
\quad timer\_update!.period = FLOW \\
client\_state' = client\_state \\
client\_seqno' = client\_seqno \\
client\_buffer' = client\_buffer \\
\textbf{where} \\
\quad next\_frag\_to\_send = \\
\qquad min \{ \quad i : FragIndex \mid \\
\qquad\qquad i \in \operatorname{dom} request\_frags\_map \land \\
\qquad\qquad request\_frags\_map(i) = NOT\_OK \\
\qquad \} \\
\hline
\end{array}
$$

If a timeout occurs and there are no more fragments to send, then this would mean that a time *REPLY* has passed from the transmission of the complete request. The protocol entity would therefore suspect that something went wrong and it will re-transmit the first fragment.

[1] The *fragment* function is described in Appendix A.

$$\begin{array}{|l}
\hline
\textit{ClientCallSendingNoReplyTimeout} \\
\Xi SessionInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer_update! : TimerUpdate \\
\hline
client_state = CALL_SENDING \\
timeout?.dest = REX_CLIENT \\
NOT_OK \notin \mathrm{ran}\ request_frags_map \\
message! \in client_buffer \\
message!.offset = 1 \\
timer_update!.type = START \\
timer_update!.source = REX_CLIENT \\
timer_update!.period = REPLY \\
\hline
\end{array}$$

14.1.3 Receipt of the first *CALL* fragment

The first fragment *CALL* that is received by the REX-server entity will have a sequence number which is greater than that held by the REX-server entity.

The fragment will be stored and the protocol entity will wait for more fragments to arrive. It knows that there are more fragments to come since a fragmented request will always consist of more than one fragment message. As a precaution, the timer is also started. The action taken on timeouts will be described later.

$$\begin{array}{|l}
\hline
\textit{ServerReceiveFirstFragmentOfCall} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : DataMessage \\
timer_update! : TimerUpdate \\
\hline
server_state \notin \{ASKED, CALL_RECEIVING\} \\
message?.type = CALL_FRAG \\
message?.dest = REX_SERVER \\
message?.seqno > server_seqno \\
server_state' = CALL_RECEIVING \\
server_seqno' = message?.seqno \\
server_buffer' = \{message?\} \\
timer_update!.type = START \\
timer_update!.source = REX_SERVER \\
timer_update!.period = CHECK \\
\hline
\end{array}$$

It is worth also pointing out that we have left open the question of whether the first fragment received is actually the first one sent by the REX-client entity. This may well not be the case.

14.1.4 Receipt of other *CALL* fragments

Subsequent fragment messages that are received will have the same sequence number as that held by the REX-server entity. We first consider the arrival of a fragment which does not complete the *CALL* request.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerReceiveNonFinalFragmentOfCall}} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
server\_state = CALL\_RECEIVING \\
message?.type = CALL\_FRAG \\
message?.dest = REX\_SERVER \\
message?.seqno = server\_seqno \\
fragment\_is\_not\_repeat \\
fragment\_is\_not\_final \\
server\_buffer' = server\_buffer \cup \{message?\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
timer\_update!.period = CHECK \\
server\_state' = server\_state \\
server\_seqno' = server\_seqno \\
\mathbf{where} \\
\quad fragment\_is\_not\_repeat = \\
\quad\quad \forall m : DataMessage \bullet \\
\quad\quad\quad m \in server\_buffer \Rightarrow m.offset \neq message?.offset \\
\quad fragment\_is\_not\_final = \\
\quad\quad \exists i : 1 \mathrel{..} max\_offset \bullet i \notin fragments\_received \\
\quad max\_offset = message?.total\_size \\
\quad fragments\_received = \\
\quad\quad \{ \quad i : FragIndex \mid \\
\quad\quad\quad\quad \exists_1 m : DataMessage \bullet \\
\quad\quad\quad\quad\quad m \in server\_buffer' \land i = m.offset \\
\quad\quad \} \\
\hline
\end{array}
$$

When the fragment which is received does complete the set of message fragments, the REX-server entity will *build*[2] the complete request again and delivers it to the server entity.

[2] The *build* function is described in Appendix A.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, \mathit{ServerReceiveFinalFragmentOfCall}\, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{RexClientInfo} \\
\Delta \mathit{RexServerInfo} \\
\mathit{message?} : \mathit{DataMessage} \\
\mathit{timer\_update!} : \mathit{TimerUpdate} \\
\mathit{request!} : \mathit{Request} \\
\hline
\mathit{server\_state} = \mathit{CALL\_RECEIVING} \\
\mathit{message?.type} = \mathit{CALL\_FRAG} \\
\mathit{message?.dest} = \mathit{REX\_SERVER} \\
\mathit{message?.seqno} = \mathit{server\_seqno} \\
\forall\, m : \mathit{DataMessage} \bullet \\
\qquad m \in \mathit{server\_buffer} \Rightarrow m.\mathit{offset} \neq \mathit{message?.offset} \\
\forall\, i : 1 \mathrel{..} \mathit{max\_offset} \bullet i \in \mathit{fragments\_received} \\
\mathit{server\_buffer}' = \mathit{server\_buffer} \cup \{\mathit{message?}\} \\
\mathit{server\_state}' = \mathit{ASKED} \\
\mathit{timer\_update!.type} = \mathit{STOP} \\
\mathit{timer\_update!.source} = \mathit{REX\_SERVER} \\
\mathit{request!} = \mathit{build}(\mathit{server\_buffer}', \mathit{SERVER}) \\
\mathit{server\_seqno}' = \mathit{server\_seqno} \\
\textbf{where} \\
\qquad \mathit{max\_offset} = \mathit{message?.total\_size} \\
\qquad \mathit{fragments\_received} = \\
\qquad\qquad \{ \qquad i : \mathit{FragIndex} \mid \\
\qquad\qquad\qquad\quad \exists_1\, m : \mathit{DataMessage} \bullet \\
\qquad\qquad\qquad\qquad m \in \mathit{server\_buffer}' \land i = m.\mathit{offset} \\
\qquad\qquad \} \\
\hline
\end{array}
$$

14.1.5 Delays when receiving $CALL$ fragments

The REX-server entity will suspect that fragments have been lost if a time $CHECK$ passes without a fragment being received. It would then try to correct the situation by sending a negative acknowledgment message informing the REX-client entity of the fragments that have been received so far.

The following schema describes this situation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\mathit{ServerCallReceivingTimeout}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{SessionInfo} \\
\mathit{timeout}? : \mathit{Timeout} \\
\mathit{message}! : \mathit{ControlMessage} \\
\mathit{timer\_update}! : \mathit{TimerUpdate} \\
\hline
\mathit{timeout}?.\mathit{dest} = \mathit{REX\_SERVER} \\
\mathit{server\_state} = \mathit{CALL\_RECEIVING} \\
\mathit{message}!.\mathit{type} = \mathit{FRAG\_NACK} \\
\mathit{message}!.\mathit{dest} = \mathit{REX\_CLIENT} \\
\mathit{message}!.\mathit{seqno} = \mathit{server\_seqno} \\
\mathit{message}!.\mathit{frags\_map} = \\
\quad \{ \quad i : \mathit{FragIndex} \mid \\
\qquad\quad \exists_1\, m : \mathit{DataMessage} \bullet \\
\qquad\qquad m \in \mathit{server\_buffer} \land i = m.\mathit{offset} \\
\quad \} \\
\mathit{timer\_update}!.\mathit{type} = \mathit{START} \\
\mathit{timer\_update}!.\mathit{source} = \mathit{REX\_SERVER} \\
\mathit{timer\_update}!.\mathit{period} = \mathit{CHECK} \\
\hline
\end{array}
$$

When the REX-client entity receives a negative acknowledgment message, it uses the information contained in this message to update its list of fragments to be sent. In this way any fragments which are lost will eventually be re-transmitted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\mathit{ClientReceiveFragNackMessageForCall}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{RexServerInfo} \\
\Delta \mathit{RexClientInfo} \\
\mathit{message}? : \mathit{ControlMessage} \\
\hline
\mathit{message}?.\mathit{type} = \mathit{FRAG\_NACK} \\
\mathit{message}?.\mathit{dest} = \mathit{REX\_CLIENT} \\
\mathit{message}?.\mathit{seqno} = \mathit{client\_seqno} \\
\mathit{client\_state} = \mathit{CALL\_SENDING} \\
\mathit{request\_frags\_map}' = \\
\quad \{ \quad i : \mathit{FragIndex};\ f : \mathit{MapFlag} \mid \\
\qquad\quad i \leqslant \mathit{max\_offset} \land \\
\qquad\quad (i \in \mathit{message}?.\mathit{frags\_map} \Rightarrow f = \mathit{OK}) \land \\
\qquad\quad (i \notin \mathit{message}?.\mathit{frags\_map} \Rightarrow f = \mathit{NOT\_OK}) \\
\quad \} \\
\mathit{client\_state}' = \mathit{client\_state} \\
\mathit{client\_seqno}' = \mathit{client\_seqno} \\
\mathit{client\_buffer}' = \mathit{client\_buffer} \\
\textbf{where} \\
\quad \mathit{max\_offset} = \#\mathit{request\_frags\_map} \\
\hline
\end{array}
$$

14.2 Fragmented responses

As we stated earlier, the interrogation may also involve fragmentation if the response is too large to send back as a single message. This is very similar to the transmission of a fragmented $CALL$ request, so we will just give the schemas and comment only if there are any major differences.

14.2.1 Accepting the response and sending the first fragment

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerLargeCallReply}} \\
\hline
\Xi RexClientInfo \\
\Delta RexServerInfo \\
response? : Response \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
server\_state = ASKED \\
response?.type = RESPONSE \\
response?.dest = REX\_SERVER \\
response?.data\_size > max\_data\_size \\
server\_state' = SENDING \\
server\_seqno' = new\_seqno \\
server\_buffer' = \\
\quad fragment(response?.data, new\_seqno, REPLY\_FRAG) \\
message! \in server\_buffer' \land message!.offset = 1 \\
response\_frags\_map' = initialized\_map \oplus \{(1, OK)\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
timer\_update!.period = FLOW \\
\textbf{where} \\
\quad new\_seqno = server\_seqno + 1 \\
\quad initialized\_map = \\
\qquad \{ \quad i : FragIndex;\ f : MapFlag \mid \\
\qquad\qquad f = NOT\_OK \land \\
\qquad\qquad \exists_1\, m : DataMessage \bullet \\
\qquad\qquad\quad m \in server\_buffer' \land i = m.offset \\
\qquad \} \\
\hline
\end{array}
$$

14.2.2 Transmission of the other response fragments

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, ServerReplySendingRemainingFragmentsTimeout \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
server\_state = SENDING \\
timeout?.dest = REX\_SERVER \\
NOT\_OK \in \operatorname{ran} response\_frags\_map \\
message! \in server\_buffer \\
message!.offset = next\_frag\_to\_send \\
response\_frags\_map' = \\
\quad response\_frags\_map \oplus \{(next\_frag\_to\_send, OK)\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
NOT\_OK \notin \operatorname{ran} response\_frags\_map' \Rightarrow \\
\quad timer\_update!.period = REPLY \\
NOT\_OK \in \operatorname{ran} response\_frags\_map' \Rightarrow \\
\quad timer\_update!.period = FLOW \\
server\_state' = server\_state \\
server\_seqno' = server\_seqno \\
server\_buffer' = server\_buffer \\
\textbf{where} \\
\quad next\_frag\_to\_send = \\
\qquad min\, \{ \quad i : FragIndex \mid \\
\qquad\qquad i \in \operatorname{dom} response\_frags\_map \land \\
\qquad\qquad response\_frags\_map(i) = NOT\_OK \\
\qquad\quad \} \\
\hline
\end{array}
$$

The REX-server entity timing-out and no more fragments of response to send:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, ServerReplySendingNoAckTimeout \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi SessionInfo \\
timeout? : Timeout \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
server\_state = SENDING \\
timeout?.dest = REX\_SERVER \\
NOT\_OK \notin \operatorname{ran} response\_frags\_map \\
message! \in server\_buffer \\
message!.offset = 1 \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
timer\_update!.period = REPLY \\
\hline
\end{array}
$$

The REX-server entity will re-transmit the first fragment of the response to re-assert that it requires an acknowledgment.

14.2.3 Receipt of the first fragment of response

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveFirstFragmentOfReply} \\
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state \notin \{IDLE, RECEIVING, CAST\_SENDING\} \\
message?.type = REPLY\_FRAG \\
message?.dest = REX\_CLIENT \\
message?.seqno > client\_seqno \\
client\_state' = RECEIVING \\
client\_seqno' = message?.seqno \\
client\_buffer' = \{message?\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = CHECK \\
\hline
\end{array}
$$

14.2.4 Receipt of other response fragments

$$
\begin{array}{|l}
\hline
\textit{ClientReceiveNonFinalFragmentOfReply} \\
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = RECEIVING \\
message?.type = REPLY\_FRAG \\
message?.dest = REX\_CLIENT \\
message?.seqno = client\_seqno \\
\forall m : DataMessage \bullet \\
\quad m \in client\_buffer \Rightarrow m.offset \neq message?.offset \\
\exists i : 1 \mathrel{..} max\_offset \bullet i \notin fragments\_received \\
client\_buffer' = client\_buffer \cup \{message?\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = CHECK \\
client\_state' = client\_state \\
client\_seqno' = client\_seqno \\
\textbf{where} \\
\quad max\_offset = message?.total\_size \\
\quad fragments\_received = \\
\qquad \{ \quad i : FragIndex \mid \\
\qquad\qquad \exists_1 m : DataMessage \bullet \\
\qquad\qquad\quad m \in client\_buffer' \land i = m.offset \\
\qquad \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClientReceiveFinalFragmentOfReply}} \\
\hline
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
response! : Response \\
\hline
client\_state = RECEIVING \\
message?.type = REPLY\_FRAG \\
message?.dest = REX\_CLIENT \\
message?.seqno = client\_seqno \\
\forall m : DataMessage \bullet \\
\quad m \in client\_buffer \Rightarrow m.offset \neq message?.offset \\
\forall i : 1 \mathrel{..} max\_offset \bullet i \in fragments\_received \\
client\_buffer' = client\_buffer \cup \{message?\} \\
client\_state' = IDLE \\
timer\_update!.type = STOP \\
timer\_update!.source = REX\_CLIENT \\
response! = build(client\_buffer', CLIENT) \\
client\_seqno' = client\_seqno \\
\textbf{where} \\
\quad max\_offset = message?.total\_size \\
\quad fragments\_received = \\
\qquad \{ \quad i : FragIndex \mid \\
\qquad\qquad \exists_1\, m : DataMessage \bullet \\
\qquad\qquad\quad m \in client\_buffer' \land i = m.offset \\
\qquad \} \\
\hline
\end{array}
$$

14.2.5 Delays when receiving fragments of response

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClientReplyReceivingTimeout}} \\
\hline
\Xi SessionInfo \\
timeout? : Timeout \\
message! : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
timeout?.dest = REX\_CLIENT \\
client\_state = RECEIVING \\
message!.type = FRAG\_NACK \\
message!.dest = REX\_SERVER \\
message!.seqno = client\_seqno \\
message!.frags\_map = \\
\quad \{ \quad i : FragIndex \mid \\
\qquad \exists_1\, m : DataMessage \bullet \\
\qquad\quad m \in client\_buffer \land i = m.offset \\
\quad \} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = CHECK \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ServerReceiveFragNackMessage \quad}} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : ControlMessage \\
\hline
message?.type = FRAG\_NACK \\
message?.dest = REX\_SERVER \\
message?.seqno = server\_seqno \\
server\_state = SENDING \\
response\_frags\_map' = \\
\quad \{ \quad i : FragIndex;\ f : MapFlag \mid \\
\qquad\quad i \leqslant max\_offset \land \\
\qquad\quad (i \in message?.frags\_map \Rightarrow f = OK) \land \\
\qquad\quad (i \notin message?.frags\_map \Rightarrow f = NOT\_OK) \\
\quad \} \\
server\_state' = server\_state \\
server\_seqno' = server\_seqno \\
server\_buffer' = server\_buffer \\
\textbf{where} \\
\quad max\_offset = \#response\_frags\_map \\
\hline
\end{array}
$$

15 Fragmentation in announcements

Announcements do not involve any responses. Hence the only way an announcement may involve fragmentation is when the $CAST$ request is too large to send as a single message. The way in which fragmentation is dealt with in announcements is similar to that encountered for interroagtion interactions. We will again just give the schemas and comment only where major differences exist.

15.1 Accepting the $CAST$ and sending the first fragment

$$
\begin{array}{l}
\textbf{ClientLargeCastRequest} \\
\hline
\Xi RexServerInfo \\
\Delta RexClientInfo \\
request? : Request \\
message! : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
client\_state = IDLE \\
request?.type = CAST \wedge request?.dest = REX\_CLIENT \\
request?.data\_size > max\_data\_size \\
client\_state' = CAST\_SENDING \\
client\_seqno' = new\_seqno \\
client\_buffer' = \\
\quad fragment(request?.data, new\_seqno, CAST\_FRAG) \\
message! \in client\_buffer' \\
message!.offset = 1 \\
request\_frags\_map' = initialized\_map \oplus \{(1, OK)\} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = FLOW \\
\textbf{where} \\
\quad new\_seqno = client\_seqno + 1 \\
\quad initialized\_map = \\
\qquad \{ \quad i : FragIndex;\ f : MapFlag \mid \\
\qquad\qquad f = NOT\_OK \wedge \\
\qquad\qquad \exists_1\, m : DataMessage \bullet \\
\qquad\qquad\quad m \in client\_buffer' \wedge i = m.offset \\
\qquad \}
\end{array}
$$

15.2 Transmission of the other *CAST* fragments

The situation in this case is **not** analogous to the the one encountered in interrogation interactions. When the REX-server entity has sent all the fragments of the *CAST* request, the timer is not activated and it changes to the *IDLE* state. This is because the protocol entity would not be expecting to receive any response or acknowledgment.

$$
\begin{array}{l}
\textit{ClientCastSendingRemainingFragmentsTimeout} \\
\hline
\Xi \mathit{RexServerInfo} \\
\Delta \mathit{RexClientInfo} \\
\mathit{timeout?} : \mathit{Timeout} \\
\mathit{message!} : \mathit{DataMessage} \\
\mathit{timer\_update!} : \mathit{TimerUpdate} \\
\hline
\mathit{client\_state} = \mathit{CAST\_SENDING} \\
\mathit{timeout?.dest} = \mathit{REX\_CLIENT} \\
\mathit{NOT\_OK} \in \operatorname{ran} \mathit{request\_frags\_map} \\
\mathit{message!} \in \mathit{client\_buffer} \\
\mathit{message!.offset} = \mathit{next\_frag\_to\_send} \\
\mathit{request\_frags\_map}' = \\
\quad \mathit{request\_frags\_map} \oplus \{(\mathit{next\_frag\_to\_send}, \mathit{OK})\} \\
\mathit{NOT\_OK} \in \operatorname{ran} \mathit{request\_frags\_map}' \Rightarrow \\
\quad \mathit{client\_state}' = \mathit{client\_state} \land \\
\quad \mathit{timer\_update!.type} = \mathit{START} \land \\
\quad \mathit{timer\_update!.source} = \mathit{REX\_CLIENT} \land \\
\quad \mathit{timer\_update!.period} = \mathit{FLOW} \\
\mathit{NOT\_OK} \notin \operatorname{ran} \mathit{request\_frags\_map}' \Rightarrow \\
\quad \mathit{client\_state}' = \mathit{IDLE} \land \\
\quad \mathit{timer\_update!.type} = \mathit{NO\_UPDATE} \\
\mathit{client\_seqno}' = \mathit{client\_seqno} \\
\mathit{client\_buffer}' = \mathit{client\_buffer} \\
\textbf{where} \\
\quad \mathit{next\_frag\_to\_send} = \\
\quad\quad \mathit{min}\, \{ \quad i : \mathit{FragIndex} \mid \\
\quad\quad\quad\quad i \in \operatorname{dom} \mathit{request\_frags\_map} \land \\
\quad\quad\quad\quad \mathit{request\_frags\_map}(i) = \mathit{NOT\_OK} \\
\quad\quad\quad \} \\
\hline
\end{array}
$$

15.3 Receipt of the first fragment of $CAST$ request

$$
\begin{array}{l}
\textit{ServerReceiveFirstFragmentOfCast} \\
\hline
\Xi \mathit{RexClientInfo} \\
\Delta \mathit{RexServerInfo} \\
\mathit{message?} : \mathit{DataMessage} \\
\mathit{timer\_update!} : \mathit{TimerUpdate} \\
\hline
\mathit{server\_state} \notin \\
\quad \{ \quad \mathit{ASKED}, \mathit{REPLYING}, \\
\quad\quad \mathit{SENDING}, \mathit{CALL\_RECEIVING} \\
\quad \} \\
\mathit{message?.type} = \mathit{CAST\_FRAG} \\
\mathit{message?.dest} = \mathit{REX\_SERVER} \\
\mathit{message?.seqno} > \mathit{server\_seqno} \\
\mathit{server\_state}' = \mathit{CAST\_RECEIVING} \\
\mathit{server\_seqno}' = \mathit{message?.seqno} \\
\mathit{server\_buffer}' = \{\mathit{message?}\} \\
\mathit{timer\_update!.type} = \mathit{START} \\
\mathit{timer\_update!.source} = \mathit{REX\_SERVER} \\
\mathit{timer\_update!.period} = \mathit{CHECK} \\
\hline
\end{array}
$$

15.4 Receipt of other $CAST$ request fragments

$$
\begin{array}{l}
\textit{ServerReceiveNonFinalFragmentOfCast} \\
\hline
\Xi \mathit{RexClientInfo} \\
\Delta \mathit{RexServerInfo} \\
\mathit{message}? : \mathit{DataMessage} \\
\mathit{timer\_update}! : \mathit{TimerUpdate} \\
\hline
\mathit{server\_state} = \mathit{CAST\_RECEIVING} \\
\mathit{message}?.\mathit{type} = \mathit{CAST\_FRAG} \\
\mathit{message}?.\mathit{dest} = \mathit{REX\_SERVER} \\
\mathit{message}?.\mathit{seqno} = \mathit{server\_seqno} \\
\forall\, m : \mathit{DataMessage} \bullet \\
\qquad m \in \mathit{server\_buffer} \Rightarrow m.\mathit{offset} \neq \mathit{message}?.\mathit{offset} \\
\exists\, i : 1 \mathrel{..} \mathit{max\_offset} \bullet i \notin \mathit{fragments\_received} \\
\mathit{server\_buffer}' = \mathit{server\_buffer} \cup \{\mathit{message}?\} \\
\mathit{timer\_update}!.\mathit{type} = \mathit{START} \\
\mathit{timer\_update}!.\mathit{source} = \mathit{REX\_SERVER} \\
\mathit{timer\_update}!.\mathit{period} = \mathit{CHECK} \\
\mathit{server\_state}' = \mathit{server\_state} \\
\mathit{server\_seqno}' = \mathit{server\_seqno} \\
\textbf{where} \\
\qquad \mathit{max\_offset} = \mathit{message}?.\mathit{total\_size} \\
\qquad \mathit{fragments\_received} = \\
\qquad\qquad \{ \quad i : \mathit{FragIndex} \mid \\
\qquad\qquad\qquad \exists_1\, m : \mathit{DataMessage} \bullet \\
\qquad\qquad\qquad\qquad m \in \mathit{server\_buffer}' \wedge i = m.\mathit{offset} \\
\qquad\qquad \}
\end{array}
$$

$$
\begin{array}{l}
\textit{ServerReceiveFinalFragmentOfCast} \\
\hline
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
request! : Request \\
\hline
server\_state = CAST\_RECEIVING \\
message?.type = CAST\_FRAG \\
message?.dest = REX\_SERVER \\
message?.seqno = server\_seqno \\
\forall\, m : DataMessage \bullet \\
\quad m \in server\_buffer \Rightarrow m.offset \neq message?.offset \\
\forall\, i : 1 \,..\, max\_offset \bullet i \in fragments\_received \\
server\_buffer' = server\_buffer \cup \{message?\} \\
server\_state' = IDLE \\
timer\_update!.type = STOP \\
timer\_update!.source = REX\_SERVER \\
request! = build(server\_buffer', SERVER) \\
server\_seqno' = server\_seqno \\
\textbf{where} \\
\quad max\_offset = message?.total\_size \\
\quad fragments\_received = \\
\qquad \{ \quad i : FragIndex \mid \\
\qquad\qquad \exists_1\, m : DataMessage \bullet \\
\qquad\qquad\quad m \in server\_buffer' \land i = m.offset \\
\qquad \} \\
\hline
\end{array}
$$

This differs from the corresponding situation for interrogation interactions in that no reponse will have to be sent back to the REX-client entity. Therefore the REX-server entity changes back to the *IDLE* state on submitting the complete *CAST* request to the server entity.

15.5 Receipt of overtaken *CAST* request fragments

We have seen in section 13.3 that non-fragmented *CAST* request messages may be overtaken by another request message. This also applies to fragment messages of *CAST* requests. We make it clear below that these will be ignored when they eventually arrive at the REX-server entity.

$$
\begin{array}{l}
\textit{ServerReceiveOvertakenFragmentOfCast} \\
\hline
\Xi SessionInfo \\
message? : DataMessage \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CAST\_FRAG \\
message?.dest = REX\_SERVER \\
message?.seqno < server\_seqno \\
timer\_update!.type = NO\_UPDATE \\
\hline
\end{array}
$$

15.6 Delays when receiving fragments of *CAST* request

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerCastReceivingTimeout}} \\
\hline
\Xi SessionInfo \\
timeout? : Timeout \\
message! : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
timeout?.dest = REX\_SERVER \\
server\_state = CAST\_RECEIVING \\
message!.type = FRAG\_NACK \\
message!.dest = REX\_CLIENT \\
message!.seqno = server\_seqno \\
message!.frags\_map = \\
\quad \{ \quad i : FragIndex \mid \\
\qquad\quad \exists_1\, m : DataMessage \bullet \\
\qquad\qquad m \in server\_buffer \land i = m.offset \\
\quad \} \\
timer\_update!.type = START \\
timer\_update!.source = REX\_SERVER \\
timer\_update!.period = CHECK \\
\hline
\end{array}
$$

The protocol requires the REX-server entity to seek corrective action (by sending a $FRAG\_NACK$ message) when an expected fragment is not received within a time $CHECK$.

However, REX does not require the REX-client entity (which is sending a fragmented $CAST$ request) to take any explicit action on receiving a $FRAG\_NACK$ message. Therefore, we have not given a schema to represent this.

16 Final details on fragmentation

The receipt of a repeat of the first fragment of a response, eliciting acknowledgment for the response:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClientReceiveRepeatReplyFragment}} \\
\hline
\Xi SessionInfo \\
message? : DataMessage \\
message! : ControlMessage \\
\hline
message?.type = REPLY\_FRAG \\
message?.dest = REX\_CLIENT \\
message?.seqno = client\_seqno \\
client\_state = IDLE \\
message?.offset = 1 \\
message!.type = REPLY\_ACK \\
message!.dest = REX\_SERVER \\
message!.seqno = message?.seqno \\
\hline
\end{array}
$$

The receipt of a repeat of the first fragment of a $CALL$ request eliciting acknowledgment for the request:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerReceiveRepeatCallFragment}} \\
\Xi SessionInfo \\
message? : DataMessage \\
message! : ControlMessage \\
\hline
message?.type = CALL\_FRAG \\
message?.dest = REX\_SERVER \\
message?.seqno = server\_seqno \\
server\_state = ASKED \\
message?.offset = 1 \\
message!.type = CALL\_ACK \\
message!.dest = REX\_CLIENT \\
message!.seqno = message?.seqno \\
\hline
\end{array}
$$

We now extend the specifications given for the receipt of explicit acknowledgments given in section 12.4 and in section 12.6, to take into account that the request or response may be fragmented:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServerReceiveReplyAckMessage\_}\mathcal{B}} \\
\Xi RexClientInfo \\
\Delta RexServerInfo \\
message? : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
message?.type = REPLY\_ACK \\
message?.dest = REX\_SERVER \\
message?.seqno = server\_seqno \\
server\_state = SENDING \\
server\_state' = IDLE \\
timer\_update!.type = STOP \\
timer\_update!.source = REX\_SERVER \\
server\_seqno' = server\_seqno \\
server\_buffer' = server\_buffer \\
response\_frags\_map' = response\_frags\_map \\
\hline
\end{array}
$$

$$
\begin{array}{l}
ServerReceiveReplyAckMessage \mathrel{\widehat{=}} \\
\qquad\qquad ServerReceiveReplyAckMessage\_\mathcal{A} \\
\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad ServerReceiveReplyAckMessage\_\mathcal{B}
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClientReceiveCallAckMessage\_}\mathcal{D}} \\
\Xi RexServerInfo \\
\Delta RexClientInfo \\
message? : ControlMessage \\
timer\_update! : TimerUpdate \\
\hline
message?.type = CALL\_ACK \\
message?.dest = REX\_CLIENT \\
message?.seqno = client\_seqno \\
client\_state = CALL\_SENDING \\
client\_state' = PROBING \\
timer\_update!.type = START \\
timer\_update!.source = REX\_CLIENT \\
timer\_update!.period = PROBE \\
\hline
\end{array}
$$

$$ClientReceiveCallAckMessage \mathrel{\widehat=} \begin{array}{c} ClientReceiveCallAckMessage\_\mathcal{C} \\ \vee \\ ClientReceiveCallAckMessage\_\mathcal{D} \end{array}$$

17 Schema Decompostion of REX

$$RequestEvent \mathrel{\widehat=} \begin{array}{c} ClientCallRequest \\ \vee \\ ClientCastRequest \\ \vee \\ ClientLargeCallRequest \\ \vee \\ ClientLargeCastRequest \end{array}$$

$$ReplyEvent \mathrel{\widehat=} \begin{array}{c} ServerCallReply \\ \vee \\ ServerLargeCallReply \end{array}$$

$$ReceptionOfControlMessage \mathrel{\widehat=} \begin{array}{c} ClientReceiveCallAckMessage \\ \vee \\ ClientReceiveFragNackMessageForCall \\ \vee \\ ServerReceiveReplyAckMessage \\ \vee \\ ServerReceiveProbeMessage \\ \vee \\ ServerReceiveFragNackMessage \end{array}$$

$$ReceptionOfDataMessage \mathrel{\widehat=} \begin{array}{c} ClientReceiveReplyMessage \\ \vee \\ ClientReceiveRepeatReplyMessage \\ \vee \\ ClientReceiveFirstFragmentOfReply \\ \vee \\ ClientReceiveNonFinalFragmentOfReply \\ \vee \\ ClientReceiveFinalFragmentOfReply \\ \vee \\ ClientReceiveRepeatReplyFragment \\ \vee \\ ServerReceiveCallMessage \\ \vee \\ ServerReceiveRepeatReplyMessage \\ \vee \\ ServerReceiveCastMessage \\ \vee \\ ServerReceiveFirstFragmentOfCall \end{array}$$

$$\lor$$
$$ServerReceiveNonFinalFragmentOfCall$$
$$\lor$$
$$ServerReceiveFinalFragmentOfCall$$
$$\lor$$
$$ServerReceiveFirstFragmentOfCast$$
$$\lor$$
$$ServerReceiveNonFinalFragmentOfCast$$
$$\lor$$
$$ServerReceiveFinalFragmentOfCast$$
$$\lor$$
$$ServerReceiveOvertakenFragmentOfCast$$
$$\lor$$
$$ServerReceiveRepeatCallFragment$$

$TimeoutEvent \mathrel{\widehat=}$

$$ClientCallingTimeout$$
$$\lor$$
$$ClientProbeTimeout$$
$$\lor$$
$$ClientCallSendingRemainingFragmentsTimeout$$
$$\lor$$
$$ClientCallSendingNoReplyTimeout$$
$$\lor$$
$$ClientReplyReceivingTimeout$$
$$\lor$$
$$ClientCastSendingRemainingFragmentsTimeout$$
$$\lor$$
$$ServerReplyingTimeout$$
$$\lor$$
$$ServerCallReceivingTimeout$$
$$\lor$$
$$ServerReplySendingRemainingFragmentsTimeout$$
$$\lor$$
$$ServerReplySendingNoAckTimeout$$
$$\lor$$
$$ServerCastReceivingTimeout$$

$RexEvent \mathrel{\widehat=}$

$$ReceptionOfControlMessage$$
$$\lor$$
$$ReceptionOfDataMessage$$
$$\lor$$
$$TimeoutEvent$$
$$\lor$$
$$RequestEvent$$
$$\lor$$
$$ReplyEvent$$

18 Conclusion

A lot of work on formal specifications is in the context of systems design. The task involved is usually that of identifying the desirable properties of the required system and expressing these in an abstract form, from which an implementation can be derived.

By contrast, preparing this formal specification involved taking the implementation-oriented details available to us, such as those on data-structures and algorithms used, and working towards a functional description at a higher level of abstraction.

The requirement was that of providing a precise, clear and unambiguous view of the basic functionality of the REX protocol. We are confident of having fulfilled this requirement and feel that the Z notation was the important tool that enabled us to do so.

Working with a formal notation systemized the way in which we derived the specification. It also made us study the protocol in more detail and thus, a more precise description was produced.

Furthermore, use of the Z notation did not constrain us in the way the information was ordered and structured. We were able to introduce aspects of the protocol in a logical manner and to do this in tutorial form.

And finally of course, the possibility of misinterpretation, which is a great pitfall of natural-language specifications, was also avoided by adopting the formal notation.

19 Acknowledgments

The work on this project was carried out in liaison with Alastair Tocher and Ed Oskiewicz of the ANSA project team, and Ben Potter, formerly of STC Technology. Their help and support is greatly appreciated.

References

[ANSA 90] *Testbench Implementation Manual* (Cambridge: Advanced Networked Systems Architecture, 1990).

[Birr 84] A.D.Birrel, B.J.Nivat 'Implementing remote procedure calls' *ACM Transactions on Computer Systems* **2, 1** (February 1984) 39-59.

[Grim 89a] A.Grimley *Formal Specification Using Z* (UKC, July 89)

[Grim 89b] A.Grimley *Formal Specification of Mail System* (UKC, April 89)

[Hayes 87] I.J.Hayes (Ed.) *Specification Case Studies* (Prentice-Hall International, 1987).

[Herb 87] A.J.Herbert, J.Monk (Editors) *The ANSA Reference Manual: release 00:03 (draft)* (Cambridge: Advanced Networked Systems Architecture, 1987).

[Lin 90] P.F.Linington *Networked Systems* (Lecture Notes - UKC, 1990)

[Mull 89] S.J.Mullender (Ed.) *Distributed Systems* (Addison-Wesley, 1989).

[Pott 91] B.Potter, J.Sinclair, D.Till *An Introduction to Formal Specification and Z* (Prentice-Hall International, 1991).

[Shriv 82] S.K.Shrivastava, F.Panzieri 'The design of a reliable remote procedure mechanism' *IEEE Transactions on Computers* **C-31, 7** (July 1982) 692-697.

[Spec 82] A.Z.Spector 'Performing remote operations efficiently on a local computer network' *Commun. ACM* **25, 4** (April 1982) 246-260.

[Spiv 88] J.M.Spivey *The Z Notation: a Reference Manual* (Cambridge University Press, 1988).

[Till 87] D.Till, B.Potter *The Specification in Z of Gateway Functions within a Communications Network* (October 1987).

[Wood 89] J.C.P.Woodcock 'Structuring specifications in Z' *Software Engineering Journal* (January 1989) 51-66.

A Appendix - Auxiliary functions

In this document we have assumed existence of functions *fragment* and *build*.

$$fragment : DataType \times SeqNo \times MessageType \rightarrow \mathbb{P}\, DataMessage$$

The *fragment* function takes some data and breaks it down into a number of fragment messages.

The function must also be given the values to be assigned to the *seqno* and *type* components of the fragment messages.

If the *type* component is $CALL\_FRAG$ or $CAST\_FRAG$ the *dest* component of the messges will be set to REX_SERVER. If the *type* is $REPLY\_FRAG$ the *dest* component will be set to REX_CLIENT.

The *offset* component of each fragment message is generated automatically by the function so as to reflect the offset of the data contained within that fragment. So, an offset of 1 will be generated for the first fragment, 2 for the second, etc..

$$build : \mathbb{P}\, DataMessage \times NonProtocolEntity \rightarrow \mathbb{P}(Request \cup Response)$$

The *build* function takes a number of fragment messages and returns a *Request* if the *NonProtocolEntity* specified is *CLIENT*, or a *Response* if the *NonProtocolEntity* specified is *SERVER*.

The *data* component of the request or response returned will be the concatenation of all the data contained in the fragment messages, in the order of their *offset*s.