Inter-Office Memorandum

To	Communication Protocols	Date	July 2, 1978
From	Ed Taft	Location	Palo Alto
Subject	State Machine for Rendezvous/Termination Protocol	Organization	PARC/CSL

XEROX

Filed on: [Maxc1]<Pup>RTPStates.press

This is a revision of *Pup Connection State Diagram*, dated October 4, 1975. Only minor changes have been made.

The attached diagram purports to illustrate Tenex's management of Pup connections according to the Rendezvous/Termination Protocol. While some aspects of this diagram are peculiar to the Tenex implementation, we are presenting it as a model for implementation of the protocol on other machines. Familiarity with *Pup Specifications* is assumed, and *Implementation of Pup in Tenex* might be of some help.

Pup connections are controlled by a Finite State Machine (FSM) consisting of states, events, actions, and transitions. At any given moment, a port is in some state, indicated by one of the labelled circles. Leading out of each state are one or more arcs, each labelled with one or more events and (optionally) an action. For example, extending down and to the right from the *Closed* state is an arc leading to the *RFC Out* state and labelled "OPENF$_C$, Send RFC". This means that if the port is in the *Closed* state and event *OPENF$_C$* occurs, the action *Send RFC* is performed and the port's state is changed to *RFC Out*.

If an event occurs and no path out of the current state is labelled with that event, we assume that a protocol violation or similar error has occurred. Such events give rise to no action and cause the state to loop back to itself.

Before going into detail, we should note that Tenex implements only a special case of the Rendezvous Protocol, namely that in which the *local* Rendezvous Port and Connection Port are the same. This topic is elaborated upon in the memo *Implementation of Pup in Tenex*.

Events

There are three classes of events that potentially cause changes to a port's state. Events may be initiated by operations performed on the port by the local process (OPENF, CLOSF), by receipt of a Rendezvous/Termination Protocol Pup from the appropriate foreign port, or by a timeout. These events will now be described in some detail.

OPENF is the name of the Tenex JSYS used to open a file. In the case of an OPENF applied to device PUP:, the OPENF may be executed in one of three modes, corresponding to the events *OPENF$_C$*, *OPENF$_L$*, and *OPENF$_N$*.

The *OPENF$_C$* operation is used when the local process desires to actively initiate a connection with a specific foreign port, while the *OPENF*$_L$ operation puts the port in a passive, *Listening* state for the purpose of accepting incoming requests from other ports. Note that a successful rendezvous requires that one party "listen" while the other "connects".

The *OPENF*$_N$ operation is a means by which a BSP port may be placed immediately in the *Open* state without Tenex performing a rendezvous. Obviously, it is assumed that the process has already performed a rendezvous by some other means. This is the mechanism by means of which servers may implement separate Rendezvous and Connection ports.

CLOSF$_N$ is the normal means by which the local process requests that a connection be terminated, while *CLOSF*$_T$ requests an abnormal termination. In the Tenex implementation, either operation may be initiated by the CLOSF JSYS, which generates a *CLOSF*$_N$ event normally but a *CLOSF*$_T$ event if a timeout error has occurred (i.e. no activity has taken place for a long time despite repeated probing of the foreign port).

The events *RFC rec'd*, *End rec'd*, *End Reply rec'd*, and *Abort rec'd* are triggered by receipt of Pups of those respective types. An incoming Pup of any of the latter three types must (a) have a Destination Port exactly matching the local port, (b) have a Source Port exactly matching the foreign port to which the connection has been established, and (c) have a Pup ID equal to the Connection ID (established by the initiating RFC). If these conditions do not hold, the Pup is discarded without generating an event. In the case of an RFC, the conditions checked depend on the current state, and the action performed depends on parameters obtained from the incoming RFC. This will be described later on.

Whenever a port enters a state (even if it is the state it was in before), a timer is initialized to a fairly short interval (on the order of a second unless the state being entered is *Dally*, in which case the interval should be somewhat longer, say 5 to 10 seconds). If this interval elapses, a *Timeout* event is generated. The *Timeout* event may cause Pups to be retransmitted (*RFC Out*, *End Out*), and is also used to force termination of the *Dally* state. In other states, the *Timeout* event is ignored. The timer used to generate this event is different from the one used to detect long-term inactivity. That one is typically on the order of several minutes; its expiration does not itself generate an event but rather signals an error to the process, which may then choose to perform a *CLOSF*$_T$ to abort the connection.

Actions

The actions *Send RFC*$_1$, *Send RFC*$_2$, and *Send RFC*$_3$ all cause an RFC Pup to be transmitted; however, the RFC's parameters are set up in a different manner for the three cases, and there are some differing auxiliary actions performed as well.

The action *Send RFC*$_1$ is performed when the local port is taking the initiative in establishing the connection. In this case, the RFC's Source Port and Connection Port fields are set to the local port address; the Destination Port is set to the desired foreign Rendezvous Port, and the Pup ID (i.e. the Connection ID) is chosen locally by some suitable means (e.g. reading a real-time clock). Note that the same Pup ID should be used for any retransmission of the RFC as was used for the initial RFC.

Send RFC$_2$ is the action performed when the local port is in the *Listening* state and an RFC is received which passes through whatever address filtering has been specified (generally it is completely wildcard). In this case, the Connection ID and foreign Connection Port are gotten from the incoming RFC. The answering RFC is then generated by exchanging the Pup Source and Destination and supplying the local port address as the Connection Port.

Send RFC$_3$ is the action performed when an RFC is received while the state is *Open* or *End Out* (meaning that we believe the connection has been fully established but the foreign process has not yet seen our answering RFC). In this case, the incoming RFC is accepted only if all the following conditions hold: (a) the port was previously in the *Listening* state; (b) the Connection Port is exactly the same as the foreign Connection Port previously remembered (in action *Send RFC*$_2$); and

(c) the Connection ID is the same as the ID already established for the connection. If these conditions are satisfied, the answering RFC is generated as in the action *Send RFC$_2$*. Requirement (a) is important, since if we were to reply to an RFC after having been the initiator of a connection, receipt of a delayed duplicate RFC could result in both parties bouncing the RFC back and forth between them indefinitely. This memory of "having been listening" should really be represented by other states in the FSM (duplicates of *Open* and *End Out*), but we have not done so in the interest of avoiding excessive clutter.

The action *Open Connection* is performed only for a port in the *RFC Out* state when an answering RFC is received. The RFC should not be accepted unless the Source and Destination Port fields (reversed) and the Pup ID match the corresponding fields of the RFC we sent previously (by action *Send RFC$_1$*). The only operation performed by this action is to remember the foreign Connection Port contained in the incoming RFC.

The actions *Send End*, *Send End Reply* and *Send Abort* are self-explanatory. Each of these is constructed using the local and foreign Connection Ports and the Connection ID for the Source Port, Destination Port, and Pup ID, respectively.

Miscellaneous

The FSM just described deals only with the Rendezvous/Termination Protocol and not with the actual transfer of data by means of the Byte Stream Protocol (or, possibly, some alternative protocol). However, this protocol interacts with BSP in two important ways.

First, BSP operations should be performed only when the connection state is *Open*, *End In*, or *End Out*. Furthermore, the local process should not be permitted to generate more output when the state is *End Out*, and should be given an end-of-file indication upon exhausting the input stream and finding the state to be *End In*. Incoming Acknowledgments should be ignored and no Datas or ADatas should be transmitted (or even *retransmitted*) when the state is *End Out*, and incoming Datas and ADatas should be ignored and no Acknowledgments transmitted when *End In*.

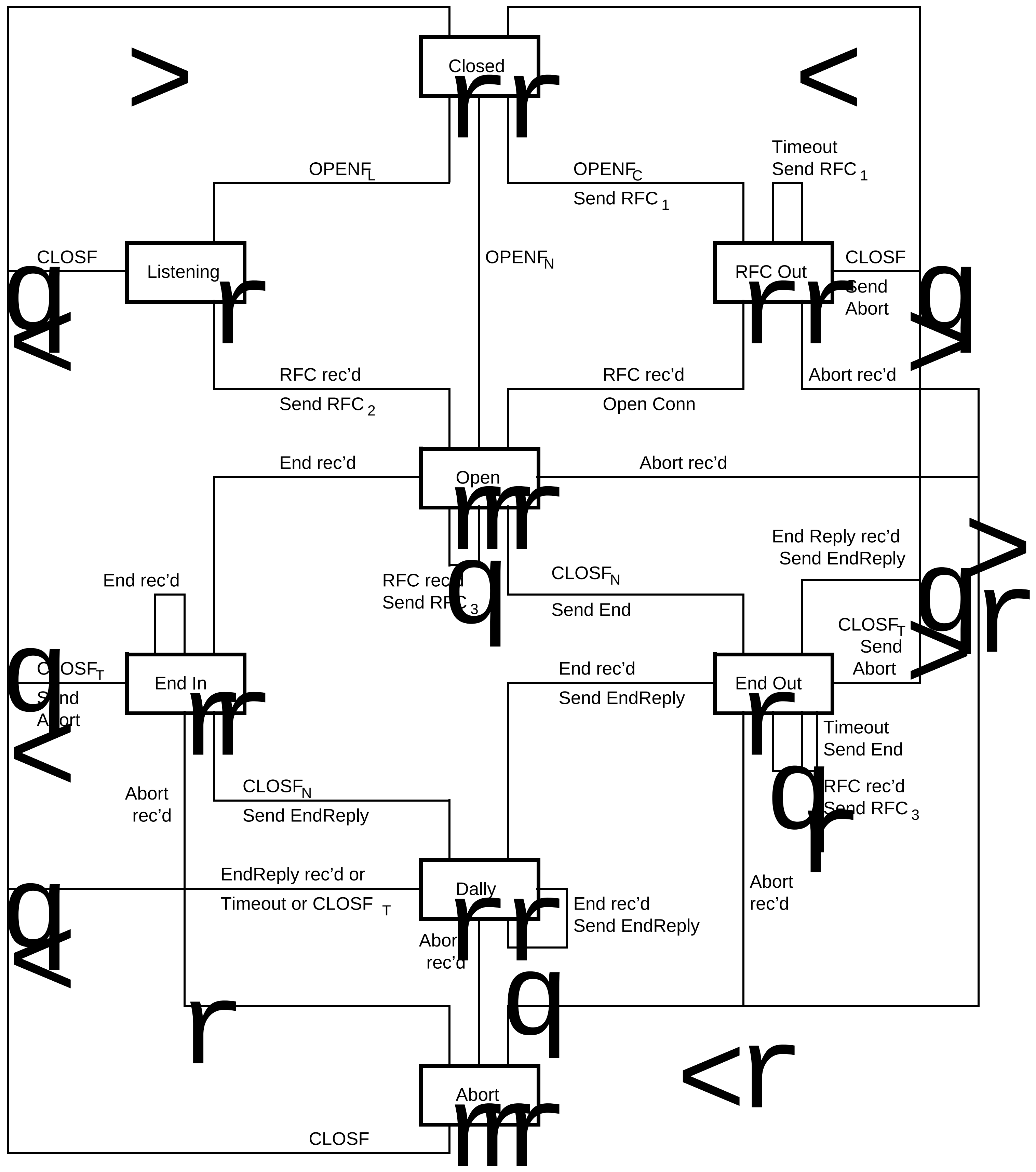
Second, the CLOSF$_N$ event must not be generated until all outgoing data has been successfully transmitted and acknowledged. This ensures clean termination of the outgoing byte stream before sending either an End or an End Reply, as required by the protocol.

The *Abort* state is included in the FSM purely as a means by which the local process may discover that the connection has been aborted by the foreign process. In most other respects, this state is equivalent to *Closed*, and the transition from *Abort* to *Closed* generates no action.

Summary

The following table enumerates all states, events, and actions in the FSM.

States	*Events*	*Actions*
Closed	OPENF$_C$	Send RFC$_1$
Listening	OPENF$_L$	Send RFC$_2$
RFC Out	OPENF$_N$	Send RFC$_3$
Open	CLOSF$_N$	Open Connection
End In	CLOSF$_T$	Send End
End Out	RFC rec'd	Send End Reply
Dally	End rec'd	Send Abort
Abort	End Reply rec'd	
	Abort rec'd	
	Timeout	



Rendezvous/Termination State Diagram