Jumpstart just-in-time signaling protocol: a formal description using extended finite state machines

Abdul Halim Zaim
Ilia Baldine
Mark Cassada
MCNC
3021 Cornwallis Road
P.O. Box 12889
Research Triangle Park, North Carolina 27709
E-mail: ibaldin@anr.mcnc.org

George N. Rouskas
Harry G. Perros
North Carolina State University
Department of Computer Science
Raleigh, North Carolina 27695

Daniel Stevenson
MCNC
3021 Cornwallis Road
P.O. Box 12889
Research Triangle Park, North Carolina 27709

Abstract. We present a formal protocol description for a just-in-time (JIT) signaling scheme running over a core dense wavelength division multiplexing (DWDM) network that utilizes optical burst switches (OBSs). We apply an eight-tuple extended finite state machine (EFSM) model to formally specify the protocol. Using the EFSM model, we define the communication between a source client node and a destination client node through an ingress and one or multiple intermediate switches. We work on on-the-fly and persistent unicast connections. The communication between the EFSMs is handled through messages. *© 2003 Society of Photo-Optical Instrumentation Engineers.* [DOI: 10.1117/1.1533795]

Subject terms: extended finite state machines; optical burst switches; just-in-time signaling protocol; formal protocol description.

Paper 020195 received May 15, 2002; revised manuscript received Jul. 16, 2002; accepted for publication Aug. 6, 2002.

1 Introduction

Recently, a great change in the protocol design has been observed in telecommunication industry. Instead of the traditional design cycle, which includes a three-step process consisting of a high-level design, a low-level design and coding and testing, a more formal design approach has been developed. The formal design approach uses methods that help the designer verify the correctness of the design decisions as they are made. For more information on formal design approaches, refer to Refs. 1–5.

In Ref. 6, Gunawan et al. classified formal description techniques into three main categories: state transition models (STMs), programming language models (PLMs), and hybrid models (HMs). After giving the advantages of different description techniques on each category, the authors mentioned that HMs are the most powerful techniques because they combine the ease and understandable structure of STMs with the elasticity and power of PLMs. Extended finite state machines (EFSMs) are also HMs.

Ordinary finite state machine (FSM) representation is not powerful enough to model in a succinct way the Jumpstart just-in-time (JIT) signaling protocol, because the protocol specifications include variables, timers, and operations based on these values (for more information about the Jumpstart protocol refer to Ref. 7). Therefore, we define an EFSM model with the addition of some variables. For further information on EFSMs, the interested readers can see Refs. 2, 4, 5 and 8. An EFSM approach is fully expressive and particularly useful as a means of describing a communication protocols. EFSM-based techniques can be applied in telecommunications more easily than most other approaches and are better suited to assist in the follow-up implementations. Therefore, in this paper we use an EFSM-based description model.

The Jumpstart* signaling protocol was first introduced in Ref. 7. The signaling architecture is based on wavelength routing and burst switching. Signaling is JIT, indicating that signaling messages travel slightly ahead of the data they describe. Signaling is out of band, with signaling packets undergoing electro-optical conversion at every hop. Data are opaque to network entities and travel through the network in bursts of varying durations, each burst preceded by its own signaling message.

Optical burst switching (OBS) is a promising direction of research and development in wavelength-routed core wavelength-division multiplexing (WDM) networks. Coupled with out-of-band signaling it promises to deliver a transparent all-optical architecture, capable of transporting digital and analog data regardless of format. JIT signaling approaches to OBS have been previously studied in the literature.[9–13] These approaches are characterized by the fact that the signaling messages are sent just ahead of the data to inform the intermediate switches. The common thread is the elimination of the round-trip waiting time before the information is transmitted (the so-called tell-and-go approach): the switching elements inside the switches are configured for the incoming burst as soon as the first signaling message announcing the burst is re-

*The Jumpstart project is in its third phase, which consists of implementation. At the time of this paper, the testbed was almost finished.

 © 2003 Society of Photo-Optical Instrumentation Engineers

Table 1 Signaling protocol functions.

Session declaration	Announce the connection to the network
Path setup	Configure resources needed to set up an all-optical path from source to destination
Data transmission	Inform intermediate switches burst arrival time and length
State maintenance	Keep up the necessary state information to maintain the connection
Path teardown	Release resources taken up to maintain the lightpath for the connection

ceived. The variations on the signaling schemes mainly differ in how soon before the burst arrival and how soon after its departure the switching elements are made available to route other bursts through use of the combination of signaling messages and timers.

The organization of the paper is as follows. Section 2 briefly explains the Jumpstart signaling protocol. The EFSM model is given in great detail in Sec. 3. Section 4 shows the channel architecture for message communication among different EFSMs. Section 5 provides the formal specification of the Jumpstart protocol, showing all state diagrams and explaining the state machines. Section 6 concludes our paper.

2 Jumpstart JIT Signaling Protocol

Jumpstart signaling uses a link-unique identifier (or label) for each message, which upon emergence on the other end of the link can be cached and mapped to a new identifier or label on the exit link. The first message in a signaling flow (SESSION DECLARATION or SETUP) serves the purpose of setting up a label-switched path, which all further messages in forward and reverse direction follow. That is, this on-the-fly setup of a label switched path is the main difference between multiprotocal label switching (MPLS) or asynchronous transfer mode (ATM) and our approach. Another difference worth noting is that in MPLS, labels are distributed upstream, in the reverse direction of the path prior to path being used. In our case we need to setup the label-switched path in the forward direction. In addition, we must setup the reverse path at the same time.

The basic signaling protocol for a JIT OBS network described in this section addresses only the connection setup procedures.

Depending on the type of the connection being set up, the signaling protocol may need to perform several functions, all described in Table 1. These phases can be accomplished by the signaling protocol in a different fashion, depending on the assumptions made about the network: the reliability of individual links, scheduling capabilities of the switches, and other factors.

In the Jumpstart network we propose to use two types of connection setups:

1. *Explicit setup and explicit teardown.* Each burst is preceded by its own Setup message and followed by its own Release message (which enables the intermediate switches to close the optical cross-connects or use them for other connections).
2. *Explicit setup and estimated teardown signaling schemes.* This is similar to explicit teardown, with the exception that the source notifies the network of the duration of its burst and the network uses this estimate to close the crossconnects. This way no Release message is needed.

An explanation of different signaling schemes can be found in Ref. 7. We define a unified signaling scheme that will enable both approaches to be used at the discretion of the caller.

2.1 *Connection Phases*

Each connection in our OBS network goes through a number of well-defined phases, as described in Ref. 7. This paper concentrates on a unicast case. Unicast connections have all of these phases; however, some of them are collapsed into a single step. For example, for short bursts the Setup message serves to

1. announce the session to the network (Session Declaration)
2. set up the path of the session (Path Setup)
3. announce the arrival of the burst (Data Transmission)

This way the Setup message combines the three phases, which are followed by either an explicit or implicit session Release. Path teardown phase may be explicit (if explicit teardown with a Release message is used) or implicit (if estimated teardown with a Timeout message is used). These simple connections lack State Maintenance phase due to their short-lived nature. This phase is intended for long-lived bursts that require the "keep-alive" message and persistent-path connections.

2.2 *Persistent Path Connection*

In the previous section we alluded to the fact that the Setup message combines the announcement, path setup, and data transmission phases. Although this approach works well for short-lived bursty connections, we can envision the need to transport a number of bursts over the same path to reduce jitter. For that purpose, we must separate the path setup phase from the data transmission phase. This is where so-called persistent path connections become necessary. In this type of connection, a path is established for a session, which may consist of any number of short bursts following the same path.

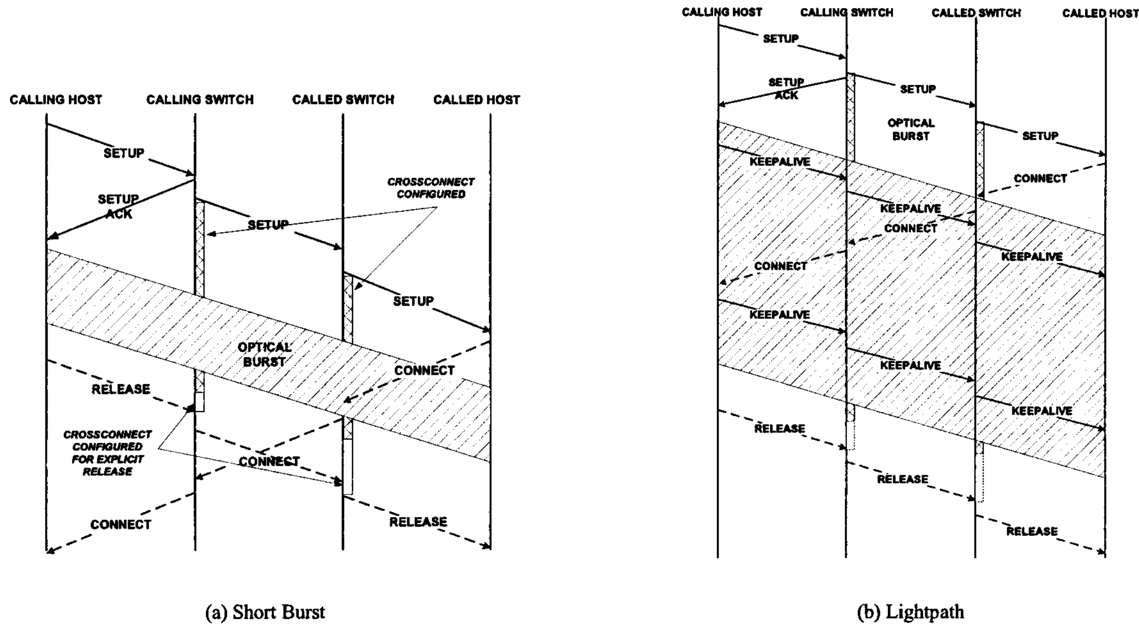


Fig. 1 Supported connection types: (a) short burst and (b) lightpath.

2.3 *Unicast Message Flows*

The next two subsections present the flow of signaling messages for unicast connections for both on-the-fly short bursts and persistent path connections.

2.3.1 *On-the-fly unicast signaling flows*

We begin by describing the signaling flows for on-the-fly routed unicast connections. These connections combine the Session Declaration, Path Setup, and Data Transmission phases into a single Setup message.

The message flows for short bursts and lightpaths are presented in Fig. 1. The presence of the Release message at the end of each connection is dictated by the type of the connection (explicit versus timed teardown).

Regardless of the type of the connection, it is initiated with a Setup message sent by the originator of the burst to its ingress switch. The ingress switch consults with delay estimation mechanism based on the destination address and returns the updated delay information to the originator by using a Setup Ack message, at the same time acknowledging the receipt of the Setup message by the network. The Setup Ack message also informs the originating node which channel/wavelength to use when sending the data burst.

The originator waits the required balance of time left based on its knowledge of the round-trip time to the ingress switch, and then sends the burst on the indicated wavelength. The Setup message at the same time is traveling across the network, informing the switches on the path of the burst arrival. If no blocking occurs on the path, the Setup message eventually reaches the destination node, which then receives the incoming burst shortly thereafter.

Upon the receipt of the Setup message, the destination node may choose to send a Connect message acknowledging the successful connection (indeed, the receipt of the Setup by the destination only guarantees that the connection has been established; it does not guarantee its successful completion, since a connection may be preempted somewhere along the path by a higher priority connection).

For long-lived bursts, the Keepalive message maintains the state of the connection, preventing it from timing out. Especially for explicit teardown, where a connection is not closed until a Release message is received, Keepalive message is used to notify the aliveness of the source. Otherwise, in case that the source is dead, if there is no Keepalive mechanism, the connection will wait for a Release forever wasting the limited crossconnect resources. However, with Keepalive mechanism, if the source does not send a Keepalive message during a specified time, a timeout occurs and the connection is closed.

One message type not already mentioned is sent if any type of failure is detected during setup or maintenance phase of the connection. This message is called Failure. It is sent to the originator of the connection and it carries with it the cause of failure, including blocking, preemption by a higher-priority connection, lack of route to host, refusal by destination, etc.

2.3.2 *Persistent path unicast signaling flow*

The only difference, from the signaling point of view, between the persistent path versus on-the-fly route unicast connections is that the Session Declaration and Path Setup phases are separated from the Data Transmission phase. While the contents of the Setup, Setup_Ack, Connect, and Release messages is slightly different for persistent path connections (they only need to carry information related to the Data Transmission phase, as opposed to the on-the-fly routed connections, for which these messages need to carry information related to the Session Declaration and Path Setup phases), their flows remain the same in both types of connections. However in persistent path connections the data transmission flows (Setup→Setup_Ack→Connect →Release) are enveloped by the Session Declaration and

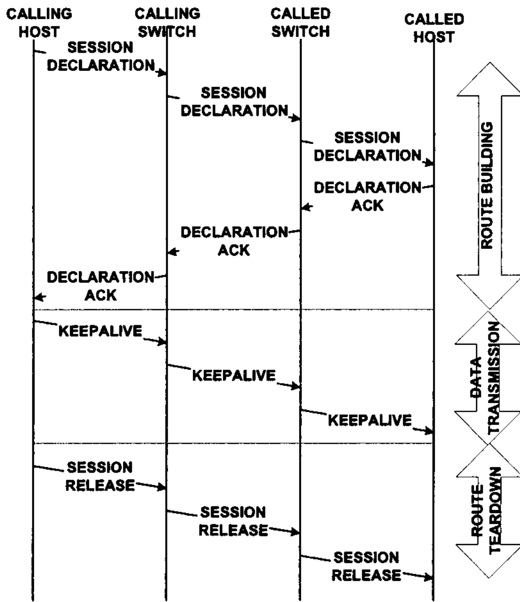
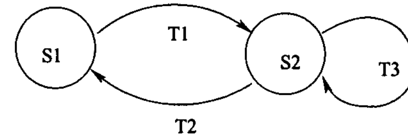


Fig. 2 Persistent path setup.



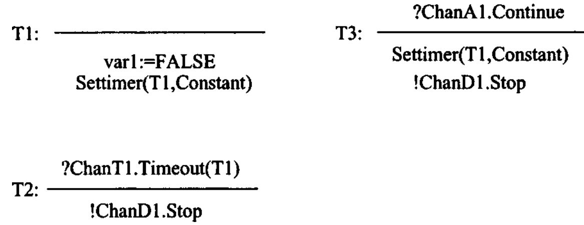


Fig. 3 Example EFSM model.

Session Release flows, which are comprised of Session Declaration, Declaration Ack, and Session Release. Figure 2 demonstrates these last two flows. For the flows describing the data transmission phase refer to the previous section.

With this type of connection, a Session Declaration message travels between the source and the destination first and sets up a persistent path, which intermediate nodes can refer to by the label contained in the message. What follows during the data transmission phase is any number of data transmission flows as described in the previous section. Once all the Data Transmission phases have been concluded, the originator of the connection will use the Session Release message to remove the persistent path from the network. Also, the network may choose to terminate a certain session. It can achieve this by sending a Session Release message to the source and the destination.

Note that the optical switching elements of the intermediate switches are not permanently configured for the path between the arrivals of the Session Declaration and the Session Release messages. Rather, their state, necessary to route the data to the destination, is cached when the Session Declaration message arrives, and is used to reset the configuration of the crossconnect every time a new Setup message announces the beginning of the new Data Transmission phase, guaranteeing that the bursts belonging to this connection travel on the same optical path.

3 Extended Finite State Model

In this model, each EFSM can be formally represented as a eight-tuple $(\Sigma, S, s, V, E, T, A, \delta)$, where

- Σ = set of messages that can be sent or received
- S = set of states
- s = initial state
- V = set of variables
- E = set of predicates that operate on variables
- T = set of timers
- A = set of actions that operate on variables
- δ = set of state transition functions, where each state transition function is formally represented as follows: $S*\Sigma*E(V)*T \rightarrow \Sigma*A(V)*S$.

There are two types of transitions: spontaneous and "when" transitions. A spontaneous transition does not have an input event on its condition part. A "when" transition, on the other hand, includes an input event satisfying the T condition. A transition is shown $S_1 \rightarrow S_2$. This means there is a transition T at state S_1, and it goes to state S_2, where T is an outgoing transition, S_1 is the head state, and S_2 is the tail state.

A transition consists of two parts: a condition part and an action part. The condition part have an input event and a predicate (Boolean expression). An action may be an output event or a statement operating on variables. A transition executed when an input event is available, and a predicate is true. Once a transition is triggered, the action part is executed. An example of an EFSM is shown in Fig. 3. In the figure, ?Chan. *m* shows an input message from given channel carrying the message *m*, and !Chan. *m* shows an output message to the indicated channel carrying the message *m*. Settimer (*T,C*) is an action defined to operate on timers. It sets the timer *T* to a value specified by *C*. Timers create a Timeout messages using timer channels. As seen in Fig. 3, three transitions are defined in the EFSM. The definitions for each transition are given below the figure. The first transition, $T1$ is a spontaneous transition, and is executed without an input event. On the other hand, $T2$ and $T3$ are when transitions because they are triggered once the input messages are received.

Protocols among different processes can often be modeled as a collection of communicating finite state machines where interactions between the processes are modeled by the exchange of messages.[8] EFSMs communicate with each other by message passing through a number of first-in-first-out (FIFO) unidirectional queues (channels)g which associate with some buffers at the endpoints of the corresponding EFSMs respectively.

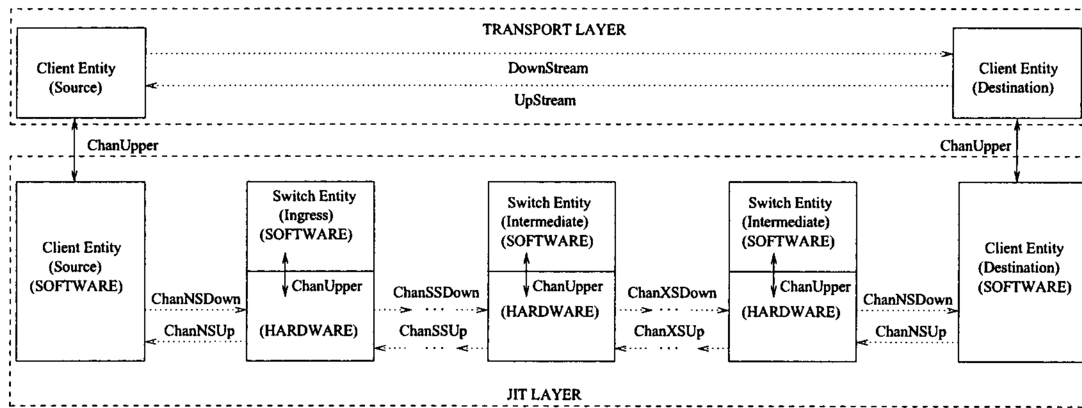


Fig. 4 Protocol stack architecture.

4 System Architecture

The relationship between different protocol entities are explained using the system architecture illustrated in Fig. 4. As seen in the figure, an upper layer source client starts the transactions by sending an Open message through the ChanUpper. JIT Layer Source Client generates a Setup message as soon as it receives the Open message from the Upper Layer using the ChanNSDown. The Upper Layer peer-to-peer connections indicate that the traffic flow from source node to the destination is called DownStream and the traffic flow from the destination to the source is called UpStream. All the messages from the ingress switch to the source client use the channel named ChanNSUp. ChanSSDown represents the channel between the ingress switch and the intermediate switch and ChanSSUp is the inverse. From the point of an intermediate switch, the channel between the intermediate switch and the entity on its downstream path is ChanXSDown, and the inverse of it is ChanXSUp whether it is another intermediate switch or the destination node. From the point of the destination node, the channel from the previous intermediate node to itself is called ChanNSDown and the inverse is ChanNSUp.

Client nodes are implemented by software, therefore there is no hardware part attached to client nodes. However, switch entities have both software and hardware parts and the connection between these two parts are represented by ChanUpper because the state diagrams related to switch entities show the behavior of hardware.

5 EFSM-Based Formal Specifications of Jumpstart JIT Protocol

The Jumpstart JIT protocol can be defined as a set of extended finite state machines communicating with each other via message transfer. The protocol consists of unicast and multicast connections. In this section, we define the state diagrams of source client, destination client, ingress switch, and intermediate switch for each type of connection. Note that for the sake of clarity, each arc in the state diagrams represents a set of transitions, and the transitions are shown in separate figures.

5.1 *Single Burst Unicast Connection*

5.1.1 *Source client sending unicast messages*

The first state machine is defined for the source client sending unicast messages. The set of messages is

$$\sum = \{\text{Open}, \text{Setup}, \text{Failure}, \text{Timeout}, \text{Setup_Ack}, \text{Connection_Failure}, \text{Close}, \text{Release}, \text{Clear_To_Send}, \text{Connect}, \text{Transmission_Complete}, \text{Keepalive}\}. \quad (1)$$

Open is generated by the Transport Layer to notify JIT Layer incoming of a burst. The Setup message is created by the Source Client's JIT Layer to set up the resources. Failure can be generated by any node to notify an error. Timeout is used for each timer specified within a Timeout message. Setup_Ack is used to acknowledge the Source Client that the Ingress switch could make the crossconnect successfully and the burst could be send. Connection_Failure is used to notify upper layers that there has been an error during the connection phase. Close is used to end the connection. The Release message is used for explicit teardown. The Clear_To_Send message is used by the Source Client's JIT layer to notify the Transport Layer that the setup process is complete and the burst can be send. Connect is generated by the Destination Client as soon as the Setup is received if a Connect is requested. Transmission_Complete notifies the upper layer that the transmission has been completed successfully. Keepalive is used for long bursts to maintain the connection until the burst ends.

The set of states S is

$$S = \{\text{IDLE}, \text{WAIT_FOR_SETUP_ACK}, \text{SETUP_PROCEEDING}, \text{DATA_TRANSMISSION}, \text{WAIT_FOR_CONNECT}\}. \quad (2)$$

The state machine waits at IDLE state until receiving a triggering event (Open for this state machine). Until it gets an acknowledgment from the ingress, it waits at

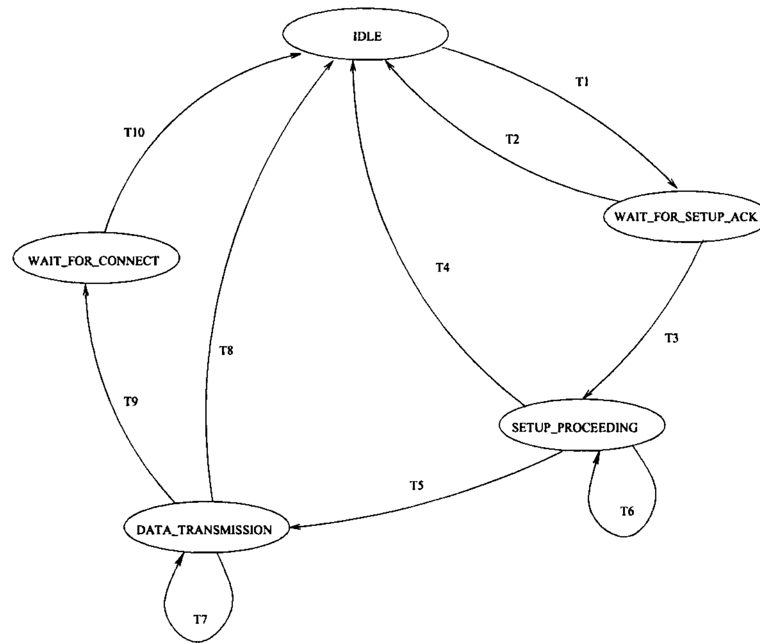


Fig. 5 State diagram for Source Client (unicast).

WAIT_FOR_SETUP_ACK. As soon as the Ack comes, the machine goes to SETUP_PROCEEDING state and stays there for the duration given in Burst_Time. The Setup_Timer times out indicating start of the data burst and the machine moves to DATA_TRANSMISSION. If the connection will be closed but the Connect has not been received, then the machine goes to the WAIT_FOR_CONNECT state and waits until the Connect comes or Conn_Timer times out.

Initial state s is the state IDLE. The set of variables is

$$V=\{\text{SA_Constant},\Delta T,\text{Conn_Constant},\text{Conn_Rcvd},\text{Rel},\text{Conn},\text{Burst_Time},\text{Burst_Delay},\text{KA_Time}\}, \tag{3}$$

where SA_Constant is used to set timer SA_Timer to an expected value equal to the duration of round trip time from source to ingress switch. If the source does not receive the Ack during that time, it indicates and error and state machine goes back to IDLE. Note that ΔT is the expected delay variation on Burst_Delay calculated by the ingress switch according to the time values in Connect message. It is used to adjust the timing information at the source. Conn_Constant is used to set the Conn_Timer, which is explained in the previous paragraph. Conn_Rcvd, Rel, and Conn, are flag variables indicating request or arrival of Connect and Release messages. Burst_Delay indicates the required delay to be waited at the source before sending the burst. Burst_Time shows the implicit teardown time calculated to end the burst. KA_Timer is the Keepalive Timer set up to send Keepalive messages.

The set of timers is

$$T=\{\text{SA_Timer},\text{SETUP_Timer},\text{Conn_Timer},\text{Burst_Timer},\text{KA_Timer}\}. \tag{4}$$

The set of actions that operate on variables is

$$A=\{\text{Settimer},\text{Update}\}. \tag{5}$$

The state diagram waits in the IDLE state until an Open message is sent by the upper layer. Once the Open message is received, the client creates a Setup message with four variables: Rel, Conn, Burst_Time, and Burst_Delay. Rel is the flag indicating whether a Release is required or not. If Rel is TRUE then a Release is required for closing the connection. The Conn variable is used to indicate whether a Connect message should be waited for or not. If Conn is set to TRUE, the protocol goes to the WAIT_FOR_CONNECT state before closing the connection. The variable Burst_Time is used to tell the burst length. If it is not specified explicitly, the protocol should wait for an explicit Close message. The variable Burst_Time together with the variable Burst_Delay is used to set the Burst_Timer. Burst_Delay is updated by the function called Update at each hop subtracting the processing time from the Burst_Delay. The state diagram and the state transitions are given in Fig. 5 and 6, respectively.

Once a Setup message is received, we change our state to WAIT_FOR_SETUP_ACK and during that transition we also set the timer setup acknowledgment timer (SA_Timer) to a predetermined value. If we do not receive an acknowledgment during this time, a timeout is generated and the state machine goes back to IDLE state generating a Release message to be sent to the Ingress switch indicating that we are closing the connection. Other possible transactions while we are at state WAIT_FOR_SETUP_ACK are receiving a Failure message from the Ingress switch or a Close message from the Upper Layer. In either case, we return to IDLE state by generating a Connection_Failure message to Upper Layer or a Release message to Ingress switch, respectively. On the other hand,

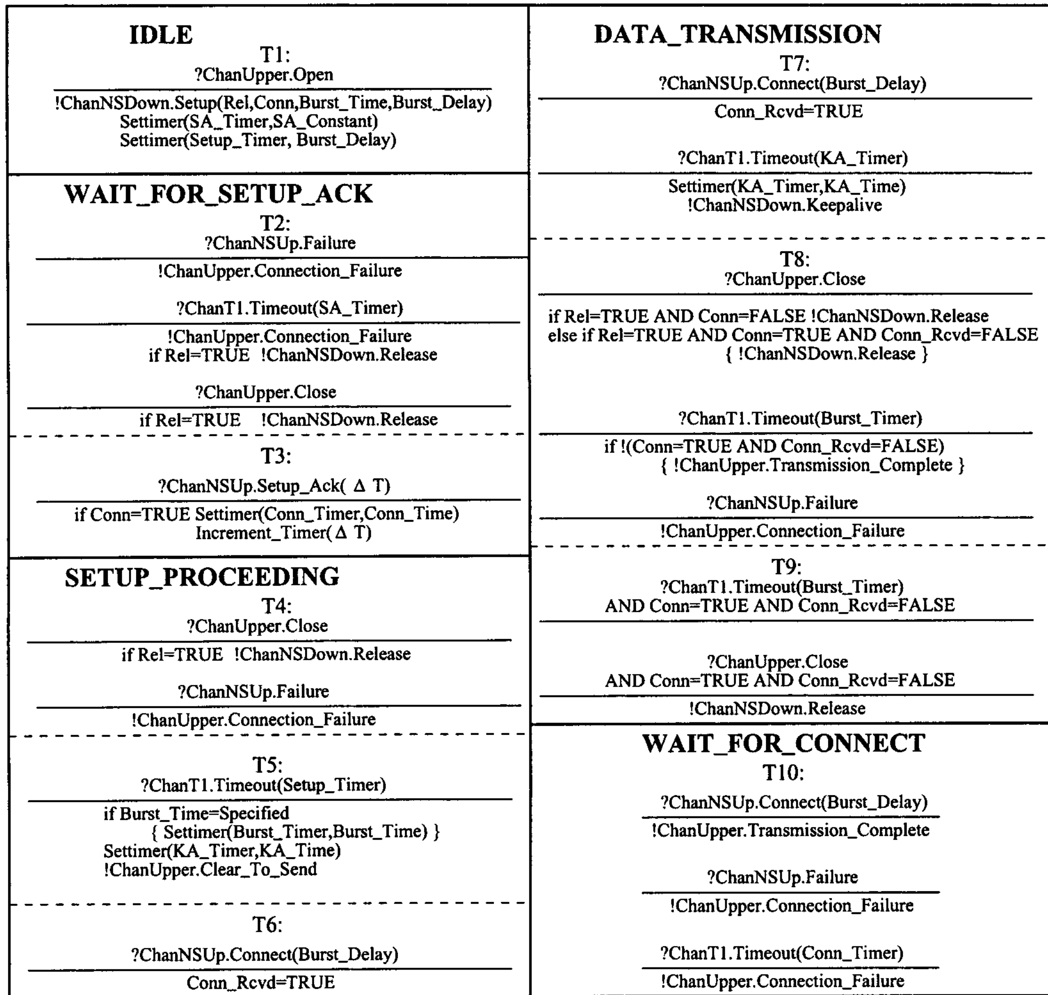


Fig. 6 State transitions for Source Client (unicast).

if we receive the acknowledgment on time, we go to SETUP_PROCEEDING state setting the timers connection timer (Conn_Timer) and setup timer (Setup_Timer).

From the SETUP_PROCEEDING state, we can go to IDLE state by receiving a Close message from the Upper Layer or a Failure message from the Ingress switch. Otherwise, we wait until the Setup_Timer times out and go to DATA_TRANSMISSION state. If we receive a Connect message meanwhile, we stay at the same state changing the variable connection received (Conn_Rcvd), which indicates that the Connect message has been received from the Ingress switch, to TRUE.

DATA_TRANSMISSION is the most complicated state. If we receive a Connect message or keepalive timer (KA_Timer) times out, we stay in the same state triggering the necessary actions. If we receive a Close, we check the status of the variables Conn and Conn_Rcvd to decide whether we will trigger transition $T8$ or $T9$. If Conn is TRUE and Connect has not been received then we go to the WAIT_FOR_CONNECT state. Otherwise we go to IDLE sending a Release message if it is required. Burst timer timeout also can trigger both $T8$ and $T9$. The decision is again based on the status of the variables Conn and Conn_Rcvd. If Conn is TRUE and Connect is not received, then we have to wait for a Connect message. Therefore, we go to WAIT_FOR_CONNECT state. Otherwise, we trigger transition $T8$. The action set for transition $T8$ with Burst_Timer timeout consists of an if-else statement checking the status of variables Rel, Conn, and Conn_Rcvd to decide on the action to be taken.

The last state is WAIT_FOR_CONNECT, where we wait only for a Connect message to arrive before we close the connection.

The state transitions use four different channels shown in Fig. 4: ChanUpper, ChanNSUp, ChanNSDown, and Chan$T1$. ChanUpper is the channel between the client node signaling protocol layer and the upper layer. ChanNSUp is the upstream channel between the client node and the ingress switch. That is, the flow is from the ingress switch to the client node. ChanNSDown is the downstream channel between the client node and the ingress switch, and the direction of the flow is from the client to the switch. Chan$T1$ is the timer channel used to receive timeout messages from the indicated timers.

5.1.2 *Destination client receiving unicast messages*

The second state machine belongs to the destination side. The role of the destination client is to complete the Setup

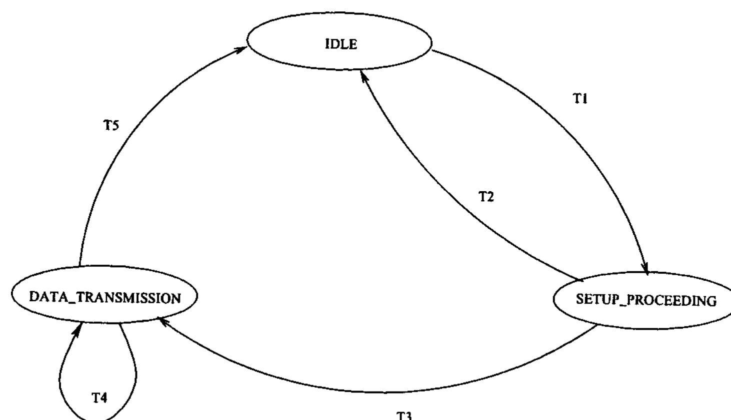


Fig. 7 State diagram for Destination Client (unicast).

process and start receiving data until seeing a Release message from the peer client, or closing the connection with a timeout. The destination client does not use the variable Rel passed by the Setup message, but for the sake of consistency, we use the same Setup message format at each state machine.

The set of messages is

$$\sum = \{\text{Open}, \text{Setup}, \text{Failure}, \text{Timeout}, \text{Setup_Complete}, \text{Close}, \text{Release}, \text{Connect}, \text{Transmission_Complete}, \text{Keepalive}\}. \quad (6)$$

Setup_Ack is not defined in this machine because the destination client does not receive nor generate an acknowledgment. The set of states S is

$$S = \{\text{IDLE}, \text{SETUP_PROCEEDING}, \text{DATA_TRANSMISSION}\}. \quad (7)$$

The destination state machine is simpler than the source machine because there is not a waiting requirement for an acknowledgment and a connect from another entity. Therefore, we can eliminate two states. Initial state s is the state IDLE. The set of variables is

$$V = \{\text{Rel}, \text{Conn}, \text{Burst_Time}, \text{Burst_Delay}, \text{KA_Timer}\}. \quad (8)$$

Although variables Rel and Conn are defined in the list of variables, they are not used. They are left on the list to be consistent with the *Setup* message structure. The set of timers is

$$T = \{\text{KA_Timer}, \text{Burst_Timer}\}. \quad (9)$$

The set of actions that operate on variables is

$$A = \{\text{Settimer}, \text{Update}\}. \quad (10)$$

The state diagram and transitions are shown in Figs. 7 and 8.

State diagram of the destination client waits in IDLE state until a Setup message comes. Once the Setup message arrives, the destination client's JIT Layer sends an Open message to the Upper Layer. If Upper Layer responds with a Close, then the destination generates a Failure message toward the source node. Otherwise, it adjusts its burst delay, sets burst timer (Burst_Timer) and keepalive timer (KA_Timer) and goes to DATA_TRANSMISSION state. If Connect is requested by the source, a Connect message is also created with the new burst delay added in it as a parameter. This parameter will be used on future estimations. Once the state machine is in DATA_TRANSMISSION state, it can either receive Keepalive messages from the source node indicating that data transmission is continuing, in which case KA_Timer is reset, or trigger transition $T5$ and go to IDLE state back. The actions triggering transaction $T5$ are a timeout event due to the Burst_Timer or the KA_Timer, receiving a Release message from the source or a Close request from the Upper Layer. KA_Timer timeout event is used to close the connection in cases where an explicit burst time is not indicated and a Release is not required. Otherwise, during normal course of data transmission, as KA_Timer is set to a value greater than keepalive message intervals, a Keepalive message is expected to reset KA_Timer before a timeout. In case a Close request comes from the Upper Layer, the protocol generates a Failure message, indicating that the connection is forced to be torn down by the destination.

5.1.3 *Ingress switch setting up a unicast connection*

This subsection gives the state diagram of an ingress switch receiving a Setup request from the source client. The role of the ingress switch receiving a Setup message is in configuring itself, finding the appropriate wavelength and port information for the data channel, and calculate estimated time for the source to start sending the data. These processes are handled in switch hardware and as fast as possible so that the switch can return an acknowledgment back to the source client with the necessary information for data transmission. As soon as the switch makes the necessary allocations inside the switch, it passes the Setup message to the next switch. The set of messages used in ingress switch state diagrams is

STATE IDLE

T1:

!ChanNSDown.Setup(Rel,Conn,Burst_Time,Burst_Delay)

!ChanUpper.Open

STATE SETUP_PROCEEDING

T2:

?ChanUpper.Close

!ChanNSUp.Failure

T3:

?ChanUpper.Setup_Complete

if Burst_Time=Specified
{ Settimer(Burst_Timer,Burst_Time+Burst_Delay) }
if Conn=TRUE !ChanNSUp.Connect(Burst_Delay)
Settimer(KA_Timer,KA_Time)

STATE DATA_TRANSMISSION

T4:

?ChanNSDown.Keepalive

Settimer(KA_Timer,KA_Time)

T5:

?ChanT1.Timeout(Burst_Timer)

!ChanUpper.Transmission_Complete

?ChanNSDown.Failure

!ChanUpper.Transmission_Failure

?ChanNSDown.Release

!ChanUpper.Transmission_Complete

?ChanUpper.Close

!ChanNSUp.Failure

?ChanT1.Timeout(KA_Timer)

!ChanUpper.Transmission_Failure

Fig. 8 State transitions for Destination Client (unicast).

$$\sum = \{\mathrm{Setup}, \mathrm{Open}, \mathrm{Failure}, \mathrm{Close}, \mathrm{Timeout}, \mathrm{Setup_Ack}, \mathrm{Release}, \mathrm{Connect}, \mathrm{Keepalive}\}. \tag{11}$$

The set of states S is

$$S = \{\mathrm{IDLE}, \mathrm{RUNNING_CHECKS}, \mathrm{DATA_TRANSMISSION}, \mathrm{WAIT_FOR_CONNECT}\}. \tag{12}$$

Unlike the source node, we do not need to use two separate states for WAIT_FOR_SETUP_ACK and SETUP_PROCEEDING because there is not any other entity sending an Ack message. Therefore, we define only one state similar to SETUP_PROCEEDING and call it RUNNING_CHECKS. Initial state s is the state IDLE. The set of variables is

$$V = \{\mathrm{Conn_Rcvd}, \Delta T, \mathrm{ErrorCode}, \mathrm{Rel}, \mathrm{Conn}, \mathrm{Burst_Time}, \mathrm{Burst_Delay}, \mathrm{KA_Time}\}. \tag{13}$$

We do not use variables SA_Constant and Conn_Constant in this machine because these are the variables used to set setup acknowledgment timer and connection timer and they are used in ingress switch state machine. The set of timers is

$$T = \{\mathrm{Burst_Timer}, \mathrm{Conn_Timer}, \mathrm{KA_Timer}\}. \tag{14}$$

The set of actions that operate on variables is

$$A = \{\mathrm{RunChecks}, \mathrm{Settimer}, \mathrm{Update}\}. \tag{15}$$

A Setup message is sent by the source client. Once the Setup message is received, the ingress switch runs some checks, e.g., cyclic redundancy code (CRC), buffer overflow, cross connect error, etc. A RunChecks function is defined in this state machine. This function returns an error code specified with the variable ErrorCode. If there is an error, this variable indicates the type of error found and the state machine returns to IDLE state. If it is NULL, the state machine goes to the DATA_TRANSMISSION state. The list of possible errors and the resulting error codes are given in Table 2.

The Ingress switch does not use Rel variable line destination client. The state diagram and the state transitions are given in Figs. 9 and 10. The Ingress switch state machine waits at the IDLE state and triggered with the arrival of a Setup message similarly with the two previous state machines. Once the Setup message comes, the switch hardware sends an Open message to the software layer of the switch and runs the checks we already mentioned. Although it is not a normally expected behavior, if the switch hardware receives a Release message from the source immediately after receiving the Setup request, it passes the Release message to the following switch and sends a Close message to the Software layer. In case, the hardware passes the error checks successfully, it sends back a Setup_Ack(ΔT) message. Here the ΔT parameter is used by the source node to determine the waiting interval between reception of the Setup_Ack and start of the data transmission. After sending back the Setup_Ack, it passes the Setup message to the next switch, updates the burst delay by subtracting its processing time from the Burst_Delay variable it receives with Setup message. If the Burst_Time is specified explicitly in the Setup message, the switch sets the Burst_Timer. After setting the connection and keepalive timers, it goes to DATA_TRANSMISSION state.

Once we are at DATA_TRANSMISSION state, we can get a burst timer timeout indicating, we reached to the estimated teardown time and we close the connection. On the action part of that transition, we have an if control that is used for deciding whether or not we must send a Connect message. If the Conn variable is set during the Setup message, indicating the source requires a Connect message back and the Connect message is received,† the switch

†The variable Conn_Rcvd is TRUE only if a Connect message is received.

Table 2 Error types and codes.

Error Type	Error Code	Explanation
no_error	0	The check returns without any error
crc_error	1	CRC error
ime_buf_overflow	2	An ingress switch message buffer overflow
gme_buf_overflow	3	An intermediate switch message buffer overflow
sigmess_state	4	A state machine error
sigmess_xcnct	5	A cross connect error
label_lut	6	A label look-up table error

sends a Connect message back to the source. On the other hand, if the burst timer times out, a Connect message is expected, but the Connect has not been received yet, then the switch goes to WAIT_FOR_CONNECT state, closing the connection. In another case where we receive a Release message from the source, we must check the status of the variables Conn and Conn_Rcvd. If Conn is TRUE, that is a Connect message is expected but the Connect has not been received, then we go to the WAIT_FOR_CONNECT state again closing the connection and creating the Release message. On the other hand, if Conn is FALSE, that is a Connect message is not expected, or Conn is TRUE and a Connect has already been received, then we go to the IDLE state again by closing the connection and creating the Release. The two other possible transitions at the DATA_TRANSMISSION state are receiving a Failure from the following switch and going to a KA_Timer timeout. In case of a Failure, we just pass it to the source node. In case of a timeout, we close the connection informing the Software Layer.

The state transitions use five different channels: ChanSSUp, ChanSSDown, ChanNSUp, ChanNSDown, and ChanT1. ChanNSUp, ChanNSDown, and Chan$T1$ have already been defined. ChanSSUp is the channel between the ingress switch and the intermediate switch with the flow from intermediate switch to ingress switch. ChanSSDown is the same channel with opposite flow direction.

5.1.4 *Intermediate switch setting up a unicast connection*

The state diagram of an intermediate switch is similar to the state diagram of an Ingress switch shown in Fig. 9. The transition diagrams, on the other hand, are also almost identical with different communication channels and only one transition deleted. The transitions for an intermediate switch are given in Fig. 11. We do not give the set of

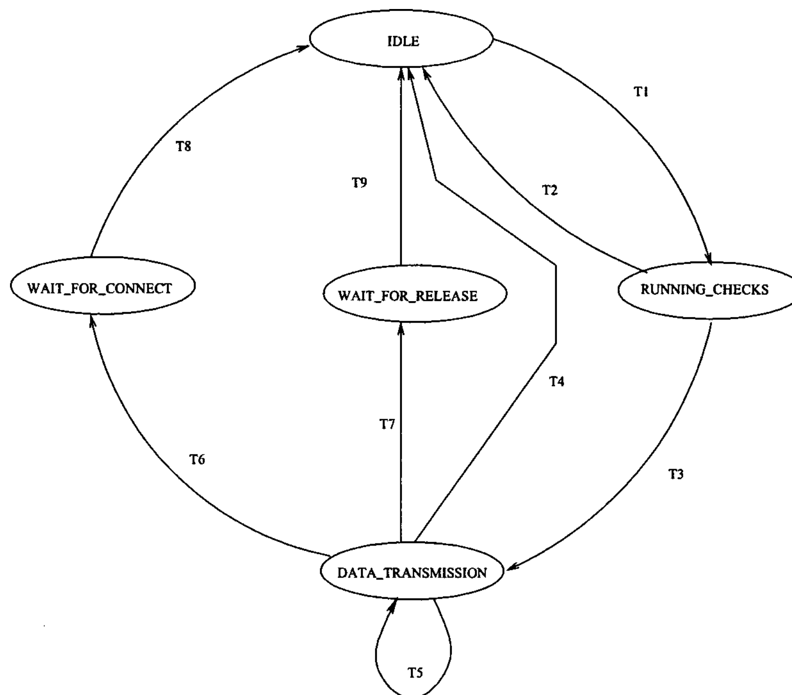


Fig. 9 State diagram for Ingress switch (unicast).

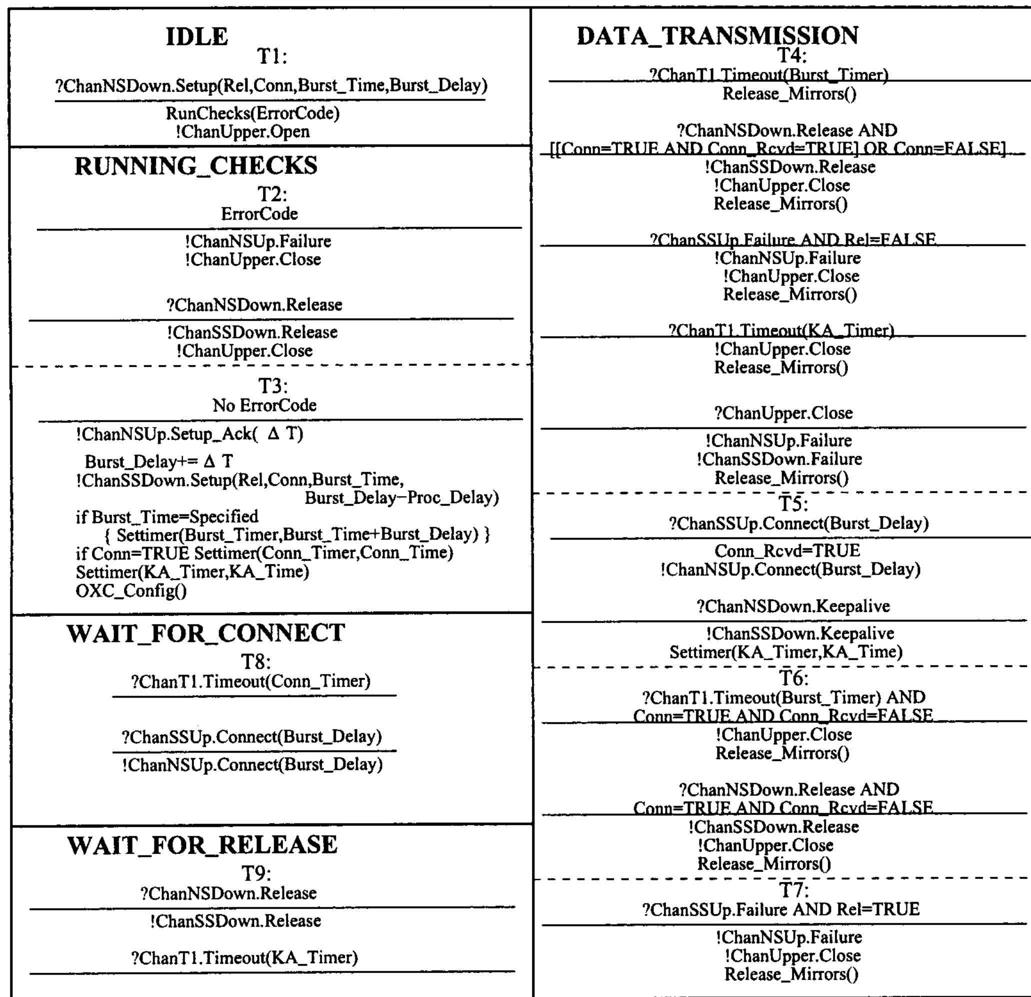


Fig. 10 State transitions for Ingress switch (unicast).

messages, states, etc., as they are all the same with Ingress switch.

For intermediate switches some channels are defined as ChanXS because it is not known whether there is a switch or a node connected to the switch to which the diagrams belong. Therefore these channels are defined anonymously. The channels between the intermediate switch and the switch connected to it are called, as in earlier cases, ChanSS. The channel to the Software Layer is also ChanUpper and the timer channel is again Chan$T1$.

The only transition different from the Ingress switch is transition $T3$. An ingress switch, after completing error controls successfully sends back to the source a Setup_Ack both for acknowledgment purposes and to inform the source about the calculated start time of data burst. On the other hand, an intermediate switch does not have such a function because in Jumpstart protocol, there is no acknowledgment process between the entities. The rest of the state machine is similar to that of the Ingress switch.

5.2 *Persistent Unicast Connection*

In Sec. 5.1, it is explained that during a persistent connection, a number of single bursts are placed in a general session framework. In other words, as illustrated in Fig. 2, a persistent path setup process is similar once a session is setup. That is, in this case, we must open a session first. Once the session is initialized, the rest is single burst setup until we close the session sending a Session Release message. Therefore, due to the space limitations, we do not give all the state machines related to persistent connection but explain only briefly the difference of the state machines with the Source Client case.

5.2.1 *Source client creating persistent unicast connection*

The set of messages is

$$\begin{aligned}\sum =\{&\text{SessionOpen}, \text{SessionDeclaration}, \text{Session_Failure},\\ &\text{SessionRelease}, \text{SessionClose}, \text{Declaration_Ack},\\ &\text{Open}, \text{Setup}, \text{Failure}, \text{Timeout}, \text{Setup_Ack},\\ &\text{Connection_Failure}, \text{Close}, \text{Release},\\ &\text{Clear_To_Send}, \text{Connect},\\ &\text{Transmission_Complete}, \text{Keepalive}\}. \end{aligned} \tag{16}$$

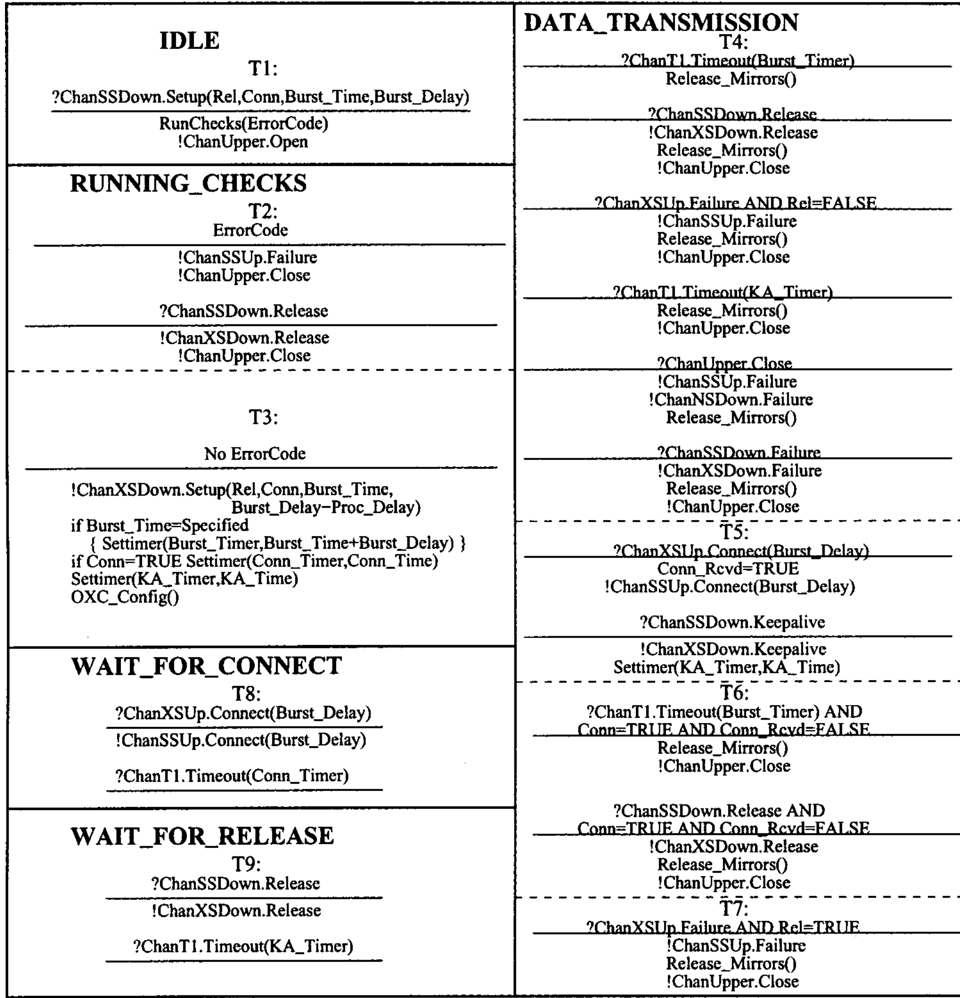


Fig. 11 State transitions for an Intermediate switch (unicast).

The set of states S is

$$S=\{\text{IDLE},\text{WAIT_FOR_DEC_ACK},\\ \text{WAIT_FOR_SETUP},\text{WAIT_FOR_SETUP_ACK},\\ \text{SETUP_PROCEEDING},\text{DATA_TRANSMISSION},\\ \text{WAIT_FOR_CONNECT}\}. \tag{17}$$

Initial state s is the state IDLE. The set of variables is

$$V=\{\text{Session_Time},\text{SKA_Time},\text{SA_Constant},\Delta T,\\ \text{Conn_Constant},\text{Conn_Rcvd},\text{Rel},\text{Conn},\\ \text{Burst_Time},\text{Burst_Delay},\text{KA_Time}\}. \tag{18}$$

The set of timers is

$$T=\{\text{Session_Timer},\text{SKA_Timer},\text{SA_Timer},\\ \text{SETUP_Timer},\text{Conn_Timer},\text{Burst_Timer},\text{KA_Timer}\}. \tag{19}$$

The set of actions that operate on variables is

$$A=\{\text{Settimer},\text{Update}\}. \tag{20}$$

In the persistent connection, we place on-the-fly connection diagrams in a framework that includes session control message flows. This time, the triggering action is a Session Declaration, which requires a Declaration Ack from the destination. Once the session is initialized successfully, the state diagram becomes almost identical to the on-the-fly diagrams. The main difference is in the control functions such as SKA_Timer timeout, which indicates a Session Keepalive Timer timeout or a Session Release message that will force the state diagram go to the IDLE state wherever it was. The state diagram and the state transitions are given in Fig. 12 and 13, respectively.

As seen in the message set, we added a set of new messages related to session handling such as SessionOpen, SessionDeclaration, etc. Compared with on-the-fly source diagrams, we added two new states for session initialization. The other five states are the same, although we changed the name of the IDLE state in Fig. 5 to WAIT_FOR_SETUP for the sake of clarity. On the other hand, in this diagram,

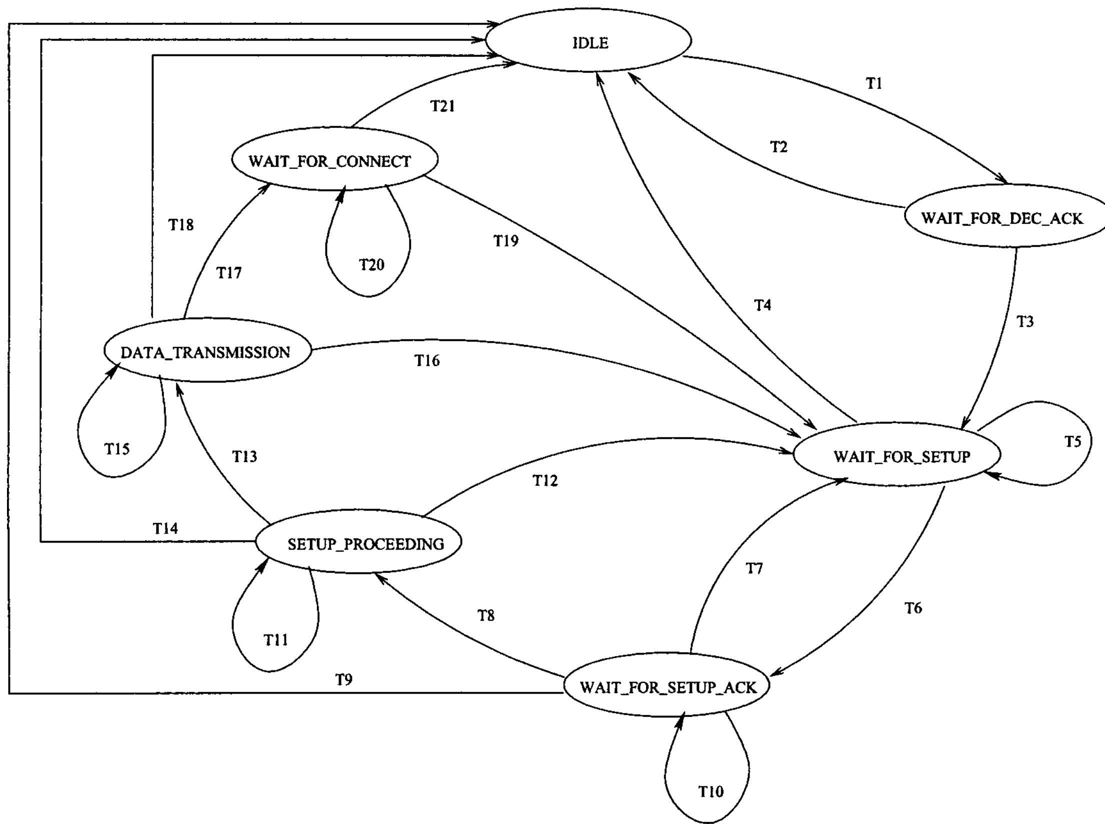


Fig. 12 State diagram for Source Client (persistent unicast).

almost every state has a self-loop due to session Keepalive Timer. In this case, we can go to a session timeout at any state once the session is initialized so we needed to add a self-loop to check that event. The other important change comes with additional transitions from every state to IDLE state. In this state machine, it is always possible to close the session, which will automatically close the connections also. Therefore, we can go to IDLE state from every state in case we receive a SessionClose request from the Upper Layer or a Failure (e.g., session failure) from the Ingress switch. The last but not the least point we need to mention is that, during Keepalive timeouts, we generate Keepalive messages. Although in practice Keepalive message generated for session continuation and message continuation will differ, in this machine, we use the same message for both purposes. The same is true for Failure messages.

5.2.2 *Destination client receiving persistent unicast session declaration*

The set of messages is

$$\sum = \{\text{SessionOpen}, \text{SessionDeclaration}, \text{Session_Failure}, \text{SessionRelease}, \text{SessionClose}, \text{Declaration_Ack}, \text{Open}, \text{Setup}, \text{Failure}, \text{Timeout}, \text{Setup_Ack}, \text{Connection_Failure}, \text{Close}, \text{Release}, \text{Clear_To_Send}, \text{Connect}, \text{Transmission_Complete}, \text{Keepalive}\}.$$

The set of states S is

$$S = \{\text{IDLE}, \text{WAIT_FOR_SETUP}, \text{SETUP_PROCEEDING}, \text{DATA_TRANSMISSION}\}. \tag{22}$$

Initial state s is the state IDLE. The set of variables is

$$V = \{\text{SKA_Time}, \text{Rel}, \text{Conn}, \text{Burst_Time}, \text{Burst_Delay}, \text{KA_Time}\}. \tag{23}$$

The set of timers is

$$T = \{\text{SKA_Timer}, \text{SA_Timer}, \text{Burst_Timer}, \text{KA_Timer}\}. \tag{24}$$

The set of actions that operate on variables is

$$A = \{\text{Settimer}, \text{Update}\} \ldots \tag{25}$$

The state diagram and the state transitions are given in Figs. 14 and 15, respectively.

IDLE

T1:
?ChanUpper.SessionOpen
!ChanNSDown.SessionDeclaration
Settimer(Session_Timer,Session_Time)

WAIT_FOR_DEC_ACK

T2:
?ChanNSUp.SessionRelease
!ChanUpper.Session_Failure

?ChanT1.Timeout(Session_Timer)
!ChanUpper.Session_Failure
!ChanNSDown.SessionRelease

?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

T3:
?ChanNSUp.Declaration_Ack
Settimer(SKA_Timer,SKA_Time)

WAIT_FOR_SETUP

T4:
?ChanNSUp.SessionRelease
!ChanUpper.SessionClose

?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

T5:
?ChanT1.Timeout(SKA_Timer)
Settimer(SKA_Timer,SKA_Time)
!ChanNSDown.Keepalive

T6:
?ChanUpper.Open
!ChanNSDown.Setup(Rel,Conn,Burst_Time,Burst_Delay)
Settimer(SA_Timer,SA_Constant)
Settimer(Setup_Timer,Burst_Delay)

WAIT_FOR_SETUP_ACK

T7:
?ChanNSUp.Failure
!ChanUpper.Connection_Failure

?ChanT1.Timeout(SA_Timer)
!ChanUpper.Connection_Failure
if Rel=TRUE !ChanNSDown.Release

?ChanUpper.Close
if Rel=TRUE !ChanNSDown.Release

T8:
?ChanNSUp.Setup_Ack(Δ T)
if Conn=TRUE Settimer(Conn_Timer,Conn_Time)
Increment_Timer(Δ T)

T9:
?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

?ChanNSUp.SessionRelease
!ChanUpper.Session_Failure

T10:
?ChanT1.Timeout(SKA_Timer)
Settimer(SKA_Timer,SKA_Time)
!ChanNSDown.Keepalive

SETUP_PROCEEDING

T11:
?ChanT1.Timeout(SKA_Timer)
Settimer(SKA_Timer,SKA_Time)
!ChanNSDown.Keepalive

?ChanNSUp.Connect(Burst_Delay)
Conn_Rcvd=TRUE

T12:
?ChanUpper.Close
if Rel=TRUE !ChanNSDown.Release

?ChanNSUp.Failure
!ChanUpper.Connection_Failure

T13:
?ChanT1.Timeout(Setup_Timer)
if Burst_Time=Specified
{ Settimer(Burst_Timer,Burst_Time) }
Settimer(KA_Timer,KA_Time)
!ChanUpper.Clear_To_Send

T14:
?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

?ChanNSUp.SessionRelease
!ChanUpper.Session_Failure

WAIT_FOR_CONNECT

T19:
?ChanNSUp.Connect(Burst_Delay)
!ChanUpper.Transmission_Complete

?ChanNSUp.Failure
!ChanUpper.Connection_Failure

?ChanT1.Timeout(Conn_Timer)
!ChanUpper.Connection_Failure

T20:
?ChanT1.Timeout(SKA_Timer)
Settimer(SKA_Timer,SKA_Time)
!ChanNSDown.Keepalive

T21:
?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

?ChanNSUp.SessionRelease
!ChanUpper.Session_Failure

DATA_TRANSMISSION

T15:
?ChanNSUp.Connect(Burst_Delay)
Conn_Rcvd=TRUE

?ChanT1.Timeout(KA_Timer)
Settimer(KA_Timer,KA_Time)
!ChanNSDown.Keepalive

?ChanT1.Timeout(SKA_Timer)
Settimer(SKA_Timer,SKA_Time)
!ChanNSDown.Keepalive

T16:
?ChanUpper.Close
if Rel=TRUE AND Conn=FALSE !ChanNSDown.Release
else if Rel=TRUE AND Conn=TRUE AND Conn_Rcvd=FALSE
{ !ChanNSDown.Release }

?ChanT1.Timeout(Burst_Timer)
if !(Conn=TRUE AND Conn_Rcvd=FALSE)
{ !ChanUpper.Transmission_Complete }

?ChanNSUp.Failure
!ChanUpper.Connection_Failure

T17:
?ChanT1.Timeout(Burst_Timer)
AND Conn=TRUE AND Conn_Rcvd=FALSE

?ChanUpper.Close
AND Conn=TRUE AND Conn_Rcvd=FALSE
!ChanNSDown.Release

T18:
?ChanUpper.SessionClose
!ChanNSDown.SessionRelease

?ChanNSUp.SessionRelease
!ChanUpper.Session_Failure

Fig. 13 State transitions for Source Client (persistent unicast).

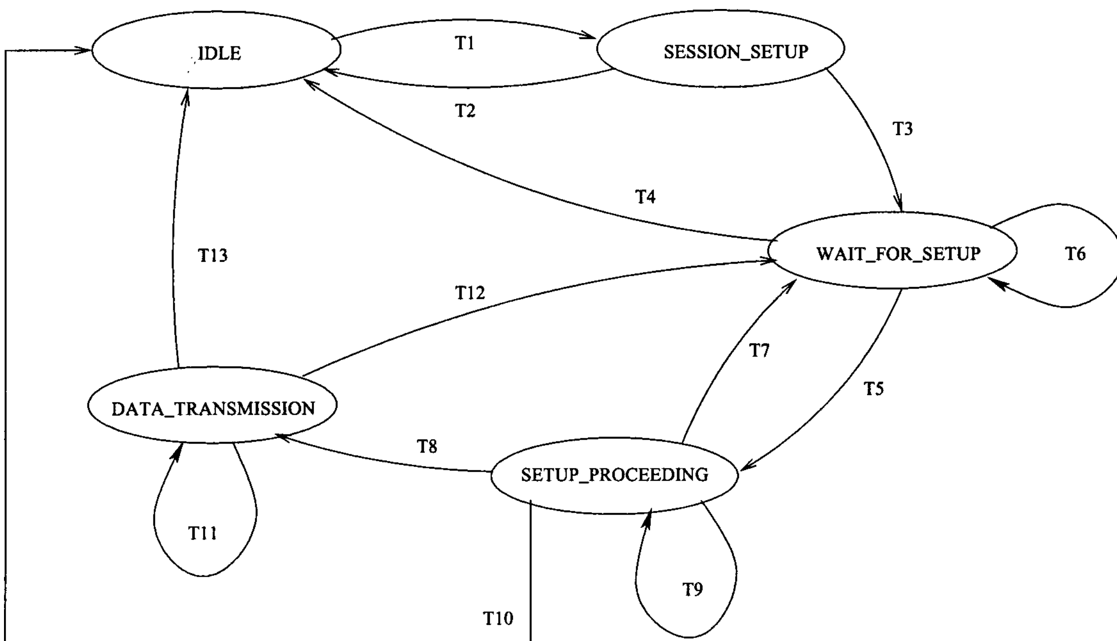


Fig. 14 State diagram for Destination Client (persistent unicast).

IDLE

T1:
?ChanNSDown.SessionDeclaration
!ChanUpper.SessionOpen

WAIT_FOR_SETUP

T2:
?ChanUpper.SessionClose

?ChanNSDown.SessionRelease
!ChanUpper.Session_Complete

T3:
?ChanUpper.SessionOpen
Settimer(SKA_Timer,SKA_Time)
!ChanNSUp.Declaration_Ack

WAIT_FOR_SETUP

T4:
?ChanNSDown.SessionRelease
!ChanUpper.Session_Complete

?ChanT1.Timeout(SKA_Timer)
!ChanUpper.Session_Failure

T5:
?ChanNSDown.Setup(Rel,Conn,Burst_Time,Burst_Delay)
!ChanUpper.Open

T6:
?ChanNSDown.Keepalive
Settimer(SKA_Timer,SKA_Time)

SETUP_PROCEEDING

T7:
?ChanUpper.Close
!ChanNSUp.Failure

T8:
?ChanUpper.Setup_Complete
if Burst_Time=Specified
{ Settimer(Burst_Timer,Burst_Time+Burst_Delay) }
if Conn=TRUE !ChanNSUp.Connect(Burst_Delay)
Settimer(KA_Timer,KA_Time)

T9:
?ChanNSDown.Keepalive
Settimer(SKA_Timer,SKA_Time)

DATA_TRANSMISSION

T11:
?ChanNSDown.Keepalive
Settimer(KA_Timer,KA_Time)
Settimer(SKA_Timer,SKA_Time)

T12:
?ChanT1.Timeout(Burst_Timer)
!ChanUpper.Transmission_Complete

?ChanNSDown.Release
!ChanUpper.Transmission_Complete

?ChanUpper.Close
!ChanNSUp.Failure

?ChanNSDown.Failure
!ChanUpper.Transmission_Failure

?ChanT1.Timeout(KA_Timer)
!ChanUpper.Transmission_Failure

T13:
?ChanNSDown.SessionRelease
!ChanUpper.Session_Complete

?ChanUpper.SessionClose

?ChanT1.Timeout(SKA_Timer)
!ChanUpper.Session_Failure

T10:
?ChanUpper.SessionClose

?ChanT1.Timeout(SKA_Timer)
!ChanUpper.Session_Failure

?ChanNSDown.SessionRelease
!ChanUpper.Session_Complete

Fig. 15 State transitions for Destination Client (persistent unicast).

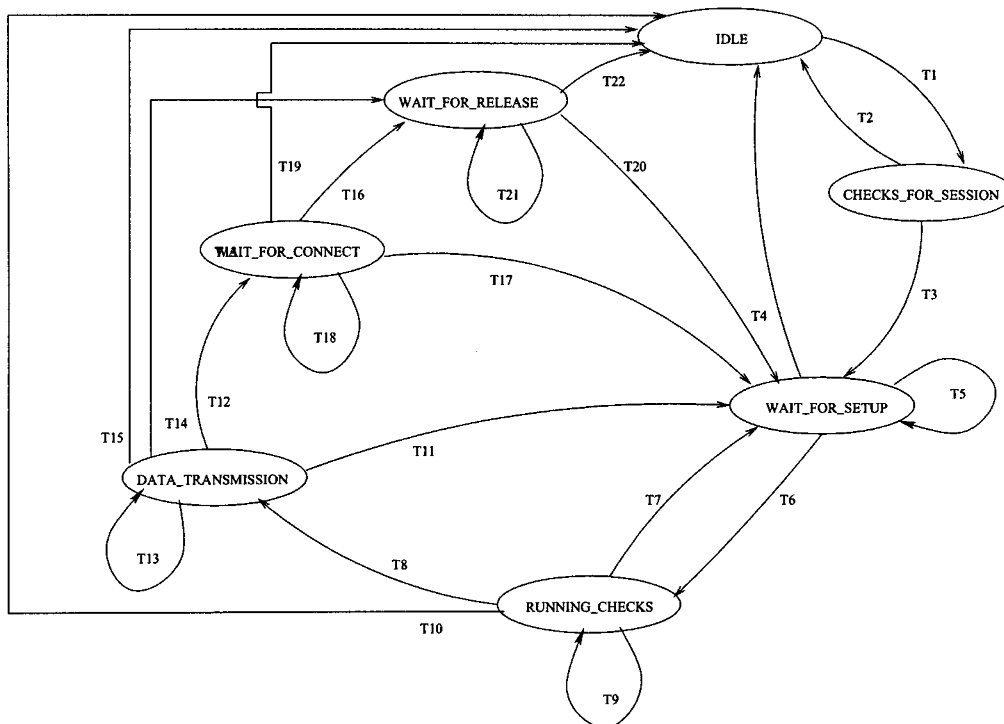


Fig. 16 State diagram for Ingress switch (persistent unicast).

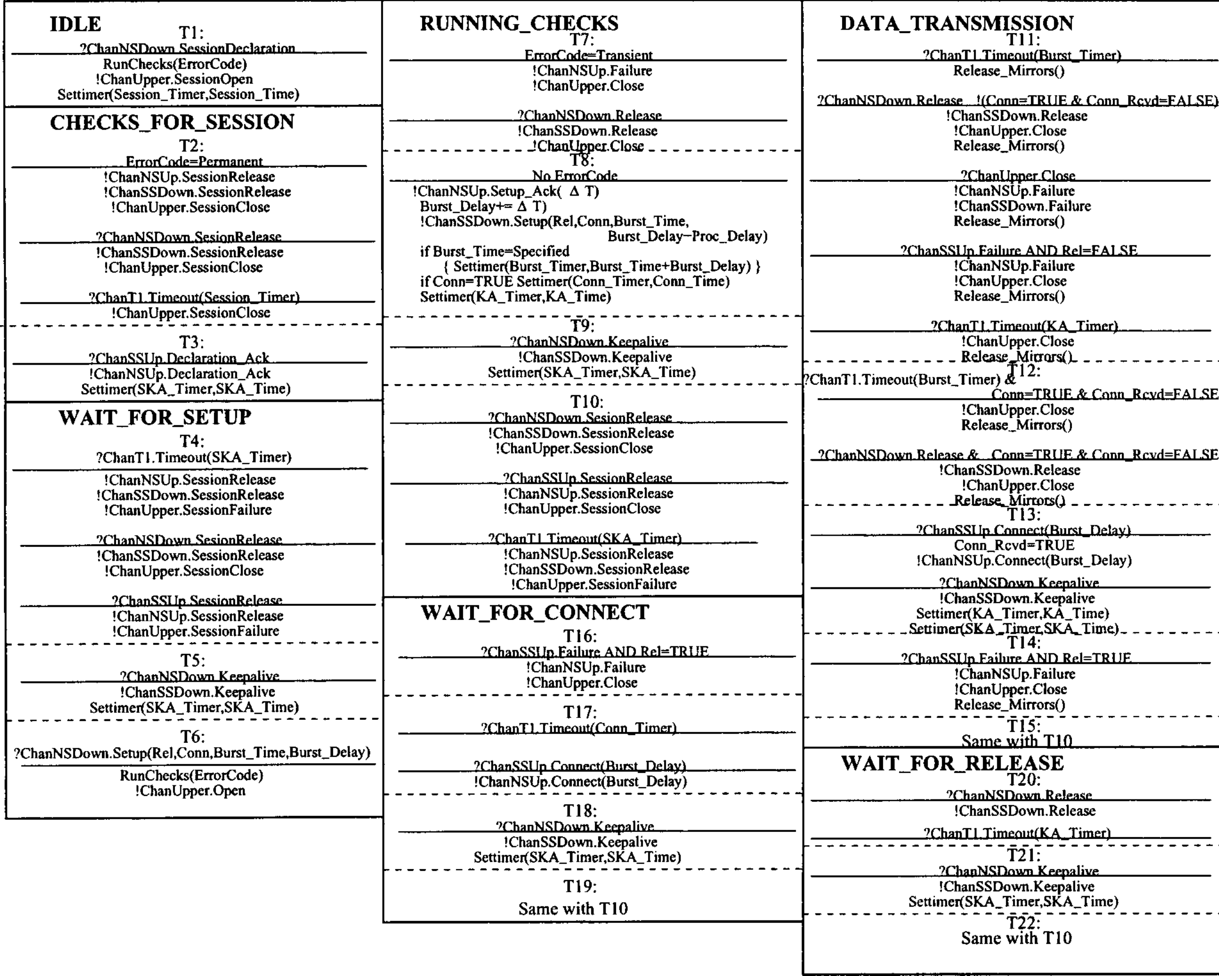


Fig. 17 State transitions for Ingress switch (persistent unicast).

5.2.3 *Ingress switch setting up a persistent unicast connection*

This subsection gives the state diagram of an Ingress switch receiving a SESSIONDECLARATION message from the source client, and configuring itself, and passing the message to the next switch. The set of messages is

$$\sum = \{\text{SessionDeclaration}, \text{SessionOpen}, \text{SessionClose}, \text{SessionRelease}, \text{Declaration_Ack}, \text{Open}, \text{Setup}, \text{Setup_Ack}, \text{Failure}, \text{Close}, \text{Timeout}, \text{Release}, \text{Connect}, \text{Keepalive}\}. \tag{26}$$

The set of states S is

$$S = \{\text{IDLE}, \text{CHECKS_FOR_SESSION}, \text{WAIT_FOR_SETUP}, \text{RUNNING_CHECKS}, \text{DATA_TRANSMISSION}, \text{WAIT_FOR_CONNECT}\}. \tag{27}$$

Initial state s is the state IDLE. The set of variables is

$$V = \{\text{Conn_Rcvd}, \Delta T, \text{ErrorCode}, \text{Rel}, \text{Conn}, \text{Burst_Time}, \text{Burst_Delay}, \text{KA_Time}, \text{SKA}, \text{Time}\}. \tag{28}$$

The set of timers is:

$$T = \{\text{Burst_Timer}, \text{Conn_Timer}, \text{KA}, \text{Timer}, \text{SKA_Timer}\}. \tag{29}$$

The set of actions that operate on variables is

$$A = \{\text{RunChecks}, \text{Settimer}, \text{Update}\}\text{g} \ldots \tag{30}$$

The state diagram and the state transitions are given in Figs. 16 and 17, respectively.

From the point of a persistent connection setup, an Intermediate switch is exactly similar to an Ingress switch, except for the channel names and the difference in handling SETUP requests, which was explained in Sec. 5.1.4. Therefore, it is not explained separately for the persistent call case.

6 Conclusions

We presented a formal description of the Jumpstart JIT signaling protocol for unicast traffic. As mentioned in Ref. 7, although JIT signaling supports four different connection setup schemes, Jumpstart covers only two of them since they require the simplest schedulers. On the other hand, while predictive reservation schemes may have a potential positive effect on the overall blocking probability of the network, the switch hardware becomes significantly more complex.

We defined two different unicast traffic flows: single-burst unicast and persistent unicast. In the single-burst unicast connection, we set a connection only for sending one burst and then close the connection. On the persistent case, the connection is setup once for the duration of a session and a number of bursts use the same connection until the session is closed. The state diagrams and transitions for both traffic flows are given in this paper. Protocol testing based on these EFSMs are also defined as another study. For testing, we define reachability trees based on the EFSMs and generate possible input and output sequences. Then using reduction techniques, we obtain the unique input-output (UIO) sequences as mentioned in Refs. 14 and 15. Future work includes the definition of multicast traffic, which is part of the Jumpstart JIT protocol specifications.

Acknowledgments

This research effort is being supported through a contract with ITIC (Intelligence Technology Innovation Center).

References

1. P. W. King, "Formalization of protocol engineering concepts," *IEEE Trans. Comput.* **40**(4), April, 1991 ().
2. H. Hansson, B. Jonsson, F. Orava, and B. Pehrson, "Formal design of communication protocols," ISS'90 (1990).
3. K. Nail and B. Sarikaya, "Testing communication protocols," IEEE Software (1992).
4. G. J. Holzmann, "Protocol Design: Redefining the State of the Art," IEEE Software (1992).
5. K. J. Turner, "The use of formal methods in communications standards."
6. E. Gunawan, T. P. Tong, and S. Nansi, *Survey of Formal Description Techniques (FDTs) For Protocol Converter Design*, pp. 422–425, IEEE Tencon, Beijing (1993).
7. I. Baldine, G. N. Rouskas, H. G. Perros, and D. Stevenson, "Jumpstart: a just-in-time signaling architecture for WDM burst-switched networks," *IEEE Commun. Mag.* **40**(2), 82–89 (2002).
8. H. Bowman, G. S. Blair, L. Blair, and A. G. Chetwynd, "Formal description of distributed multimedia systems: an assessment of potential techniques," *Comput. Commun.* (Dec. 1995).
9. J. Y. Wei and R. I. McFarland, "Just-in-time signaling for WDM optical burst switching networks," *J. Lightwave Technol.* **18**(12), 2019–2037 (2000).
10. M. Yoo, C. Qiao, and S. Dixit, "QoS performance of optical burst switching in IP-over-WDM networks," *IEEE J. Sel. Areas Commun.* **18**(10), 2062–2071 (2000).
11. J. S. Turner, "Terabit burst switching," *J. High-Speed Net.* **8**(1), 3–16 (1999).
12. C. Qiao and M. Yoo, "Optical burst switching (OBS)—a new paradigm for an optical internet," *J. High-Speed Net.* **8**(1), 69–84 (1999).
13. M. Yoo and C. Qiao, "Just-enough-time (JET): a high speed protocol for bursty traffic in optical networks," in *Proc. IEEE/LEOS Tech. Global Info. Infra*. pp. 26–27 (1997).
14. R. Lai, "A survey of communication protocol testing," *J. Syst. Software* (2001).
15. D. P. Sidhu and T.-K. Leung, "Formal methods for protocol testing: a detailed study," *IEEE Trans. Software Eng.* **15**(4), 413–426 (1989).

Abdul Halim Zaim received his BSc degree (with honor) in computer science and engineering from Yildiz Technical University, Istanbul, Turkey, in 1993, his MSc degree in computer engineering from Bogazici University, Istanbul, Turkey, in 1996, and his PhD degree in electrical and computer engineering from North Carolina State University, Raleigh, in 2001. As a teaching assistant and lecturer, he taught several courses at Istanbul and Yeditepe Universities between 1993 and 1997. In 1998 to 1999, he was with Alcatel, Raleigh, North Carolina. He is currently a postdoctoral research associate at North Carolina State University (NCSU) (working at MCNC under contract with NCSU) and an adjunct professor at NCSU. His research interests include computer performance evaluation, satellite and high-speed networks, network protocols, optical networks, and computer network design.

Ilia Baldine attended Moscow State University, Moscow, Russia, and received his BS degree in 1993 in computer science from the Illinois Institute of Technology and his MS degree in 1995 and his PhD degree in 1998 in computer science from North Carolina State University. He joined the Advanced Network Research Group at MCNC in 1998 and has been an active participant in JiNao, Celestial, Helios, and Jumpstart projects. Dr. Baldine has published a numerous papers on network security, all-optical networks, and other related topics.

Mark Cassada received his BS and MS degrees from North Carolina Statue University. His studies included antenna theory, electronic circuits, switched power supplies, and field programmable gate array (FPGA) design. He has worked in the telecommunications field for over 8 years. Throughout the course of his work he has focused on FPGA analog and digital board designs. Currently, he is a network hardware engineer/team leader for the Advanced Networking Research Group at MCNC.

George N. Rouskas is a professor of computer science at North Carolina State University. He received his Diploma in computer engineering from the National Technical University of Athens (NTUA), Greece, in 1989, and his MS and PhD degrees in computer science from the College of Computing, Georgia Institute of Technology, Atlanta, in 1991 and 1994, respectively. He received a 1997 National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award, and coauthored a paper that received the Best Paper Award at the 1998 SPIE conference on all-optical networking. He also received the 1995 Outstanding New Teacher Award from the Department of Computer Science, North Carolina State University, and the 1994 Graduate Research Assistant Award from the College of Computing, Georgia Tech. During the 2000 and 2001 academic year he spent a sabbatical term at Vitesse Semiconductor, Morrisville, North Carolina, and in May and June 2000 he was an invited professor at the University of Evry, France. He was a coguest editor for the *IEEE Journal on Selected Areas in Communications*, Special Issue on Protocols and Architectures for Next Generation Optical WDM Networks, published in October 2000, and is on the editorial boards of the *IEEE/ACM Transactions on Networking, Computer Networks, and Optical Networks*. He is a senior member of the IEEE, and a member of the ACM and the Technical Chamber of Greece.

Harry G. Perros is a professor of computer science, an Alumni Distinguished Graduate Professor, and program coordinator of the master of science degree in computer networks at North Carolina State University. He received his BSc degree in mathematics in 1970 from Athens University, Greece, his MSc degree in operational research with computing from Leeds University, England, in 1971, and his PhD degree in operations research from Trinity College Dublin, Ireland, in 1975. He has held visiting faculty positions at INRIA, Rocquencourt, France in 1979, NORTEL, Research Triangle

Park, North Carolina in 1988 and 1989 and 1995 and 1996, and the University of Paris 6, France, in 1995 and 1996, 2000, and 2002. He has published extensively in the area of performance modeling of computer and communication systems and has organized several national and international conferences. In 1994 he published the monograph *Queueing Networks with Blocking: Exact and Approximate Solutions* (Oxford Press) and in 2001 the textbook *An Introduction to ATM networks* (Wiley). He chairs the International Federation for Information Processing (IFIP) Working Group 6.3 on the Performance of Communication Systems, is a member of IFIP Working Groups 7.3 and 6.2, and is senior member of the IEEE. His current research interests are in the areas of optical networks and satellites.

Daniel Stevenson received his MS degree in physics from the University of North Carolina and has worked in various commercial R&D organizations including Bell Labs, GTE, and Nortel Networks. He currently directs Advanced Network Research at MCNC, where he pursues interest in optical networking.