Formal Description of the Jumpstart Just-In-Time Signaling Protocol Using EFSM*

A.Halim Zaim[a], Ilia Baldine[a], Mark Cassada[a], George N. Rouskas[b], Harry G. Perros[b], and Dan Stevenson[a]

[a]MCNC, 3021 Cornwallis Rd. P.O.Box 12889, Research Triangle Park, NC 27709 USA
[b]NCSU, Department of Computer Science, Raleigh, NC, USA

ABSTRACT

We present a formal protocol description for a Just-In-Time (JIT) signaling scheme running over a core dWDM network which utilizes Optical Burst Switches (OBS). 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 worked on single burst connections that means setting up the connection just before sending a single burst and then closing the connection as soon as the burst is sent. The communication between the EFSMs is handled through message transfer between protocol entities.

1. INTRODUCTION

In recent years, a great change in the protocol design is observed in telecommunication industry. Instead of the traditional design cycle, which includes a three step process consisting of a high level design, 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.[1–5]

Extended Finite State Machine (EFSM) approach is fully expressive and particularly useful as a means of describing a communication protocol. EFSM-based techniques can be applied in telecommunications easier than most other approaches, and are better suited to assist in the followup implementations. EFSMs allow generation of test suites easier than traditional techniques and scale better than traditional FSM models. Therefore, in this study we used an EFSM based description model.

Jumpstart signaling protocol was first introduced in.[6] The signaling architecture is based on wavelength routing and burst switching. Signaling is Just-in-time(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 is opaque to network entities and travels 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 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 optical burst switching (OBS) have been previously studied in the literature.[7–11] 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 received. The variations on the signaling schemes

*This research effort is being supported through a contract with ARDA (Advanced Research and Development Activity, http://www.ic-arda.org).

(Send correspondence to Ilia Baldine)
Ilia Baldine: E-mail: ibaldin@anr.mcnc.org
A.H.Z., M.C. and D.S.: E-mail: {ahzaim,mcc, stevenso}@anr.mcnc.org
G.N.R and H.G.P: E-mail: {rouskas, hp}@csc.ncsu.edu

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

2. JUMPSTART JUST-IN-TIME 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 MPLS or 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 Just-In-Time OBS network described in this section only addresses the connection setup procedures.

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

Table 1. Signaling Protocol Functions

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 setup:

Explicit setup and explicit teardown - each burst is preceded by its own *Setup* message and followed by its own *Release* message (which allows the intermediate switches to close the optical crossconnects or use them for other connections)

Explicit setup and estimated teardown signaling schemes - 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.

Explanation of different signaling schemes can be found in.[6] We will 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.[6] 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 that is out of scope of this paper.

2.2. 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 Figures 1 and 2. The presence of the *Release* message at the end of each connection is dictated by the type of the connection (explicit vs. timed teardown).

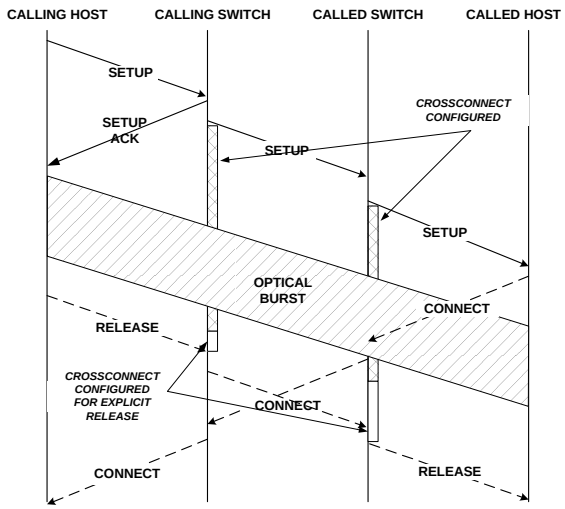

Figure 1. Single Burst

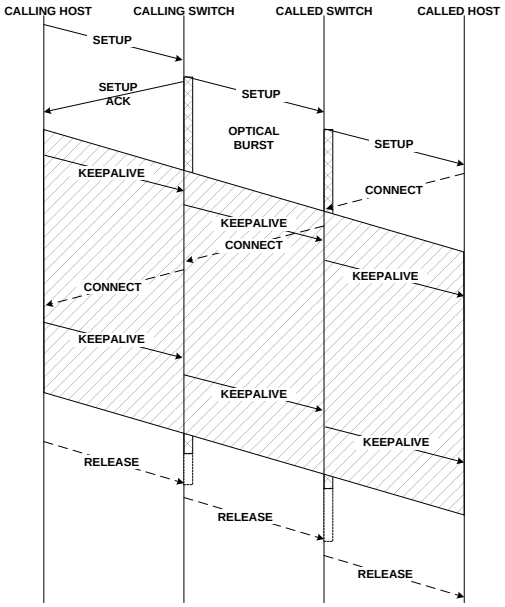

Figure 2. Lightpath

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 mentioned above 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.

3. EXTENDED FINITE STATE MODEL

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[6]). Therefore, we define an Extended Finite State Machine (EFSM) model with the addition of some variables. For further information on EFSMs, the interested readers may read ([2, 4, 5, 12]). In this model, each EFSM can be formally represented as a eight-tuple $(\sum, S, s, V, E, T, A, \delta)$ where:

$\sum$: 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:

$Sx \sum xE(V)xT \longrightarrow \sum xA(V)xS$

There are two types of transitions: spontaneous and 'when' transitions. A spontaneous transition doesn't have an input event on its condition part. A 'when' transition, on the other hand includes an input event satisfying the condition. A transition is shown $S_1 \overset{T}{\rightarrow} S_2$. This means there is a transition T at state S_1 and it goes to state S_2. 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 EFSM is shown in Figure 3.

In Figure 3, *?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

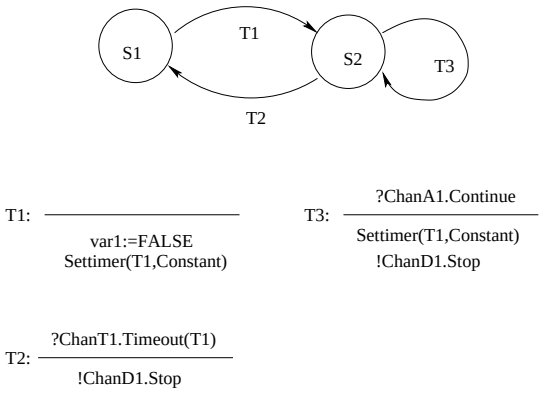

Figure 3. An Example EFSM Model

channels. As seen in Figure 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. *T2* and *T3*, on the other hand 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.[12] EFSMs communicate with each other by message passing through a number of first-in-first-out (FIFO) unidirectional queues (channels), which associate with some buffers at the endpoints of the corresponding EFSMs respectively.

4. SYSTEM ARCHITECTURE

The relationship between different protocol entities are explained using the system architecture illustrated in Figure 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.

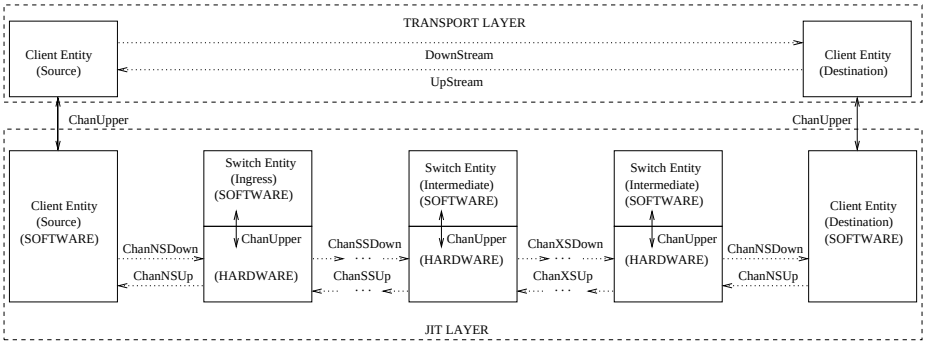

Figure 4. Protocol Stack Architecture

Client nodes are implemented by software, therefore there is not a hardware part attached to client nodes. However, switch entities have both a software and a hardware parts and the connection between these two parts are represented by ChanUpper because the state diagrams related with switch entities show the behavior of hardware.

5. EFSM-BASED FORMAL SPECIFICATIONS OF JUMPSTART JIT PROTOCOL

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 unicast 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 are:

$$\sum = \begin{array}{c} \{Open, Setup, Failure, Timeout, Setup\_Ack, \\ Connection\_Failure, Close, Release, Clear\_To\_Send, \\ Connect, Transmission\_Complete, Keepalive\} \end{array} \quad (1)$$

Open is generated by the Transport Layer to notify JIT Layer incoming of a burst. *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 *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. *Release* message is used for explicit teardown. *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 are:

$$S = \begin{array}{c} \{IDLE, WAIT\_FOR\_SETUP\_ACK, \\ SETUP\_PROCEEDING, DATA\_TRANSMISSION, \\ WAIT\_FOR\_CONNECT\} \end{array} \quad (2)$$

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 *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*. 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 *WAIT_FOR_CONNECT* state and waits until Connect comes or Conn_Timer times out.

Initial state s is the state $IDLE$. The set of variables are:

$$V = \begin{array}{l} \{SA\_Constant, \Delta T, Conn\_Constant, Conn\_Rcvd, \\ Rel, Conn, Burst\_Time, Burst\_Delay, KA\_Time\} \end{array} \quad (3)$$

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*. 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*, *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 are:

$$T = \{SA\_Timer, SETUP\_Timer, Conn\_Timer, Burst\_Timer, KA\_Timer\} \quad (4)$$

The set of actions that operate on variables are:

$$A = \{Settimer, Update\} \quad (5)$$

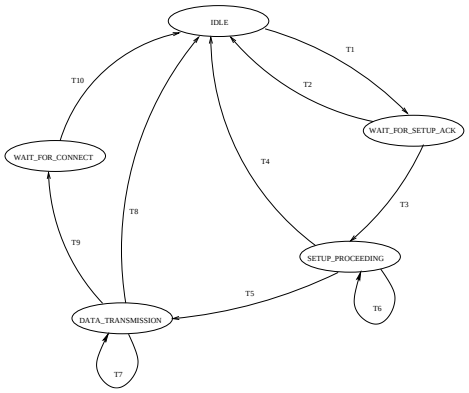

Figure 5. State Diagram for Source Client (Unicast)

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. *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 Figures 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 don't 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 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, 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 only wait for a *Connect* message to arrive before we close the connection.

The state transitions use four different channels shown in Figure 4: *ChanUpper*, *ChanNSUp*, *ChanNSDown*, and *ChanT1*. *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. *ChanT1* is the timer channel used to receive timeout messages from the indicated timers.

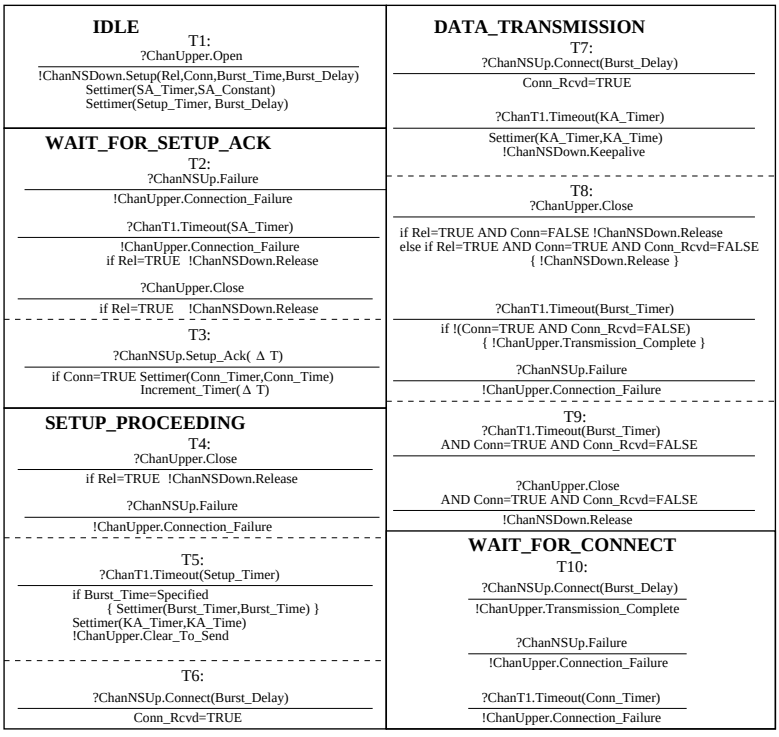

Figure 6. State Transitions for Source Client (Unicast)

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* 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 are:

$$\sum = \begin{array}{l} \{Open, Setup, Failure, Timeout, Setup\_Complete, Close, \\ Release, Connect, Transmission\_Complete, Keepalive\} \end{array} \quad (6)$$

*Setup*_Ack is not defined in this machine because the destination client does not receive nor generate an acknowledgment. The set of states S are:

$$S = \{IDLE, SETUP\_PROCEEDING, DATA\_TRANSMISSION\} \quad (7)$$

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 are:

$$V = \{Rel, Conn, Burst\_Time, Burst\_Delay, KA\_Time\} \quad (8)$$

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 are:

$$T = \{KA\_Timer, Burst\_Timer\} \quad (9)$$

The set of actions that operate on variables are:

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

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

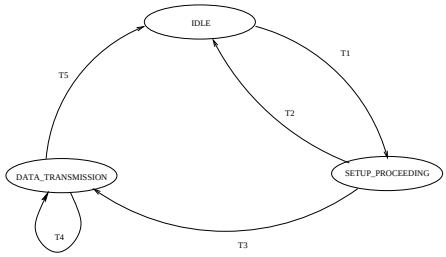

Figure 7. State Diagram for Destination Client (Unicast)

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

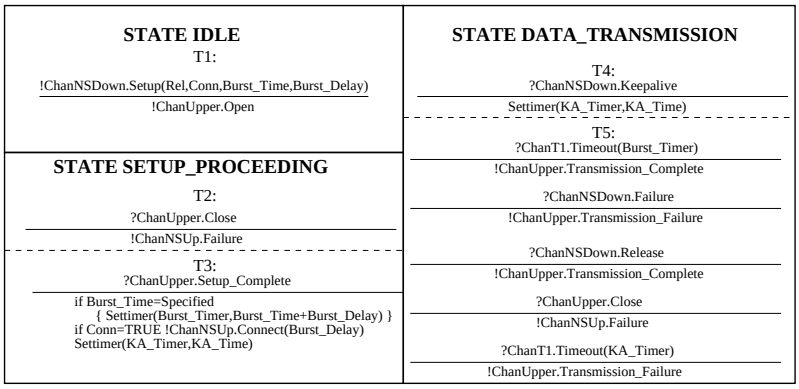

Figure 8. State Transitions for Destination Client (Unicast)

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 are:

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

The set of states S are:

$$S = \{IDLE, RUNNING\_CHECKS, DATA\_TRANSMISSION, WAIT\_FOR\_CONNECT\} \tag{12}$$

Unlike the source node, we don't 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 are:

$$V = \{Conn\_Rcvd, \Delta T, ErrorCode, Rel, Conn, Burst\_Time, Burst\_Delay, KA\_Time\} \tag{13}$$

We don't 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 are:

$$T = \{Burst\_Timer, Conn\_Timer, KA\_Timer\} \tag{14}$$

The set of actions that operate on variables are:

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

Setup message is sent by the source client. Once the *Setup* message is received, the ingress switch runs some checks, e.g. 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.

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
eme_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

Ingress switch does not use Rel variable line destination client. The state diagram and the state transitions are given in Figure 10.

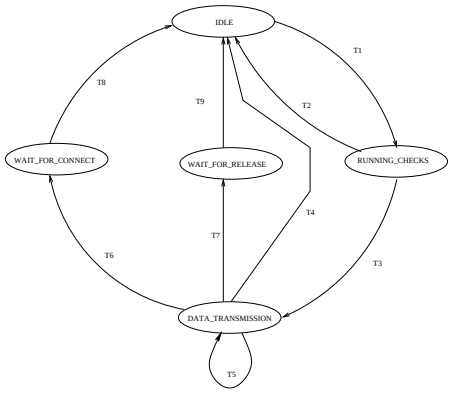

Figure 9. State Diagram for Ingress Switch (Unicast)

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 mentioned above. 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)

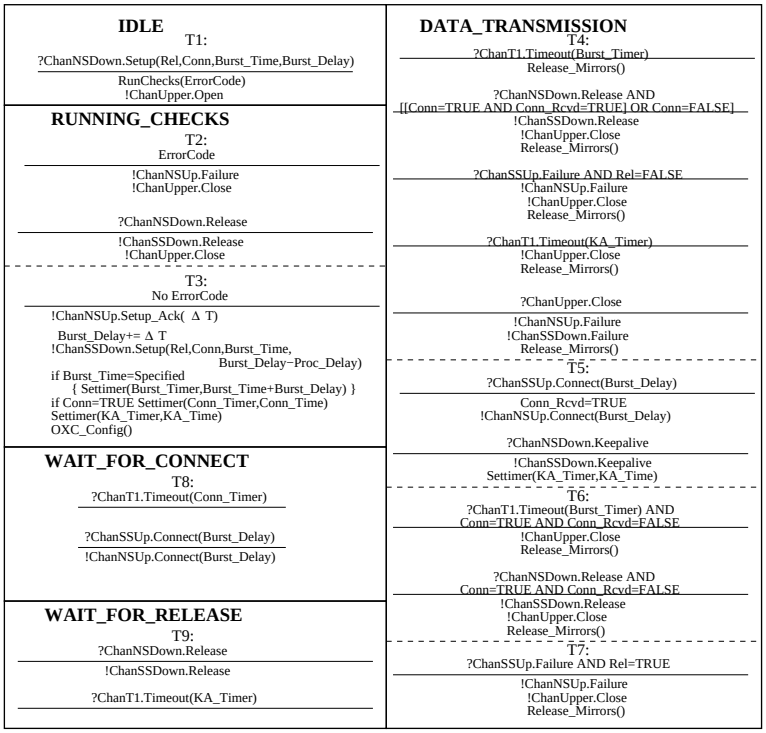

Figure 10. State Transitions for Ingress Switch (Unicast)

message. 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 we need to send a *Connect* message or not. 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 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 need to 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 *IDLE* state again by closing the connection and creating the Release. The two other possible transitions at *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 *ChanT1* 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.

[†]the variable Conn_Rcvd is TRUE only if a Connect message is received

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 Figure 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 is given in Figure 11. We will not give the set of messages, states, etc...as they are all the same with Ingress switch.

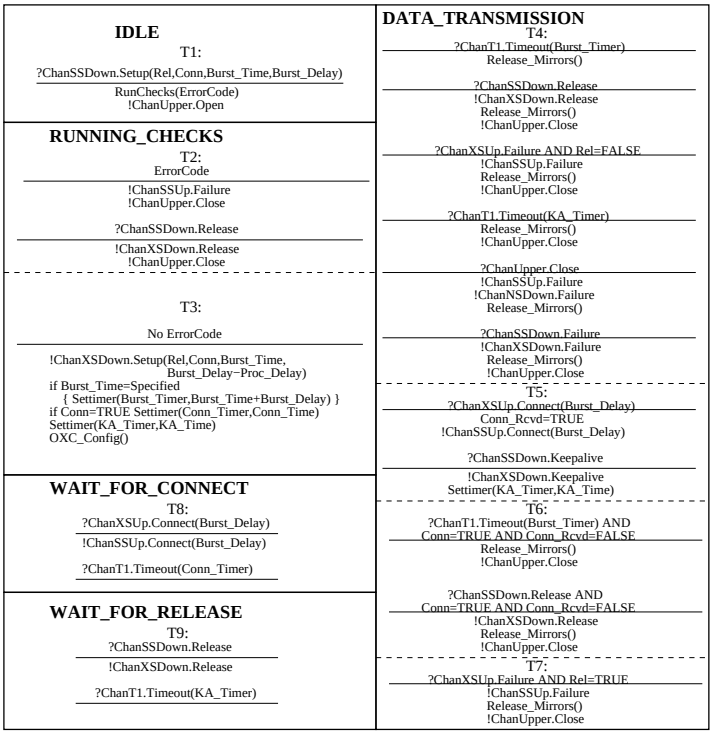

Figure 11. State Transitions for an Intermediate Switch (Unicast)

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 the diagrams belong to. Therefore these channels are defined anonymously. The channels between the intermediate switch and the switch connected to it is called as in earlier cases as ChanSS. The channel to the Software Layer is also ChanUpper and the timer channel is again ChanT1.

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 not an acknowledgment process between each entities. The rest of the state machine is similar to the Ingress switch's state machine.

6. CONCLUSION

In this paper, we present a formal description of the Jumpstart Just-in-time signaling protocol for unicast traffic. We defined unicast traffic flow for a single burst. In the single burst unicast connection, we set a connection only for sending one burst and then close the connection. The state diagrams and transitions for single burst unicast traffic flow are given in this paper. The future work includes the definition of persistent and multicast traffic which is part of the Jumpstart Just-in-time protocol specifications, and protocol testing based on the formal definitions.

REFERENCES

1. P.W. King, Formalization of Protocol Engineering Concepts, IEEE Transactions on Computers, vol.40, no.4, April, 1991.
2. H. Hansson, B. Jonsson, F.Orava, and B. Pehrson, Formal Design of Communication Protocols, ISS'90.
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. I. Baldine, G.N. Rouskas, H.G. Perros, D. Stevenson, Jumpstart: a just-in-time signaling architecture for WDM burst-switched networks, IEEE Communications Magazine , Volume: 40 Issue: 2 , Feb 2002 Page(s): 82 -89.
7. J.Y.Wei and R.I.McFarland,"Just-in-time Signaling for WDM Optical Burst Switching Networks", J.Lightwave Tech., vol.18,no.12,Dec.2000, pp.2019-37.
8. M.Yoo, C.Qiao and S.Dixit,"QoS Performance of Optical Burst Switching in IP-over-WDM Networks", JSAC, vol.18,no.10,Oct.2000, pp.2062-71.
9. J.S.Turner,"Terabit Burst Switching", J.High Speed Networks, vol.8,no.1,Jan.1999,pp.3-16.
10. C.Qiao and M.Yoo, "Optical Burst Switching (OBS)-A New Paradigm for an Optical Internet", J.High Speed Net.,vol.8,no.1, Jan.1999,pp.69-84.
11. M.Yoo and C.Qiao, "Just-enough-time (JET): A High Speed Protocol for Bursty Traffic in Optical Networks", IEEE/LEOS Tech. Global Info. Infra. Aug.1997, pp.26-27.
12. H. Bowman, G.S. Blair, L. Blair and A.G. Chetwynd, Formal Description of Distributed Multimedia Systems: An Assessment of Potential Techniques, Computer Communications, December, 1995.