AFRL-IF-RS-TR-2004-194
Final Technical Report
July 2004

REGIONAL TESTBED OPTICAL ACCESS NETWORK (PROJECT HELIOS)

MCNC-RDI

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J182

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

APPROVED: /s/
ROBERT L. KAMINSKI
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** July 2004	**3. REPORT TYPE AND DATES COVERED** **FINAL** **Feb 00 – Jun 03**

4. TITLE AND SUBTITLE

REGIONAL TESTBED OPTICAL ACCESS NETWORK (PROJECT HELIOS)

5. FUNDING NUMBERS

C - F30602-00-C-0034
PE - 62301E
PR - J182
TA - 23
WU - 01

6. AUTHOR(S)

Illia Bladine, Laura E. Jackson, Mrugendra Singhai, Dan Stevenson, Steve Thorpe, George Rouskas, Dhaval Thaker, Sudhin Bengeri, Marc J. Beacken, John C. Goetjen, Henry Fuchs, Kostas Daniilidis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MCNC-RDI
P O Box 13910
3021 Cornwallis Road
Research Triangle Park NC 27709-3910

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFG
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRLIF-RS-TR-2004-194

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert L. Kaminski/IFG/(315) 330-1865 Robert.Kaminski@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

This final technical report presents the results and conclusions from a set of research objectives posed by the Helios project. The objectives included: The study of all-optical LAN architectures and protocols; the establishment of a testbed for the purpose of demonstrating high-bandwidth applications and collecting and analyzing statistical profiles of their traffic; and the study of propagation of analog signals through all-optical networks. The report is structured as follows: Sections 2, 3 and 4 describe MCNC-RDI's work on the HiPeR-1 protocol and scheduler for broadcast WDM LAN architectures. Section 5 describes the application testbed established between MCNC-RDI, NCSU and UNC-CH using NCNI infrastructure for the purpose of accumulating statistical information about traffic demands of high-bandwidth applications. Section 6 describes the application cluster established within MCNC-RDI in order to test high-bandwidth/high-compute demand applications. Section 7 describes the research performed at Lucent Bell Labs concerned with transporting analog signals over DWDM optical links including physical layer impairments and adaptation layer studies. Section 8 describes the work performed jointly by UNC-CH and the University of Pennsylvania to establish a multimedia network testbed for telepresence applications. The Appendices present some of the technical details related to the HiPeR-1 scheduling protocol developed and implemented by MCNC-RDI.

14. SUBJECT TERMS

Optical Networks, Local Area Networks, Network Protocols, Dense Wave Length Division Multiplexing

15. NUMBER OF PAGES 59

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Introduction 1

2 dWDM Broadcast and Select MAC 5 2
- 2.1 Overview of Helios Protocol 2
- 2.2 Constants and Parameters 2
- 2.3 High Level Node Design 2
  - 2.3.1 In Software: the Driver 4
  - 2.3.2 In Hardware: the Adapter 4
- 2.4 Frames and Superframes 4
  - 2.4.1 A Superframe: The Implementation of a Schedule 4
  - 2.4.2 Frame Format 5
- 2.5 Helios Network Operation 9
  - 2.5.1 Helios Modes of Operation 10
  - 2.5.2 Time Synchronization 12
  - 2.5.3 The Signaling Controller 14
  - 2.5.4 Election 17
  - 2.5.5 Time Measurement 19
  - 2.5.6 Join 21
  - 2.5.7 Routine 23
  - 2.5.8 Scheduling 27
  - 2.5.9 Error Detection, Reporting, and Recovery (ERR) Mode 27
- 2.6 New Scheduler Approaches 31
  - 2.6.1 Helios Greedy Scheduling Algorithm 31

3 Multicast and QoS support in all-optical broadcast LANs 34
- 3.1 Multicast in the Helios Architecture 34
- 3.2 QoS Support in the Helios Architecture 34
- 3.3 The Scheduling andWavelength Assignment Problem 36

4 Helios Protocol Testbed 37
- 4.1 Testbed setup 37
- 4.2 Software 37

5 Helios Applications Testbed 39
- 5.1 METRO AREA DWDM NETWORK 39
- 5.2 APPLICATIONS 40

6 VCC Collaboration and Weather Modeling 42
- 6.1 Virtual Collaborative Center Overview 42
- 6.2 Just In Time Optical Burst Switching Protocol Software Development 42
  - 6.2.1 GridFTP over JIT 42
  - 6.2.2 Grid Information Retrieval 42
- 6.3 High Performance Computing and Collaborative Applications 43
- 6.4 AN-MSI Collaboration Center 43
- 6.5 Helios/VCC Testbed Integration 43
  - 6.5.1 Adding an Additional Compute Node to the VCC Testbed 43
  - 6.5.2 Condor Installation and Configuration 45
- 6.6 Executing Environmental Simulations 46

Appendix A: HiPeR-1 Timers and Counters 48
Appendix B: HiPeR-1 Signals 49
Appendix C: HiPeR-1 Constants 51
Appendix D: HiPeR-1 Shared Memory 52

Appendix E: HiPeR-1 Guardbands........53
References........54

List of Figures

1 A Helios network with 8 nodes connected to the Passive Star Coupler........2
2 Benchmark parameter values........3
3 A high-level hardware view of the Helios NIC hardware........4
4 Superframe for a network with N=7, C=4; λ3 is the receive wavelength for the master node N1........5
5 Structure of a frame in the Helios network........6
6 SYNCSCHED Frame payload........7
7 ARP frame payload........8
8 OCC frame payload........9
9 Logical representation of memory........11
10 Software state machine signaling_controller........13
11 Receiver hardware state machine for scheduler election: >elect<........18
12 Transmitter hardware state machine for scheduler election: <elect>........18
13 Receive hardware state machine for time measurement: >tm<........20
14 Transmit hardware state machine for time measurement: <tm>........20
15 Receive hardware state machine for join: >join<........21
16 Transmit hardware state machine for join: <join>........22
17 Receive hardware state machine for candidate and slave nodes: >routine<........23
18 Transmit hardware state machine for candidate and slave nodes: <routine>........26
19 Receive hardware state machine for the master node: >scheduling<........28
20 Transmit hardware state machine for the master node: <scheduling>........29
21 Performance of Helios scheduler........32
22 Datagram flow........37
23 Application Testbed Topology........39
24 Cluster Topology........44
25 Weather Model........47

List of Tables

1 Frame types and functions........5
2 Field lengths in Helios frames........6
3 Flags in the Helios frame header........6
4 SYNCSCHED frame payload fields........8
5 SYNCSCHED frame flags........8
6 ARP frame payload fields........8
7 OCC frame payload fields........9
8 The modes of operation for a Helios node........10
9 Example traffic matrix........31
10 Minimum frame size vs. changing parameter values........53

1 Introduction

This report presents the results and conclusions from a set of research objectives posed by the Helios project. The objectives included

- The study of all-optical LAN architectures and protocols
- The establishment of a testbed for the purpose of demonstrating high-bandwidth applications and collecting and analyzing statistical profiles of their traffic
- The study of propagation of analog signals through all-optical networks

MCNC-RDI was the prime contractor on this project, with Lucent Bell Labs, North Carolina State University (NCSU) and University of North Carolina at Chapel Hill (UNC CH) serving as sub-contractors.

This document consists of a number of sections each dedicated to a specific part of the Helios project. For convenience, this report references external documents, which are submitted with it. The breakdown of research areas between the participants of the project was as follows:

MCNC RDI: prime contractor. Study and implementation of an all-optical broadcast LAN protocol named HiPeR-l. It included the design and specification of the protocol and implementation of the protocol and the associated unicast scheduler in an emulated environment.

NCSU: subcontractor. Study of multicast and Quality of Service (QoS) extensions to the HiPeR-l scheduler.

UNC CH: subcontractor. Establishment and testing of a high-performance/high bandwidth demand multimedia testbed

University of Pennsylvania: subcontractor. Establishment and testing of a high-performance/high bandwidth demand multimedia testbed

Lucent Bell Labs: subcontractor. Study of the effects of propagation of analog RF signals over all-optical networks as well as necessary adaptation layers

The report is structured as follows:

Sections 2, 3 and 4 describe MCNC-RDI's work on the HiPeR-l protocol and scheduler for broadcast WDM LAN architectures. Section 5 describes the Application testbed established between MCNC-RDI, NCSU and UNC-CH using NCNI infrastructure for the purpose of accumulating statistical information about traffic demands of high-bandwidth applications. Section 6 describes the application cluster established withing MCNC-RDI in order for the purpose of testing high-bandwidth/high-compute demand applications.

The Appendices present some of the technical details related to the HiPeR-l scheduling protocol developed and implemented by MCNC-RDI.

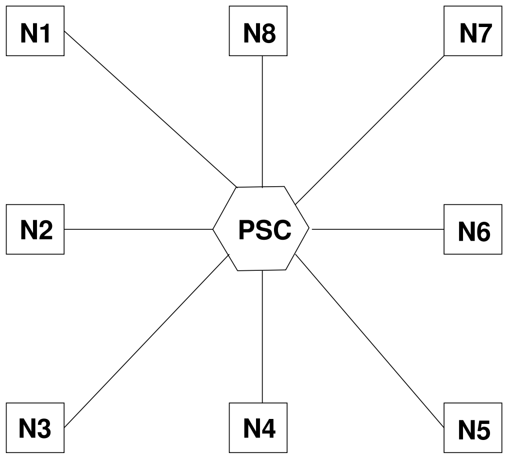


Figure 1: A Helios network with 8 nodes connected to the Passive Star Coupler

2 dWDM Broadcast and Select MAC

2.1 Overview of Helios Protocol

This chapter describes the details of the design of the HiPeR-l protocol as implemented in the Helios project. A Helios/HiPeR-l network can consist of a (potentially large) number of end hosts (referred to as nodes), which are connected to a Passive Star Coupler (PSC), a passive all-optical device that allows us to create a broadcast environment in the network without employing expensive electro-optical interfaces. Communication between nodes will occur on multiple wavelengths; thus the Helios network is a type of single-hop WDM network. Each Helios node will be equipped with an optical NIC to facilitate control communication between the nodes in the network. All communication will be done using either IPv4 or IPv6 protocol. The number of wavelengths utilized by the HiPeR-l protocol is assumed to be smaller than the number of nodes in the network.

Communication in a Helios network is collision-free due to the use of a non-preemptive gated scheduling protocol. A scheduling node calculates and disseminates the schedule. There are two types of nodes in a Helios network: candidate nodes, which are eligible to serve as the scheduling node, and slave nodes, which are not. We make this distinction because we can envision a network composed of servers and workstations, where the workstations may lack the necessary computing resources to perform the scheduling node's duties. Furthermore, workstations may allow low priority access, making them vulnerable to security attacks that could disrupt the network.

The Helios network utilizes a Fast Tunable Transmitter - Slowly Tunable Receiver (FTT-STR) approach; for packet transmission and scheduling purposes the lasers are tunable and the receivers are fixed. However, the receivers can be retuned occasionally in order to balance the load in the network. Helios differs from all other WDM networks currently under development in several respects: it operates within a broadcast-and-select environment, it is collision-free, and it is packet-switched instead of circuit-switched.

2.2 Constants and Parameters

Refer to the Appendices for a complete list of important constants and parameters. Figure 2 gives the values for the most common parameters. Those without a formula are constants, while the rest can be calculated from the constants.

2.3 High Level Node Design

Figure 3 depicts the high-level design of a Helios network adapter, highlighting the various hardware, software and firmware modules and their interactions. On the transmit side, data packets are received by the driver and forwarded to the transmit path of the adapter. On the receive side, frames received by the optical module are differentiated into

Name	Value	Units	Formula	Definition
N	200			Number of nodes in network
C	10			Number of wavelengths in network
c	300000	km/s		Speed of light in a vacuum
Lmax	250	m		Max length of fiber span in network
Cf	200000	km/s	2c/3	Speed of light in fiber
MAX_PSCO	1.25	microsec	Lmax/Cf	Maximum PSC offset
ND	2.500	microsec	2(Lmax)x10^6 / (Cf x 10^3)	Network Diameter
tick	12	ns		Clock tick
LS	1	Gb/s		Line speed
pkt	600	octets		Packet size
slot	4.8	microsec	8(pkt)x10^6 / (LSx10^9)	Slot length
TL	9	microsec		Tuning latency between 2 lambdas
	1125	octets	(TLx10^-6)(LSx10^9) / 8	
	2	slots	ceil(TL/slot)	
schedchunk	5	octets	1 octet for lambda + 2 octets for T_start + 2 octets for T_last_slot	Single node-lambda schedule length
min{S+S}	50	octets	C*schedchunk	Min length of SYNC+SCHED frame
mem	32	MB		Total node memory for packet queues
glb{Smax_true}	256	ms	8(mem) / LS	GLB on True Max Superframe Length
Smin	30	slots	C(ceil(TL/pkt) + ceil(min{S+S}/pkt))	Min superframe length
	144	microsec		
Smax	42667	slots	ceil(.8 * glb{Smax_true} x 10^3 / slot)	Max superframe length (80%)
	204.8	ms	.8 * glb{Smax_true}	

Figure 2: Benchmark parameter values

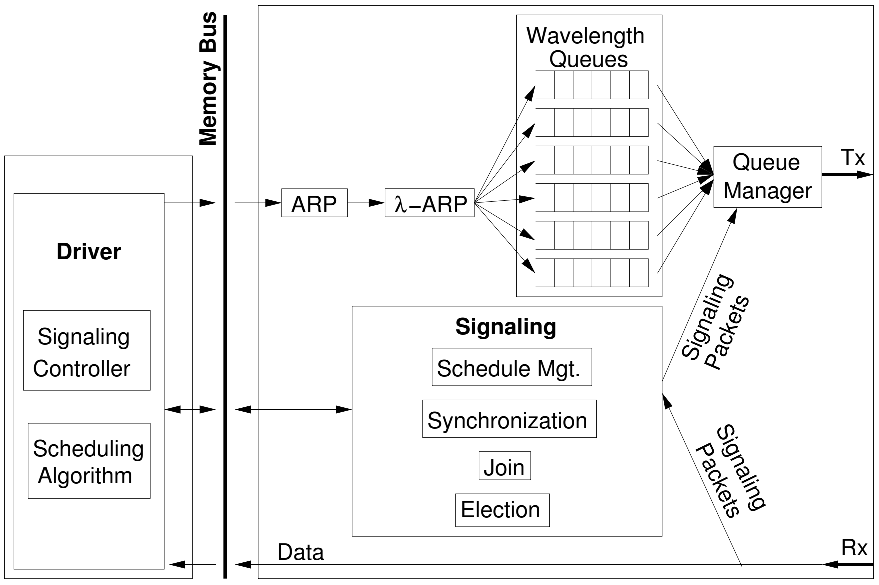


Figure 3: A high-level hardware view of the Helios NIC hardware

signaling and data packets and forwarded either to the signaling module or to the driver for processing. The following sections describe in more detail the functionality of each module.

2.3.1 In Software: the Driver

The **Driver** module consists of two sub-modules. The **Signaling Controller** coordinates the operation of all other software and hardware modules. The **Scheduling Algorithm** calculates new schedules based on queue occupancies provided by all the nodes in the network; it is called relatively infrequently, either in response to changes in the traffic pattern or simply periodically.

2.3.2 In Hardware: the Adapter

The **Signaling** module of the adapter contains four sub-modules that govern the necessary signaling actions: **Schedule Management** forms and processes frames related to scheduling, **Synchronization** enables all communication to occur in hard real time, **Join** contains the procedure for a node to join a `Helios` network, and **Election** is invoked when a master node fails and all candidate nodes take part in the election of a new one. Section 2.5 describes the signaling protocol in detail.

The **ARP** and **λ-ARP** tables enable a `Helios` node to perform IP-to-MAC address resolution and MAC-to-receive-wavelength resolution, respectively. The master node keeps track of the ARP and **λ-ARP** mappings and distributes them via ARP frames to all other nodes. Outgoing IP packets are buffered in the **Wavelength Queues** on a per-wavelength basis prior to transmission. The **Queue Manager** serves the wavelength queues and controls which frames are transmitted.

2.4 Frames and Superframes

2.4.1 A Superframe: The Implementation of a Schedule

The time required to complete the transmissions of one full schedule in HiPeR-l is referred to as a superframe. A superframe further consists of frames, which are continuous sequences of octets transmitted by nodes on individual wavelengths. Helios uses non-preemptive schedules; in other words, within each superframe a node transmits on a particular wavelength at most once.

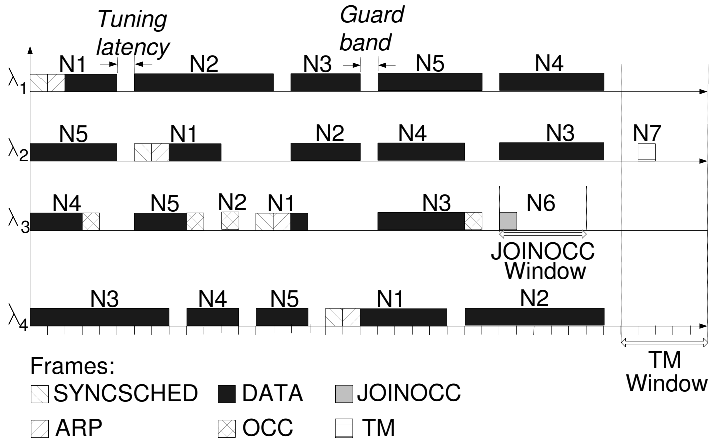

Tuning latency
Guard band
λ_1 N1 N2 N3 N5 N4
λ_2 N5 N1 N2 N4 N3 N7
λ_3 N4 N5 N2 N1 N3 N6
JOINOCC Window
λ_4 N3 N4 N5 N1 N2
TM Window
Frames:
SYNCSCHED
DATA
JOINOCC
ARP
OCC
TM

Figure 4: Superframe for a network with N=7, C=4; λ_3 is the receive wavelength for the master node N1.

Frame	Function	Frame Type
DATA	Carries regular data	0x01
MDATA	Carries multicast data	0x02
TM	Measures roundtrip delay to the PSC	0x03
OCC	Transmits queue occupancies to the scheduling node	0x04
JOINOCC	Identical to OCC except for the join_flag in the header	0x05
SYNCSCHED	Carries scheduling information to the nodes	0x06
ARP	Carries MAC address to wavelength index mapping (λARP)	0x07
OAM	Carries error and management information about network state	0x08
AVAIL[1,2]	Announces the availability of a scheduling server to become a scheduling node during scheduler election process	0x09

Table 1: Frame types and functions

The master node calculates the schedule based on other nodes' packet queue occupancies, which it learns through the OCC frames sent by other nodes during routine network operation. Once calculated, the schedule is then broadcast on each wavelength inside the SYNCSCHED frame, which the master node transmits on every wavelength every superframe. A schedule contains **windows**, or intervals of time, during which a particular node may transmit a frame.

Figure 4 shows the position of various frames within a superframe. In this example, N1 is the master node and its receive wavelength is λ3. There is a JOINOCC window on λ3 (with a JOINOCC frame in it), and there is an attached TM window at the end of the superframe. Two nodes are in different stages of joining the network: N6 is sending a JOINOCC frame containing its queue occupancy information to the master node so that it can be included in the next schedule. Meanwhile, N7 is performing Time Measurement; its TM frame can be seen inside the TM window. Time measurement is the first operation a new node must perform when joining the network, in order to synchronize frame reception and transmission. The complete list of the types of frames that may be transmitted during a superframe are shown in Table 1. In the following section each frame type is discussed in detail.

2.4.2 Frame Format

Each frame consists of a header, a variable length payload, and a trailer. Frame structure is illustrated in Figure 5, and each field is described in Table 2. The header contains the Frame Type indicator, one octet of flags, the payload length indicator, and the source and destination addresses. The trailer contains a timestamp and a CRC32 checksum field. The last column of Table 1 shows the possible values that can be placed in the frame type field of the Helios frame header.

Table 2 also shows the length of each field in a Helios frame. The payload length field is allocated two octets for

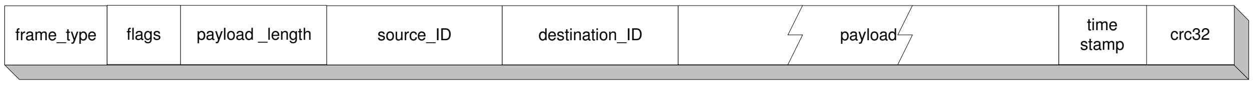


Figure 5: Structure of a frame in the Helios network.

Field Name	Description	Field Length (in octets)
`frame_type`	Frame type indicator	1
`flags`	Frame flags	1
`payload_length`	Indicates the length of the frame payload	2
`source_ID`	MAC address of the originator of the frame	16
`destination_ID`	MAC address of the destination of the frame	16
`payload`	contains frame payload	variable ≤ 556
`time_out`	Time stamp which marks the departure time of the frame	4
`crc32`	CRC32 checksum of the entire frame	4
Total		*600*

Table 2: Field lengths in Helios frames

future expansion, when it will be possible to use packets longer than 600 octets (the current maximum transfer unit). The flags field contains a number of flags, shown in Table 3, that are used by the nodes to indicate the state of the protocol.

2.4.2.1 Frame Addressing The Helios addressing scheme is compatible with both IPv4 and IPv6 address formats to allow direct mapping of addresses from those protocols into the Helios MAC addresses. IPv6 addresses can be mapped directly onto the Helios MAC addresses and used as a replacement for MAC addresses. IPv4 addresses will require padding as described in Section 2.5.4 of RFC2373. In short, an IPv4 address will be represented as eighty 0's, followed by sixteen 1's, followed by the IPv4 address of the node interface.

Similarly, multicast addresses can be used as destination MAC addresses for multicast communications in Helios. Helios will utilize the link-local multicast addresses reserved in IPv4 and IPv6. For the all-nodes multicast group used internally for sending signaling messages, Helios will utilized the following values:

IPv4:	*`224.0.0.250`*
IPv6:	*`ff02:1`*

2.4.2.2 DATA Frame The DATA frame payload contains an IPv4 or IPv6 packet. Use of the timestamp field is optional.

2.4.2.3 MDATA Frame The MDATA frame payload contains an IPv4 or IPv6 multicast packet. Use of the timestamp field is optional.

Flag Name	Purpose
`more_frames`	Indicates that more frames of the same type will follow this frame (Used only by SYNCSCHED and ARP frames)
`join_flag`	Indicates that this frame is a JOINOCC frame (Used only by OCC frames)

Table 3: Flags in the Helios frame header

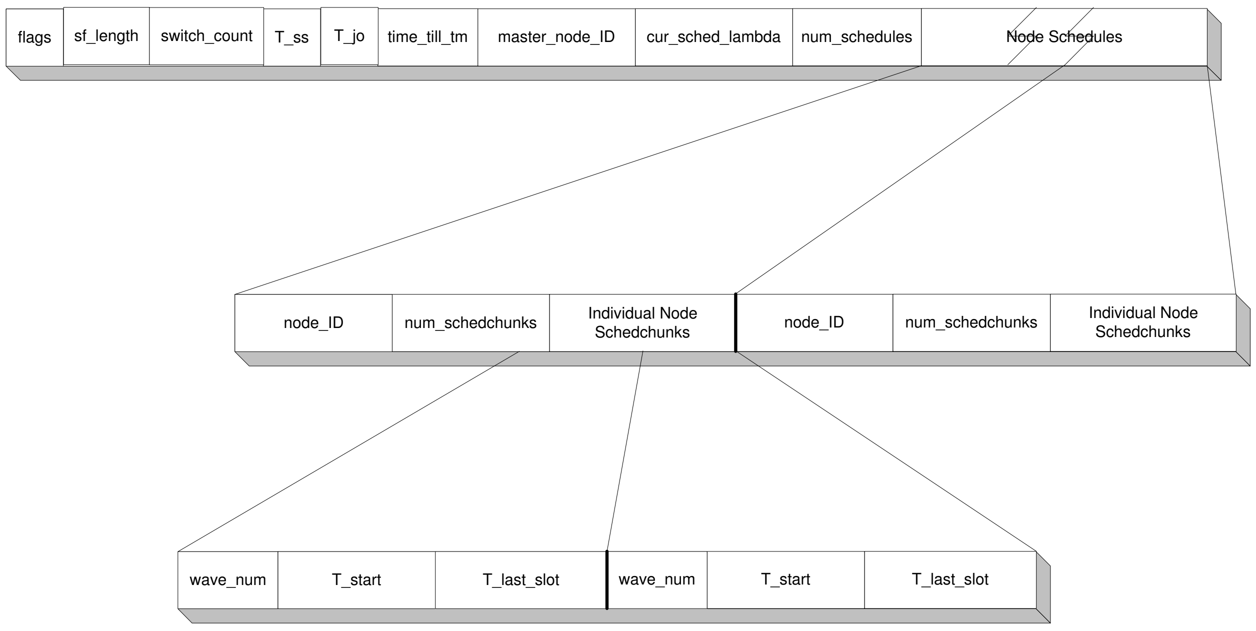


Figure 6: SYNCSCHED Frame payload

2.4.2.4 SYNCSCHED Frame The SYNCSCHED frame is sent to the all-nodes link-local multicast address (see 2.4.2.1). It carries node-specific scheduling information (shown in Figure 6) from the scheduling node to all nodes. SYNCSCHED frames transmitted on a particular wavelength λ_i will only contain node schedules for those nodes listening to λ_i. Upon receipt of a SYNCSCHED frame, each node stores its own schedule until the time comes to start using the new schedule. Special flags in the header indicate the transition phase from one schedule to the next.

In a network with a large number of nodes, the schedules for all the nodes listening on a particular wavelength may not fit into a single SYNCSCHED frame. In this case multiple consecutive instances of the SYNCSCHED frame are scheduled and transmitted on that wavelength. A node's schedule is never fragmented across frames; if a node's complete schedule cannot fit into the remainder of a SYNCSCHED frame, it is transmitted in the next one. These multiple instances of the SYNCSCHED frame are transmitted in sequence and non-preemptively. To indicate that more frames of the same type follow, the `more_frames` flag in the header is set in all consecutive frames except the last one.

Each SYNCSCHED frame consists of the Helios header, the SYNCSCHED payload, and the trailer. As shown in Figure 6, the payload itself consists of a header and the attached node schedules. Table 4 describes each field in the SYNCSCHED frame payload. The flags field of the SYNCSCHED frame contains several single-bit flags, which are described in Table 5.

2.4.2.5 ARP Frame The scheduling node transmits ARP frames on every wavelength in order to disseminate the MAC address, IP address, and wavelength ID for all nodes in the network. Each ARP frame carries an integral number of such mappings. If all available mappings do not fit into a single ARP frame, the scheduling node may schedule and transmit a number of ARP frames. Like multiple SYNCSCHED frames sent on one wavelength, the multiple ARP frames are transmitted in sequence and non-preemptively, and the `more_frames` flag in the header is set in all consecutive frames except the last one. But the transmission of ARP frames differs from that of SYNCSCHED frames in an important way: whereas the SYNCSCHED frames sent on one wavelength differ from the SYNCSCHED frames sent on another wavelength, the same ARP frames are transmitted on every wavelength.

Each ARP frame consists of the Helios header, the ARP payload, and the trailer. The structure of the ARP payload is shown in Figure 7. Table 6 contains a description and the field length for each field in the ARP frame payload.

2.4.2.6 TM Frame A TM window is a quiet time provided on each wavelength at the end of a schedule in order to allow new nodes to measure their delay to the PSC, called the `psc_offset`. A new node transmits a timestamped TM

Field Name	Description	Field Length (in octets)
`flags`	Current state of the schedule and protocol	1
`sf_length`	Length of the superframe/schedule in slots	2
`switch_count`	Countdown to the new schedule (in conjunction with active_bit)	1
`T_ss`	Offset (in slots, from the start of the superframe) of this SYNCSCHED frame	2
`time_till_tm`	Time (in slots) from the SYNCSCHED frame to the TM window (If the flags show the presence of a TM window in this superframe)	2
`T_jo`	Offset (in slots, from the start of the superframe) of the JOINOCC window	2
`master_node_ID`	Scheduling node's MAC address	16
`cur_sched_lambda`	Scheduling node's listening wavelength	1
`num_schedules`	Number of individual node schedules in this frame	1
`node_ID`	Address of the node for which the following schedule is intended	16
`num_schedchunks`	Number of schedchunks in the node's schedule	1
`wave_num`	ID of the wavelength for this schedchunk	1
`T_start`	Offset (in slots, from the start of the superframe) of the first slot in which the node may transmit on this wavelength	2
`T_last_slot`	Offset (in slots, from the start of the superframe) of the last slot in which the node may transmit on this wavelength	2

Table 4: SYNCSCHED frame payload fields

Flag name	Purpose
`tm_bit`	Indicates the presence (1) or absence (0) of a TM window in this superframe
`active_bit`	Indicates whether the information in this SYNCSCHED is for current (1) or future (0) use

Table 5: SYNCSCHED frame flags

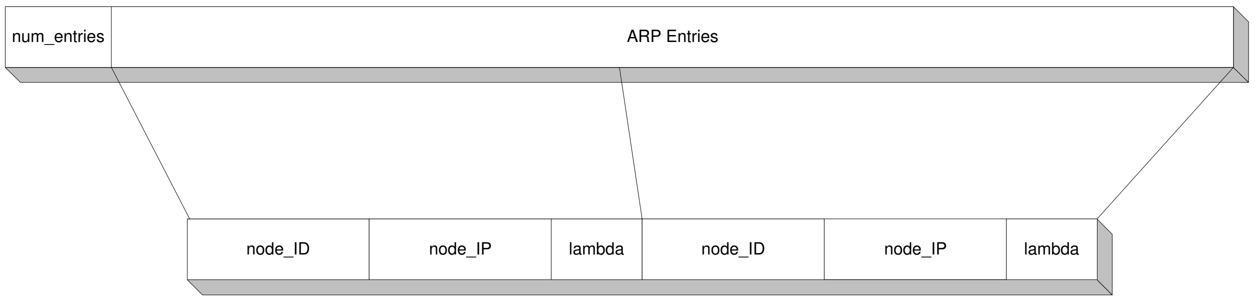


Figure 7: ARP frame payload

Field Name	Description	Field Length (in octets)
`num_entries`	Indicates the number of ARP entries in this frame	1
`node_ID`	Contains the MAC address of the node in the mapping	16
`node_IP`	Contains the IP address of the node in the mapping	16
`lambda`	Contains the wavelength number in the mapping	1

Table 6: ARP frame payload fields

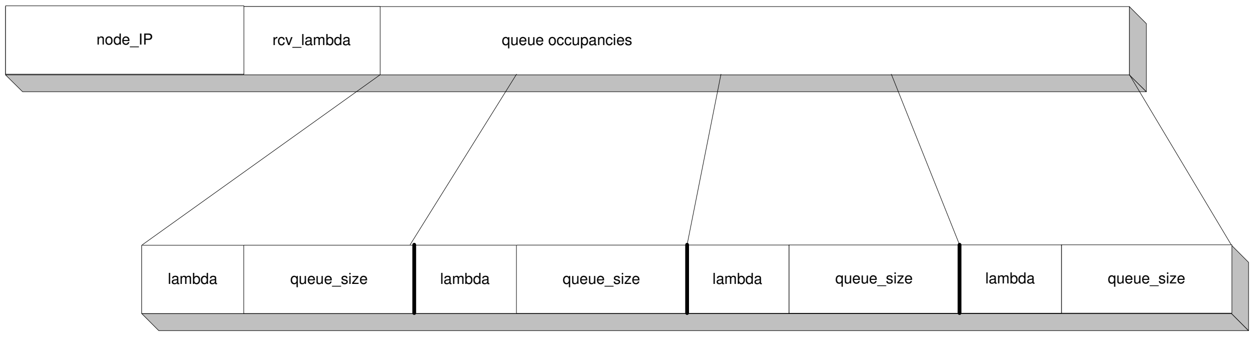


Figure 8: OCC frame payload

Field Name	Description	Field Length (in octets)
`node_IP`	IP address of the source node	16
`rcv_lambda`	Receive wavelength number of the source node	1
`lambda`	Wavelength number for the queue	1
`queue_size`	Queue size of the associated wavelength	2

Table 7: OCC frame payload fields

frame to itself during the TM window; the difference between the timestamp and the receipt time of the TM frame is the roundtrip delay to the PSC. The `psc_offset` is one-half the roundtrip time.

A TM frame consists of the Helios header, an empty payload, and the trailer.

2.4.2.7 OCC Frame Each node in the network informs the scheduling node of its packet queue occupancies by transmitting an OCC frame. Using this aggregate information, the scheduling node can produce a new schedule that better accomodates nodes' current load demands. The scheduling node must always reserve enough time on its receive wavelength for each node in the network to send its OCC frame.

An OCC frame consists of the Helios header, the OCC payload, and the trailer. The structure of the OCC payload is shown in Figure 8. Table 7 describes the fields of the OCC frame payload in detail.

2.4.2.8 JOINOCC Frame When a new node joins the Helios network, it sends a JOINOCC frame to the scheduler to indicate its presence. A JOINOCC frame is simply an OCC frame with the `join_flag` set in the Helios frame header. The main difference between the two frames is the time at which they are transmitted. A node in the Helios network routinely transmits an OCC frame during its allocated time on the scheduling node's receive wavelength. However, a new node that is not yet a part of the Helios network transmits a JOINOCC frame on the scheduling node's receive wavelength during the JOINOCC window in the schedule, given by the `T_jo` field of the SYNCSCHED frame.

2.4.2.9 AVAIL Frames A candidate node participating in scheduler election (Section 2.5.4) uses AVAIL[1,2] frames to indicate it is available to become the master node in the network. The frames consist of a standard Helios header, an empty payload, and a trailer. Use of the timestamp field is optional.

2.4.2.10 OAM Frame OAM frames are similar in spirit to OAM ATM cells. They carry additional management information between the nodes. The format and exact function of this frame type remains presently undefined.

2.5 Helios Network Operation

The operation of a node in the Helios network can be divided into six modes, as shown in Table 8. Corresponding to each mode of operation are two hardware state machines, the receive and the transmit hardware state machines.

Mode	Function	Hardware State Machines	
		Transmit	Receive
Time Measurement	a new node measures its propagation delay to the PSC	**`<tm>`**	**`>tm<`**
Join	a new node contacts the master node with its bandwidth requests	**`<join>`**	**`>join<`**
Election	a candidate node vies to become the master node	**`<elect>`**	**`>elect<`**
Routine	a node transmits according to the schedule	**`<routine>`**	**`>routine<`**
Scheduling	same as Routine, plus creates new schedules	**`<scheduling>`**	**`>scheduling<`**
ERR	error detection, reporting, and recovery		

Table 8: The modes of operation for a Helios node

Each machine begins and ends in the idle state: they are triggered out of the idle state by a signal from the software state machine called **`signaling_controller`**, and they usually terminate by sending a signal back to **`signaling_controller`** and returning to the idle state. Figure 9 collects most of the variables and parameters used in the hardware state machines and shows their logical locations in memory.

Following a brief overview of the different modes of operation in Section 2.5.1, we cover the important issue of timing in Section 2.5.2. Next we describe the **`signaling_controller`** in Section 2.5.3, and finally we discuss each mode in detail in Sections 2.5.4 through 2.5.9.

2.5.1 Helios Modes of Operation

When the network comes up after having been completely powered down, no master node has yet been designated, no frames are traveling, and no synchronization information is available. The first task during this initialization phase is the election of a master node; candidate nodes enter **Election Mode** while slave nodes sleep. The operation of Election Mode assumes that candidate nodes are equipped with slowly tunable receivers; otherwise, a network administrator must designate the master node.

Once a master node has been elected, it circulates the scheduling and synchronization information in SYNC-SCHED frames, enabling other nodes to join the network. A node formally joins the `Helios` network by proceeding through the **Time Measurement** and **Join** modes. In Time Measurement, a node calculates its `psc_offset`, the propagation delay to the PSC. All times are measured locally, and the transmissions are done in relation to the PSC time. Since collisions can occur only at the PSC, each node uses its `psc_offset` to ensure that its transmissions reach the PSC at the exact time prescribed by the schedule.

Following Time Measurement a node enters **Join Mode**. The node first lets the master node know of its traffic demands via the JOINOCC frame, so that the current schedule can be expanded to include this new demand. The joining node must then wait to hear a new schedule that includes its request.

It is possible for a collision to occur when two or more nodes attempt to join a `Helios` network at the same time. Two nodes assigned to the same listening wavelength could experience a collision during Time Measurement, or two nodes may transmit a JOINOCC frame to the master node during the same JOINOCC window. In either case, the collision does not interfere with the normal operation of the rest of the network; a collision during the transmission of a TM frame (respectively, a JOINOCC frame) is isolated to the TM window (respectively, the JOINOCC window). The protocol includes backoff algorithms to resolve such contention.

After successfully joining the network, a new node enters **Routine Mode**, where it remains indefinitely unless an error condition occurs. During Routine Mode, the receive hardware extracts the schedule from the arriving SYNC-SCHED frames and forwards incoming data frames to the driver. Meanwhile, the transmit hardware transmits control frames and data frames from its wavelength queues onto the appropriate outgoing wavelengths, according to the current schedule. These transmissions include sending an OCC frame to the master node, once per superframe, to communicate its packet queue occupancies; from the OCC frames the master node can calculate a schedule. In contrast to the Time Measurement and Join modes, Routine Mode is collision-free. The `psc_offset`, first measured during Time Measurement, is also measured periodically during Routine Mode, in a collision-free manner.

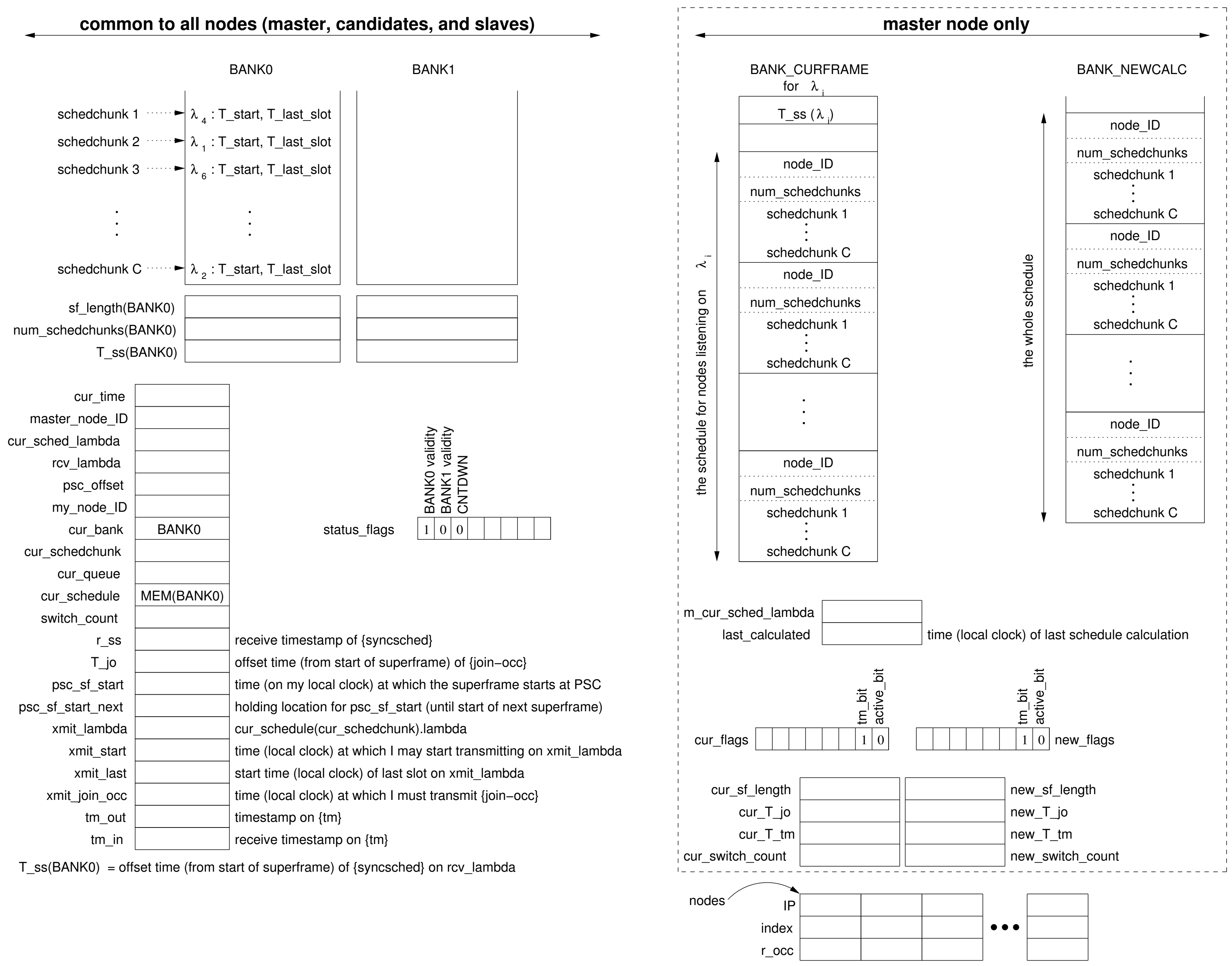

common to all nodes (master, candidates, and slaves)
BANK0
BANK1
schedchunk 1
schedchunk 2
schedchunk 3
schedchunk C
λ 4 : T_start, T_last_slot
λ 1 : T_start, T_last_slot
λ 6 : T_start, T_last_slot
λ 2 : T_start, T_last_slot
sf_length(BANK0)
num_schedchunks(BANK0)
T_ss(BANK0)
cur_time
master_node_ID
cur_sched_lambda
rcv_lambda
psc_offset
my_node_ID
cur_bank
BANK0
cur_schedchunk
cur_queue
cur_schedule
MEM(BANK0)
switch_count
r_ss
receive timestamp of {syncsched}
T_jo
offset time (from start of superframe) of {join−occ}
psc_sf_start
time (on my local clock) at which the superframe starts at PSC
psc_sf_start_next
holding location for psc_sf_start (until start of next superframe)
xmit_lambda
cur_schedule(cur_schedchunk).lambda
xmit_start
time (local clock) at which I may start transmitting on xmit_lambda
xmit_last
start time (local clock) of last slot on xmit_lambda
xmit_join_occ
time (local clock) at which I must transmit {join−occ}
tm_out
timestamp on {tm}
tm_in
receive timestamp on {tm}
T_ss(BANK0) = offset time (from start of superframe) of {syncsched} on rcv_lambda
BANK0 validity
BANK1 validity
CNTDWN
status_flags
1 0 0
master node only
BANK_CURFRAME
for λ i
T_ss (λ i)
node_ID
num_schedchunks
schedchunk 1
schedchunk C
the schedule for nodes listening on λ i
BANK_NEWCALC
the whole schedule
m_cur_sched_lambda
last_calculated
time (local clock) of last schedule calculation
tm_bit
active_bit
cur_flags
1 0
new_flags
cur_sf_length
cur_T_jo
cur_T_tm
cur_switch_count
new_sf_length
new_T_jo
new_T_tm
new_switch_count
nodes
IP
index
r_occ

Figure 9. Logical representation of memory

2.5.2 Time Synchronization

Time synchronization is necessary so that nodes can transmit data onto wavelengths according to a pre-established schedule, preventing collisions. Since collisions can only occur at the PSC, the PSC naturally lends itself to being a common point of reference. In order for a node's transmission to arrive at the PSC at the time prescribed by the schedule, a node must know how long its signal needs to travel to the PSC (its `psc_offset`). Each node calculates its `psc_offset` through Time Measurement (Section 2.5.5). The schedule disseminated by the master node lists the *offset times* that a node can transmit on the different wavelengths; each offset time is the relative time since the start of the superframe. Therefore, to use the schedule, a node must learn when the superframe will start at the PSC. This task is performed first during Join (Section 2.5.6) and again during each superframe as a part of Routine Mode (Section 2.5.7). In addition to the `psc_offset`, three other quantities are needed to calculate the start time of the superframe at the PSC; for a node with its receiver tuned to λ_i, these quantities are:

1. `T_ss` : the offset time that the SYNCSCHED frame is scheduled to appear on λ_i within the superframe. The node copies the value from `{ss.T_ss}` (a field in the SYNCSCHED frame) into the local variable `T_ss`.

2. `sf_length(cur_bank)` : the length of the superframe in slots. The node copies the value from `{ss.sf_length}` (a field in the SYNCSCHED frame) into the local variable `sf_length(cur_bank)` associated with the current schedule bank, `cur_bank`.

3. `r_ss` : the receive timestamp of the SYNCSCHED frame at the node (according to the node's clock). The node copies the current time from its own clock into the local variable `r_ss` at the instant the SYNCSCHED frame arrives.

Using these three quantities and the `psc_offset`, the beginning of the next superframe at the PSC, called `psc_sf_start_next`, is:

$$\texttt{psc\_sf\_start\_next} = \texttt{r\_ss} - \texttt{T\_ss} - \texttt{psc\_offset} + \texttt{sf\_length}$$

The quantity `psc_sf_start_next` is updated during Routine Mode by the receive hardware (**`>routine<`** for slaves and candidates; **`>scheduling<`** for the master node). When the transmit hardware reaches the end of the schedule and is ready to start transmissions in the next superframe, it copies the value held in `psc_sf_start_next` into `psc_sf_start`. This mechanism prevents the current value from being overwritten when the start time for the next superframe is calculated.

2.5.2.1 Transmitting frames on time

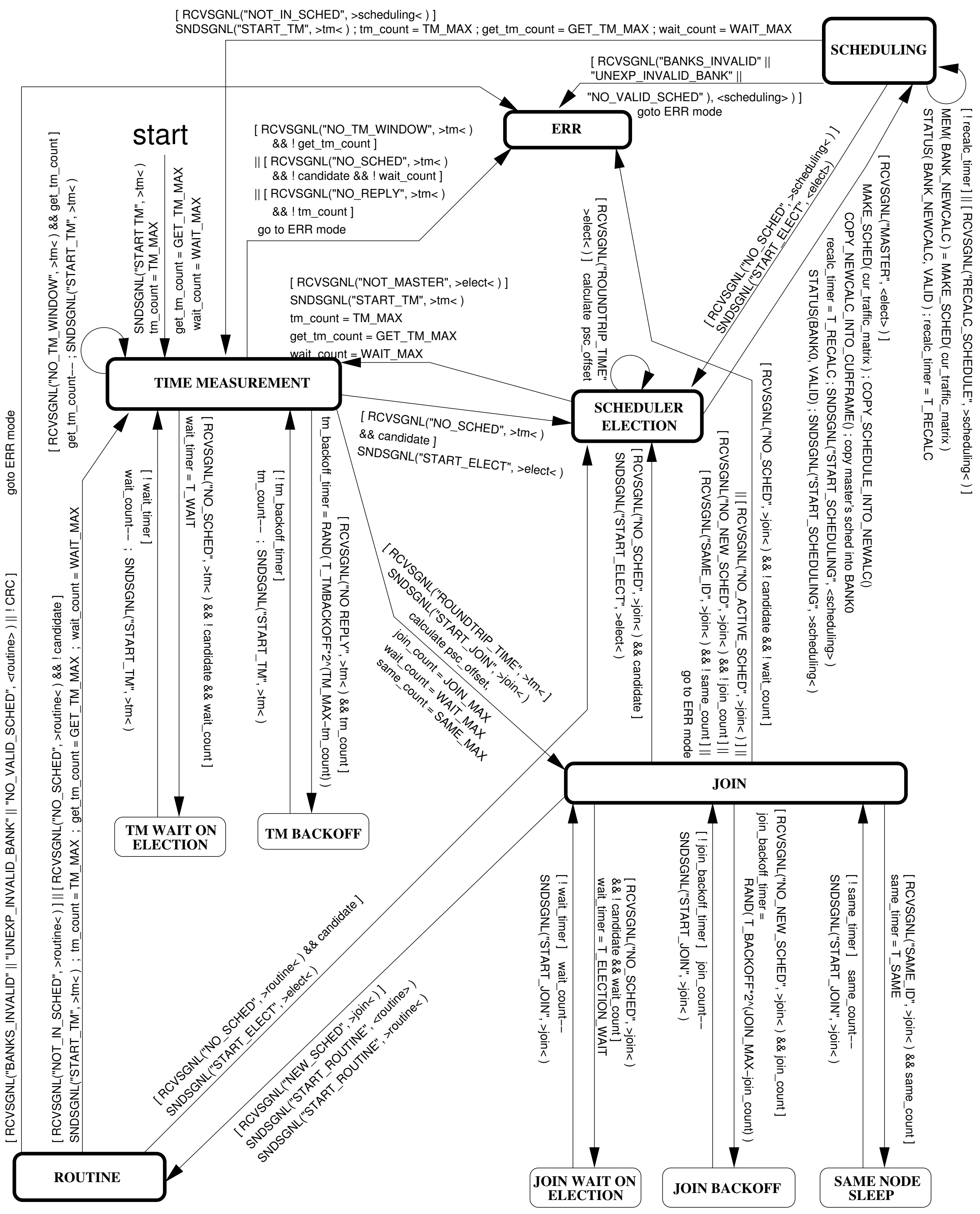
We now show how this calculation will aid in the transmission of data frames according to the schedule. Suppose that the schedule lists the offset time (again, relative to the start of the superframe) that node A can begin transmitting on λ_i to be `T_i_start`. That is, node A's transmission must arrive at the PSC at `t_sf_start + T_i_start`. In order for its transmission to reach the PSC at this time, node A must begin transmitting at time `psc_sf_start + T_i_start - psc_offset`.

2.5.2.2 Network size and the need for pipelining

Because of the relatively small dimensions of the Helios network we need not worry about the SYNCSCHED frame arriving too late to be used for the next superframe's transmission (recall that the reception of the current superframe and the transmission of the next superframe overlap in time). Consider the following example:

Let the link speed for a single wavelength be 1 Gbps and network radius be 250 m. Assuming the speed of light in fiber is approximately 200,000 km/s, the amount of data stored in one radius of a maximum-sized network will be on the order of 1000 bits, which is smaller than a single slot (see constants). This figure must be doubled since the maximum `psc_offset` is equal to the network radius. Therefore a node must begin the transmissions of the next superframe within one slot's time of the end of the current superframe. As long as the SYNCSCHED frame never appears as the last frame on a given wavelength, the node will have enough time to calculate the start of the next superframe.

In the future, when the network dimensions and speed increase, we can use pipelining to get around this "simultaneity" problem.



[RCVSGNL("NOT_IN_SCHED", >scheduling<)]
SNDSGNL("START_TM", >tm<) ; tm_count = TM_MAX ; get_tm_count = GET_TM_MAX ; wait_count = WAIT_MAX
SCHEDULING
[RCVSGNL("BANKS_INVALID" || "UNEXP_INVALID_BANK" || "NO_VALID_SCHED"), <scheduling>)] goto ERR mode
ERR
start
[RCVSGNL("NO_TM_WINDOW", >tm<) && ! get_tm_count] || [RCVSGNL("NO_SCHED", >tm<) && ! candidate && ! wait_count] || [RCVSGNL("NO_REPLY", >tm<) && ! tm_count] go to ERR mode
SNDSGNL("START TM", >tm<) tm_count = TM_MAX get_tm_count = GET_TM_MAX wait_count = WAIT_MAX
[RCVSGNL("NO_TM_WINDOW", >tm<) && get_tm_count] get_tm_count-- ; SNDSGNL("START_TM", >tm<)
[RCVSGNL("NOT_MASTER", >elect<)] SNDSGNL("START_TM", >tm<) tm_count = TM_MAX get_tm_count = GET_TM_MAX wait_count = WAIT_MAX
TIME MEASUREMENT
[RCVSGNL("ROUNDTRIP_TIME" >elect<)] calculate psc_offset
[RCVSGNL("NO_SCHED", >scheduling<)] SNDSGNL("START_ELECT", <elect>)
[RCVSGNL("MASTER", <elect>)] MAKE_SCHED(cur_traffic_matrix) ; COPY_SCHEDULE_INTO_NEWALC() COPY_NEWCALC_INTO_CURFRAME() ; copy master's sched into BANK0 recalc_timer = T_RECALC ; SNDSGNL("START_SCHEDULING", <scheduling>) STATUS(BANK0, VALID) ; SNDSGNL("START_SCHEDULING", >scheduling<)
[! recalc_timer] || [RCVSGNL("RECALC_SCHEDULE", >scheduling<)] MEM(BANK_NEWCALC) = MAKE_SCHED(cur_traffic_matrix) STATUS(BANK_NEWCALC, VALID) ; recalc_timer = T_RECALC
SCHEDULER ELECTION
[RCVSGNL("NO_SCHED", >tm<) && candidate] SNDSGNL("START_ELECT", >elect<)
goto ERR mode
[! wait_timer] wait_count-- ; SNDSGNL("START_TM", >tm<)
[RCVSGNL("NO_SCHED", >tm<) && ! candidate && wait_count] wait_timer = T_WAIT
[! tm_backoff_timer] tm_count-- ; SNDSGNL("START_TM", >tm<)
[RCVSGNL("NO REPLY", >tm<) && tm_count] tm_backoff_timer = RAND(T_TMBACKOFF*2^(TM_MAX-tm_count))
[RCVSGNL("ROUNDTRIP_TIME", >tm<] SNDSGNL("START_JOIN", >join<) calculate psc_offset, join_count = JOIN_MAX wait_count = WAIT_MAX same_count = SAME_MAX
[RCVSGNL("NO_SCHED", >join<) && candidate] SNDSGNL("START_ELECT", >elect<)
[RCVSGNL("NO_SCHED", >join<) && ! candidate && ! wait_count] || [RCVSGNL("NO_ACTIVE_SCHED", >join<)] || [RCVSGNL("NO_NEW_SCHED", >join<) && ! join_count] || [RCVSGNL("SAME_ID", >join<) && ! same_count] || go to ERR mode
TM WAIT ON ELECTION
TM BACKOFF
JOIN
[RCVSGNL("BANKS_INVALID" || "UNEXP_INVALID_BANK" || "NO_VALID_SCHED", <routine>) || ! CRC]
[RCVSGNL("NOT_IN_SCHED", >routine<)] || [RCVSGNL("NO_SCHED", >routine<) && ! candidate] SNDSGNL("START_TM", >tm<) ; tm_count = TM_MAX ; get_tm_count = GET_TM_MAX ; wait_count = WAIT_MAX
[RCVSGNL("NO_SCHED", >routine<) && candidate] SNDSGNL("START_ELECT", >elect<)
[RCVSGNL("NEW_SCHED", >join<)] SNDSGNL("START_ROUTINE", <routine>) SNDSGNL("START_ROUTINE", >routine<)
[! wait_timer] wait_count-- SNDSGNL("START_JOIN", >join<)
[RCVSGNL("NO_SCHED", >join<) && ! candidate && wait_count] wait_timer = T_ELECTION_WAIT
[! join_backoff_timer] join_count-- SNDSGNL("START_JOIN", >join<)
[RCVSGNL("NO_NEW_SCHED", >join<) && join_count] join_backoff_timer = RAND(T_BACKOFF*2^(JOIN_MAX-join_count))
[! same_timer] same_count-- SNDSGNL("START_JOIN", >join<)
[RCVSGNL("SAME_ID", >join<) && same_count] same_timer = T_SAME
ROUTINE
JOIN WAIT ON ELECTION
JOIN BACKOFF
SAME NODE SLEEP

Figure 10: Software state machine **signaling_controller**

2.5.3 The Signaling Controller

The state machine **`signaling_controller`** is shown in Figure 10. There are six main states in **`signaling_controller`**, corresponding to the six modes of operation: Time Measurement, Scheduler Election, Join, Routine, Scheduling, and ERR. In addition, there are five minor states: two are related to Time Measurement (Wait On Election and TM Backoff), while the other three are related to Join (Wait On Election, Join Backoff, and Same Node Sleep). We now describe each state in turn.

2.5.3.1 Time Measurement State The **`signaling_controller`** is awaiting a signal from the hardware state machine **`>tm<`**. A successful signal is:

- ROUNDTRIP_TIME from **`>tm<`**.

  Event: **`>tm<`** has obtained the needed data (`tm_in` an `tm_out`) so that **`signaling_controller`** can calculate the `psc_offset`.

  Transition: Move to the Join state and start **`>join<`**.

Unsuccessful signals are:

- NO_TM_WINDOW from **`>tm<`**.

  Event: None of the SYNCSCHED frames that **`>tm<`** encountered (out of NO_TM_MAX SYNCSCHED frames) indicated that a tm window was present in the superframe.

  Transition: If this failure has occurred less than GET_TM_MAX times, restart **`>tm<`** (i.e., self-transition); else, move to ERR state.

- NO_SCHED from **`>tm<`**.

  Event: **`>tm<`** failed to hear a SYNCSCHED within T_GET_SCHED.

  Transition: If a candidate node, move to Scheduler Election state; else, if this failure has occurred less than WAIT_MAX times, move to Wait On Election state; else, move to ERR state.

- NO_REPLY from **`>tm<`**.

  Event: **`>tm<`** failed to hear the echo of its tm frame.

  Transition: If this failure has occurred less than TM_MAX times, move to TM Backoff state; else, move to ERR state.

2.5.3.2 Scheduler Election State The **`signaling_controller`** is awaiting a signal from the hardware state machine **`>elect<`** or **`<elect>`**. A successful signal is:

- SCHEDULER from **`<elect>`**.

  Event: **`<elect>`** has just transmitted the second AVAIL frame, winning the election.

  Transition: Move to the Scheduling state and start **`>scheduling<`**.

An unsuccessful signal is:

- NOT_SCHEDULER from **`>elect<`**.

  Event: **`>elect<`** heard something (a SYNCSCHED or an AVAIL frame) which caused it to lose the election.

  Transition: Move to Time Measurement state and start **`>tm<`**.

2.5.3.3 Join State The **`signaling_controller`** is awaiting a signal from the hardware state machine **`>join<`**. A successful signal is:

- NEW_SCHED from **`>join<`**.

  Event: **`>join<`** has received a SYNCSCHED frame that contains its own `my_node_ID`.

  Transition: Move to Routine state and start **`<routine>`** and **`>routine<`**.

Unsuccessful signals are:

- NO_SCHED from **`>join<`**.

  Event: **`>join<`** failed to hear a SYNCSCHED within T_GET_SCHED.

  Transition: If a candidate node, move to Scheduler Election state; else, if this failure has occurred less than WAIT_MAX times, move to Wait On Election state; else, move to ERR state.

- NO_NEW_SCHED from **`>join<`**.

  Event: **`>join<`** failed to hear a SYNCSCHED that included scheduling information for `my_node_ID`.

  Transition: If this failure has occurred less than JOIN_MAX times, move to Join Backoff state; else, move to ERR state.

- SAME_ID from **`>join<`**.

  Event: **`>join<`** heard a SYNCSCHED that included scheduling information for `my_node_ID`.

  Transition: If this failure has occurred less than SAME_MAX times, move to Same Node Sleep state; else, move to ERR state.

- NO_ACTIVE_SCHED from **`>join<`**.

  Event: None of the SYNCSCHED frames that **`>join<`** encountered (out of INACTIVE_MAX SYNCSCHED frames) had the `active_bit` set.

  Transition: Move to ERR state.

2.5.3.4 Routine State The **`signaling_controller`** could remain in this state indefinitely, while **`<routine>`** transmits according to the schedule and **`>routine<`** receives incoming data and signaling frames. Only an unsuccessful signal (triggered by an error condition in either **`<routine>`** or **`>routine<`**) will cause a transition out of the Routine state. Possible unsuccessful signals are:

- BANKS_INVALID from **`<routine>`**.

  Event: **`<routine>`** was just started, but was unable to transmit anything at all, because BANK0 was invalid.

  Transition: Move to ERR state.

- UNEXP_INVALID_BANK from **`<routine>`**.

  Event: **`<routine>`** completed transmissions for the current schedule (held in `cur_bank`) and discovered that cur_bank had been marked invalid.

  Transition: Move to ERR state.

- NO_VALID_SCHED from **`<routine>`**.

  Event: **`<routine>`** was ready to switch from an out-of-date schedule to a new one, but discovered that the reserve memory bank was marked invalid.

  Transition: Move to ERR state.

- NO_SCHED from **`>routine<`**.

  Event: **`>routine<`** failed to hear a SYNCSCHED within T_GET_SCHED.

  Transition: Move to ERR state.

- NOT_IN_SCHED from **`>routine<`**.

  Event: >routine< heard a SYNCSCHED that failed to include scheduling information for `my_node_ID`.

  Transition: Move to ERR state.

2.5.3.5 Scheduling State The **`signaling_controller`** could remain in this state indefinitely, while **`<scheduling>`** transmits according to the schedule and **`>scheduling<`** receives incoming data and signaling frames. Only an unsuccessful signal (triggered by an error condition in either **`<scheduling>`** or **`>scheduling<`**) will cause a transition out of the Scheduling state. Possible unsuccessful signals, listed below, are identical to those listed for Routine State in Section 2.5.3.4, except that they come from the state machines **`<scheduling>`** or **`>scheduling<`**.

- BANKS_INVALID from **`<scheduling>`**.

  Event: **`<scheduling>`** was just started, but was unable to transmit anything at all, because BANK0 was invalid.

  Transition: Move to ERR state.

- UNEXP_INVALID_BANK from **`<scheduling>`**.

  Event: **`<scheduling>`** completed transmissions for the current schedule (held in `cur_bank`) and discovered that `cur_bank` had been marked invalid.

  Transition: Move to ERR state.

- NO_VALID_SCHED from **`<scheduling>`**.

  Event: **`<scheduling>`** was ready to switch from an out-of-date schedule to a new one, but discovered that the reserve memory bank was marked invalid.

  Transition: Move to ERR state.

- NO_SCHED from **`>scheduling<`**.

  Event: **`>scheduling<`** failed to hear a SYNCSCHED within T_GET_SCHED.

  Transition: Move to ERR state.

- NOT_IN_SCHED from **`>scheduling<`**.

  Event: **`>scheduling<`** heard a SYNCSCHED that failed to include scheduling information for `my_node_ID`.

  Transition: Move to ERR state.

2.5.3.6 ERR State ERR is a terminal, absorbing state; an error message is printed to the screen and then the **`signaling_controller`** halts.

2.5.3.7 TM Wait On Election State The **`signaling_controller`** arrives at this state from the Time Measurement state because no schedules are heard and the node is a slave (i.e., cannot participate in scheduler election). The **`signaling_controller`** remains here for a time T_ELECTION_WAIT. There is only one opportunity to exit TM Wait On Election state:

Event: The `wait_timer` expires.

Transition: Move to the Time Measurement state.

2.5.3.8 TM Backoff State The **`signaling_controller`** arrives at this state from the Time Measurement state, because it did not hear the echo of its TM transmission, possibly due to a collision. The **`signaling_controller`** remains here for a random amount of time (exponential backoff). There is only one opportunity to exit TM Backoff state:

Event: The `tm_backoff_timer` expires.

Transition: Move to the Time Measurement state.

2.5.3.9 Join Wait On Election State The **`signaling_controller`** arrives at this state from the Join state because no schedules are heard and the node is a slave (i.e., cannot participate in scheduler election). The **`signaling_controller`** remains here for a time T_ELECTION_WAIT. There is only one opportunity to exit Join Wait On Election state:

Event: The `wait_timer` expires.

Transition: Move to the Join state.

2.5.3.10 Join Backoff State The **`signaling_controller`** arrives at this state from the Join state, because it did not hear a SYNCSCHED frame containing scheduling information for `my_node_ID`. The **`signaling_controller`** remains here for a random amount of time (exponential backoff). There is only one opportunity to exit Join Backoff state:

Event: The `join_backoff_timer` expires.

Transition: Move to the Join state.

2.5.3.11 Same Node Sleep State The **`signaling_controller`** arrives at this state from the Join state, because before the node could join the network, a SYNCSCHED was heard containing scheduling information for `my_node_ID`, possibly meaning that another node in the network possesses the same node ID. The **`signaling_controller`** remains here for a time T_SAME. There is only one opportunity to exit Same Node Sleep state:

Event: The `same_timer` expires.

Transition: Move to the Join state.

2.5.4 Election

Whenever a candidate node fails to detect the presence of a master node, *i.e.* no SYNCSCHED frames are heard within a pre-determined amount of time, then the candidate node enters Election Mode. This situation can occur when the network comes up after having been completely powered down, or when an operational master node suddenly fails.

Slave nodes, in the meantime, are capable neither of serving as a master node nor of participating in the election of one. Therefore, whenever a slave node fails to detect the presence of a master node, it enters a sleep state for a short time. Upon emerging, it listens for SYNCSCHED frames that indicate the presence of a master node, and if none is heard, it sleeps again. A slave node may re-enter the sleep state a fixed number of times before giving up (and moving to Error Mode).

Election Mode, as it is described below, assumes that candidate nodes are equipped with slowly tunable receivers. If candidate nodes are only equipped with fixed receivers, then a network administrator must designate the master node.

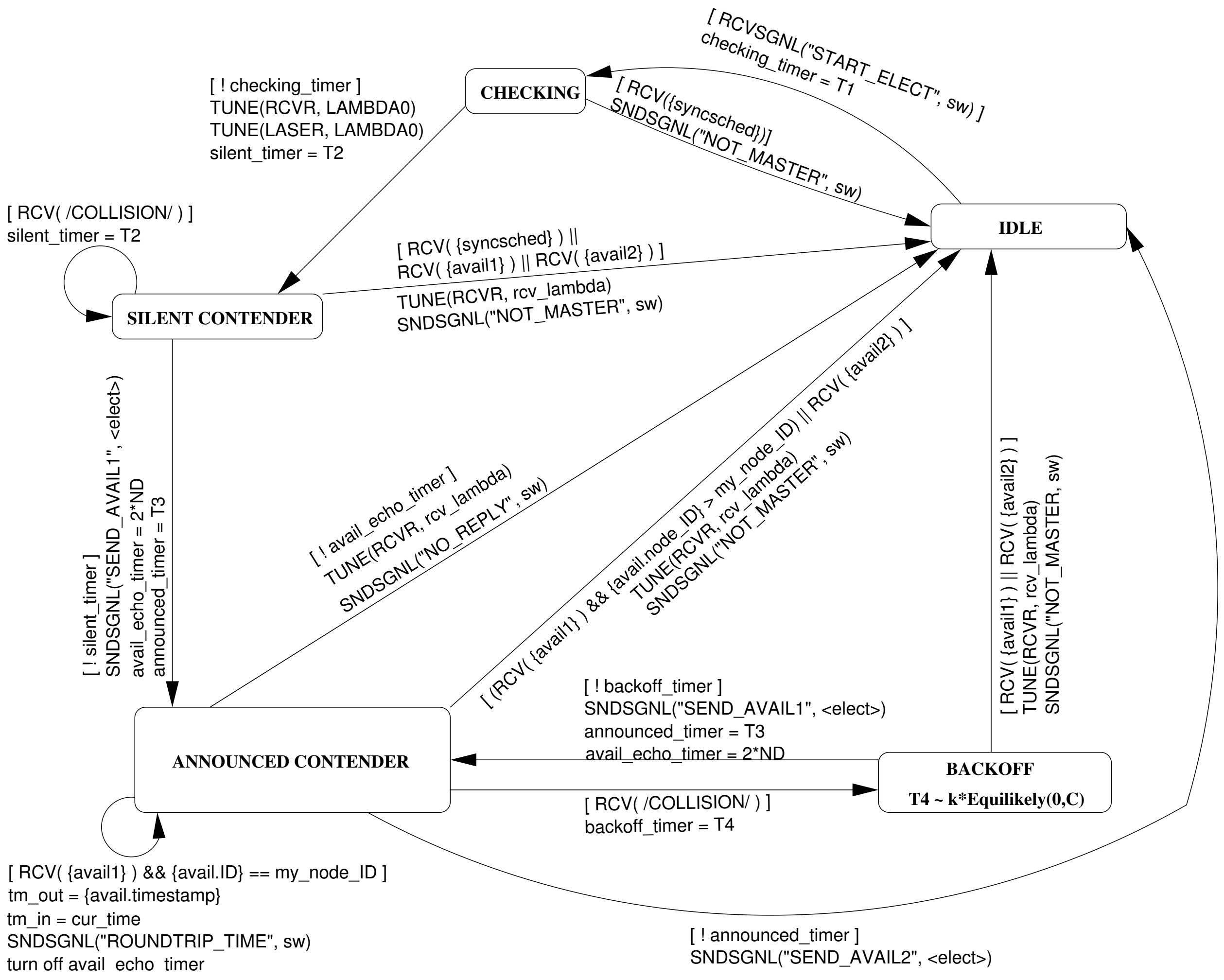


Figure 11: Receiver hardware state machine for scheduler election: **`>elect<`**

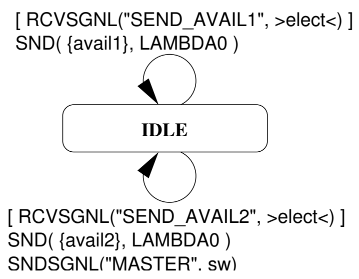


Figure 12: Transmitter hardware state machine for scheduler election: **`<elect>`**

2.5.4.1 Scheduler Election with Slowly Tunable Receivers Scheduler Election is illustrated in the receive and transmit hardware state machines **>elect<** and **<elect>**. A software signal to **>elect<** begins Scheduler Election, moving the state machine from IDLE to CHECKING state. The node listens on its original wavelength for a SYNCSCHED frame, which would indicate the presence of a master node. If none is heard within a time T1, the node moves to SILENT CONTENDER, tuning both its receiver and its transmitter to λ_0. There, it listens for either a SYNCSCHED frame, indicating the presence of a master node, or an AVAIL frame, indicating that another node is in the ANNOUNCED-CONTENDER state; in either case, the node drops out of Scheduler Election and becomes a non-scheduling node. As such, the node must wait to see SYNCSCHED frames generated by the newly elected master node and then join the network by proceeding through Time Measurement and Join modes.

If neither a SYNCSCHED nor an AVAIL is heard within a time T2, the node transmits an AVAIL1 frame on λ_0 and, after hearing its own transmission, becomes an ANNOUNCED-CONTENDER. Now it listens on λ_0 for a time T3; so long as the node hears no AVAIL with a higher-valued MAC address (({avail.node_ID}) during the interval T3, it will win the election and become the master node.

However, while in the ANNOUNCED-CONTENDER state, the node could hear an AVAIL with a higher-valued MAC address. In this case, the node will take itself out of the election and become a non-scheduling node; the other node with the higher MAC address has precedence in the scheduler election process.

If, on the other hand, our node detects a collision while in the ANNOUNCED-CONTENDER state, it enters the BACKOFF state for a random amount of time (T4). Other nodes involved in the collision will also enter the BACKOFF state, each choosing a different T4. The node whose T4 expires first will try again to transmit AVAIL1. (If there's a tie, a collision occurs and the involved nodes return to the BACKOFF state.) Any successfully transmitted AVAIL will cause the nodes waiting in BACKOFF to become non-scheduling nodes.

To prevent two or more nodes from mistakenly believing they have emerged victorious from Scheduler Election, the time durations T2 and T3 must obey a particular relationship. Recall that ND is defined to be the longest one-way propagation time between any two nodes. Then we have the following relationship:

$$2 * \mathtt{ND} < \mathtt{T3} < \mathtt{T2}$$

(First Inequality) If more than one node is an ANNOUNCED-CONTENDER, then this inequality ensures that the node with the highest-valued MAC address will win. (in particular, it ensures that all nodes with lower-valued MAC addresses will wait long enough in state ANNOUNCED-CONTENDER to hear the AVAIL from the node with highest address.)

(Second Inequality) Suppose node B is busy retuning its receiver to λ_0, transitioning from CHECKING to SILENT-CONTENDER, and that the retuning is completed just after node A's AVAIL1 has passed by. Then this inequality will ensure that node B will hear node A's AVAIL2 before node B becomes an ANNOUNCED-CONTENDER itself.

2.5.4.2 Time Measurement within Scheduler Election When a node reaches the SILENT-CONTENDER state, both its transmitter and receiver are tuned to λ_0. When a node then transmits AVAIL1, it becomes an ANNOUNCED-CONTENDER and sets the announced_timer for T3. Since the node should hear the echo of its own AVAIL1 transmission (provided its receiver is functional), it takes advantage of this opportunity to execute Time Measurement, that is, to calculate its psc_offset. The longest amount of time a node would have to wait to hear the echo is ND. But the announced_timer requires that the node remain in the ANNOUNCED-CONTENDER state for a time T3 before becoming the master node. Therefore, the avail_echo_timer should be set for a time longer than ND but less than T3. Since the inequality $2 * \mathtt{ND} < \mathtt{T3}$ must hold (Section 2.5.4.1), then we could set the avail_echo_timer for $2 * \mathtt{ND}$.

If the AVAIL echo is heard, the avail_echo_timer is turned off. Otherwise, the avail_echo_timer will expire before the announced_timer expires, causing the node to abort Scheduler Election and then move into the ERR Mode. Bundling Time Measurement with Scheduler Election produces a master node that knows its psc_offset and has a functioning transmitter and receiver.

2.5.5 Time Measurement

When a new node wishes to join a functioning Helios network, it must first synchronize its system time with that of the network through a process called Time Measurement. Next, the node must execute the Join process, which lets the master node know of its presence so that the current schedule can be expanded to include the new node's traffic

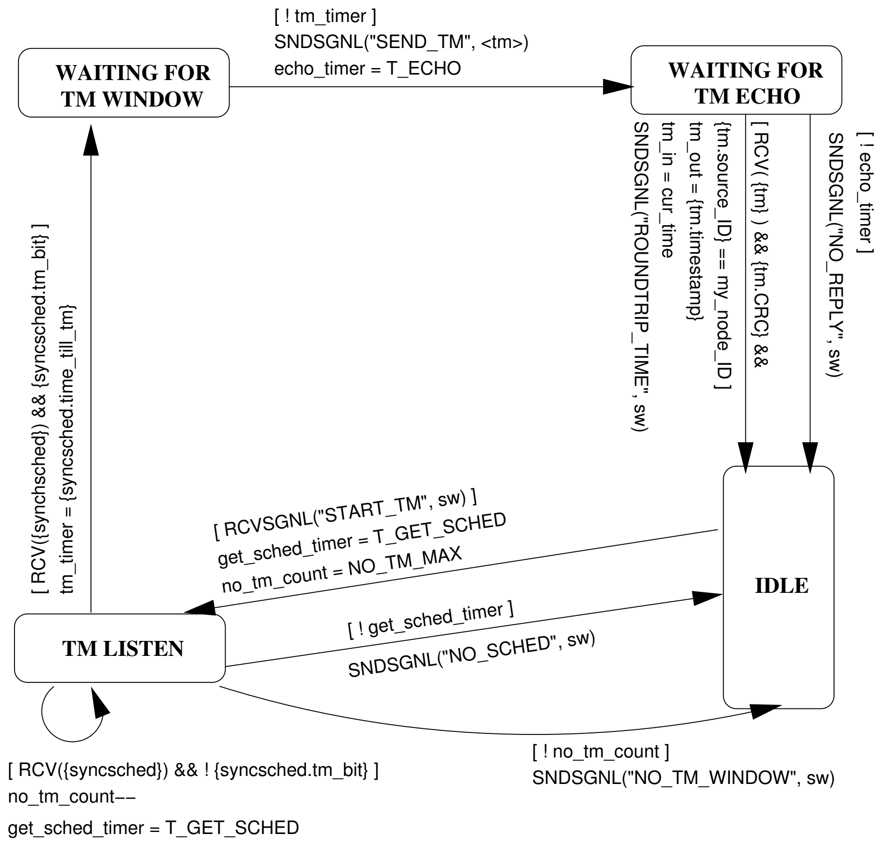

WAITING FOR TM WINDOW
WAITING FOR TM ECHO
[! tm_timer]
SNDSGNL("SEND_TM", <tm>)
echo_timer = T_ECHO
[RCV({tm}) && {tm.CRC} &&
{tm.source_ID} == my_node_ID]
tm_out = {tm.timestamp}
tm_in = cur_time
SNDSGNL("ROUNDTRIP_TIME", sw)
[! echo_timer]
SNDSGNL("NO_REPLY", sw)
[RCV({synchsched}) && {syncsched.tm_bit}]
tm_timer = {syncsched.time_till_tm}
IDLE
[RCVSGNL("START_TM", sw)]
get_sched_timer = T_GET_SCHED
no_tm_count = NO_TM_MAX
TM LISTEN
[! get_sched_timer]
SNDSGNL("NO_SCHED", sw)
[! no_tm_count]
SNDSGNL("NO_TM_WINDOW", sw)
[RCV({syncsched}) && ! {syncsched.tm_bit}]
no_tm_count−−
get_sched_timer = T_GET_SCHED

Figure 13: Receive hardware state machine for time measurement: **>tm<**

demands. Figure 13 and Figure 14show the receive and transmit hardware state machines for time measurement, respectively.

To synchronize its system time, a node must calculate its psc_offset, the time needed for a transmission to reach the PSC. The TM frame is the mechanism for achieving this goal. The master node from time to time (at least every TM_FREQUENCY superframes) will place a TM window at the end of a superframe on all wavelengths. The master node will then announce the presence of a TM window by setting a bit in the SYNCSCHED frame, {ss.tm_bit}. Further, the SYNCSCHED frame includes the duration of time until the TM window will appear ({ss.time_till_tm}); this value varies from wavelength to wavelength, since SYNCSCHED frames appear on each wavelength at different points in time.

A software signal to **>tm<** begins the Time Measurement process. The node listens until it hears a SYNCSCHED frame with the {ss.tm_bit} set, indicating that a TM frame is attached to the end of this superframe. It then sets the tm_timer for the amount {ss.time_till_tm}, waits for the timer to expire, and then transmits a timestamped

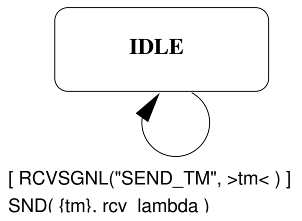
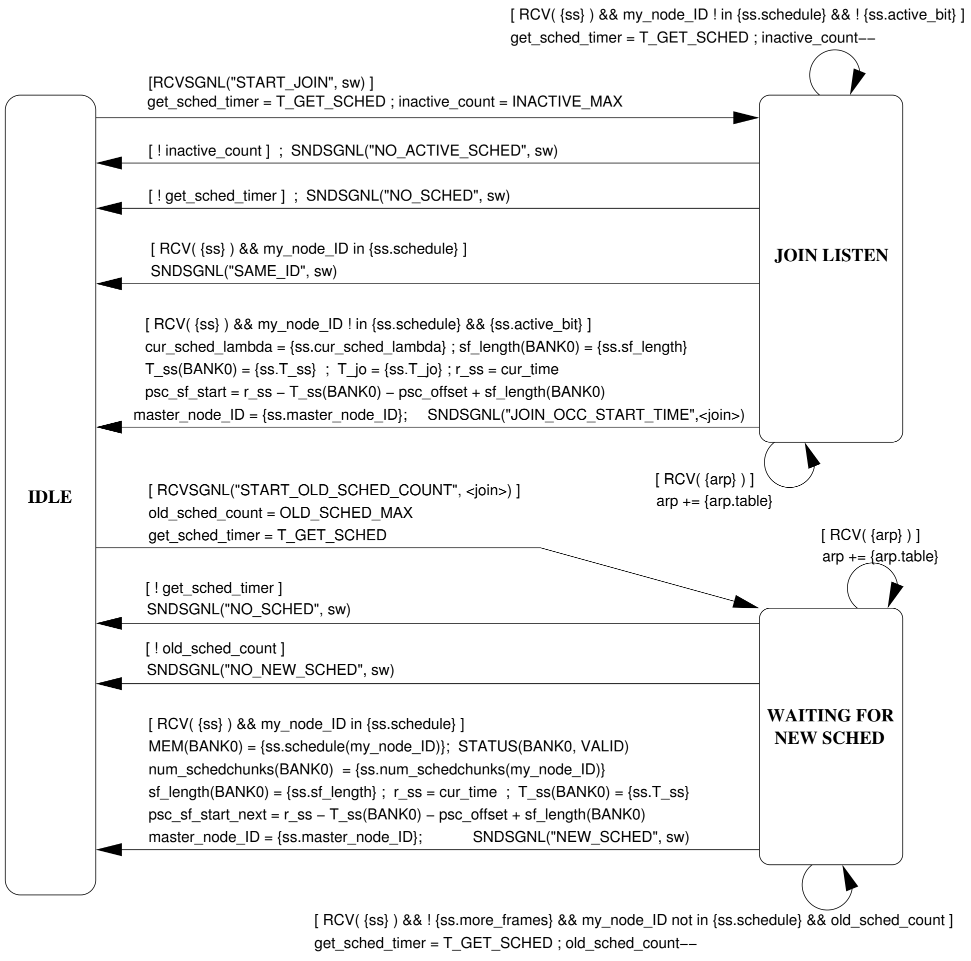


Figure 14: Transmit hardware state machine for time measurement: **<tm>**



[RCV({ss}) && my_node_ID ! in {ss.schedule} && ! {ss.active_bit}]
get_sched_timer = T_GET_SCHED ; inactive_count−−
[RCVSGNL("START_JOIN", sw)]
get_sched_timer = T_GET_SCHED ; inactive_count = INACTIVE_MAX
[! inactive_count] ; SNDSGNL("NO_ACTIVE_SCHED", sw)
[! get_sched_timer] ; SNDSGNL("NO_SCHED", sw)
[RCV({ss}) && my_node_ID in {ss.schedule}]
SNDSGNL("SAME_ID", sw)
JOIN LISTEN
[RCV({ss}) && my_node_ID ! in {ss.schedule} && {ss.active_bit}]
cur_sched_lambda = {ss.cur_sched_lambda} ; sf_length(BANK0) = {ss.sf_length}
T_ss(BANK0) = {ss.T_ss} ; T_jo = {ss.T_jo} ; r_ss = cur_time
psc_sf_start = r_ss − T_ss(BANK0) − psc_offset + sf_length(BANK0)
master_node_ID = {ss.master_node_ID}; SNDSGNL("JOIN_OCC_START_TIME",<join>)
IDLE
[RCV({arp})]
arp += {arp.table}
[RCVSGNL("START_OLD_SCHED_COUNT", <join>)]
old_sched_count = OLD_SCHED_MAX
get_sched_timer = T_GET_SCHED
[RCV({arp})]
arp += {arp.table}
[! get_sched_timer]
SNDSGNL("NO_SCHED", sw)
[! old_sched_count]
SNDSGNL("NO_NEW_SCHED", sw)
WAITING FOR NEW SCHED
[RCV({ss}) && my_node_ID in {ss.schedule}]
MEM(BANK0) = {ss.schedule(my_node_ID)}; STATUS(BANK0, VALID)
num_schedchunks(BANK0) = {ss.num_schedchunks(my_node_ID)}
sf_length(BANK0) = {ss.sf_length} ; r_ss = cur_time ; T_ss(BANK0) = {ss.T_ss}
psc_sf_start_next = r_ss − T_ss(BANK0) − psc_offset + sf_length(BANK0)
master_node_ID = {ss.master_node_ID}; SNDSGNL("NEW_SCHED", sw)
[RCV({ss}) && ! {ss.more_frames} && my_node_ID not in {ss.schedule} && old_sched_count]
get_sched_timer = T_GET_SCHED ; old_sched_count−−

Figure 15: Receive hardware state machine for join: **`>join<`**

TM frame on its receive wavelength. When the node hears its own transmission of the TM frame, it copies the frame's timestamp and the current time into the variables `tm_out` and `tm_in`, respectively, and signals the **`signaling_controller`**, which divides the difference of these two values by two to yield the `psc_offset`.

2.5.6 Join

For a new node, the Join process can be broken into two parts: letting the master node know of its presence, and waiting for the master node to include it in the schedule.

2.5.6.1 Contacting the master node The new node must learn when the JOINOCC window will occur, so that it can transmit a JOINOCC frame to the master node. It listens on its receive wavelength until it receives a SYNCSCHED frame with the `ss.active_bit` set. (This measure ensures that the schedule included in the SYNCSCHED frame is the one currently in effect.) From SYNCSCHED, it extracts the following data fields and stores them in its corresponding local variables:

`{ss.cur_sched_lambda}` : the master node's listening wavelength. (gets copied into `cur_sched_lambda`)

`{ss.sf_length}` : the length in slots of the superframe. (gets copied into `sf_length(BANK0)`)

`{ss.T_ss}` : the offset time (relative to the start of the superframe) of the SYNCSCHED frame on the node's receive wavelength. (gets copied into `T_ss(BANK0)`)

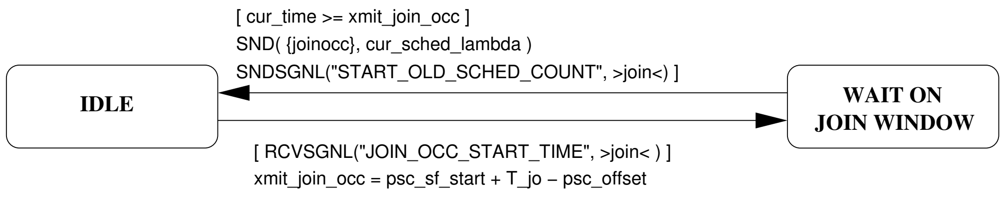

[cur_time >= xmit_join_occ]
SND({joinocc}, cur_sched_lambda)
SNDSGNL("START_OLD_SCHED_COUNT", >join<)]
IDLE
WAIT ON JOIN WINDOW
[RCVSGNL("JOIN_OCC_START_TIME", >join<)]
xmit_join_occ = psc_sf_start + T_jo – psc_offset

Figure 16: Transmit hardware state machine for join: **<join>**

{ss.T_jo} : the offset time (relative to the start of the superframe) of the JOINOCC window on the master node's receive wavelength. (gets copied into T_jo)

Additionally, the node stores the time from its local clock that the SYNCSCHED frame arrived (the "receive timestamp") in the local variable r_ss. From these 5 values, the node can calculate the time (from its local clock) that the start of the superframe occurred at the PSC:

$$\texttt{psc\_sf\_start} = \texttt{r\_ss} - \texttt{T\_ss}(\texttt{BANK0}) - \texttt{psc\_offset} + \texttt{sf\_length}(\texttt{BANK0})$$

The node can now calculate the time (from its local clock) that it must transmit a JOIN-OCC frame in order to hit the JOINOCC window:

$$\texttt{xmit\_join\_occ} = \texttt{psc\_sf\_start} + \texttt{T\_jo} - \texttt{psc\_offset}$$

The node must include a checksum in the JOINOCC frame so that the master node can determine whether it has received the correct information, since it is possible for a collision to occur when two or more nodes attempt to send a JOINOCC frame at the same time.

2.5.6.2 Waiting to be included

Since the new node's receive wavelength is not necessarily the same as the master node's receive wavelength, the new node will be unable to directly detect a collision in the JOINOCC window. Once the JOINOCC frame has been sent, the only way for the new node to learn that it has successfully been included in the network is to receive a new schedule (via the SYNCSCHED frame) which includes its own MAC address (my_node_ID). This new schedule will indicate the windows in which the new node may transmit on each wavelength.

To handle the case of a collision, the new node sets a counter (old_sched_count) to the value OLD_SCHED_MAX after it transmits a JOINOCC frame. While waiting to hear a new schedule containing its own MAC address, the node decrements old_sched_count each time it hears a SYNCSCHED that lacks its MAC address. If the counter should reach zero, the new node notifies the **signaling_controller** and exits the Join process. The **signaling_controller** may either retry the Join process or, after repeated failures, simply give up (*i.e.* enter ERR mode).

If, on the other hand, the new node hears a new schedule containing its own MAC address, then it copies the necessary timing information from the SYNCSCHED frame into the corresponding local variable locations, and signals the **signaling_controller** that it has successfully joined the network (via the "NEW_SCHED" signal).

2.5.6.3 Backoff Algorithms

If a new node exits the TM receive hardware state machine **>tm<** with the signal "NO_REPLY" to **signaling_controller**, **signaling_controller** may execute an exponential backoff algorithm. (A total of TM_MAX failures of this kind are allowed before giving up on Time Measurement and moving to ERR Mode.) The tm_backoff_timer is assigned the value $\texttt{RAND}(1..\texttt{T\_TMBACKOFF} * 2^{\texttt{TM\_MAX}-\texttt{tm\_count}})$. Each time the node picks a random uniformly-distributed number whose bounds are growing larger.

If a new node exits **>tm<** with the signal "NO_TM_WINDOW" to **signaling_controller**, then **signaling_controller** decrements the counter get_tm_count and immediately restarts Time Measurement, without backing off. A total of GET_TM_MAX failures of this kind are allowed before moving to the ERR Mode. (A backoff algorithm could be added to the handling of this type of failure.)

If a joining node exits **>join<** with the signal "NO_NEW_SCHED" to **signaling_controller**, then **signaling_controller** may execute an exponential backoff algorithm. (A total of JOIN_MAX failures of this kind are allowed before moving to ERR Mode.) The join_backoff_timer is assigned the value of $\texttt{RAND}(\texttt{T\_BACKOFF} * 2^{\texttt{JOIN\_MAX}-\texttt{join\_count}})$.

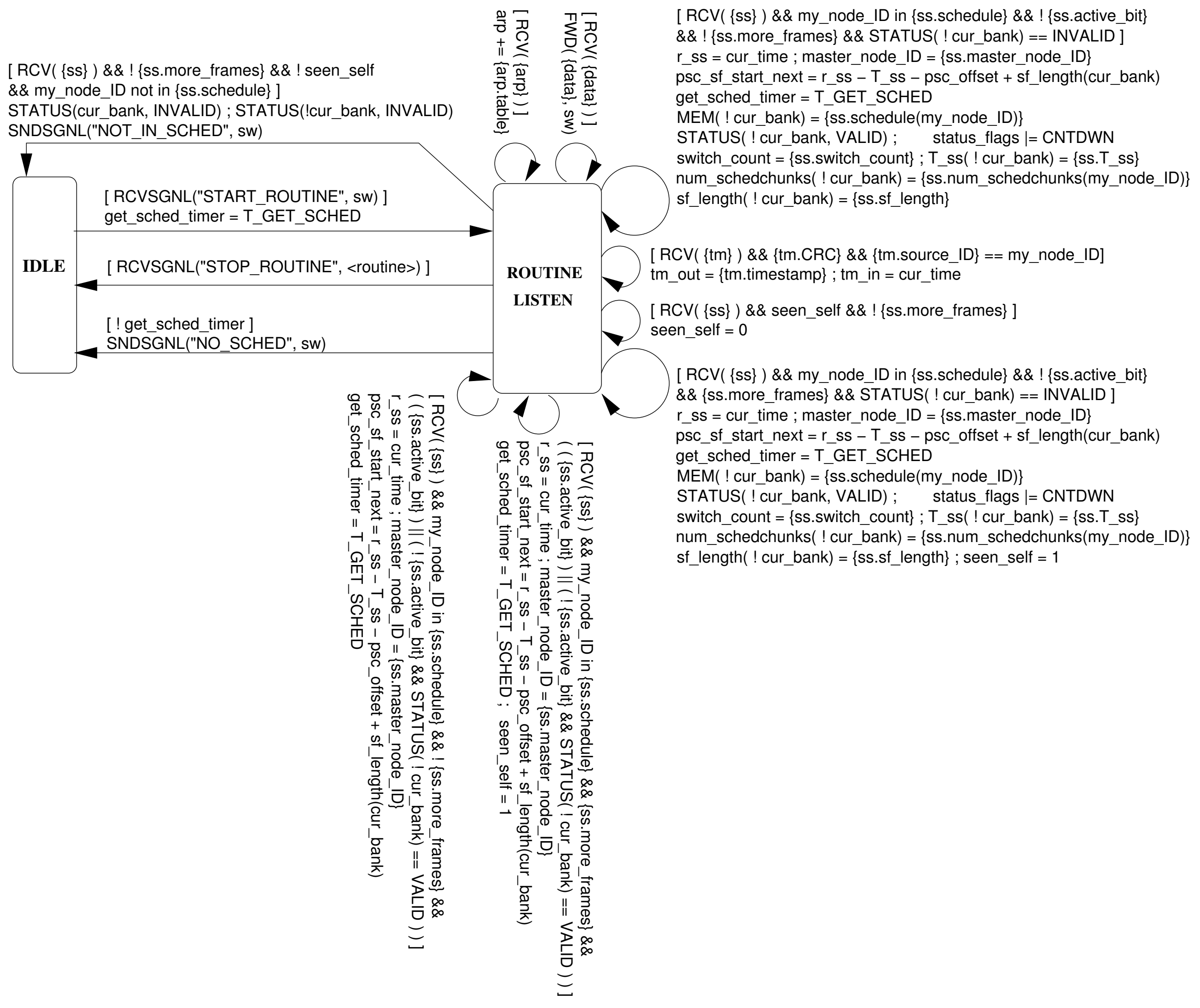

IDLE
ROUTINE LISTEN
[RCV({ss}) && ! {ss.more_frames} && ! seen_self
&& my_node_ID not in {ss.schedule}]
STATUS(cur_bank, INVALID) ; STATUS(!cur_bank, INVALID)
SNDSGNL("NOT_IN_SCHED", sw)
[RCVSGNL("START_ROUTINE", sw)]
get_sched_timer = T_GET_SCHED
[RCVSGNL("STOP_ROUTINE", <routine>)]
[! get_sched_timer]
SNDSGNL("NO_SCHED", sw)
[RCV({data})]
FWD({data}, sw)
[RCV({arp})]
arp += {arp.table}
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.active_bit}
&& ! {ss.more_frames} && STATUS(! cur_bank) == INVALID]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED
MEM(! cur_bank) = {ss.schedule(my_node_ID)}
STATUS(! cur_bank, VALID) ; status_flags |= CNTDWN
switch_count = {ss.switch_count} ; T_ss(! cur_bank) = {ss.T_ss}
num_schedchunks(! cur_bank) = {ss.num_schedchunks(my_node_ID)}
sf_length(! cur_bank) = {ss.sf_length}
[RCV({tm}) && {tm.CRC} && {tm.source_ID} == my_node_ID]
tm_out = {tm.timestamp} ; tm_in = cur_time
[RCV({ss}) && seen_self && ! {ss.more_frames}]
seen_self = 0
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.active_bit}
&& {ss.more_frames} && STATUS(! cur_bank) == INVALID]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED
MEM(! cur_bank) = {ss.schedule(my_node_ID)}
STATUS(! cur_bank, VALID) ; status_flags |= CNTDWN
switch_count = {ss.switch_count} ; T_ss(! cur_bank) = {ss.T_ss}
num_schedchunks(! cur_bank) = {ss.num_schedchunks(my_node_ID)}
sf_length(! cur_bank) = {ss.sf_length} ; seen_self = 1
[RCV({ss}) && my_node_ID in {ss.schedule} && {ss.more_frames} &&
(({ss.active_bit}) || (! {ss.active_bit} && STATUS(! cur_bank) == VALID))]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED ; seen_self = 1
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.more_frames} &&
(({ss.active_bit}) || (! {ss.active_bit} && STATUS(! cur_bank) == VALID))]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED

Figure 17: Receive hardware state machine for candidate and slave nodes: **>routine<**

2.5.7 Routine

We now describe the operation of the receive and transmit hardware of a nonscheduling node, *i.e.* candidate and slave nodes. A new node enters Routine Mode once it has successfully joined the network; that is, during **>join<** it received a SYNCSCHED frame that included its own MAC address in the schedule, and it then exited **>join<** with the message "NEW_SCHED" to **signaling_controller**. The main functions of the receive hardware **>routine<** are to forward incoming data frames to the **signaling_controller** and to extract the schedule from the SYNCSCHED frame. The transmit hardware **<routine>** meanwhile transmits control frames and data frames from its wavelength queues onto the appropriate outgoing wavelengths, according to the current schedule.

2.5.7.1 Receive Hardware We first describe **>routine<** qualitatively, and then explicitly describe the transitions possible at each state. The state machine diagram for**>routine<** is shown in Figure 17. When a SYNCSCHED frame is received, **>routine<** first checks whether its own MAC address (`my_node_ID`) is included in the schedule. If the node has for some reason been left out of the schedule, **>routine<** notifies the **signaling_controller** with the "NOT_IN_SCHED" signal and returns to idle. The **signaling_controller** then exits Routine Mode and moves to ERR mode.

If, on the other hand, the node's `my_node_ID` is in the schedule, then **`>routine<`** copies synchronization information from SYNCSCHED and next checks whether the `{ss.active_bit}` is set. As long as the active bit is set, the node will continue to operate according to the current schedule (located in `cur_bank`). However, if the active bit is not set, then the schedule being disseminated in the SYNCSCHED frame is a newly calculated schedule that will go into effect after `{ss.switch_count}` more superframes. That is, `switch_count` (the local variable into which the value `{ss.switch_count}` is copied) represents the number of remaining superframes following the current one in which the old schedule will still be used.

When **`>routine<`** encounters a SYNCSCHED frame without the `{ss.active_bit}` set, it checks the status of the reserve memory bank (i.e., `!cur_bank`). If the status is `INVALID`, then all the new synchronization and scheduling information for the new schedule has yet to be copied into the reserve memory bank (i.e., into `!cur_bank`). After copying this information, **`>routine<`** sets this bank's status to `VALID`. In this way, **`>routine<`** doesn't waste effort recopying the new schedule's information into !cur_bank a total of `{ss.switch_count}` times. That is, if **`>routine<`** encounters a SYNCSCHED frame without the `{ss.active_bit}` set but finds the status of !cur_bank to be already `VALID`, then it recognizes that it has already copied the new information into `!cur_bank`.

There are three states in **`>routine<`**: Idle, Routine Listen, and In Schedule. We now describe each state in turn.

1. Idle State. The only transition out of the Idle state is:

   - START_ROUTINE signal from **`signaling_controller`**.

     Event: The node has successfully joined the network.

     Transition: Move to Routine Listen state.

2. Routine Listen. There are 6 transitions out of the Routine Listen state; the first two are self-transitions.

   - Receipt of a DATA frame.

     Event: A DATA frame was received on the listening wavelength.

     Transition: Forward the frame to the frame handling layer and return to the Routine Listen state.

   - Receipt of an ARP frame.

     Event: An ARP frame was received on the listening wavelength.

     Transition: Copy the new information into the ARP table and return to the Routine Listen state.

   - STOP_ROUTINE signal from **`<routine>`**.

     Event: **`<routine>`** has encountered an error condition and is returning to idle.

     Transition: Move to Idle state.

   - `get_sched_timer` expired.

     Event: Failed to receive a SYNCSCHED within T_GET_SCHED.

     Transition: Move to Idle state.

   - Receipt of a SYNCSCHED which contains `my_node_ID`.

     Event: A SYNCSCHED frame was received that contains scheduling information for this node.

     Transition: Save important timing information and move to the In Schedule state.

   - Receipt of a SYNCSCHED which *does not* contain `my_node_ID`.

     Event: A SYNCSCHED frame was received that unexpectedly fails to contain scheduling information for this node.

     Transition: Mark the status of both memory banks (the current and the reserve bank) INVALID, send signal NOT_IN_SCHED to **`signaling_controller`**, and move to the Idle state.

3. Routine Listen. **`>routine<`** can only arrive at this state after the receipt of a SYNCSCHED which contains `my_node_ID`. There are 5 transitions out of the In Schedule state; the first is a self-transition, the second is triggered by a signal, and the final three involve checking another field in the newly-arrived SYNCSCHED frame.

- Receipt of an ARP frame.

  Event: An ARP frame was received on the listening wavelength.

  Transition: Copy the new information into the ARP table and return to the In Schedule state.

- STOP_ROUTINE signal from **`<routine>`**.

  Event: **`<routine>`** has encountered an error condition and is returning to idle.

  Transition: Move to Idle state.

- `{ss.active_bit}` was set.

  Event: The active bit in the newly-arrived SYNCSCHED frame was set, indicating that no countdown has begun to switch to a new schedule.

  Transition: Reset `get_sched_timer` and move to Routine Listen state.

- `{ss.active_bit}` was *not* set and the status of `!cur_bank` is VALID.

  Event: Countdown has begun to switch to a new schedule, and the new schedule has *already* been copied into the reserve memory bank (`!cur_bank`).

  Transition: Reset `get_sched_timer` and move to the Routine Listen state.

- `{ss.active_bit}` was *not* set and the status of `!cur_bank` is INVALID.

  Event: Countdown has begun to switch to a new schedule, but the new schedule has *not yet* been copied into the reserve memory bank (`!cur_bank`).

  Transition: Copy the new scheduling information into `!cur_bank`, save important timing information, and move to the In Schedule state.

2.5.7.2 Transmit Hardware The state machine **`<routine>`** shown in Figure 18 can only begin after the Join process has succeeded. The final task in the Join process was to place the current schedule and relevant synchronization information into the memory bank `BANK0`; therefore **`<routine>`** begins by setting `cur_bank` to `BANK0`. After confirming `BANK0`'s status to be `VALID`, **`<routine>`** is ready to begin the first superframe.

At the start of any superframe, **`<routine>`** :

1. sets `cur_schedule` to point to the schedule contained in `cur_bank`. (Note that the status of `cur_bank` has already been confirmed `VALID`.)

2. sets the index `cur_schedchunk` to zero. This index will be incremented after the node completes its transmissions on each successive wavelength; the node then can recognize that it is done with the current superframe when `cur_schedchunk` reaches the value $\texttt{num\_schedchunks}(\texttt{cur\_bank}) - 1$.

3. sets `psc_sf_start` to `psc_sf_start_next`. Recall that `psc_sf_start` represents the time (according the node's local clock) that the superframe began at the PSC. The value of `psc_sf_start_next` could have been set in one of two ways: either **`>join<`** set the value (true only for the first superframe after the node joins the network), or **`>routine<`** set the value (true for all other superframes).

Once these tasks have been completed, **`<routine>`** is ready to begin transmissions according to the information contained in the current schedchunk.

At the start of any schedchunk, **`<routine>`** :

1. sets `xmit_lambda` to the wavelength in `cur_schedchunk`.

2. sets `cur_queue` to point to the queue for `xmit_lambda`.

3. calculates two time references that govern its transmissions: (a) `xmit_start`, the time it may begin transmitting on `xmit_lambda`, and (b) `xmit_last`, the last instant at which it may start the transmission of a frame on `xmit_lambda`.

At this point **`<routine>`** needs only to wait until `xmit_start` arrives in order to begin transmitting; the first frame it transmits depends on the value of `xmit_lambda`:

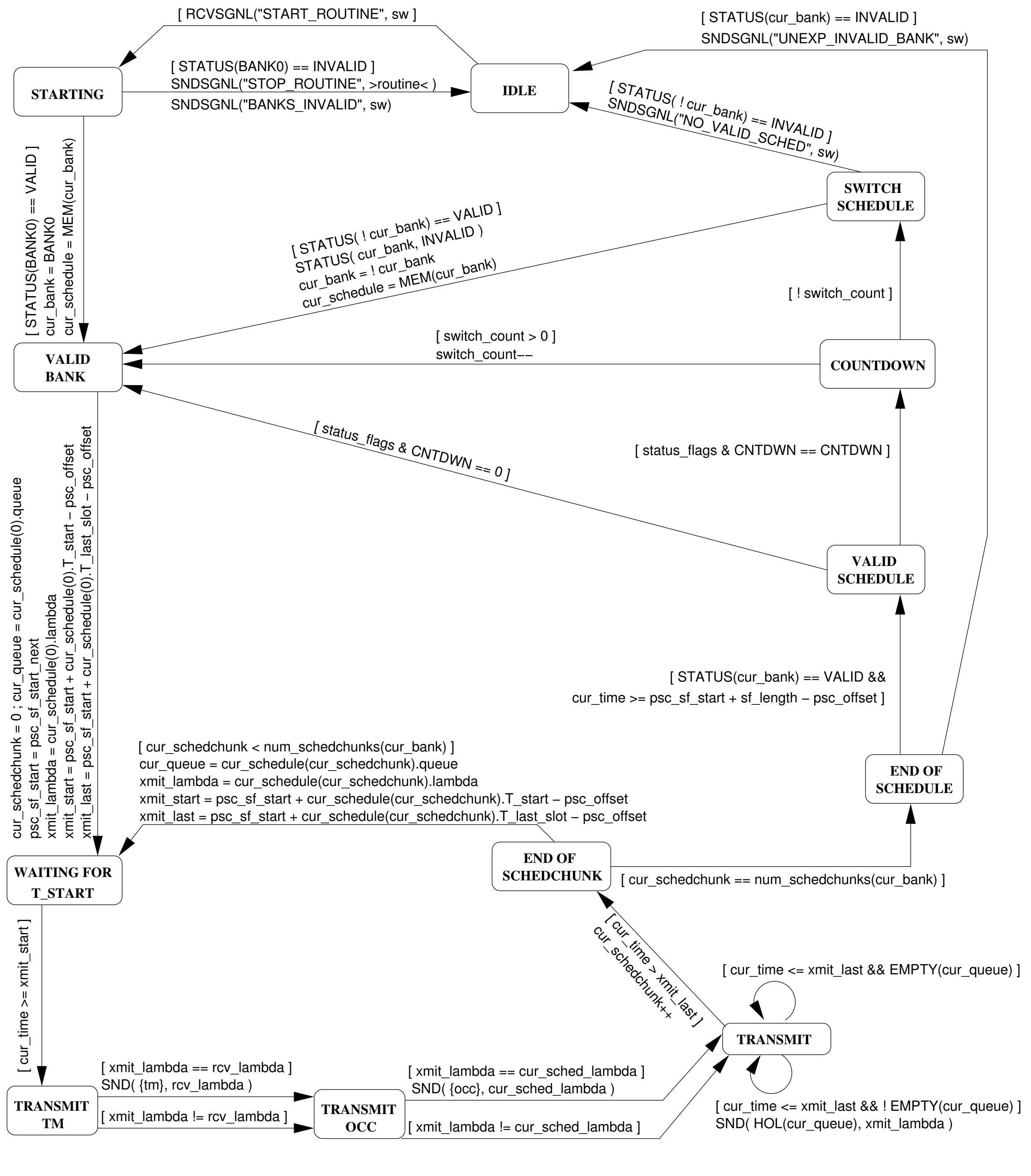

[RCVSGNL("START_ROUTINE", sw]
[STATUS(cur_bank) == INVALID]
SNDSGNL("UNEXP_INVALID_BANK", sw)
[STATUS(BANK0) == INVALID]
SNDSGNL("STOP_ROUTINE", >routine<)
SNDSGNL("BANKS_INVALID", sw)
STARTING
IDLE
[STATUS(! cur_bank) == INVALID]
SNDSGNL("NO_VALID_SCHED", sw)
SWITCH SCHEDULE
[STATUS(BANK0) == VALID]
cur_bank = BANK0
cur_schedule = MEM(cur_bank)
[STATUS(! cur_bank) == VALID]
STATUS(cur_bank, INVALID)
cur_bank = ! cur_bank
cur_schedule = MEM(cur_bank)
[! switch_count]
[switch_count > 0]
switch_count−−
VALID BANK
COUNTDOWN
[status_flags & CNTDWN == 0]
[status_flags & CNTDWN == CNTDWN]
cur_schedchunk = 0 ; cur_queue = cur_schedule(0).queue
psc_sf_start = psc_sf_start_next
xmit_lambda = cur_schedule(0).lambda
xmit_start = psc_sf_start + cur_schedule(0).T_start − psc_offset
xmit_last = psc_sf_start + cur_schedule(0).T_last_slot − psc_offset
VALID SCHEDULE
[STATUS(cur_bank) == VALID &&
cur_time >= psc_sf_start + sf_length − psc_offset]
[cur_schedchunk < num_schedchunks(cur_bank)]
cur_queue = cur_schedule(cur_schedchunk).queue
xmit_lambda = cur_schedule(cur_schedchunk).lambda
xmit_start = psc_sf_start + cur_schedule(cur_schedchunk).T_start − psc_offset
xmit_last = psc_sf_start + cur_schedule(cur_schedchunk).T_last_slot − psc_offset
END OF SCHEDULE
WAITING FOR T_START
END OF SCHEDCHUNK
[cur_schedchunk == num_schedchunks(cur_bank)]
[cur_time > xmit_last]
cur_schedchunk++
[cur_time <= xmit_last && EMPTY(cur_queue)]
[cur_time >= xmit_start]
TRANSMIT
[xmit_lambda == rcv_lambda]
SND({tm}, rcv_lambda)
[xmit_lambda == cur_sched_lambda]
SND({occ}, cur_sched_lambda)
TRANSMIT TM
[xmit_lambda != rcv_lambda]
TRANSMIT OCC
[xmit_lambda != cur_sched_lambda]
[cur_time <= xmit_last && ! EMPTY(cur_queue)]
SND(HOL(cur_queue), xmit_lambda)

Figure 18: Transmit hardware state machine for candidate and slave nodes: **`<routine>`**

1. If `xmit_lambda` = `cur_sched_lambda` (the receive wavelength of the master node), then the first frame **`<routine>`** transmits must be an `{OCC}` frame, to inform the master node of its queue occupancies.
2. Otherwise, if `xmit_lambda` = `rcv_lambda` (the node's own receive wavelength), then the first frame **`<routine>`** transmits must be a `{TM}` frame, to carry out "Routine Time Measurement".
3. Otherwise, **`<routine>`** may transmit DATA frames from `cur_queue`.

Recall that DATA frames may be of variable length, with none exceeding `L_max`. The node transmits DATA frames from `cur_queue` back to back, without waiting for the beginning of a new slot. Just prior to transmitting each frame, the node checks to make sure that the current time has not exceeded `xmit_last`. When `xmit_last` has passed, transmissions on this wavelength must cease; the end of the current schedchunk has arrived. The index `cur_schedchunk` is incremented and then tested against `num_schedchunks(cur_bank)` to determine whether the end of the schedule has arrived. If not, **`<routine>`** proceeds to the next schedchunk.

But if **`<routine>`** *has* reached the end of the schedule, it next checks whether a countdown has started (counting down the superframes until the time to switch from the current to the reserve memory bank). If the countdown has not yet begun, then **`<routine>`** simply starts over at the beginning of the current schedule in `cur_bank`. If the countdown has begun, then **`<routine>`** must determine whether it should switch now to the reserve memory bank. This task is accomplished by considering the value of `switch_count`, which gives the number of remaining superframes for which the old schedule should still be used. If `switch_count` is positive, **`<routine>`** decrements `switch_count` and starts over at the beginning of the old schedule (in `cur_bank`). If `switch_count` has reached zero, then **`<routine>`** marks `cur_bank INVALID` and switches to the reserve memory bank, by setting `cur_bank` to `!cur_bank`.

2.5.8 Scheduling

2.5.8.1 Receive Hardware The receive state machine **`>scheduling<,`** shown in Figure 19, retains all the functionality of **`>routine<`**, but possesses two extra transitions to aid in the collection of information needed to compute the schedule. Each of the additional transitions is a self-transition from the Routine Listen state:

- Receipt of an OCC frame.

  Event: An OCC frame was received on the listening wavelength.

  Transition: Forward the frame to the **`signaling_controller`** and return to the Routine Listen state.

- Receipt of a JOINOCC frame.

  Event: An JOINOCC frame was received on the listening wavelength.

  Transition: Forward the frame to the **`signaling_controller`** and return to the Routine Listen state.

2.5.8.2 Transmit Hardware The transmit state machine **`<scheduling>,`** shown in Figure 20, retains all the functionality of **`<routine>`**. However, the transition from the End of Schedchunk state to the End of Schedule state becomes split into two, in order to aid in the transmission of a newly-calculated schedule. Both transitions first check to make sure the end of schedule has been reached, by verifying that `cur_schedchunk+1` equals `num_schedchunks(cur_bank)`. Next, the status of `BANK_NEWCALC` is checked. If INVALID, no action is taken. If VALID, then a newly-calculated schedule has been placed in `BANK_NEWCALC` by the software; **`<scheduling>`** copies the new schedule into `BANK_CURFRAME` so that it can be disseminated in the next superframe.

2.5.8.3 Routine Time Measurement All nodes, whether master, candidate, or slave, will make repeat measurements of their `psc_offset` once per superframe. Upon the arrival of the regularly-scheduled transmission window on its own receive wavelength, the node will first transmit a TM frame. The receive hardware, **`>routine<`**, will hear this transmission and will compute an updated value of `psc_offset`: (`cur_time - {tm.timestamp}`) / 2.

2.5.9 Error Detection, Reporting, and Recovery (ERR) Mode

2.5.9.1 Events that trigger the transition to ERR mode A node enters ERR mode from another state in the software state machine. Several events can cause the transition into ERR mode.

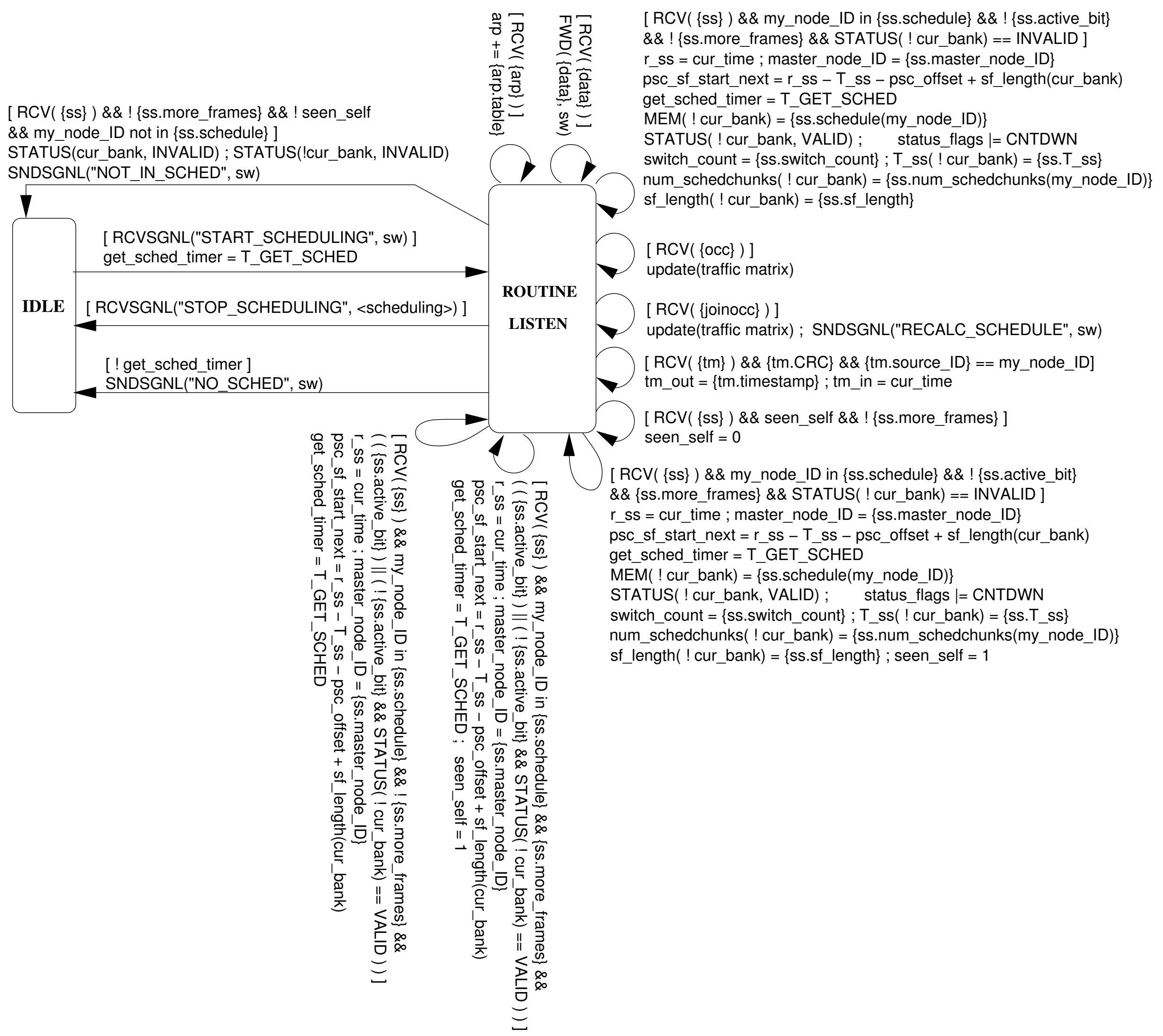
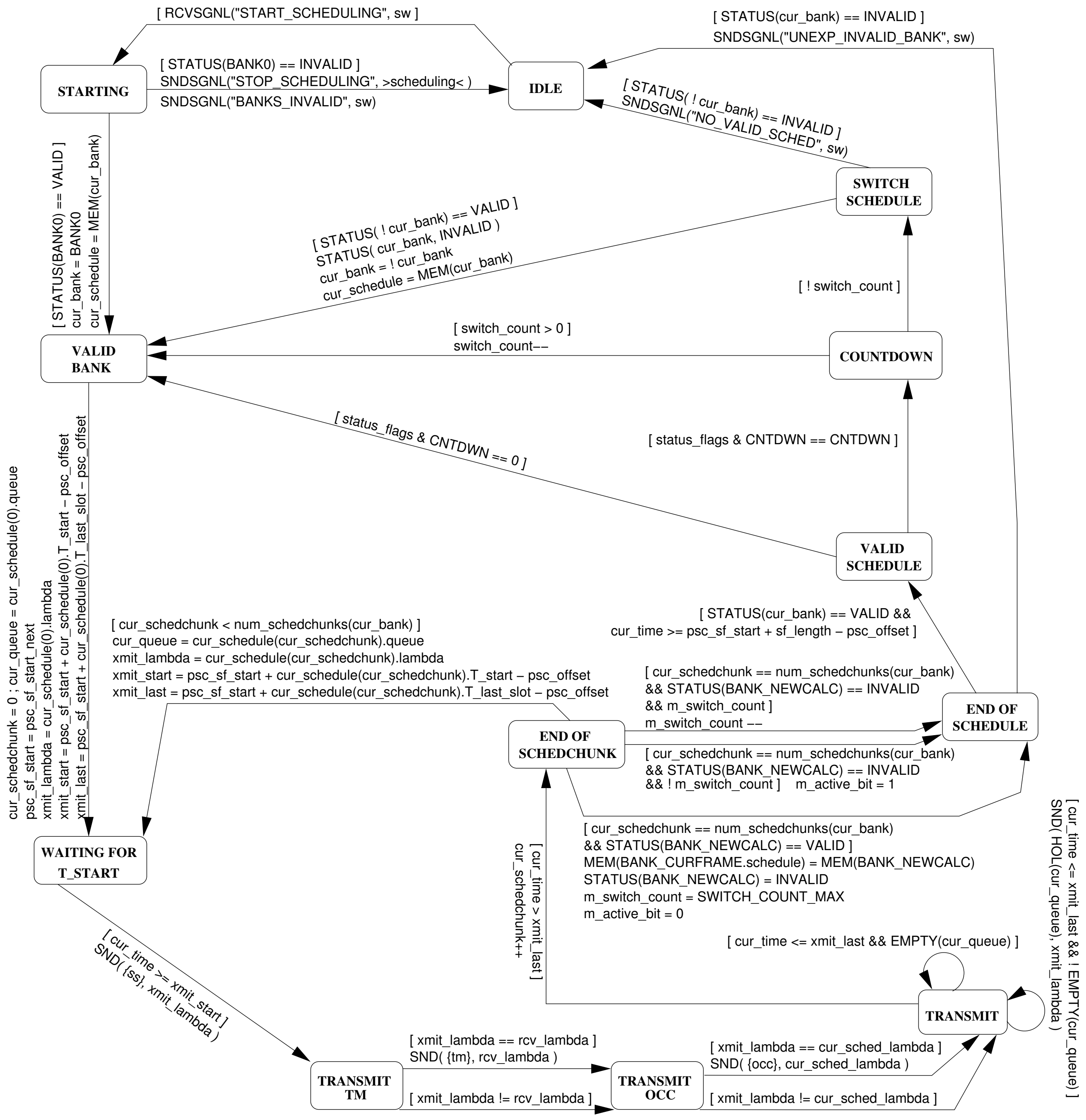

[RCV({arp})]
arp += {arp.table}
[RCV({data})]
FWD({data}, sw)
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.active_bit}
&& ! {ss.more_frames} && STATUS(! cur_bank) == INVALID]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED
MEM(! cur_bank) = {ss.schedule(my_node_ID)}
STATUS(! cur_bank, VALID) ; status_flags |= CNTDWN
switch_count = {ss.switch_count} ; T_ss(! cur_bank) = {ss.T_ss}
num_schedchunks(! cur_bank) = {ss.num_schedchunks(my_node_ID)}
sf_length(! cur_bank) = {ss.sf_length}
[RCV({ss}) && ! {ss.more_frames} && ! seen_self
&& my_node_ID not in {ss.schedule}]
STATUS(cur_bank, INVALID) ; STATUS(!cur_bank, INVALID)
SNDSGNL("NOT_IN_SCHED", sw)
[RCVSGNL("START_SCHEDULING", sw)]
get_sched_timer = T_GET_SCHED
IDLE
ROUTINE
LISTEN
[RCVSGNL("STOP_SCHEDULING", <scheduling>)]
[! get_sched_timer]
SNDSGNL("NO_SCHED", sw)
[RCV({occ})]
update(traffic matrix)
[RCV({joinocc})]
update(traffic matrix) ; SNDSGNL("RECALC_SCHEDULE", sw)
[RCV({tm}) && {tm.CRC} && {tm.source_ID} == my_node_ID]
tm_out = {tm.timestamp} ; tm_in = cur_time
[RCV({ss}) && seen_self && ! {ss.more_frames}]
seen_self = 0
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.active_bit}
&& {ss.more_frames} && STATUS(! cur_bank) == INVALID]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED
MEM(! cur_bank) = {ss.schedule(my_node_ID)}
STATUS(! cur_bank, VALID) ; status_flags |= CNTDWN
switch_count = {ss.switch_count} ; T_ss(! cur_bank) = {ss.T_ss}
num_schedchunks(! cur_bank) = {ss.num_schedchunks(my_node_ID)}
sf_length(! cur_bank) = {ss.sf_length} ; seen_self = 1
[RCV({ss}) && my_node_ID in {ss.schedule} && {ss.more_frames} &&
(({ss.active_bit}) || (! {ss.active_bit} && STATUS(! cur_bank) == VALID))]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED ; seen_self = 1
[RCV({ss}) && my_node_ID in {ss.schedule} && ! {ss.more_frames} &&
(({ss.active_bit}) || (! {ss.active_bit} && STATUS(! cur_bank) == VALID))]
r_ss = cur_time ; master_node_ID = {ss.master_node_ID}
psc_sf_start_next = r_ss – T_ss – psc_offset + sf_length(cur_bank)
get_sched_timer = T_GET_SCHED

Figure 19: Receive hardware state machine for the master node: **`>scheduling<`**



STARTING
IDLE
SWITCH SCHEDULE
VALID BANK
COUNTDOWN
VALID SCHEDULE
END OF SCHEDCHUNK
END OF SCHEDULE
WAITING FOR T_START
TRANSMIT
TRANSMIT TM
TRANSMIT OCC
[RCVSGNL("START_SCHEDULING", sw]
[STATUS(BANK0) == INVALID]
SNDSGNL("STOP_SCHEDULING", >scheduling<)
SNDSGNL("BANKS_INVALID", sw)
[STATUS(cur_bank) == INVALID]
SNDSGNL("UNEXP_INVALID_BANK", sw)
[STATUS(! cur_bank) == INVALID]
SNDSGNL("NO_VALID_SCHED", sw)
[STATUS(BANK0) == VALID]
cur_bank = BANK0
cur_schedule = MEM(cur_bank)
[STATUS(! cur_bank) == VALID]
STATUS(cur_bank, INVALID)
cur_bank = ! cur_bank
cur_schedule = MEM(cur_bank)
[! switch_count]
[switch_count > 0]
switch_count−−
[status_flags & CNTDWN == 0]
[status_flags & CNTDWN == CNTDWN]
[STATUS(cur_bank) == VALID &&
cur_time >= psc_sf_start + sf_length − psc_offset]
cur_schedchunk = 0 ; cur_queue = cur_schedule(0).queue
psc_sf_start = psc_sf_start_next
xmit_lambda = cur_schedule(0).lambda
xmit_start = psc_sf_start + cur_schedule(0).T_start − psc_offset
xmit_last = psc_sf_start + cur_schedule(0).T_last_slot − psc_offset
[cur_schedchunk < num_schedchunks(cur_bank)]
cur_queue = cur_schedule(cur_schedchunk).queue
xmit_lambda = cur_schedule(cur_schedchunk).lambda
xmit_start = psc_sf_start + cur_schedule(cur_schedchunk).T_start − psc_offset
xmit_last = psc_sf_start + cur_schedule(cur_schedchunk).T_last_slot − psc_offset
[cur_schedchunk == num_schedchunks(cur_bank)
&& STATUS(BANK_NEWCALC) == INVALID
&& m_switch_count]
m_switch_count −−
[cur_schedchunk == num_schedchunks(cur_bank)
&& STATUS(BANK_NEWCALC) == INVALID
&& ! m_switch_count]   m_active_bit = 1
[cur_schedchunk == num_schedchunks(cur_bank)
&& STATUS(BANK_NEWCALC) == VALID]
MEM(BANK_CURFRAME.schedule) = MEM(BANK_NEWCALC)
STATUS(BANK_NEWCALC) = INVALID
m_switch_count = SWITCH_COUNT_MAX
m_active_bit = 0
[cur_time > xmit_last]
cur_schedchunk++
[cur_time <= xmit_last && EMPTY(cur_queue)]
[cur_time <= xmit_last && ! EMPTY(cur_queue)]
SND(HOL(cur_queue), xmit_lambda)
[cur_time >= xmit_start]
SND({ss}, xmit_lambda)
[xmit_lambda == rcv_lambda]
SND({tm}, rcv_lambda)
[xmit_lambda != rcv_lambda]
[xmit_lambda == cur_sched_lambda]
SND({occ}, cur_sched_lambda)
[xmit_lambda != cur_sched_lambda]

Figure 20: Transmit hardware state machine for the master node: **`<scheduling>`**

From software state Time Measurement:

1. The signal "NO_TM_WINDOW" was received from **>tm<** and `get_tm_count` == 0. Software has repeatedly attempted to perform time measurement, and it has failed GET_TM_MAX times. Each of these failures was caused by no TM window appearing out of NO_TM_MAX superframes.

2. The signal "NO_SCHED" was received from **>tm<** and the node is not a scheduling server and `wait_count` == 0. Whereas a scheduling server immediately moves to Scheduler Election upon receiving the "NO_SCHED" signal, a slave instead transitions to the Wait On Election state for a time T_WAIT – allowing the servers time to elect a scheduler – before re-attempting **>tm<**. A slave allows a total of WAIT_MAX failures of this type before entering to ERR mode.

3. The signal "NO_REPLY" was received from **>tm<** and `tm_count` == 0. Software has repeatedly attempted to perform time measurement, and it has failed TM_MAX times with the "NO_REPLY" error. This error results from the node having failed to receive its own transmission of the TM frame, possibly because either the transmitter or the receiver is broken. Much less likely, it is possible that a collision occurred TM_MAX times during the TM window, even though exponential backoff was used in between attempts at **>tm<.**

From software state Join:

1. The signal "NO_ACTIVE_SCHED" was received from **>join<**. The **>join<** state machine has received INACTIVE_MAX SYNCSCHED frames with `{ss.active_bit}` not set. This error is likely the result of a malfunction at the master node.

2. The signal "NO_NEW_SCHED" was received from **>join<**, and `join_count` == 0. The node has received OLD_SCHED_MAX schedules that failed to contain `my_node_ID`. This error may have resulted from a collision in the JOIN-OCC frame, meaning that the master node never received the joining node's request to be included in the schedule. After waiting an exponential backoff, software re-attempts **>join<**. Software allows JOIN_MAX failures of type "NO_NEW_SCHED" before moving to ERR mode.

3. The signal "NO_SCHED" was received from **>join<** and the node is not a scheduling server and `wait_count` == 0. Whereas a scheduling server immediately moves to Scheduler Election upon receiving the "NO_SCHED" signal, a slave instead transitions to the Wait On Election state for a time T_WAIT – allowing the servers time to elect a scheduler – before re-attempting **>join<**. A slave allows a total of WAIT_MAX failures of this type before entering to ERR mode.

From software state Routine Non-Scheduler:

1. CRC failure

2. The signal "BANKS_INVALID" was received from **<routine>**. At the start of **<routine>**, `BANK0` should hold a valid schedule, placed there by **>join<**. Therefore, if **<routine>** finds `BANK0` invalid, it moves to ERR mode.

3. The signal "UNEXP_INVALID_BANK" was received from **<routine>**. When the transmitting hardware reaches the end of the schedule in `cur_bank`, it checks the status of `cur_bank`. If the status is unexpectedly invalid, then **>routine<** has encountered an error condition and has marked `cur_bank` invalid in order to silence the transmitter. The node then moves to ERR mode.

4. The signal "NOT_IN_SCHED" was received from **>routine<**. If **>routine<** receives a `{syncsched}` which fails to include `my_node_ID`, then the node has been accidentally left out of the schedule and cannot continue to transmit. Hence the node moves to ERR mode.

	λ_1	λ_2	λ_3	sum
n_1	4	1	3	8
n_2	2	3	2	7
n_3	3	2	1	6
n_4	2	3	1	6
n_5	1	1	2	4
sum	12	10	9	

Table 9: Example traffic matrix

From software state Routine Scheduler:

1. CRC failure
2. The signal "BANKS_INVALID" was received from **`<scheduler>`**. At the start of **`<scheduler>`**, `BANK0` should hold a valid schedule, placed there by **`>join<`**. Therefore, if **`<scheduler>`** finds `BANK0` invalid, it moves to ERR mode.
3. The signal "UNEXP_INVALID_BANK" was received from **`<scheduler>`**. When the transmitting hardware reaches the end of the schedule in `cur_bank`, it checks the status of `cur_bank`. If the status is unexpectedly invalid, then **`>scheduler<`** has encountered an error condition and has marked `cur_bank` invalid in order to silence the transmitter. The node then moves to ERR mode.
4. The signal "NOT_IN_SCHED" was received from **`>scheduler<`**. If **`>scheduler<`** receives a `{syncsched}` which fails to include `my_node_ID`, then the node has been accidentally left out of the schedule and cannot continue to transmit. Hence the node moves to ERR mode.

2.5.9.2 Response to Error Conditions For certain error conditions, we anticipate being able to design a solution to correct the problem or minimize its impact, without forcing the node to drop out of the network completely. The handling of error conditions presents an opportunity for future work. At present, when a node reaches ERR mode, an error message will be displayed and the node will cease to be a part of the Helios network.

2.6 New Scheduler Approaches

2.6.1 Helios Greedy Scheduling Algorithm

Recall that the master node receives an OCC frame, containing packet queue occupancies, from each node once per superframe. The master node may also receive a JOIN-OCC frame, containing packet queue occupancies, from a new node wishing to join the network. From this information, the master node can build the traffic matrix A, an $N \times C$ matrix, where N is the number of nodes in the network, C is the number of wavelengths, and entry a_{ij} represents the number of slots requested by node i for transmission on λ_j. For a network of $C = 3$ wavelengths and $N = 5$ nodes, a sample traffic matrix is shown in Table 9.

`Helios` uses a one-pass greedy scheduling algorithm, the pseudocode for which is shown in Algorithm 1. The algorithm creates a schedule from $t = 0$ forward in time without backtracking, always attempting to schedule the highest priority node on the highest priority wavelength. Higher priority is assigned to nodes (respectively, wavelengths) that have higher corresponding row-sums (respectively, column-sums) in the traffic matrix A. In the sample traffic matrix above, the nodes have been renumbered in order of largest row-sum to smallest, such that n_1 has the largest row-sum and n_N has the smallest, with ties being broken arbitrarily. The same was done for the wavelengths: λ_1 has the largest column-sum and λ_C has the smallest. The traffic matrix gives rise to two lower bounds on the schedule length. The maximum column-sum is the *channel bound*; a schedule can be no shorter than the total demand for any one wavelength. The maximum row-sum plus C tuning latencies is called the *node bound*; in order to meet the demand of n_1, a schedule must be at least long enough for n_1 to transmit all its traffic and tune to each of the $C = 3$ wavelengths. The maximum of the channel and node bounds is the greatest lower bound on the schedule length.

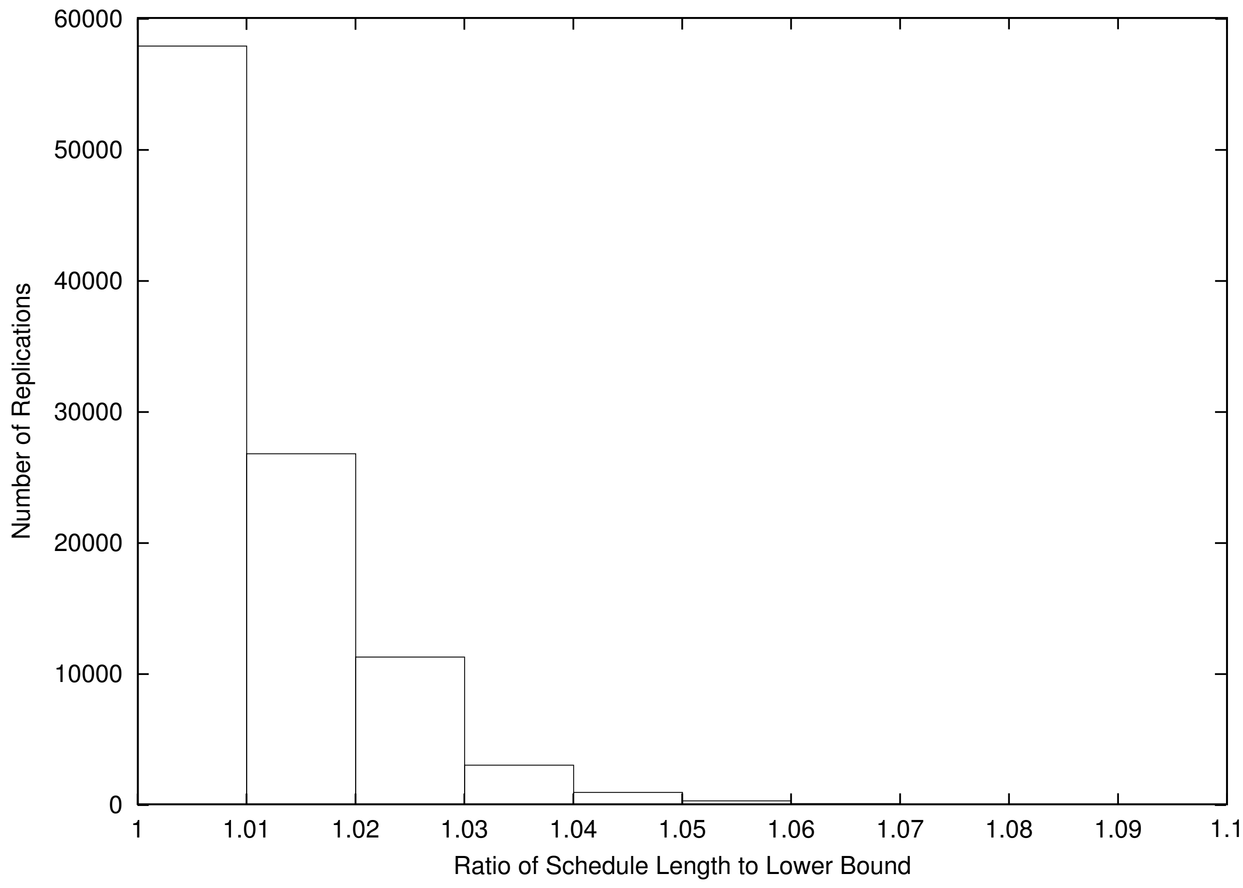


Figure 21: Performance of `Helios` scheduler

2.6.1.1 Running time and results The original scheduler developed in a previous work at NCSU ([9], [10]) produces schedules very close to the lower bound in length, but requires a prohibitively long runtime. In particular, the original scheduler has a worst-case runtime of $O(CN^{4)}$. The scheduler developed for `Helios` is a straightforward greedy scheduler that has a worst-case runtime of $O(C^2N^2)$. This speedup is substantial because the number of nodes is expected to be much larger than the number of channels. Moreover, the faster scheduler can be readily implemented in hardware, resulting in an additional gain in speed. To achieve these gains in speed and simplicity, the new scheduler produces schedules that are not as close to optimal as those produced by the original scheduler. However, the faster scheduler's results are "reasonably close" to optimal. In simulations with various patterns of network traffic demand, the new scheduler produces schedules within 5% of the lower bound, approximately 95% of the time.

The histogram shown in Figure 21 corresponds to a network with balanced traffic demand; that is, each node determines its demand for each wavelength by drawing from the same distribution (here, equally likely over the set {0,1,...,20}). For each set of traffic demands, we examined the ratio of the length of the schedule generated by the new scheduler to the lower bound. The histogram was created from 100,000 replications. The height of each box shows the number of times the ratio fell within the range indicated. For example, nearly 58,000 or 58% of the replications resulted in ratios between 1.00 and 1.01. Furthermore, in 95% of the replications, the new scheduler produced a schedule that was no more than 3% longer than the lower bound (corresponding to ratios between 1.00 and 1.03).

2.6.1.2 Schedule Correction for Node Synchronization The master node must send a `{SYNCSCHED}` on each wavelength early enough in the the superframe for the receiving nodes to make use of the synchronization information contained in the frame. In particular, each node in the network uses `r_ss`, the receive timestamp of the `{SYNCSCHED}` frame that arrives during superframe `n`, to calculate `psc_sf_start_next`, the start time of the upcoming superframe (superframe `n+1`).

Algorithm 1 Helios scheduling algorithm

```
/* initialize each entry in the schedule to 0 */
for ($t = 0 \ldots 2(glb)$ ) { /* the schedule length will not exceed $2(glb)$ */
    for ($\lambda = 1 \ldots C$)
        schedule[$t$][$\lambda$] = 0 ;
} /* end for */
/* ****************************************************************** */
/* initialize remainingDemand to the sum of all the $a_{n\lambda}$'s */
remainingDemand = 0 ;
for ($\lambda = 1 \ldots C$) {
    for ($n = 1 \ldots N$)
        remainingDemand = remainingDemand + $a[n][\lambda]$ ;
} /* end for */
/* ****************************************************************** */
/* begin scheduling! */
$t = 0$ ;
while (remainingDemand > 0) and ($t$ < 2(glb)) { /* while there is still unmet demand */
    for ($\lambda = 1 \ldots C$) {
        if (schedule[$t$][$\lambda$] == 0) { /* if no task has been assigned yet to this $\lambda$ at this slot */
            $n = 1$ ;
            while ( ($n \leq N$) AND ((unavailable[$n$][$t$] == 1) OR ($a[n][\lambda]$ == 0)) )
                $n = n + 1$ ; /* find an available node with unfulfilled demand on this $\lambda$ */
            if ($n \leq N$ ) {
                for ($i = t$ to $t + a[n][\lambda] - 1$)
                    schedule[$i$][$\lambda$] = $n$ ;
                for ($i = t$ to $t + a[n][\lambda] - 1 + tuneLatency$)
                    unavailable[$n$][$i$] = 1 ;
                remainingDemand = remainingDemand - $a[n][\lambda]$ ;
                $a[n][\lambda]$ = 0 ;
            } /* end if */
        } /* end if */
    } /* end for */
    $t = t + 1$ ;
} /* end while */
```

A `{SYNCSCHED}` frame may not be sent after offset `sf_length - ND`. If, after a schedule has been created, the master node first visits any wavelength during this “illegal interval”, then the schedule must be made artificially longer, by adding to the end of the schedule an amount of time equal to no more than `sf_length - ND`.

3 Multicast and QoS support in all-optical broadcast LANs

3.1 Multicast in the `Helios` Architecture

The `Helios` network nodes are equipped with fast tunable transmitters and slowly tunable receivers to form what is known as a FTT-STR architecture. For functions such as packet transmission and scheduling which operate at fine time scales (i.e., on the order of packet transmission times), the lasers are considered tunable and the receivers are considered fixed-tuned. The tunability of optical receivers is invoked only at longer time scales (i.e., on the order of seconds or hundreds of milliseconds) to address the issues of load balancing and multicast. In other words, we distinguish two regions of network operation: during the *normal operation* phase, the optical receivers remain fixed-tuned to their home channels, while during the *reconfiguration* phase [5], the receivers are slowly retuned to new home channels in order to optimize the network for the next normal operation phase.

Our objective was to design algorithms for native multicast over a FTT-STR environment. As a first step, we conducted a comprehensive survey of protocols and scheduling algorithms for multi-destination traffic in broadcast WDM networks. We classified the protocols based on the underlying strategy used to transmit multicast packets, as well as on their assumptions regarding the network architecture. We described in detail a number of existing approaches for scheduling both single- and multi-destination traffic. We discussed the advantages and disadvantages of each scheme, and we also identified the regions of network operation for which each strategy is most appropriate. The results of our work were published in [12]. One of the most important findings of our literature survey was that all existing protocols and algorithms assume that receivers are tunable. Therefore, we focused our efforts on developing new strategies to support multicast traffic in the fixed-receiver `Helios` network.

Let us assume that we have some information regarding the long-term multicast traffic demands in the network, including the number and composition of multicast groups, and let us further assume that this information is collected using the Helios protocol implemented at each node. Then, the problem of supporting multicast traffic in a FTT-STR broadcast WDM architecture is an optimization problem, whereby optical receivers must be assigned home channels such that a performance metric is optimized. The performance metric of interest in `Helios` is the *multicast throughput*, defined as the number of multicast completions per unit time, where a multicast completion refers to the transmission of a multicast packet to all members of its multicast group. We refer to this problem as the *multicast wavelength assignment* (MWA) problem, and we have shown in [11] that it is NP-hard.

The complexity of the MWA problem derives from two conflicting objectives that must be simultaneously satisfied. On the one hand, it is important to balance the traffic load across the different channels, while on the other hand it is desirable to assign receivers in the same multicast group to the same home channel to keep the multicast throughput high (otherwise, a multicast packet has to be transmitted multiple times, once to the home channel of the various receivers in its group). The problem is further complicated by the fact that multiple groups may not be disjoint, i.e., a given receiver may be part of multiple groups.

We have developed a number of heuristics for the MWA problem, which are described in detail in [11]. Here we provide a summary of their operation. The `Join` class of heuristics starts with each of the N receivers assigned to a separate channel, and repeatedly joins the receivers from two different channels by assigning them to a single channel, until the number of home channels is equal to the number $W, W < N$ in the network. The `GreedyJoin` heuristic applies a greedy rule in joining two sets of receivers, while the `RandomJoin` heuristic randomly joins two sets at each step. The `Split` class of heuristics starts with all N receivers in the network assigned to a single home channel, and then repeatedly selects one receiver to assign to one of the other $W-1$ channels. The `Join` class and `Split` class of heuristics take advantage of the *monotonicity* properties of the multicast throughput that were first derived in [8]. The `MLPT` heuristic takes a different approach. It first uses the largest processing time first (LPT) scheduling algorithm, which provides good load balancing, to come up with an initial wavelength assignment, which it then improves through an iterative approach.

Based on a wide range of results presented in [11], the `GreedyJoin` heuristic appears to provide the best approach for the MWA problem.

3.2 QoS Support in the `Helios` Architecture

The basic `Helios` scheduling algorithm is appropriate for best-effort traffic but does not provide any QoS guarantees. We have modified this scheduling algorithm [7] to provide native support for the differentiated services (DiffServ) architecture currently being standardized by the Internet Engineering Task Force (IETF). Providing bandwidth and/or delay guarantees in a multiwavelength environment is an inherently complicated task, due to the need to coordinate packet transmissions among the nodes across multiple wavelengths while at the same time attempting to meet packet deadlines; the problem becomes all the more difficult when the transmitting nodes have to account for non-negligible tuning delays. We provide a brief summary of the scheduling algorithm here; details and numerical results are available in [7]. Our work has also been published in [2].

The algorithm consists of two steps. First, an initial schedule is built based on traffic reservations for the two classes of DiffServ traffic that require bandwidth and/or delay guarantees, the Expedited Forwarding (EF) class and the Assured Forwarding (AF) class. This schedule is such that all nodes can meet the QoS guarantees for their EF and AF traffic. This initial schedule is then extended to assign transmission slots for best-effort (BE) traffic, using an algorithm that ensures two important properties in the final schedule: first, that the QoS of the EF and AF traffic is not compromised for any node; and second, that best-effort transmissions are assigned to the various nodes in a *max-min* fair fashion. This latter property guarantees that the excess bandwidth in a `Helios` network is allocated fairly among the network flows. Another important feature of our guaranteed-service scheduling algorithms is that they require only small changes to the basic `Helios` scheduling algorithm. Numerical results in [7] using our WDM simulator (see below) indicate that the algorithm works as expected and can provide QoS guarantees compatible with the DiffServ framework.

A significant contribution of our work was the implementation of a highly extensible simulator for evaluating the performance of the scheduling algorithms. Our simulator builds upon the functionality provided by the DiffServ model contributed by Nortel Networks to the popular simulator tool `ns-2`. Before our work, `ns-2` lacked support for WDM (i.e., multi-channel) links. Our WDM simulator was integrated into `ns-2` by mapping a model of a `Helios` node into an `ns-2` topology. The details of the mapping can be found in [7], while the computer code is available at [1] and can be easily incorporated into an existing `ns-2` installation. We believe that our simulator addresses an important need and we hope that it will be useful to other researchers in the field.

3.3 The Scheduling and Wavelength Assignment Problem

We also considered a scheduling problem, which we call the *Scheduling and Wavelength Assignment* (SWA) problem, arising in *Helios*-like optical networks based on WDM technology. Consider a WDM network with n nodes interconnected via a broadcast star that supports $m < n$ distinct wavelengths. Nodes communicate by exchanging fixed-length packets, and the time it takes to transmit a packet is taken as the unit of time. Since there are fewer wavelengths than nodes, packet transmissions by several nodes may share a single wavelength, and the problem of scheduling these packet transmissions arises. At the same time, an important objective in such a network is load balancing across the different wavelengths, since it has been shown that network performance deteriorates significantly if the traffic load concentrates on a few wavelengths [3, 4, 5].

Let us assume that the long-term traffic requirements of the nodes are known. Let us also assume that the nodes are equipped with fast-tunable transmitters, so that no cost is incurred when a transmitter switches from one wavelength to another, but that receivers are fixed-tuned to a certain wavelength (these are tunability characteristics of nodes in the `Helios` network). Then, the SWA problem is such that all flows of packets between a source-destination pair must be simultaneously assigned a wavelength and scheduled for transmission. Note that, minimizing the makespan for this problem implies balancing the traffic across the various wavelengths by properly assigning the fixed-tuned receivers to wavelengths.

In [6], we proved that the SWA problem is NP-complete for both the preemptive and the non-preemptive cases. Furthermore, we proposed two efficient approximation algorithms. The first is for the preemptive case and it is based on a natural decomposition of the problem to the classical multiprocessor scheduling and open-shop problems. For the non-preemptive case, we proved that a naive implementation of list scheduling produces a schedule that can be m times far from the optimum, where m is the number of processors (equivalently, WDM channels). Finally, we developed a more refined version of list scheduling algorithm and we proved it to be a 2-approximation algorithm for both the off-line and the on-line contexts.

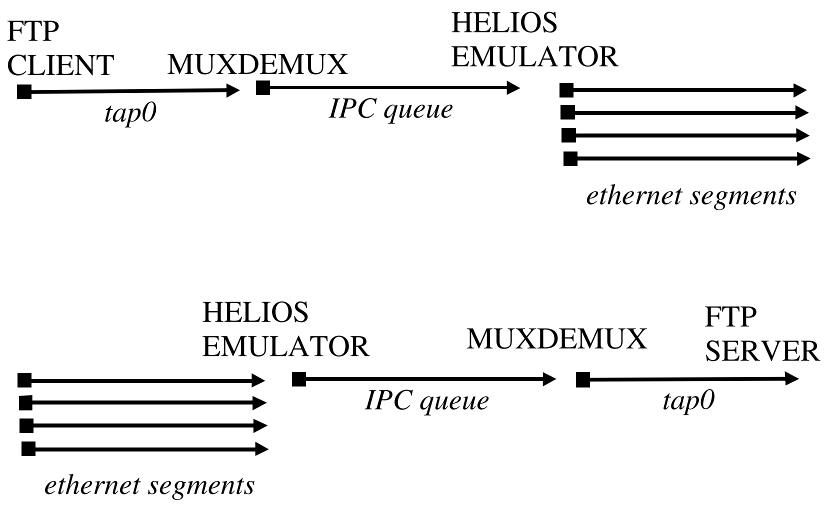


Figure 22: Datagram flow

4 Helios Protocol Testbed

4.1 Testbed setup

The goal of the Helios Protocol Testbed was to verify the validity of the HiPeR-l protocol definiton in a near-hard real-time environment with some hardware elements. The testbed was set up using off-the shelf components - 6 x86 PCs, each equipped with 5 Ethernet 100BT NICs and 5 Ethernet hubs. Each PC was connected to each one of the hubs, thus creating an isolated segment on each of the hubs. Four of the segments were used to model wavelengths and the fifth was used for administrative access. The "tuning" of each of the nodes' receiver or transmitter was achieved by selecting the appropriate interface to transmit or receive information. Additionally, on the receive side, any frames received on the NICs currently not deisgnated as receive wavelength (usually 3 out of 4) were discarded by each node locally.

In order to provide near-hard real-time environment, each of the PCs was loaded with RedHat linux 6.2 (kernel 2.2.16) augmented with KURT (Kansas University Real-Time linux) kernel patches. These patches allow user-space processes (as opposed to the kernel) to achieve low-millisecond timing resolution. Additionally, the PCs were synchronized using NTP to within hundreds of microseconds through a stratum 1 NTP server feeding off the GPS signal via a Trimble antenna placed on the roof of the lab building. The server was located on the same subnet to ensure precise synchronization.

4.2 Software

From the start the software for testing the HiPeR-l protocol on this testbed was written in such a way that regulat ip-based applications (like ftp, http and ssh) could be used to test the protocol. The protocol emulator driver was a user-space application runing near-hard real-time that accepted IP datagrams from other applications, applied HiPeR-l scheduling and sent them out. On the receive side, HiPeR-l frames received on various "wavelengths" were decapsulated and forwarded back to the ip applications. This was achieved by writing a small pseudo-network driver that presented a small "fake" IP interface to the applications running on the PC. Frames sent by applications to this interface were forwarded to the HiPeR-l emulator driver via SysV IPC message queues. The opposite took place for frames received by the HiPeR-l emulator driver. Figure 22 shows a typical setup with an FTP client.

The emulator driver implemented HiPeR-l parsing and a subset of state machines defined for the protocol (some of the state machines, notably the Time Measurement state machine, were not tested due to the limitations of the testbed, instead the time sychronization between nodes was achieved via NTP driven by GPS signals). This allowed us to test the definition of the frames as well as state machines in a real-time environment, but without implementing them in hardware.

The implementation was successfully tested using such applications as ftp and ssh. Using the HiPeR-l emulator driver nodes were able to communicate with each other according to the HiPeR-l protocol. The code for the implementation is included as part of this project report.

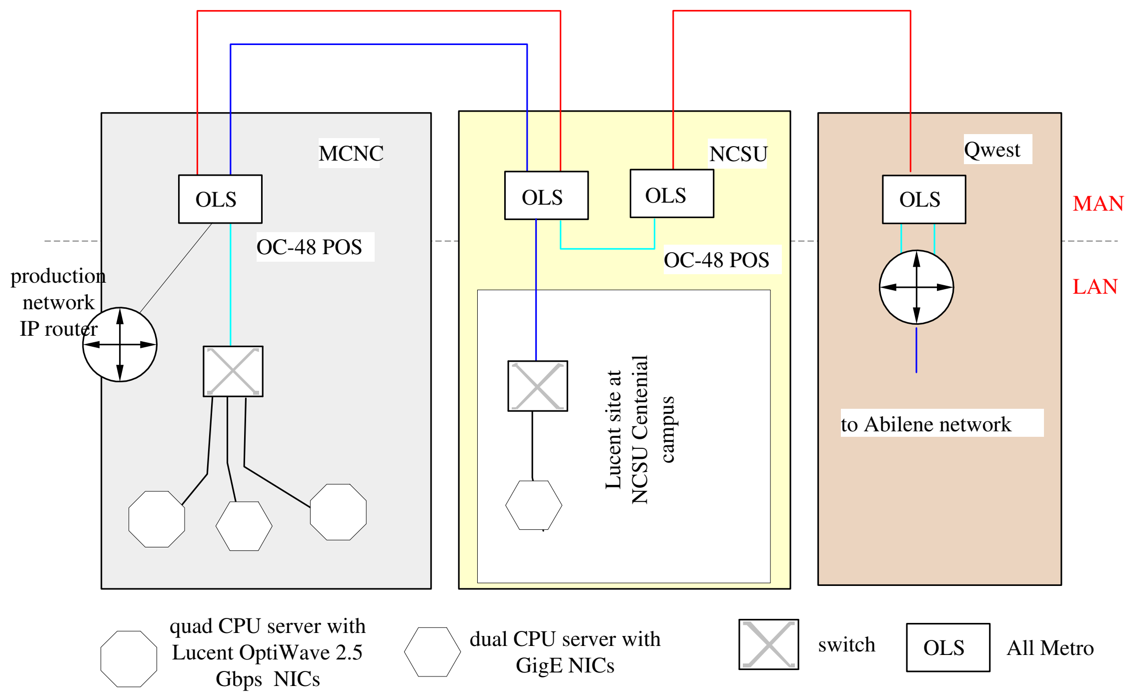

MCNC
NCSU
Qwest
OLS
OLS
OLS
OLS
OC-48 POS
OC-48 POS
MAN
LAN
production network IP router
Lucent site at NCSU Centenial campus
to Abilene network
quad CPU server with Lucent OptiWave 2.5 Gbps NICs
dual CPU server with GigE NICs
switch
OLS
All Metro

Figure 23: Application Testbed Topology

5 Helios Applications Testbed

5.1 METRO AREA DWDM NETWORK

Prior to starting the Helios project Lucent made an equipment donation to MCNC of a two pairs of OLS-40 multiplexers These were installed and went into production in operational the fall of 1999. The OLS-40 systems were used to provide a pair of wavelengths for production use as part of MCNC operated statewide Internet service provided for State Government and the University system. An addition pair of wavelengths (OC-48) were available for use in the Helios project. The major effort associated with this task was to connect the servers located in our lab at MCNC to the Helios project server located in the Lucent test lab using these facilities.

Although a straightforward task, progress in this effort was frustratingly slow. The difficulties seem to have been largely communication, motivation and coordination of configuration and installation activities crossing several organizational and institutional boundaries. Specifically, establishing these connections required the cooperation of staff from MCNC, NCSU and Lucent. We also encountered technical challenges related to lack of good debugging tools. Ultimately we did establish connectivity between MCNC and Lucent over the OLS at 2.5 Gbps. Figure illustrates the equipment involved.

As indicated above, four 2.5 Gbps wavelengths were available on each of two spans although only tow of these were utilized. One span ran between MCNC and the NCSU Centennial campus, the second between the Centennial campus and the Qwest point of presence in Raleigh. One of these wavelengths was used to carry production IP traffic from MCNC to the Internet 2 PoP at Qwest in Raleigh. The second wavelength was used to connect servers between MCNC and the Lucent office on the NCSU Centennial campus at NCSU. At each of the locations a Lucent OptistarES IP switch was used to interconnect servers at the two locations. The two quad CPU servers were equipped with OptiWave 2.5 Gbps network adapters (also provided by Lucent). The dual header servers were equipped with 1 Gbps Ethernet adapters. All server network adapters attached to ES switches.

Ultimately closing of the Lucent site at the NCSU Centennial campus caused us to discontinue the use of these network connections. This Lucent office had been the home of one of the servers used in the applications testbed. We explored options for locating the server to other space on the centennial campus but we were unsuccessful in these efforts. Consequently we returned the server to our lab at MCNC and it is currently operational. The OLS-40 continues to be used for the production IP network that MCNC operates for the statewide university system.

At the end of the project these servers became part of a Grid computing cluster that is being used as a testbed in other research efforts at MCNC.

5.2 APPLICATIONS

Two data intensive applications were targeted for use of the server cluster described above, meteorology and Internet traffic analysis. The meteorology application, unfortunately, did not make use of the cluster until after completion of the project.

A. Traffic Measurements and Analysis

The traffic monitoring team consisted of researchers from MCNC, the University of North Carolina Department of Computer Science, the Mathematical Sciences Research Center of Bell Labs, and the Computer Science Research Center of Bell Labs:

The overall project plan was to collect packet trace files from the North Carolina Internet2 GIGO, move the files to the Helios cluster, process the files to form GigaPoP and TCP flows, create primary S objects for analysis, and analyze the data within the S environment.

The NC Internet2 GigaPop features a distributed architecture consisting of a SONET ring with major gateway nodes at four locations, UNC, MCNC, NCSU and Duke. Access from each campus to the gigapop ring is via a GiGE Ethernet connection. Data collection was carried out at the UNC gateway. The data consisted of time stamped packet headers with unencrypted addresses. The data collection device consisted of commodity PCs running BSD Unix equipped with Syskonnect GiGE Ethernet adapters. The data collection code was written by UNC and was based on the use of TCPDump.

Collection and analysis of data posed significant non-technical issues involving privacy protection through "en-crypting" packet source and destination header fields. The practice, common in the traffic measurement community, is an anathema to the traffic analysis community, since it eliminates significant fields of scientific inquiry involving source and destination correlations. To bridge this gap the PI negotiated a Non-disclosure agreement with the UNC administration. The agreement permitted collection and analysis of unencrypted packet headers, but disclosure of this data was restricted in ways that barred exposure of identities of any of the network's users. UNC was also given the right to pre-review publications based on analysis of the collected data and could restrict content that it felt disclosed private information. We contend that this agreement is a model that can be widely used to resolve conflicts between privacy protection and scientific inquiry in the traffic measurement and analysis research communities.

The data consist of 42 hours of collection on the 1 Gb/s Ethernet (GbE) link that connects the UNC campus to the NC GigaPoP ring. These 42 traces were taken 6 traces per day for one week. The hours were scattered throughout a day (8:30, 11:00, 13:30, 16:00, 19:30, and 22:00) in order to take into account of busy and non-busy hour traffic load.

Following traffic collection effort, we proceeded with processing these data sets into TCP flows. The traffic volume per hour ranges from a 1.5GB to 12GB. It took from 3 to 15 hours on a quad CPUs machine on the Helios cluster to extract header information and correlate both inbound and outbound traffic in order to sort them into TCP flows. These flow files were fed into Splus to create S objects. The S objects were then subjected to extensive statistical analysis. This analysis included inter-arrival plots, Weibull probability plots, quantile plots, bandwidth analysis, and power spectrum analysis. The results from these analysis have indicated a good fit between our model and data.

Our first target was investigation of HTTP protocol traffic. Additional protocols like FTP (file transfer), SMTP (email), and NNTP (news) were also analyzed along with audio and video real-time streaming application traffic which are based on UDP protocol.

Lucent has developed a new sniffer by modifications to on-board code of an off the shelf GiGE network adapter. The salient feature of this sniffer is the use of on-board time-stamping. Our Syskonnect-based sniffer did off-board time-stamping. This process involved accumulation of multiple packets in the adapter buffer. When the software was interrupted to serve this buffer the current clock value was stamped on all the retrieved packet headers. This relatively low resolution process masks some of the timing events we want to be able to observe. We developed plans with UNC to take additional data with the new higher resolution sniffer, but ultimately not undertake this data collection.

In addition to the above effort Lucent has obtained an another sniffer (referred to as DAG boards) for OC-12 PoS links through its relationship with University of Waikato. We are contemplating deploying these sniffers on some of the production IP network links operated by MCNC. Specifically under consideration were the three connections

MCNC maintains to the commodity Internet through Qwest, Sprint and UUNET and the Abilene connection. This would have provided for a much richer set of statistics since approximately 60 universities across the state and all NC State Government offices share the commodity Internet links. This activity was abandoned once we realized that it would require us spending the rest of our careers negotiating agreements with the legal staff at 55 different universities.

The original data collection we undertook targeted 1) development of "network aware" traffic generators for performance testing of routers and switches. and 2) application specific statistical models that could be used for traffic engineering studies. The functional objective for the anticipated measurements with the DAG board sniffers is to be able to collect and analyze traffic data with sufficiently low time delays so that it might be useful for network operations. Lucent ultimately collected such data from an Internet Service Provider and used the Helios applications cluster for data storage and processing for this effort. B. Other Applications

Towards the later stages of the project the DARPA program manager requested that the PI identify applications that potentially needed multiple gigabit per second network connections that could justify connection to the existing Supernet backbone network With significant help and input from the local research community, the PI identified four candidate applications and briefed them to DARPA.

The recommended approach for providing Supernet connectivity involved using BossNet to connect UPenn, to HSCC and upgrading the existing Abilene connection for NCNI to OC-48. In addition there were various requirements for local router upgrades to support local access. This was the hands down lowest cost approach of several considered.

Connecting NCNI to Abilene and UPenn to HSCC required an expanded peering relationship between Abilene and Supernet. Various conversations with UCAID lead us to conclude that they will not support an expanded peering relationship, although this has never been directly stated. Subsequently, we came to see the recommended approach as not being electro-politically viable.

6 VCC Collaboration and Weather Modeling

6.1 Virtual Collaborative Center Overview

MCNC's Virtual Collaborative Center (VCC)[1], funded by NASA beginning in 2001, was established to accomplish the goal of bringing additional university based applications to the commodity Internet and to Internet 2 by providing critical network expertise not otherwise readily available, and by providing the research and development of middle-ware and collaborative tools needed to facilitate networked collaboration. Its objectives include targeting several categories of end users including high demand collaborative applications requiring bandwidth, latency and or jitter performance end to end; and collaborative applications requiring high demand network access with distributed computing and large data sets.

As part of the VCC effort, MCNC has established a VCC testbed to support this research. The testbed features two Linux based clusters with several Tera-bytes of available storage, a Virtual Private Network to protect access to the systems, Cisco Catalyst switches which enable communications among the systems, and additional equipment of various types.

- The VCC testbed has been used in various ways to support the following research areas:
- Just In Time Optical Burst Switching Protocol Software Development
- GridFTP over JIT
- Grid Information Retrieval
- High Performance Computing and Collaborative Applications
- AN-MSI Collaboration Center

The remainder of this section describes the above items in greater detail.

6.2 Just In Time Optical Burst Switching Protocol Software Development

MCNC is developing a Just-In-Time (JIT) signaling protocol for use on Optical Burst Switching (OBS) based Dense Wavelength Division Multiplexing (dWDM) networks. The Jumpstart project's[2] goals include the creation of a signaling protocol and architecture for such networks, along with a prototype implementation of the protocol with software components associated with both the optical switches and the client-side nodes, as well as hardware components associated with the switches.

6.2.1 GridFTP over JIT

From the VCC project's perspective, the advent of the Jumpstart project's (described elsewhere in this document) JIT signaling protocol on Optical Burst Switch networks holds promise of reduced latency and increased throughput for a variety of applications, including Grid-related software such as the Globus middleware suite. The VCC has thus been pursuing integration of the results of the JIT research with Globus-related technologies.

Bulk-data transfer is viewed as an early beneficiary of the JIT research; therefore GridFTP[3] is being looked at first. GridFTP at present utilizes IP under the covers for its data transport. In a series of stages we have been moving towards porting GridFTP over OBS on the VCC testbed.

6.2.2 Grid Information Retrieval

VCC team members are leading the Grid Information Retrieval (GridIR) Working Group[4], which was officially accepted by the Global Grid Forum[5] in December 2002. GridIR is a newly proposed initiative to implement a specific

[1] http://www.vcc.cnidr.org
[2] http://jumpstart.anr.mcnc.org
[3] http://www.globus.org/datagrid/deliverables/C2WPdraft2.pdf
[4] http://www.gridir.org
[5] http://www.ggf.org

architecture for realizing information retrieval on the The Open Grid Services Architecture (OGSA)[6] grid computing platform. The working group has met once and has published an early draft of the Grid Information Retrieval Requirements draft document.

A prototype software implementation is underway on the VCC testbed, that will develop into the GridIR reference implementation. The software includes the ability to do rudimentary collection and index management. Its expected release date is summer, 2003.

6.3 High Performance Computing and Collaborative Applications

We have endeavored to benefit the wider scientific community through access to the high-performance computational technologies in the VCC: the clusters' hardware; software such as MPI, PVM, etc; and Grid middleware such as Condor, Globus, OGSA, etc.

For example, during the previous year, we have worked with multiple external collaborators, including but not limited to the University of North Carolina at Chapel Hill's Carolina Environmental Program, North Carolina State University's Computer Science Department, and Barons Advanced Meteorological Systems' Environmental Modeling Center. These external collaborators' projects are funded by a variety of sources, including the U.S. Environmental Protection Agency (EPA), the U.S. National Security Administration (NSA), the Electric Power Research Institute (EPRI), as well as directly by the collaborators' home institutions.

6.4 AN-MSI Collaboration Center

The Advanced Networking with Minority-Serving Institutions (AN-MSI) Center is intended to help members of the AN-MSI community share a variety of instructional and support resources, learn about current projects and activities, develop collaborative project ideas, and make new friends and partners. The Center is also a place where AN-MSI faculty, staff and students can find a variety of tools and resources designed to support interdisciplinary teaching, learning, research, and information technology technical assistance.

6.5 Helios/VCC Testbed Integration

As part of the Helios effort, MCNC and MCNC-RDI have:

- augmented the VCC testbed to include an additional compute node
- installed and configured the Condor[7] high throughput middleware on the new compute node
- submitted environmental simulations from the VCC testbed to the new compute node for execution

The remainder of this section describes the above items in greater detail.

6.5.1 Adding an Additional Compute Node to the VCC Testbed

A 546 Mhz dual-CPU Pentium III compute node was networked to the VCC testbed to augment the computational power of the two clusters. The new compute node is named sunspot0, and it resides in the main building on the MCNC-RDI campus. The rest of the VCC testbed resides in another building on the same campus. A 1 Gbps ethernet cable was used to connect the sunspot0 node to the testbed; the relevant parts of the newly configured testbed are shown in Figure 24.

The two clusters, louie0..louie7 and duey0..duey7, are each made up of 8 dual-CPU AMD Athlon 1800+ systems. The addition of the sunspot0 node brings the total number of CPUs within the portion of the VCC testbed shown in the figure to 34.

The primary storage for the testbed is provided by a a Fibre Channel DotHill 7100 SANNet device, which has a total of 3.6 Terabytes of capacity. Approximately 700 Gigabytes of this is allocated to the head node of the louie cluster (louie0), and approximately 350 Gigabytes is allocated to the head node of the duey cluster (duey0); from

[6]http://www.globus.org/ogsa

[7]http://www.cs.wisc.edu/condor

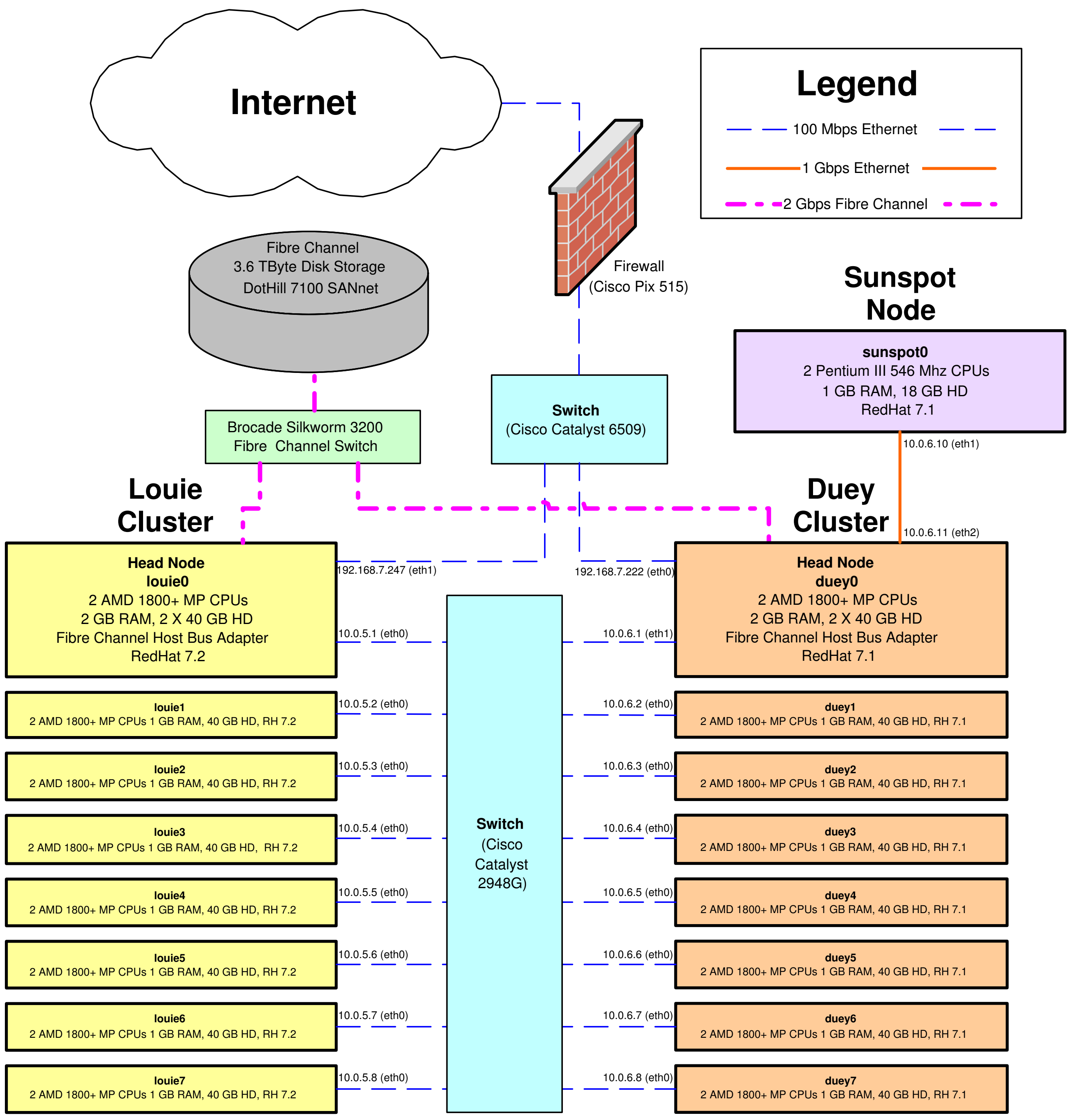


Figure 24: Cluster Topology

the respective head nodes these partitions are mounted to the slave nodes using NFS. The rest of the Fibre Channel device's storage is allocated to other machines within the VCC testbed that are not shown within the figure.

At last count in May of 2003, there were thirty active user accounts on the system. These accounts are (except for a handful of users) provided only on the louie cluster and are administered via NIS so their login information propagates across all of the louie nodes. Users access the testbed by first logging into louie's head node (louie0), and from there they can do development/testing either on the head node or by connecting to any of louie's slave nodes.

Direct login access to the duey cluster's nodes was not provided to most users for two reasons. First, this is the more experimental of the two clusters. It is used for testing various kernel flavors as part of the Jumpstart related activities, and therefore these nodes are much more likely to be shutdown and restarted at any given time than the nodes within the louie cluster. Second, and perhaps more importantly, not allowing user accounts on the duey cluster is a strategic decision made to provide a "carrot" to entice users of the louie cluster to avail themselves of Grid computing middleware to access remote computational resources. Taking advantage of appropriate middleware such as Condor, users can simply submit their computations to the system without regard as to where the execution will actually take place - the middleware manages execution on behalf of the user. In this way, the user is free to concentrate on setting up their computations with less effort devoted towards the mechanics of identifying suitable resources for the actual job execution.

The sunspot0 node, added to the VCC testbed by the Helios project, was also configured with almost no user accounts, for the same reasons described above for the duey cluster. Sunspot0 was networked directly to a Gigabit Ethernet card on the duey head node (duey0). The duey0 node was configured to use Proxy ARP in order to allow all of the louie and duey nodes to communicate with sunspot0. Proxy ARP allows duey0's eth1 interface to respond to ARP requests for sunspot0's 10.0.6.10 eth1 address, in addition to duey0's own eth1 address of 10.0.6.1. The networking system within duey0's kernel passes IP traffic along, among sunspot0 and all of the other nodes within the VCC testbed.

6.5.2 Condor Installation and Configuration

Condor middleware is available from the University of Wisconsin-Madison's Computer Science Department. It is a mature product which has been undergoing development and periodic releases since 1988.

Here is a brief overview of Condor, excerpted from the Condor project's web pages[8] :

> Condor is a workload management system for compute-intensive jobs. It provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
>
> Condor enables effective harnessing of wasted CPU power from otherwise idle compute nodes or desktop workstations. For instance, Condor can be configured to only use desktop machines where the keyboard and mouse are idle. Should Condor detect that a machine is no longer available (such as a key press detected), in many circumstances Condor is able to transparently produce a checkpoint and migrate a job to a different machine which would otherwise be idle. Condor does not require a shared file system across machines - if no shared file system is available, Condor can transfer the job's data files on behalf of the user, or Condor may be able to transparently redirect all the job's I/O requests back to the submit machine. As a result, Condor can be used to seamlessly combine all of an organization's computational power into one resource.
>
> The ClassAd mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (jobs) with resource offers (machines). Jobs can easily state both job requirements and job preferences. Likewise, machines can specify requirements and preferences about the jobs they are willing to run. These requirements and preferences can be described in powerful expressions, resulting in Condor's adaptation to nearly any desired policy.
>
> Condor can be used to build Grid-style computing environments that cross administrative boundaries. Condor's "flocking" technology allows multiple Condor compute installations to work together. Condor incorporates many of the emerging Grid-based computing methodologies and protocols.

[8]http://www.cs.wisc.edu/condor/description.html

A 2-CPU Condor "pool" was configured on the sunspot0 node; then it and the louie and duey Condor pools (16 CPUs each) were all configured such that jobs submitted to any of the pools can migrate among any of the nodes, depending on where the available CPU cycles are. In this way, users submitting Condor jobs from the louie head node might have their jobs executed on any of 34 available CPUs in the testbed - regardless of whether they have an account on the execution machine.

Coincidentally, at the very first moment when the sunspot0 node was added the the available Condor pools, compute jobs immediately began migrating to it. The VCC team has been collaborating with North Carolina State University's optical networking group since February 2003. Their work involves sophisticated simulation models of optical burst switching networks, and unfortunately the computational facilities at their NCSU lab were severely limiting their ability to carry out meaningful performance studies. Jing Teng, a PhD candidate in this department, already had several thousand jobs under submission from the louie0 node. Using the ClassAd mechanism, some of his jobs immediately found a match when sunspot0 came on line, and the CPUs became fully utilized. This increased throughput was achieved without Jing ever becoming aware of the new sunspot0 node's availability - illustrating one of the key benefits of the Condor middleware.

6.6 Executing Environmental Simulations

To fully exercise the newly added sunspot0 node, the VCC team submitted several executions of an environmental simulation to the system. The model chosen was the Community Multiscale Air Quality Modeling System (CMAQ), as this particular model was already in use within a separate VCC collaboration.

CMAQ is a modeling system for urban to regional scale air quality simulation of tropospheric ozone, acid deposition, visibility and fine particulate matter. Composed of a chemical transport model (CCTM), a meteorology data preprocessor (MCIP), initial and boundary conditions processors (ICON/BCON), and a photolysis rate processor (JPROC), CMAQ is designed as a "one atmosphere" model to treat multiple pollutants simultaneously at several spatial scales. For more detailed information on the particulars of the model, see the CMAQ homepage[9].

An example of the results from the CMAQ model as simulated on the sunspot0 node is shown in Figure 25. The output files were visualized using the Package for Analysis and Visualization of Environmental data (PAVE), which generated this image of ground level ozone as simulated over the eastern U.S. at the (simulated) time of 1:00 AM on June 30, 1999.

The authors are grateful to Elizabeth Adams, Rohit Mathur, and Zac Adelman of the University of North Carolina at Chapel Hill's Carolina Environmental Program for their assistance with the CMAQ modeling system.

[9] http://www.epa.gov/asmdnerl/models3/overview.html

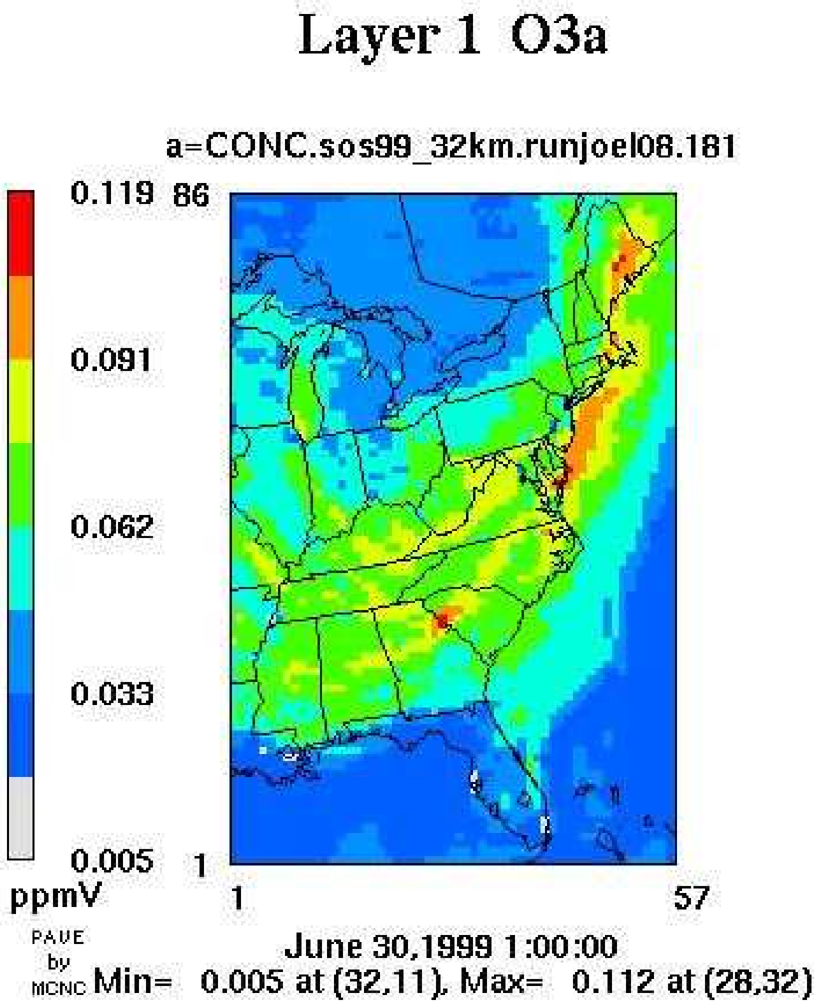


Figure 25: Weather Model

Appendix A: HiPeR-l Timers and Counters

Name	Max Value	Used By	Description
checking_timer	T1	>elect<	a node listens on LAMBDA_RCV to check whether a scheduler already exists
silent_timer	T2	>elect<	a node listens for other AVAIL frames
avail_echo_timer	T3 - epsilon	>elect<	a node listens for its own AVAIL transmission to verify that its own receiver is working
announced_timer	T3	>elect<	a node listens for other AVAIL frames with higher MAC addresses
backoff_timer	T4	>elect<	collision resolution
get_sched_timer	T_GET_SCHED	>tm<	a node listens for a SYNCSCHED to learn when the TM window will occur (i.e. TM_LATENCY)
no_tm_count	NO_TM_MAX	>tm<	a node keeps track of how many SYNCSCHED frames go by w/o a TM window
get_tm_count	GET_TM_MAX	software	software allows >tm< to fail (as a result of not hearing a SYNCSCHED w/ a TM window) for a total of GET_TM_MAX times, before giving up
tm_timer	{ss.time_till_tm}	>tm<	a node waits for TM window to arrive, so that it can send a TM frame
echo_timer	T_ECHO	>tm<	a node listens for its own transmission
get_sched_timer	T_GET_SCHED	>join<	a node listens for a SYNCSCHED to learn when the JOIN-OCC window is scheduled
same_count	SAME_MAX	software	software allows >join< to fail (as a result of hearing a SYNCSCHED with its own ID in it) for a total of SAME_MAX times, before giving up
same_timer	T_SAME	software	a node waits (sleeps) for T_SAME before retrying >join<
old_sched_count	OLD_SCHED_MAX	>join<	a node waits to hear a SYNCSCHED with a new schedule (i.e. one which includes NODE_ID)
sched_switch_count	C_SCHED_SWITCH	>routine<	Countdown to new schedule
get_sched_timer	T_GET_SCHED	>routine<	A SYNCSCHED should be heard every superframe during routine mode

Appendix B: HiPeR-l Signals

Name	From	To	Description
START_ELECT	sw	>elect<	begin scheduler election
SEND_AVAIL1	>elect<	<elect>	the node moves from being a silent contender to being an announced contender
SEND_AVAIL2	>elect<	<elect>	the node moves from being an announced contender to being the master node
MASTER	<elect>	sw	the node has won the scheduler election
NOT_MASTER	>elect<	sw	the node has lost the election; another node has become (or is becoming) the master node
ROUNDTRIP_TIME	>elect<	sw	the node has successfully measured the round-trip Xmission time of AVAIL to the PSC
NO_REPLY	>elect<	sw	the node failed to receive its own AVAIL
START_TM	sw	>tm<	begin time measurement
NO_SCHED	>tm<	sw	failed to hear a SYNCSCHED within T_GET_SCHED
NO_TM_WINDOW	>tm<	sw	saw NO_TM_MAX superframes in a row without the tm bit set (indicating presence of a TM window)
SEND_TM	>tm<	<tm>	TM window has arrived, so send TM frame now
ROUNDTRIP_TIME	>tm<	sw	the node has successfully measured the roundtrip transmission time of a TM frame to the PSC
NO_REPLY	>tm<	sw	the node failed to receive its own TM frame
START_JOIN	sw	>join<	start join process
NO_SCHED	>join<	sw	failed to hear a SYNCSCHED within T_GET_SCHED
NO_SCHED	>tm<	sw	failed to hear a SYNCSCHED within T_GET_SCHED
JOIN_OCC_START_TIME	>join<	<join>	the receiver informs the xmitter when the JOINOCC window will occur
SAME_ID	>join<	sw	before sending JOINOCC, a SYNCSCHED was heard which included the node's ID
START_REXMIT_TIMER	<join>	>join<	xmitter has just sent a JOIN-OCC frame; wait for T_REXMIT to hear a SYNCSCHED with a new schedule (one which includes NODE_ID)
NO_NEW_SCHED	>join<	sw	REXMIT_TIMER has expired without having heard a SYNCSCHED with a new schedule (one which includes NODE_ID)
NO_ACTIVE_SCHED	>join<	sw	inactive_count expired; no active sched heard within INACTIVE_MAX
NEW_SCHED	>join<	sw	a SYNCSCHED was heard with a new schedule (one which includes NODE_ID)
START_XMIT	sw	<routine>	begin routine mode of xmitting frames according to the schedule
BANKS_INVALID	<routine>	sw	at the start of routine mode, neither bank holds a valid schedule; cannot xmit
BANKS_INVALID	<scheduling>	sw	at the start of scheduling mode, neither bank

Name	From	To	Description
NO_VALID_SCHED	<routine>	sw	upon switching to the other bank, it was found to be invalid
NO_VALID_SCHED	<scheduling>	sw	upon switching to the other bank, it was found to be invalid
UNEXP_INVALID_BANK	<routine>	sw	at the end of a schedule, the current bank is unexpectedly found marked invalid
UNEXP_INVALID_BANK	<scheduling>	sw	at the end of a schedule, the current bank is unexpectedly found marked invalid
START_ROUTINE	<routine>	>routine<	begin routine mode of receiving xmissions and listening for SYNCSCHED
START_SCHEDULING	sw	>scheduling<	begin scheduling receive state machine
START_SCHEDULING	sw	<scheduling>	begin scheduling transmit state machine
RECALC_SCHEDULE	>scheduling<	sw	a JOINOCC was received; must recalculate schedule
NO_SCHED	>routine<	sw	get_sched_timer expires; no SYNCSCHED was received in the last T_GET_SCHED
NO_SCHED	>scheduling<	sw	get_sched_timer expires; no SYNCSCHED was received in the last T_GET_SCHED
NOT_IN_SCHED	>routine<	sw	SYNCSCHED was received which failed to include NODE_ID
NOT_IN_SCHED	>scheduling<	sw	SYNCSCHED was received which failed to include NODE_ID
START_OLD_SCHED_COUNT	<join>	>join<	begin looking for a SYNCSCHED containing a newly calculated schedule
STOP_ROUTINE	<routine>	>routine<	BANK0 is invalid at the very start of routine mode
STOP_SCHEDULING	<scheduling>	>scheduling<	BANK0 is invalid at the very start of scheduling mode

Appendix C: HiPeR-l Constants

Name	Used By	Description
TM_FREQUENCY	scheduling algorithm	The scheduling node includes a TM window every TM_FREQUENCY superframes
BANK0, BANK1	<routine> >routine< <scheduling> >schedul<	Two memory banks in which a schedule can be stored.
NO_TM_MAX	>tm<	Max number of SYNCSCHED frames w/o a TM window that a node can hear before exiting >tm<
T_GET_SCHED	>tm< >join<	Max time within which a node must hear a SYNCSCHED frame (otherwise there's no scheduler or its receiver is broken)
T_ECHO	>tm<	Max time within which a node must hear its own transmission of a TM frame (otherwise its receiver is broken)
OLD_SCHED_MAX	>join<	After sending a JOIN-OCC frame, the max number of SYNCSCHED frames w/o a new schedule (one which includes NODE_ID) that may be heard before exiting >join<
T1	>elect<	checking_timer in >elect<
T2	>elect<	silent_timer in >elect<
T3	>elect<	announced_timer in >elect<
T3 - epsilon	>elect<	avail_echo_timer in >elect<
K	>elect<	constant used in calculating backoff time in >elect<
LAMBDA0	>elect<	
NODE_ID		MAC address of a node
VALID, INVALID	>routine< <routine>	possible values for BANK0 and BANK1; indicates whether a bank holds a valid schedule or no
CNTDWN	>routine< <routine> >scheduling< <scheduling>	all 0's except a 1 in the countdown position; if the schedule currently being disseminated is not yet "active", then the countdown bit in status_flags is set
BANK_NEWCALC	<scheduling>	the memory bank in which software writes a newly-calculated schedule
BANK_CURFRAME	<scheduling>	the memory bank holding the pre-built SYNCSCHED frame which contains the schedule that is currently being disseminated
SAME_MAX	signaling controller	max number of times >join< is allowed to experience the error "SAME_ID"
T_SAME	signaling controller	length of time a node will sleep after experiencing the error "SAME_ID", before reattempting >join<

Appendix D: HiPeR-l Shared Memory

Name	Description
cur_bank	Currently used schedule memory bank. Can be equal to either BANK0 or BANK1
cur_queue	Currently serviced wavelength queue
cur_schedchunk	Currently serviced schedchunk of the schedule
cur_sched_lambda	current wavelength on which the scheduling node is listening
cur_schedule	Pointer to the current schedule stored in either BANK0 or BANK1.
cur_time	current time
num_schedchunks(BANKx)	number of schedchunks contained in the schedule in BANKx
psc_offset	one-way propagation delay to the PSC
psc_sf_start	the time (according to node's local clock) at which the PSC sees the superframe start
psc_sf_start_next	holding location for psc_sf_start (until the start of the next superframe
rcv_lambda	a node's current receive wavelength
r_ss	receive timestamp of {syncsched}
sf_length(BANKx)	the superframe length for the schedule contained inBANKx
switch_count	number of superframes in which to use the schedule in cur_bank before switching to ! cur_bank
T_jo	offset (from the start of the superframe) of JOINOCC
T_ss(BANKx)	offset (from the start of the superframe) of SYNCSCHED
tm_in	the arrival time of the reflected TM
tm_out	the timestamp of TM (time at which it was sent out)
xmit_join_occ	Time (according to node's local clock) at which to send a JOINOCC
xmit_lambda	cur_schedule(cur_schedchunk).lambda
	the current lambda on which a node may transmit
xmit_last	the latest instant in time (according to node's local clock) at which the node may transmit on xmit_lambda
xmit_start	the time (according to node's own clock) at which the node may begin transmitting on xmit_lambda

parameter							units
time sync tolerance	10	40	100	40	10	40	ns
guard band	50	200	500	200	20	80	bits
min superframe	2.5	2.5	2.5	2.5	1	1	Gbps
serial bit rate	5000	20000	50000	20000	2000	8000	bits
2^N min superframe	8192	32768	65536	32768	2048	16384	bits
N	13	15	16	15	11	14	bits

Table 10: Minimum frame size vs. changing parameter values

Appendix E: HiPeR-l Guardbands

There is a relationship between timing tolerance, link rate, guard band size, and minimum superframe size. Table 10 shows the minimum frame size resulting from various values of these parameters. The experimental protocol will be designed to support timing accuracy to within 100 ns. This total tolerance budget will be allocated among various system components as follows.

5 ns propagation difference between transmit and receive paths for a node. This implies that the pair of fibers for each node are cut to within five feet of the same length (and that index of refraction variations from fiber to fiber are not significant).

50 ns local clock synchronization error

40 ns other

The system timing tolerance budget above, plus other sources of errors, define the size of superframe guard bands which are used to tolerate imperfections in the synchronization methods. At a data rate of 2.5 Gbps per wavelength and a 10 ns system timing tolerance specification, the minimum guard band size will be 512 bits. This figure suggests a minimum superframe size of 65,536 bits. The resulting minimum time interval between successive superframes is about 25 μseconds.

References

[1] WDM support in ns-2. In *http://www.csc.ncsu.edu/faculty/GRouskas/NS/*.

[2] Ilia Baldine, Laura Jackson, and George N. Rouskas. Helios: A Broadcast Optical Architecture. In *Proceedings of Networking 2002*, pages 887–898, May 19-24 2002.

[3] Ilia Baldine and George N. Rouskas. "HiPeR-l: A High Performance Reservation Protocol with look-ahead for Broadcast WDM Networks ". In *Proceedings of INFOCOM*, pages 1272–1279, April 1997.

[4] Ilia Baldine and George N. Rouskas. Reconfiguration and Dynamic Load Balancing in Broadcast WDM Networks. In *Proceedings of INFOCOM*, volume 1, pages 49–64, June 1999.

[5] Ilia Baldine and George N. Rouskas. Traffic adaptive WDM networks: A study of reconfiguration issues. *IEEE/OSA Journal of Lightwave Technology*, 19(4):433–455, April 2001.

[6] Evripidis Bampis and George N. Rouskas. "The Scheduling and Wavelength Assignment Problem in Optical WDM Networks". *IEEE/OSA Journal of Lightwave Technology*, 20(5):782–789, May 2002.

[7] Sudhin Bengeri. Differentiated services support for the Helios optical WDM testbed. Master's thesis, North Carolina State University, http://www.lib.ncsu.edu/etd/public/etd-16201418610131981/etd.pdf, August 2001.

[8] Zeydy Ortiz, George N. Rouskas, and Harry G. Perros. Scheduling of multicast traffic in tunable-receiver WDM networks with non-negligible tuning latencies. In *Proceedings of SIGCOMM*, pages 301–310, September 1997.

[9] George N. Rouskas and Vijay Sivaraman. Packet scheduling in broadcast WDM networks with arbitrary transceiver tuning latencies. *IEEE/ACM Transactions on Networking*, 5(3):359–370, June 1997.

[10] Vijay Sivaraman and George N. Rouskas. A reservation protocol for broadcast WDM networks and stability analysis. *Computer Networks*, 32(2):211–277, February 2000.

[11] Dhaval Thaker. Multicasting in a partially tunable broadcast WDM network. Master's thesis, North Carolina State University, http://www.lib.ncsu.edu/etd/public/etd-120143410141221/etd.pdf, May 2001.

[12] Dhaval Thaker and George N. Rouskas. Multi-Destination Communication in Broadcast WDM Networks: A Survey. *Optical Networks*, 3(1):34–44, January/February 2002.