Residential Ethernet (RE)
(a working paper)

The following paper represents an initial attempt to codify the content of multiple IEEE 802.3 Residential Ethernet (RE) Study Group slide presentations. The author has also taken the liberty to expand on various slide-based proposals, with the goal of triggering/facilitating future discussions.

For the convenience of the author, this paper has been drafted using the style of IEEE standards. The quality of the figures and the consistency of the notation should not be confused with completeness of technical content.

Rather, the formality of this paper represents an attempt by the author to facilitate review by interested parties. Major changes and entire clause rewrites are expected before consensus-approved text becomes available.

JggDvj2005Apr16
March 6, 2006

Residential Ethernet (RE) (a working paper)

Draft 0.143

Contributors:
See page 4.

Abstract: This working paper provides background and introduces possible higher level concepts for the development of Residential Ethernet (RE).
Keywords: residential, Ethernet, isochronous, real time

Contributors

This working paper is based on contributions or review comments from the people listed below. Their listing doesn't necessarily imply they agree with the entire content or the author's interpretation of their input.

Jim Battaglia	Pioneer
Alexei Beliaev	Gibson
Dirceu Cavendish	NEC Labs America
George Claseman	Micrel
Feifei (Felix) Feng	Samsung Electronics
John Nels Fuller	Independent
Geoffrey M. Garner	Samsung Electronics
Kevin Gross	Cirrus Logic
Jim Haagen-Smit	HP
David V James	JGG
Dennis Lou	Pioneer
Michael D. Johas Teener	Broadcom
Fred Tuck	EchoStar

Version history

Version	Date	Author	Comments
0.082	2005Apr28	DVJ	Updates based on 2005Apr27 meeting discussions – Restructure document presentation order – Provide list of contributors, with appropriate disclaimer – Provide version history, for convenience of frequent reviewers – Fix page numbering for easy review (continuous count from start) – Fix clause numbering cross-reference bug (period after number) – Urban recording session (see 5.1.4) added for completeness – Conflicting traffic (see 5.1.5) added for completeness – Changed ‘ping’ to ‘refresh’, within the context of SRP – Changes the multicast addressing for classA frames – Refined state machines
0.085	2005May11	DVJ	– Updated front-page list of contributors – Updated book for continuous pages (Clause 1 discontinuity fixed) – Miscellaneous editing fixes – Initial pinging description added. – Previous Clause 9 (identifier assignments) moved to format clause. – The *subType* identifier assignments now specified. – The bunching annex (work in progress) now includes: A more typical age-based classA prioritization assumption. Other parameters of interest (idle and full-load durations). (Further thought on queue sizing, to avoid discards, is needed.)
0.088	2005Jun03	DVJ	– Application latency scenarios clarified. Generalized based on Norm Finn concerns. Clarified/corrected based on Kevin Gross comments. – Subscription revised, to converge with Felix presentation. – Bursting and bunching scenarios revised for applicability and clarity.
0.090	2005Jun06	DVJ	– Misc editorials in bursting and bunching annex.
0.092	2005Jun10	DVJ	– Extensive cleanup of Clause 5 subscription protocols, based on 2005Jun08 teleconference review comments.

Version	Date	Author	Comments
0.121	2005Jun24	DVJ	– Extensive cleanup of clock-synchronization protocols, base on 2005Jun22 teleconference review comments. Affected areas include: Subclause 5.1: Revised, based comments from Alexei Subclause 5.5: Time-synchronization overview updated Clause 7: Time-synchronization descriptions added Note that the state machines have now become obsolete. Annex I: Time-synchronization code added
0.125	2005Jun30	DVJ	– Grand-master description provided. – Clock deviation moved from code to state machine. – Clock-synchronization code enhanced and split into distinct. core (one per station) and port components. – Code cleanups, corresponding to the above.
0.127	2005Jul04	DVJ	– Pacing descriptions greatly enhanced. Miscellaneous error/clarity fixes, primarily clock related 5.7—A better overview provided Clause 9—Detailed state machines provided Synchronized time-of-day clock/Limitations of current approaches has been migrated to Annex D, where other alternatives are listed.
0.133	2005Jul12	DVJ	– Update of contributors list. – Pacing assumptions/objectives/strategies added to Clause 5. – Pacing state machine in Clause 8 simplified: migrated shapers into transmit code, eliminating shapers A,B,C merged Transmit100Mbs and Transmit1Gbs into TransmitTx
0.134	2005Jul17	DVJ	– Pacing disclaimers, based on preceding meeting discussions.l classB could be found to be unnecessary 3-cycle to 2-cycle delay reduction may not be worth its complexity. – Miscellaneous fixups. – Additions to pacing, resisions of state machines. – Excess material deletions.
0.135	2005Jul30	DVJ	– Alternative rate-based pacing protocol provided. – The stream-addressing alternatives have been better organized.
0.138	2005Sep21	DVJ	– Revised transmit state machines for clarity, based on review feedback.
0.141	2005Nov13	DVJ	– Revised timeSync clause to better mimic firmware implementation. – Clarified 3-port and 6-port topologies within the annex. – Notation updated to track state machine format changes.
—	TBD	—	—

Background

This working paper is highly preliminary and subject to changed. Comments should be sent to its editor:

David V. James
3180 South Ct
Palo Alto, CA 94306
Home: +1-650-494-0926
Cell: +1-650-954-6906
Fax: +1-360-242-5508
Email: dvj@alum.mit.edu

Formats

In many cases, readers may elect to provide contributions in the form of exact text replacements and/or additions. To simplify document maintenance, contributors are requested to use the standard formats and provide checklist reviews before submission. Relevant URLs are listed below:

General: http://grouper.ieee.org/groups/msc/WordProcessors.html
Templates: http://grouper.ieee.org/groups/msc/TemplateTools/FrameMaker/
Checklist: http://grouper.ieee.org/groups/msc/TemplateTools/Checks2004Oct18.pdf

Topics for discussion

Readers are encouraged to provide feedback in all areas, although only the following areas have been identified as specific areas of concern.

a) Terminology. Is classA an OK way to describe the traffic within an RE stream?
Alternatives:
synchronous traffic? isochronous traffic? RE traffic? quasi-synchronous traffic?

TBDs

Further definitions are needed in the following areas:

a) ClassA addressing models: review, select, and revise.
b) Pacing models: review, select, and revise.

Contents

List of figures

List of tables

Residential Ethernet (RE) (a working paper)

This document and has no official status within IEEE or alternative SDOs.
Feedback to: dvj@alum.mit.edu
(See page 4 for the list of contributors.)

1. Overview

1.1 Scope and purpose

This working paper is intended to supplement Ethernet with real-time capabilities, with the scope and purpose listed below:

Scope: Residential Ethernet provides time-sensitive delivery between plug-and-play stations over reliable point-to-point full-duplex cable media. Time-sensitive data transmissions use admission control negotiations to guarantee bandwidth allocations with predictable latency and low-jitter delivery. Device-clock synchronization is also supported. Ensuring real-time services through routers, data security, wireless media, and developing new PMDs are beyond the scope of this project.

Purpose: To enable a common network for existing home Ethernet equipment and locally networked consumer devices with time-sensitive audio, visual and interactive applications and musical equipment. This integration will enable new applications, reduce overall installation cost/complexity and leverage the installed base of Ethernet networking products, while preserving Ethernet networking services. An appropriately enhanced Ethernet is the best candidate for a universal home network platform.

1.2 Introduction

1.2.1 Documentation status

This working paper is intended to identify possible architectures for Residential Ethernet (RE), the title currently assigned to an IEEE Study Group. Although this Study Group intends to become a formal IEEE 802 Working Group, the first step in this process (approval of a PAR) has not occurred.

This working paper attempts to represent opinions of its contributors (see page 4), although numerous others contributed to its content. The documented is formatted to minimize the difficulties associated with porting the text into a yet-to-be-defined standards document, although numerous changes and clause partitioning would be expected before that occurs.

1.2.2 Background

Ethernet has successfully propagated from the data center to the home, becoming the wired home computer interconnect of choice. However, insufficient support of real-time services has limited Ethernet's success as a consumer audio-video interconnects, where IEEE Std 1394 Serial Bus and Universal Serial Bus (USB) have dominated the marketplace.

This working paper for Residential Ethernet (RE) supports time-sensitive network traffic (called classA traffic), as well as legacy IEEE 1394 traffic, while associating the interconnect with Ethernet commodity pricing and relatively seamless frame-transport bridging.

1.2.3 Design objectives

Design objectives for Residential Ethernet (RE) protocols include the following:

a) Scalable. Time-sensitive classA transfers can be supported over multiple speed links:
   1) 100 Mb/s. Normal (~1500 bytes, or 120μs) and classA frames coexist on 100 Mb/s links.
   2) 1 Gb/s. Jumbo (~8,200 bytes, or 66μs) and classA frames can coexist on 1 Gb/s links.

b) Compatible. Existing devices and protocols are supported, as follows:
   1) Interoperable. Communications of existing 802.3 stations are not degraded by classA traffic.
   2) Heterogeneous. Existing 1394 A/V devices can be bridged over RE connections.

c) Efficient. Time-sensitive transmissions are efficient as well as robust:
   1) Bandwidth is independently managed on non-overlapping paths.
   2) ClassA transmissions are limited to the links between talker and listener stations.
   3) Up to 75% of the link bandwidth can be allocated for classA transmissions.

d) Applicable. Time-sensitive transmission characteristics are applicable to the marketplace.
   1) Precise. A common synchronous clock allows playback times to be precisely synchronized.
   2) Low latency. Talker and listener delays are less than human perceptible delays, for interactive home (see 5.1.2 and 5.1.3) and between-home (telephone or internet based) applications.

e) Predictable. Subject to the (c3) constraint, classA traffic is unaffected by the network topology or the traffic loads offered by other stations.

1.2.4 Strategies

Strategies for achieving the aforementioned objectives include the following:

a) Subscription. ClassA transmission bandwidths are limited to prenegotiated bandwidths.

b) Pacing. ClassA transmissions are limited to subscription-negotiated per-cycle bandwidths. (The 125μs cycle is consistent with existing IEEE 1394 A/V and telecommunication systems.)
   1) Topology. Bandwidths can be guaranteed over arbitrary non-cyclical topologies.
   2) Presence. Subscription protocols can readily detect the presence/absence of talker streams.

c) Simplicity. Simplicity is achieved by utilizing well behaved protocols:
   1) Only duplex point-to-point Ethernet links are supported.
   2) PLLs. Precise global clock synchronization eliminates the need for PLLs within bridges.
   3) Plugs. Self-administered stream identifiers are based on talker-managed plug identifiers. (This eliminates the need to define/provide/configure stream identifier servers.)
   4) RSVP. Subscription is based on a layer-2 simplification of the RSVP protocols, called SRP. (SRP allows listeners to autonomously/robustly adapt to spanning tree topology changes).

1.2.5 Interoperability

RE interoperates with existing Ethernet, but the scope of RE services is limited to the RE cloud, as illustrated in Figure 1.1; normal best-effort services are available everywhere else. The scope of the RE cloud is limited by a non-RE capable bridge or a half-duplex link, neither of which can support RE services.

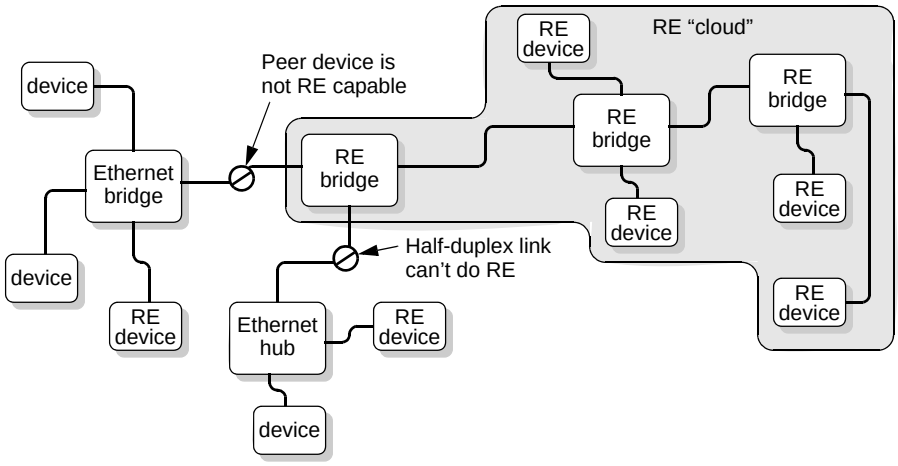


Figure 1.1—Topology and connectivity

Separation of RE devices is driven by the requirements of RE bridges to support subscription (bandwidth allocation), time-of-day clock-synchronization, and (preferably) of pacing of time-sensitive transmissions.

1.2.6 Document structure

The clauses and annexes of this working paper are listed below. The recommended reading order for first-time readers is Clause 5 (an overview), Clause F (critical considerations), Clause 7/8 (details of design). Other clauses provide useful background and reference material.

- Clause 1: Overview
- Clause 2: References
- Clause 3: Terms, definitions, and notation
- Clause 4: Abbreviations and acronyms
- Clause 5: Architecture overview
- Clause 6: Frame formats
- Clause 7: Timer synchronization
- Clause 8: Subscription state machines
- Clause 9: Endpoint shaping state machines (proposal 1)
- Clause 10: Transmit state machines (proposal 2)
- Clause 11: Transmit state machines (proposal 3)
- Annex A: Bibliography
- Annex B: Background material
- Annex C: Encapsulated IEEE 1394 frames
- Annex D: Review of possible alternatives
- Annex E: Time-of-day format considerations
- Annex G: Denigrated alternatives
- Annex F: Bursting and bunching considerations
- Annex H: Frequently asked questions (FAQs)
- Annex I: C-code illustrations

2. References

NOTE—This clause should be skipped on the first reading (continue with Clause 5).
This references list is highly preliminary, references will be added as this working paper evolves.

The following documents contain provisions that, through reference in this working paper, constitute provisions of this working paper. All the standards listed are normative references. Informative references are given in Annex A. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this working paper are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

ANSI/ISO 9899-1990, Programming Language-C.[1,2]

IEEE Std 802.1D-2004, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges.

[1]Replaces ANSI X3.159-1989

[2]ISO documents are available from ISO Central Secretariat, 1 Rue de Varembe, Case Postale 56, CH-1211, Geneve 20, Switzerland/Suisse; and from the Sales Department, American National Standards Institute, 11 West 42 Street, 13th Floor, New York, NY 10036-8002, USA

3. Terms, definitions, and notation

NOTE—This clause should be skipped on the first reading (continue with Clause 5).
This text has been lifted from the P802.17 draft standard, which has a relative comprehensive list.
Terms and definitions are expected to be added, revised, and/or deleted as this working paper evolves.

3.1 Conformance levels

Several key words are used to differentiate between different levels of requirements and options, as described in this subclause.

3.1.1 may: Indicates a course of action permissible within the limits of the standard with no implied preference ("may" means "is permitted to").

3.1.2 shall: Indicates mandatory requirements to be strictly followed in order to conform to the standard and from which no deviation is permitted ("shall" means "is required to").

3.1.3 should: An indication that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited ("should" means "is recommended to").

3.2 Terms and definitions

For the purposes of this working paper, the following terms and definitions apply. The Authoritative Dictionary of IEEE Standards Terms [B2] should be referenced for terms not defined in the clause.

3.2.1 audience: The set of listeners associated with a common streamID.

3.2.2 best-effort: Not associated with an explicit service guarantee.

3.2.3 bridge: A functional unit interconnecting two or more networks at the data link layer of the OSI reference model.

3.2.4 clock master: A bridge or end station that provides the link clock reference.

3.2.5 clock slave: A bridge or end station that tracks the link clock reference provided by the clock master.

3.2.6 cyclic redundancy check (CRC): A specific type of frame check sequence computed using a generator polynomial.

3.2.7 destination station: A station to which a frame is addressed.

3.2.8 frame: The MAC sublayer protocol data unit (PDU).

3.2.9 grand clock master: The clock master selected to provide the network time reference.

3.2.10 jitter: The variation in delay associated with the transfer of frames between two points.

3.2.11 latency: The time required to transfer information from one point to another.[3]

3.2.12 link: A unidirectional channel connecting adjacent stations (half of a span).

3.2.13 listener: A sink of a stream, such as a television or acoustic speaker.

3.2.14 local area network (LAN): A communications network designed for a small geographic area, typically not exceeding a few kilometers in extent, and characterized by moderate to high data transmission rates, low delay, and low bit error rates.

3.2.15 MAC client: The layer entity that invokes the MAC service interface.

3.2.16 management information base (MIB): A repository of information to describe the operation of a specific network device.

3.2.17 maximum transfer unit (MTU): The largest frame (comprising payload and all header and trailer information) that can be transferred across the network.

3.2.18 medium (plural: **media):** The material on which information signals are carried; e.g., optical fiber, coaxial cable, and twisted-wire pairs.

3.2.19 medium access control (MAC) sublayer: The portion of the data link layer that controls and mediates the access to the network medium. In this working paper, the MAC sublayer comprises the MAC datapath sublayer and the MAC control sublayer.

3.2.20 multicast: Transmission of a frame to stations specified by a group address.

3.2.21 multicast address: A group address that is not a broadcast address, i.e., is not all-ones, and identifies some subset of stations on the network.

3.2.22 network: A set of communicating stations and the media and equipment providing connectivity among the stations.

3.2.23 pacer: A credit-based entity that partions residual bandwidths between two classes of frames.

3.2.24 packet: A generic term for a PDU associated with a layer-entity above the MAC sublayer.

3.2.25 path: A logical concatenation of links and bridges over which streams flow from the talker to the listener.

3.2.26 plug-and-play: The requirement that a station perform classA transfers without operator intervention (except for any intervention needed for connection to the cable).

3.2.27 protocol implementation conformance statement (PICS): A statement of which capabilities and options have been implemented for a given Open Systems Interconnection (OSI) protocol.

3.2.28 service discovery: The process used by listeners or controlling stations to identify, control, and configure talkers.

3.2.29 shaper: A credit-based entity that limits short-term transmission bandwidths to a specified rate.

3.2.30 simple reservation protocol (SRP): The subscription protocol used to allocate and sustain paths for streaming classA traffic.

[3]Delay and latency are synonyms for the purpose of this working paper. Delay is the preferred term.

3.2.31 span: A bidirectional channel connecting adjacent stations (two links).

3.2.32 source station: The station that originates a frame.

3.2.33 station: A device attached to a network for the purpose of transmitting and receiving information on that network.

3.2.34 stream: A sequence of frames passed from the talker to listener(s), which have the same streamID.

3.2.35 subscription: The process of establishing committed paths between the talker and one or more listeners.

3.2.36 talker: The source of a stream, such as a cable box or microphone.

3.2.37 topology: The arrangement of links and stations forming a network, together with information on station attributes.

3.2.38 transmit (transmission): The action of a station placing a frame on the medium.

3.2.39 transparent bridging: A bridging mechanism that is transparent to the end stations.

3.2.40 unicast: The act of sending a frame addressed to a single station.

3.3 Service definition method and notation

The service of a layer or sublayer is the set of capabilities that it offers to a user in the next higher (sub)layer. Abstract services are specified in this working paper by describing the service primitives and parameters that characterize each service. This definition of service is independent of any particular implementation (see Figure 3.1).

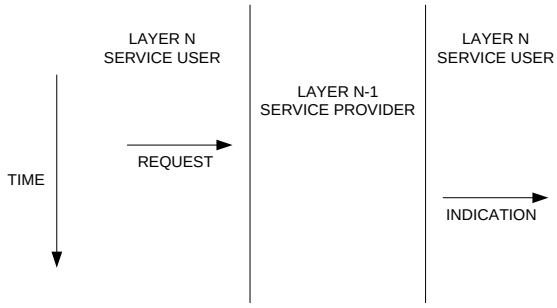


Figure 3.1—Service definitions

Specific implementations can also include provisions for interface interactions that have no direct end-to-end effects. Examples of such local interactions include interface flow control, status requests and indications, error notifications, and layer management. Specific implementation details are omitted from this service specification, because they differ from implementation to implementation and also because they do not impact the peer-to-peer protocols.

3.3.1 Classification of service primitives

Primitives are of two generic types.

a) REQUEST. The request primitive is passed from layer N to layer N-1 to request that a service be initiated.
b) INDICATION. The indication primitive is passed from layer N-1 to layer N to indicate an internal layer N-1 event that is significant to layer N. This event can be logically related to a remote service request, or can be caused by an event internal to layer N-1.

The service primitives are an abstraction of the functional specification and the user-layer interaction. The abstract definition does not contain local detail of the user/provider interaction. For instance, it does not indicate the local mechanism that allows a user to indicate that it is awaiting an incoming call. Each primitive has a set of zero or more parameters, representing data elements that are passed to qualify the functions invoked by the primitive. Parameters indicate information available in a user/provider interaction. In any particular interface, some parameters can be explicitly stated (even though not explicitly defined in the primitive) or implicitly associated with the service access point. Similarly, in any particular protocol specification, functions corresponding to a service primitive can be explicitly defined or implicitly available.

3.4 State machines

3.4.1 State machine behavior

The operation of a protocol can be described by subdividing the protocol into a number of interrelated functions. The operation of the functions can be described by state machines. Each state machine represents the domain of a function and consists of a group of connected, mutually exclusive states. Only one state of a function is active at any given time. A transition from one state to another is assumed to take place in zero time (i.e., no time period is associated with the execution of a state), based on some condition of the inputs to the state machine.

The state machines contain the authoritative statement of the functions they depict. When apparent conflicts between descriptive text and state machines arise, the order of precedence shall be formal state tables first, followed by the descriptive text, over any explanatory figures. This does not override, however, any explicit description in the text that has no parallel in the state tables.

The models presented by state machines are intended as the primary specifications of the functions to be provided. It is important to distinguish, however, between a model and a real implementation. The models are optimized for simplicity and clarity of presentation, while any realistic implementation might place heavier emphasis on efficiency and suitability to a particular implementation technology. It is the functional behavior of any unit that has to match the standard, not its internal structure. The internal details of the model are useful only to the extent that they specify the external behavior clearly and precisely.

3.4.2 State table notation

NOTE—The following state machine notation was used within 802.17, due to the exactness of C-code conditions and the simplicity of updating table entries (as opposed to 2-dimensional graphics).
Early state table descriptions can be converted (if necessary) into other formats before publication.

Each row of the table is preferably provided with a brief description of the condition and/or action for that row. The descriptions are placed after the table itself, and linked back to the rows of the table using numeric tags.

3.4.2.1 Parallel-execution state tables

State machines may be represented in tabular form. The table is organized into two columns: a left hand side representing all of the possible states of the state machine and all of the possible conditions that cause transitions out of each state, and the right hand side giving all of the permissible next states of the state machine as well as all of the actions to be performed in the various states, as illustrated in Table 3.1. The syntax of the expressions follows standard C notation (see 3.13). No time period is associated with the transition from one state to the next.

Table 3.1—State table notation example

Current		Row	Next	
state	**condition**		**action**	**state**
START	sizeOfMacControl > spaceInQueue	1	—	START
	passM == 0	2		
	—	3	TransmitFromControlQueue();	FINAL
FINAL	SelectedTransferCompletes()	1	—	START
	—	2	—	FINAL

Row START-1: Do nothing if the size of the queued MAC control frame is larger than the PTQ space.
Row START-2: Do nothing in the absence of MAC control transmission credits.
Row START-3: Otherwise, transmit a MAC control frame.

Row FINAL-1: When the transmission completes, start over from the initial state (i.e., START).
Row FINAL-2: Until the transmission completes, remain in this state.

Each combination of current state, next state, and transition condition linking the two is assigned to a different row of the table. Each row of the table, read left to right, provides: the name of the current state; a condition causing a transition out of the current state; an action to perform (if the condition is satisfied); and, finally, the next state to which the state machine transitions, but only if the condition is satisfied. The symbol "—" signifies the default condition (i.e., operative when no other condition is active) when placed in the condition column, and signifies that no action is to be performed when placed in the action column. Conditions are evaluated in order, top to bottom, and the first condition that evaluates to a result of TRUE is used to determine the transition to the next state. If no condition evaluates to a result of TRUE, then the state machine remains in the current state. The starting or initialization state of a state machine is always labeled "START" in the table (though it need not be the first state in the table). Every state table has such a labeled state.

Each row of the table is preferably provided with a brief description of the condition and/or action for that row. The descriptions are placed after the table itself, and linked back to the rows of the table using numeric tags.

3.4.2.2 Called state tables

A RETURN state is the terminal state of a state machine that is intended to be invoked by another state machine, as illustrated in Table 3.2. Once the RETURN state is reached, the state machine terminates execution, effectively ceasing to exist until the next invocation by the caller, at which point it begins execution again from the START state. State machines that contain a RETURN state are considered to be only instantiated when they are invoked. They do not have any persistent (static) variables.

Table 3.2—Called state table notation example

Current		Row	Next	
state	**condition**		**action**	**state**
START	sizeOfMacControl > spaceInQueue	1	—	FINAL
	passM == 0	2		
	—	3	TransmitFromControlQueue();	RETURN
FINAL	MacTransmitError();	1	errorDefect = TRUE	RETURN
	—	2	—	

Row START-1: The size of the queued MAC control frame is less than the PTQ space.
Row START-2: In the absence of MAC control transmission credits, no action is taken.
Row START-3: MAC control transmissions have precedence over client transmissions.

Row FINAL-1: If the transmission completes with an error, set an error defect indication.
Row FINAL-2: Otherwise, no error defect is indicated.

3.5 Arithmetic and logical operators

In addition to commonly accepted notation for mathematical operators, Table 3.3 summarizes the symbols used to represent arithmetic and logical (boolean) operations. Note that the syntax of operators follows standard C notation (see 3.13).

Table 3.3—Special symbols and operators

Printed character	Meaning
&&	Boolean AND
||	Boolean OR
!	Boolean NOT (negation)
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
=	Assignment operator
//	Comment delimiter

3.6 Numerical representation

NOTE—The following notation was taken from 802.17, where it was found to have benefits:
– The subscript notation is consistent with common mathematical/logic equations.
– The subscript notation can be used consistently for all possible radix values.

Decimal, hexadecimal, and binary numbers are used within this working paper. For clarity, decimal numbers are generally used to represent counts, hexadecimal numbers are used to represent addresses, and binary numbers are used to describe bit patterns within binary fields.

Decimal numbers are represented in their usual 0, 1, 2, … format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0-9,A-F) digits followed by the subscript 16, except in C-code contexts, where they are written as `0x123EF2` etc. Binary numbers are represented by a string of one or more binary (0,1) digits, followed by the subscript 2. Thus the decimal number “26” may also be represented as “$1A_{16}$” or “11010_2”.

MAC addresses and OUI/EUI values are represented as strings of 8-bit hexadecimal numbers separated by hyphens and without a subscript, as for example “01-80-C2-00-00-15” or “AA-55-11”.

3.7 Field notations

3.7.1 Use of italics

All field names or variable names (such as *level* or *myMacAddress*), and sub-fields within variables (such as *thisState.level*) are italicized within text, figures and tables, to avoid confusion between such names and similarly spelled words without special meanings. A variable or field name that is used in a subclause heading or a figure or table caption is also italicized. Variable or field names are not italicized within C code, however, since their special meaning is implied by their context. Names used as nouns (e.g., subclassA0) are also not italicized.

3.7.2 Field conventions

This working paper describes values that are packetized or MAC-resident, such as those illustrated in Table 3.2.

Table 3.4—Names of fields and sub-fields

Name	Description
newCRC	Field within a register or frame
thisState.level	Sub-field within field *thisState*
thatState.rateC[*n*].*c*	Sub-field within array element *rateC*[*n*]

Run-together names (e.g., *thisState*) are used for fields because of their compactness when compared to equivalent underscore-separated names (e.g., *this_state*). The use of multiword names with spaces (e.g., “This State”) is avoided, to avoid confusion between commonly used capitalized key words and the capitalized word used at the start of each sentence.

A sub-field of a field is referenced by suffixing the field name with the sub-field name, separated by a period. For example, *thisState.level* refers to the sub-field *level* of the field *thisState*. This notation can be continued in order to represent sub-fields of sub-fields (e.g., *thisState.level.next* is interpreted to mean the sub-field *next* of the sub-field *level* of the field *thisState*).

Two special field names are defined for use throughout this working paper. The name *frame* is used to denote the data structure comprising the complete MAC sublayer PDU. Any valid element of the MAC sublayer PDU, can be referenced using the notation *frame.xx* (where *xx* denotes the specific element); thus, for instance, *frame.serviceDataUnit* is used to indicate the *serviceDataUnit* element of a frame.

Unless specifically specified otherwise, reserved fields are reserved for the purpose of allowing extended features to be defined in future revisions of this working paper. For devices conforming to this version of this working paper, nonzero reserved fields are not generated; values within reserved fields (whether zero or nonzero) are to be ignored.

3.7.3 Field value conventions

This working paper describes values of fields. For clarity, names can be associated with each of these defined values, as illustrated in Table 3.5. A symbolic name, consisting of upper case letters with underscore separators, allows other portions of this working paper to reference the value by its symbolic name, rather than a numerical value.

Table 3.5—*wrap* field values

Value	Name	Description
0	STANDARD	Standard processing selected
1	SPECIAL	Special processing selected
2,3	—	Reserved

Unless otherwise specified, reserved values allow extended features to be defined in future revisions of this working paper. Devices conforming to this version of this working paper do not generate nonzero reserved values, and process reserved fields as though their values were zero.

A field value of TRUE shall always be interpreted as being equivalent to a numeric value of 1 (one), unless otherwise indicated. A field value of FALSE shall always be interpreted as being equivalent to a numeric value of 0 (zero), unless otherwise indicated.

3.8 Bit numbering and ordering

Data transfer sequences normally involve one or more cycles, where the number of bytes transmitted in each cycle depends on the number of byte lanes within the interconnecting link. Data byte sequences are shown in figures using the conventions illustrated by Figure 3.2, which represents a link with four byte lanes. For multi-byte objects, the first (left-most) data byte is the most significant, and the last (right-most) data byte is the least significant.

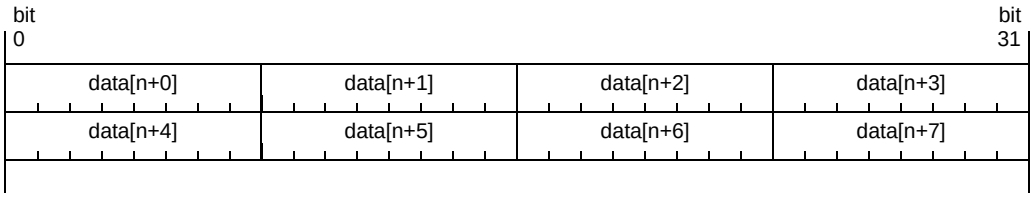


Figure 3.2—Bit numbering and ordering

Figures are drawn such that the counting order of data bytes is from left to right within each cycle, and from top to bottom between cycles. For consistency, bits and bytes are numbered in the same fashion.

NOTE—The transmission ordering of data bits and data bytes is not necessarily the same as their counting order; the translation between the counting order and the transmission order is specified by the appropriate reconciliation sublayer.

3.9 Byte sequential formats

Figure 3.3 provides an illustrative example of the conventions to be used for drawing frame formats and other byte sequential representations. These representations are drawn as fields (of arbitrary size) ordered along a vertical axis, with numbers along the left sides of the fields indicating the field sizes in bytes. Fields are drawn contiguously such that the transmission order across fields is from top to bottom. The example shows that *field1*, *field2*, and *field3* are 1-, 1- and 6-byte fields, respectively, transmitted in order starting with the *field1* field first. As illustrated on the right hand side of Figure 3.3, a multi-byte field represents a sequence of ordered bytes, where the first through last bytes correspond to the most significant through least significant portions of the multi-byte field, and the MSB of each byte is drawn to be on the left hand side.

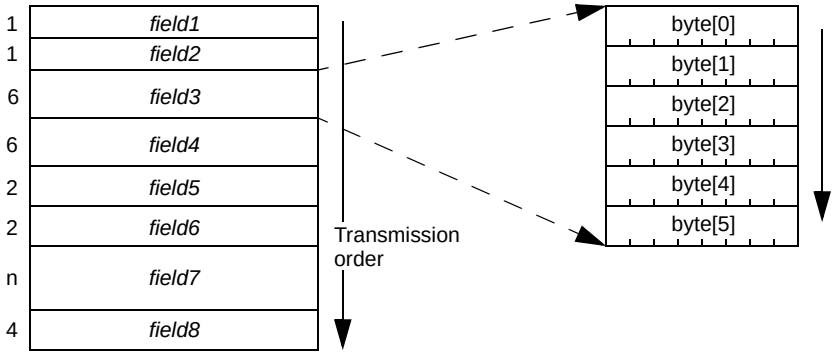


Figure 3.3—Byte sequential field format illustrations

NOTE—Only the left-hand diagram in Figure 3.3 is required for representation of byte-sequential formats. The right-hand diagram is provided in this description for explanatory purposes only, for illustrating how a multi-byte field within a byte sequential representation is expected to be ordered. The tag “Transmission order” and the associated arrows are not required to be replicated in the figures.

3.10 Ordering of multibyte fields

In many cases, bit fields within byte or multibyte objects are expanded in a horizontal fashion, as illustrated in the right side of Figure 3.4. The fields within these objects are illustrated as follows: left-to-right is the byte transmission order; the left-through-right bits are the most significant through least significant bits respectively.

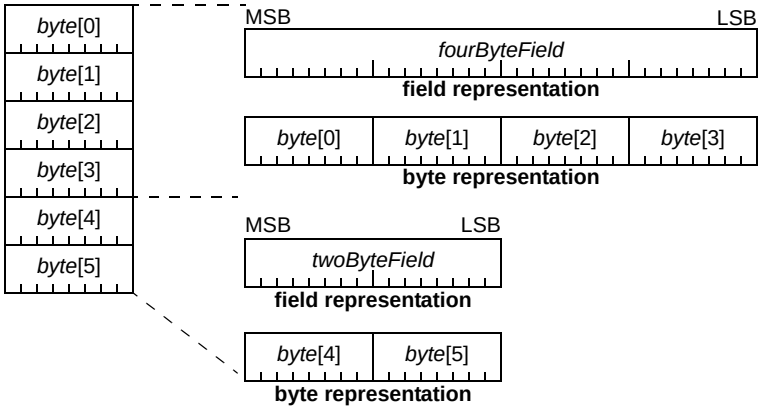


Figure 3.4—Multibyte field illustrations

The first *fourByteField* can be illustrated as a single entity or a 4-byte multibyte entity. Similarly, the second *twoByteField* can be illustrated as a single entity or a 2-byte multibyte entity.

NOTE—The following text was taken from 802.17, where it was found to have benefits:
The details should, however, be revised to illustrate fields within an RE frame header *serviceDataUnit*.

To minimize potential for confusion, four equivalent methods for illustrating frame contents are illustrated in Figure 3.5. Binary, hex, and decimal values are always shown with a left-to-right significance order, regardless of their bit-transmission order.

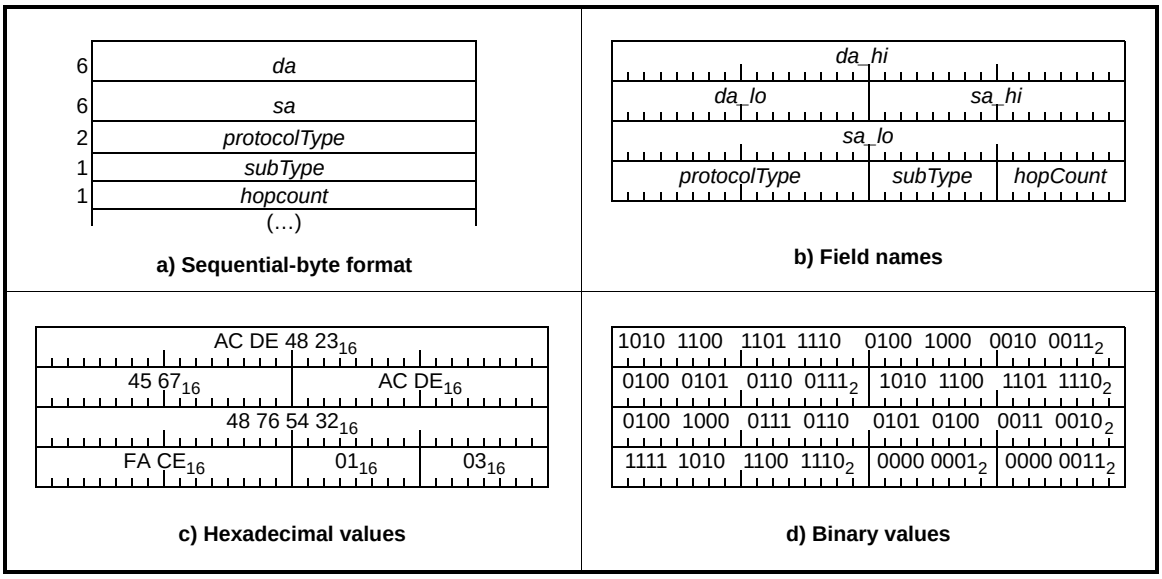


Figure 3.5—Illustration of fairness-frame structure

3.11 MAC address formats

The format of MAC address fields within frames is illustrated in Figure 3.6.

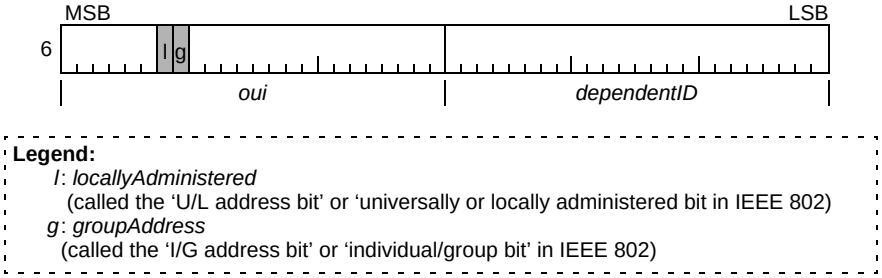


Figure 3.6—MAC address format

3.11.1 *oui*: A 24-bit organizationally unique identifier (OUI) field supplied by the IEEE/RAC for the purpose of identifying the organization supplying the (unique within the organization, for this specific context) 24-bit *dependentID*. (For clarity, the *locallyAdministered* and *groupAddress* bits are illustrated by the shaded bit locations.)

3.11.2 *dependentID*: An 24-bit field supplied by the *oui*-specified organization. The concatenation of the *oui* and *dependentID* provide a unique (within this context) identifier.

To reduce the likelihood of error, the mapping of OUI values to the *oui/dependentID* fields are illustrated in Figure 3.7. For the purposes of illustration, specific OUI and *dependentID* example values have been assumed. The two shaded bits correspond to the *locallyAdministered* and *groupAddress* bit positions illustrated in Figure 3.6.

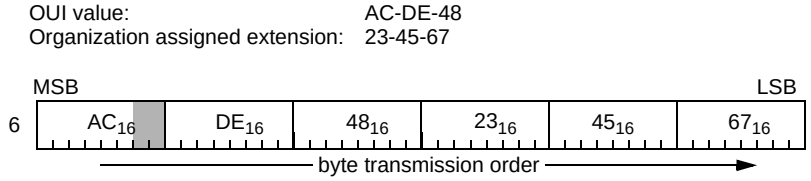


Figure 3.7—48-bit MAC address format

3.12 Informative notes

Informative notes are used in this working paper to provide guidance to implementers and also to supply useful background material. Such notes never contain normative information, and implementers are not required to adhere to any of their provisions. An example of such a note follows.

NOTE—This is an example of an informative note.

3.13 Conventions for C code used in state machines

Many of the state machines contained in this working paper utilize C code functions, operators, expressions and structures for the description of their functionality. Conventions for such C code can be found in Annex I.

4. Abbreviations and acronyms

NOTE—This clause should be skipped on the first reading (continue with Clause 5).
This text has been lifted from the P802.17 draft standard, which has a relative comprehensive list. Abbreviations/acronyms are expected to be added, revised, and/or deleted as this working paper evolves.

This working paper contains the following abbreviations and acronyms:

BER	bit error ratio
CRC	cyclic redundancy check
FCS	frame check sequence
FIFO	first in first out
GARP	Generic Attribute Registration Protocol
HEC	header error check
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
ITU	International Telecommunication Union
LAN	local area network
LSB	least significant bit
MAC	medium access control
MAN	metropolitan area network
MIB	management information base
MSB	most significant bit
MTU	maximum transfer unit
OAM	operations, administration, and maintenance
OSI	open systems interconnect
PDU	protocol data unit
PHY	physical layer
RE	Residential Ethernet
RFC	request for comment
RPR	resilient packet ring
SRP	simple reservation protocol
TDM	time division multiplexing
VOIP	voice over internet protocol

5. Architecture overview

5.1 Latency constraints

5.1.1 Interactive audio delay considerations

The latency constraints of the RE environment are based on the sensitivity of the human ear. To be comfortable when playing music, the delay between the instrument and the human ear should not exceed 10-to-15 ms, as illustrated in Figure 5.1. The individual hop delays must be considerably smaller, since instrument-sourced audio traffic may pass through multiple links and processing devices before reaching the ear, as illustrated in 5.1.2 and 5.1.3.

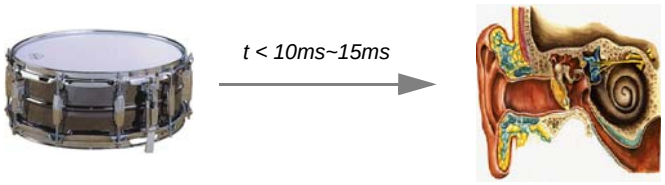


Figure 5.1—Interactive audio delay considerations

5.1.2 Home recording session

To illustrate hop-latency requirements, consider RE usage for a home recording session, as illustrated in Figure 5.2. The audio inputs (microphone and guitar) are converted, passed through a bridge, mixed within a laptop computer, converted at the speaker, and return to the performer's ear through the air.

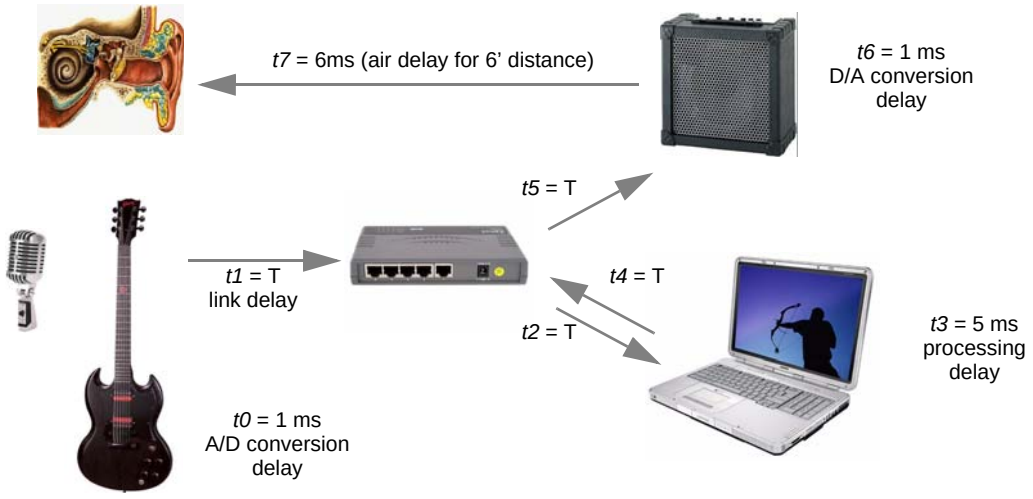


Figure 5.2—Home recording session

A fixed time T is assumed for each passage through a link, based on potential buffering and conflicting-traffic delays. Due to multiple link hops and the latency contributions, the constraints on the value of T are much less than the constraining 15ms instrument-to-ear latency, as illustrated in Equation 5.1.

$$t0 + t1 + t2 + t3 + t4 + t5 + t6 + t7 < 15\text{ ms} \quad (5.1)$$
$$1\text{ms} + T + T + 5\text{ms} + T + T + 1\text{ms} + 6\text{ms} < 15\text{ms}$$
$$4 \times T + 13\text{ms} < 15\text{ms}$$
$$T < 0.5\text{ ms}$$

To better understand the range of possibilities, consider an extremely aggressive implementation of end-point stations could reduce the link-latency requirements. For example, more aggressive end-point processing delays {*t0*=0.25 ms, *t3*=2 ms, *t6*=0.25 ms, *t7*=6 ms} would yield a constraint of T<1.6 ms.

Note that these aggressive processor delays are unlikely to decrease as the MIPs rating of processors increase, due to the inherent delays associated with finite input response (FIR) filters and efficiencies achieved through block-processing. For example, 16-sample block processing of a 128-point FIR filter implies an inherent 80-cycle delay (16 for input block accumulation, 64 for filtering). With a 40 kHz sampling rate, this corresponds to a theoretical processing-latency limitation of 2 ms.

These numbers are only approximations; actual values (as determined by the marketplace) could vary substantially. For audiophiles, an overall processing latency of 5 ms may be desired; for discount shoppers, an overall latency of 15 ms may be tolerable. Larger ad-hoc networks of cascaded 4-port or 8-port bridges may be present. As with golden speaker cables, purchases may be based on perceptions of quality (the bridge latency specification), rather than reality (perceivable latencies).

5.1.3 Garage jam session

As another example, consider RE usage for a garage jam session, as illustrated in Figure 5.3. The audio inputs (microphone and guitar) are converted, passed through a guitar effects processor, two bridges, mixed within an audio console, return through two bridges, and return to the ear through headphones.

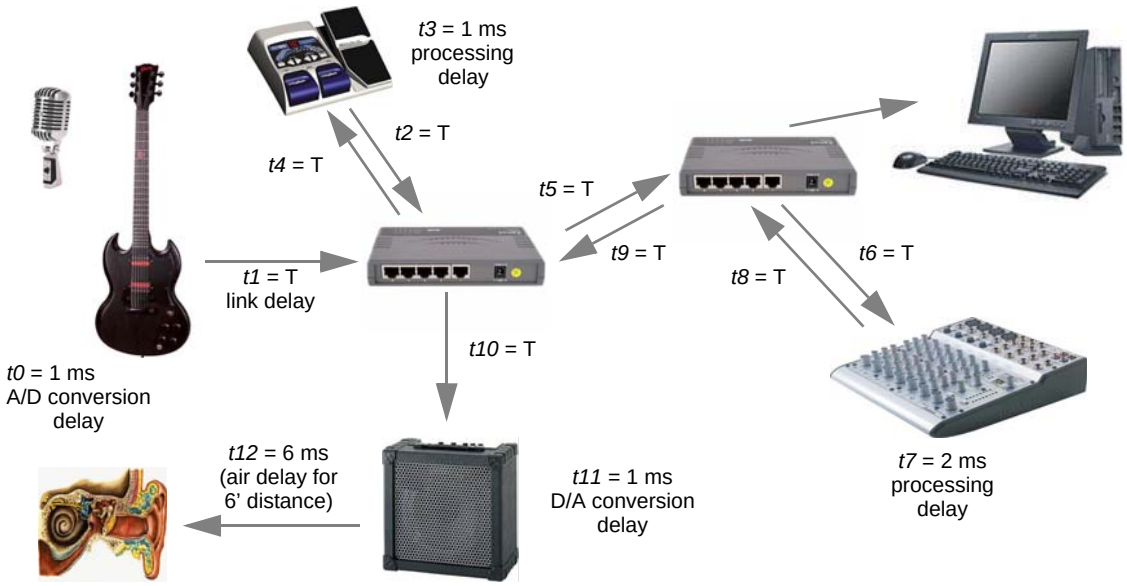


Figure 5.3—Garage jam session

Again, a fixed time T is assumed for each passage through a link, based on potential buffering and conflicting-traffic delays. Due to multiple hops and the latency contributions, the constraints yield a T value that is much less than the constraining 15ms instrument-to-ear latency (see Equation 5.2).

$$t0 + t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + t9 + t10 + t11 + t12 < 15\text{ ms} \quad (5.2)$$
$$1\text{ms} + T + T + 1\text{ms} + T + T + T + 2\text{ms} + T + T + T + 1\text{ms} + 6\text{ms} < 15\text{ms}$$
$$8 \times T + 11\text{ms} < 15\text{ms}$$
$$T < 0.5\text{ ms}$$

To better understand the range of possible latencies, consider extremely aggressive implementations of end-point stations. For example, more aggressive end-point processing delays {*t0*=0.25 ms, *t3*=0.25 ms, *t7*=2 ms, *t11*=0.25 ms, *t12*=6 ms} would yield a constraint of T<0.78 ms.

5.1.4 Urban home recording session

Within urban environments, headphones may be preferred to audio speakers, as illustrated in Figure 5.4 (a small modification of Figure 5.2). The audio inputs (microphone and guitar) are converted, passed through a bridge, mixed within a laptop computer, converted at the headphones, and near immediately presented to the performer's ear.

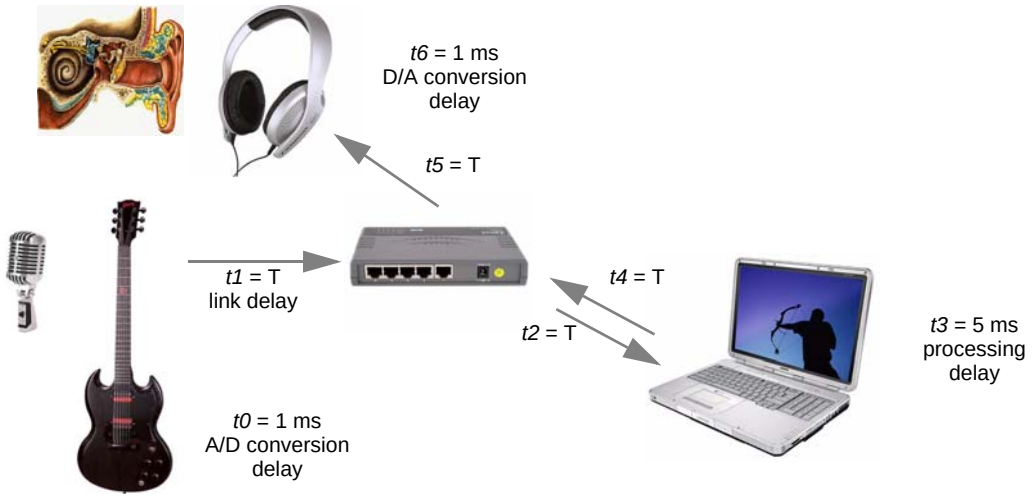


Figure 5.4—Urban recording session

While the earphones eliminate the air-to-ear hop-count delays, the sensitivity to delays is increased for the case of a vocal performer due to a comb filter formed by the interaction of headphone sound and sound conducted through the head. Remaining below the 0.5 to 5 ms range where comb filtering is prevalent is impractical, since the {*t0*=1 ms, *t3*=5 ms, *t6*=1 ms} values already exceed the 0.5 ms limitation.

Professionals believe that increasing latency to 5 ms or more within such headphone-feedback environments is preferred over operation in the 0.5 to 5 ms range where comb filtering is prevalent. Again, due to multiple hops and the latency contributions, the constraints yield a T value that is much less than the constraining 15ms instrument-to-ear latency (see Equation 5.3).

$$t0 + t1 + t2 + t3 + t4 + t5 + t6 < 15\text{ ms} \quad (5.3)$$
$$1\text{ms} + T + T + 5\text{ms} + T + T + 1\text{ms} < 15\text{ ms}$$
$$4 \times T + 7\text{ms} < 15\text{ ms}$$
$$T < 2\text{ms}$$

To better understand the range of possible latencies, consider extremely aggressive implementations of end-point stations. For example, more aggressive end-point processing delays {*t0*=0.25 ms, *t3*=2 ms, *t6*=0.25 ms} would yield a T<3.1 ms constraint.

5.1.5 Conflicting data transfers

Home networks may carry data traffic as well as time-sensitive traffic, as illustrated in Figure 5.3. During musical performances (or evening A/V screenings), high bandwidth computer-to-server transfers could occur over the same data-transfer links, as illustrated in Figure 5.5.

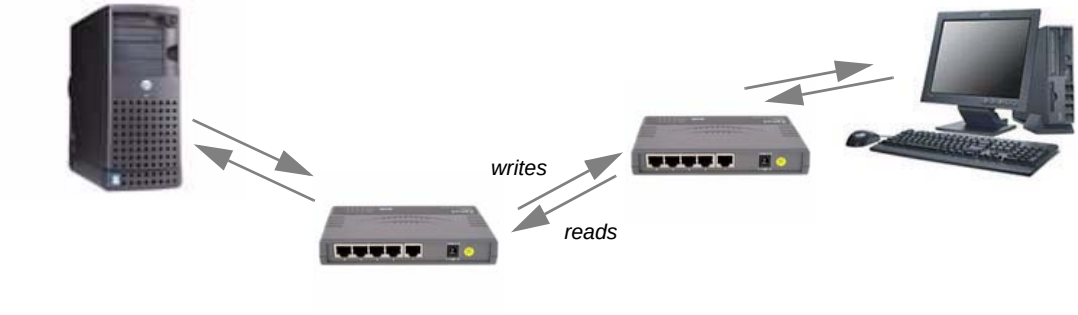


Figure 5.5—Conflicting data transfers

With the high data-transfer rates of disks and disk-array systems, the bandwidth capacity of residential Ethernet links could (if not otherwise limited) easily be reached. Thus, some form of prioritized bridging is necessary to ensure robust delivery of time-sensitive traffic.

5.2 Service classes

Editors' Notes: To be removed prior to final publication.
The classA and classC service classes have consensus among the contributors to this working paper. The concept of classB services was included in IEEE Std 802.17-2004 and is being included for consideration by universal plug and play (UP&P), congestion management (CM), or legacy applications.

This working paper defines three service classes (A, B, or C) with which the data transfer is associated, as summarized in Table 5.1. The classA service provides low-jitter transfer of traffic (and therefore lower worst-case delays) up to its allocated rate. Traffic above the allocated rate is rejected. The classB service provides bounded delay transfer of traffic. The classC service provides best-effort data-transfer services.

Table 5.1—Service classes and their quality-of-service relationships

class of service		qualities of service		
class	**examples of use**	**jitter**	**guaranteed bandwidth**	**type**
A	real time	low	yes	allocated
B	near real time	bounded		
C	best effort	unbounded	no	opportunistic

Link capacity required to support the classA and classB service is allocated via provisioning and these services can be characterized as allocated services. The provisioning activity is expected to ensure that the

aggregate service commitment on each link does not exceed that link's capacity. The allocation rates distributed by provisioning regulates access to these guaranteed services.

Link capacity has to be ensured to support classA and classB service guarantees. This is done by allocating bandwidth through provisioning that prevents over-provisioning the links, using a subscription protocol (see 5.4).

5.3 Architecture overview

5.3.1 Abstract concepts

From the perspective of end-point stations, RE systems supports classA data-frame traffic, called streams. Each stream has one talker and one or more listeners, as illustrated in Figure 5.6-a.

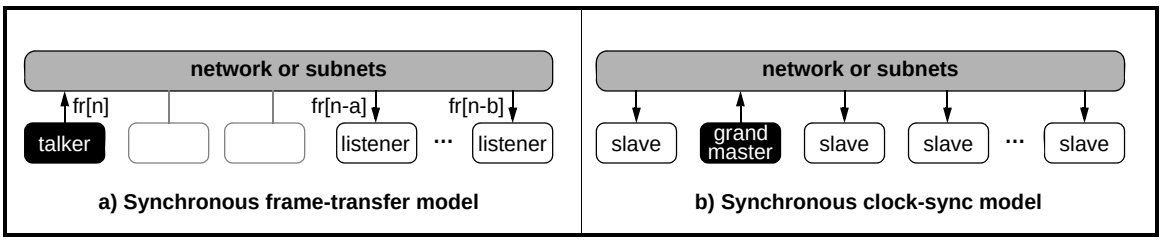


Figure 5.6—Hierarchical control

The delay between the talker and listener(s) is nominally a fixed number of 125μs cycles, although the number of cycles may be cable-length and/or bridge topology dependent. Additional delays can be inserted by the application(s), when synchronization between multiple listeners is required, since the talker's data can be time-stamped and all clocks are synchronized.

To reduce costs (and support GPS-inaccessible locations), synchronized clocks are provided by the interconnect. All classA talkers provide clock references, but only one of these stations is nominated to be the clock master; the others are called clock slaves (see Figure 5.6-b). The selected clock master is called the grand clock master, oftentimes abbreviated as "grand master".

Clock synchronization involves synchronizing the clock-slave clocks to the reference provided by the grand clock master. Tight accuracy is possible with matched-length duplex links, since bidirectional messages can cancel the cable-delay effects.

5.3.2 Detailed illustrations

In many cases, abstract illustrations (see Figure 5.6) are insufficient to illustrate expected behaviors. Thus, more detailed illustrations are oftentimes used to also show bridges and spans within the network cloud, as illustrated in Figure 5.7.

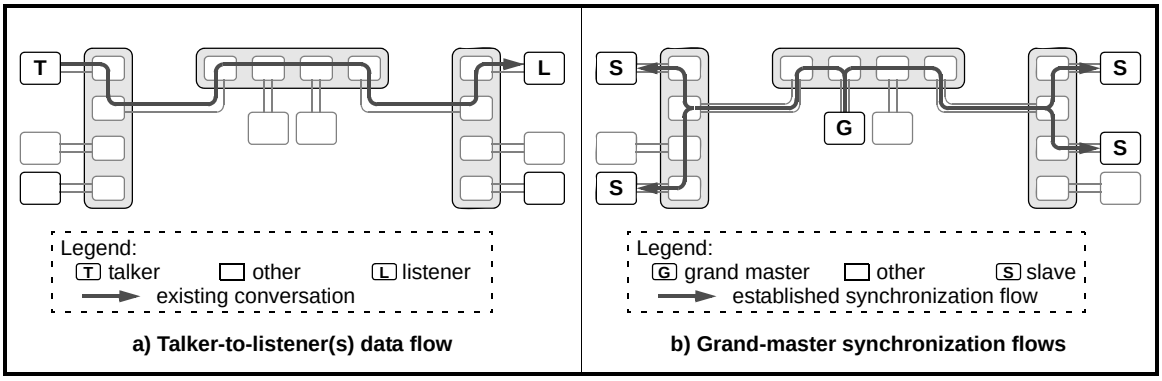


Figure 5.7—Hierarchical flows

5.3.3 Architecture components

The architecture of a home RE system involves the following protocols:

a) Discovery (beyond the scope of this working paper).
A controller discovers the proper streamID/bandwidth parameters to allow the listener to subscribe to the desired talker-sourced stream.

b) Subscription. The controller commands the listener to establish a path from the talker.
Subscription may pass or fail, based on availability of routing-table and link-bandwidth resources.

c) Synchronization. The distributed clocks in talkers and listeners are accurately synchronized.
Synchronized clocks avoid cycle slips and playback-phase distortions.

d) Pacing. The transmitted classA traffic is paced to avoid other classA traffic disruptions.

5.4 Subscription

Editors' Notes: To be removed prior to final publication.
The following SRP text is obsolete, having been replaced by Felix Feng's GARP based protocol.
Some of the general concepts remain valid, so this text is being retained for informational purposes.

5.4.1 Simple Reservation Protocol (SRP) overview

Subscription involves explicit negotiation for bandwidth resources, performed in a distributed fashion, flowing over the paths of intended communication. This subscription protocol are called the Simple Reservation Protocols (SRP). SRP represents an instance of the Generic Attribute Registration Protocol (GARP), with similar objectives to the layer-3 based Resource Reservation Protocol (RSVP). SRP shares many of the baseline RSVP and GARP features, including the following:

— SRP is simplex, i.e. reservations apply to unidirectional data flows.
— SRP is receiver-oriented, i.e., the receiver of a stream initiates and maintains the resource reservation used for that stream.
— SRP maintains "soft" state in bridges, providing graceful support for dynamic membership changes and automatic adaptations to changes in network topology.
— SRP is not a routing protocol, but depends on transparent bridging and STP routing protocols.

SRP simplicity is derived from its restricted layer-2 ambitions, as follows.

— SRP is symmetric, i.e. the listener-to-talker path is the inverse of the talker-to-listener path.
— SRP does not provide for transcoding; any stream is fully characterized by its streamID and bandwidth.

The viability of SRP is enhanced by basing its protocols on GARP, a protocol defined within IEEE Std 802.1D. Specifically, the RequestJoin and RequestLeave messages correspond to primitives defined within GARP.

SRP is defined to be a general 1-to-N resource-reservation scheme, although this discussion focuses on subscription of classA bandwidth resources. The SRP protocols could, however, be used to reserve other resource-limited resources, such as buffer allocations, latency targets, and frame-loss rates.

NOTE—SRP is thought to be applicable to N-to-N topologies, as well as 1-to-N topologies. However, the detailed review of N-to-N topologies (which would be necessary to verify the feasibility of such extensions) is beyond the scope of this working paper.

5.4.2 Soft reservation state

SRP takes a "soft state" approach to managing the reservation state in bridges. SRP soft state is created and periodically refreshed by listener generated RequestJoin messages; this state is deleted if no matching RequestJoin messages arrive before the expiration of a "cleanup timeout" interval. Listener's may also force state deletions by generating an explicit RequestLeave message.

RequestJoin messages are idempotent. When a route changes, the next RequestJoin message will initialize the path state to the new route, and future RequestJoin messages will establish state there. The state on the now-unused segment of the route will be deleted after a timeout interval. Thus, whether a RequestJoin message is "new" or a "refresh" is determined separately by each station, depending upon the existence of state at that station.

SRP soft state is also deleted in the continued absence of associated talker-generated ConfirmJoin messages; the listener's registration is discarded if no matching ConfirmJoin indication arrives before the expiration of a "cleanup timeout" interval. Thus, talker stations or agents may implicitly deregister by stopping its ConfirmJoin confirmations, or explicitly deregister by sending distinct ConfirmGone messages.

Editors' Notes: To be removed prior to final publication.
Additional discussions may be appropriate to discuss operation of the ConfirmGone messages.

SRP sends it messages as layer-2 datagrams with no reliability enhancement. Periodic transmissions by listener/talker stations and agents is expected to handle the occasional loss of an SRP message.

In the steady state, state is refreshed on a hop-by-hop basis to allow merging. Propagation of a change stops when and if it reaches a point where merging causes no resulting state change. This minimizes the SRP control traffic and is essential for scaling to large audiences.

5.4.3 Subscription bandwidth constraints

The SRP subscription protocols limit cumulative bandwidth allocations to a fixed percentage less than the capacity of the link, much like IEEE 1394 limits isochronous traffic to less than the capacity of its bus. This guarantees that high priority management information can be transmitted across the link. For RE systems, classA traffic is limited to 75% of the capacity of any RE link. Enforcement of such a limit is done in multiple ways:

a) Subscription. Requests for establishing classA transmission paths are rejected if the cumulative bandwidths of all paths would consume more than 75% of the link bandwidth.

b) Transmit queue hardware of RE stations (including bridges) discards classA content that (if transmitted) would cause classA traffic to exceed 75% of the transmit link capacity. Details are TBD.

Method (b) is desired to recovery from unexpected transient conditions (typically topology changes) that result in admission control violations, and is also useful for managing misbehaving devices

5.4.4 Controller entities

Subscription when a relative-intelligent controller discovers the need to establish a classA path between talker and listener entities. For example, user interactions with a television (called the controller) may cause streams flowing between the content source (called the talker) and speakers (the listeners), as illustrated in Figure 5.8.

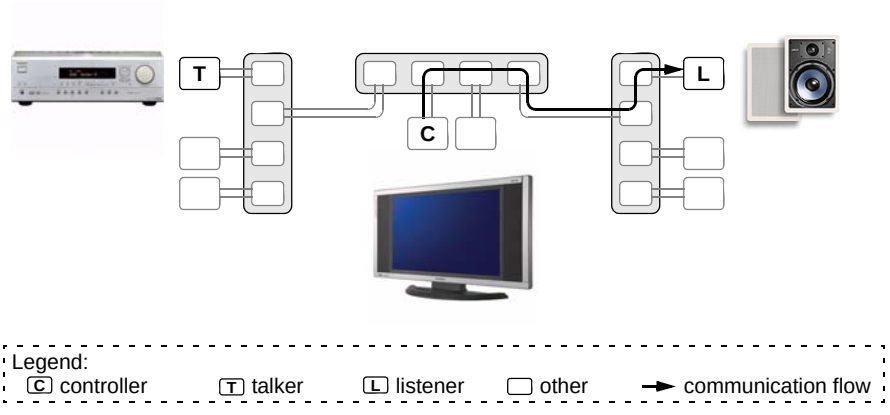


Figure 5.8—Controller activation

A controller can potentially simplify the listener by reducing the need to providing user interface and device-discovery capabilities. However, a controller could also reside within talker and/or listener components. However, actions between controllers and talker/listener stations are beyond the scope of this working paper.

5.4.5 Bridge-resident agents

Subscription facilities register classA communication paths from a talker to one or more listeners. Streams of time-sensitive data can then flow over these established paths, as illustrated by the dark arrow paths in Figure 5.9-a. Maintaining these established paths involves active participation of agents within the end-point talker, local listener, local talker, and end-point listener entities, as illustrated in Figure 5.9-b.

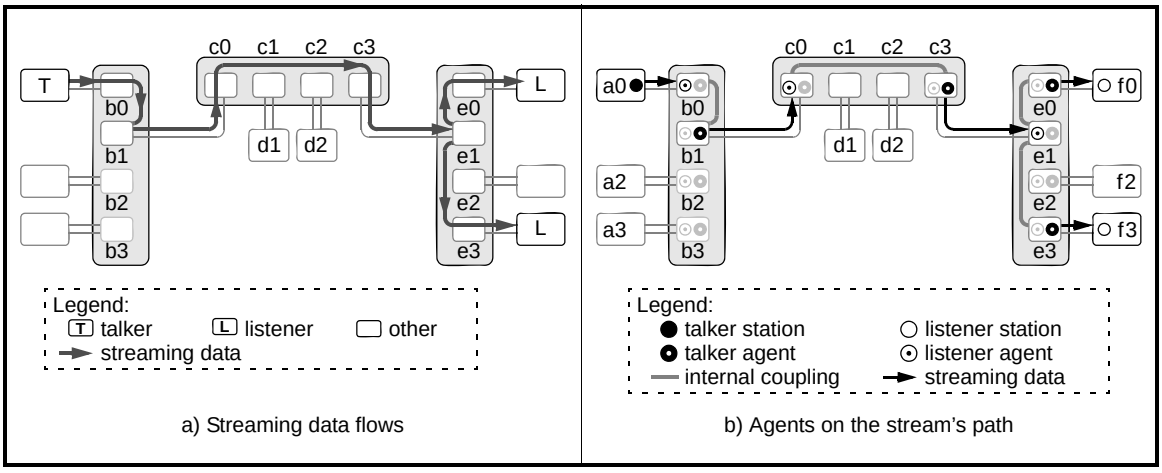


Figure 5.9—Agents on an established path

The talker stations/agents are responsible for maintaining an account consisting of {streamID, bandwidth} pairs, one for each of their distinct flows. Requests for additional link bandwidth are checked against these accounts and denied if the cumulative bandwidth would exceed 75% of the link capacity.

For each of the registered talker agents within a bridge, the listener agent remains active until all but the last talker agent registration is discarded. Thus, the talker agent in an upstream station receives its deregistration notice only after the last of the downstream listener stations has been deregistered.

The listener agent uses the same RequestJoin messages to establish and to maintain the path. This reduces design complexity and (most importantly) automatically re-routes stream flows after topology changes.

5.4.6 Registration

Registering a new listener and talker starts with a RequestJoin message sent from the listener *f0* towards the talker *a0*, as illustrated by the dark arrow (1a) in Figure 5.10-a. These registration messages are not forwarded directly, but activate cooperative listener and talker agents with the bridge.

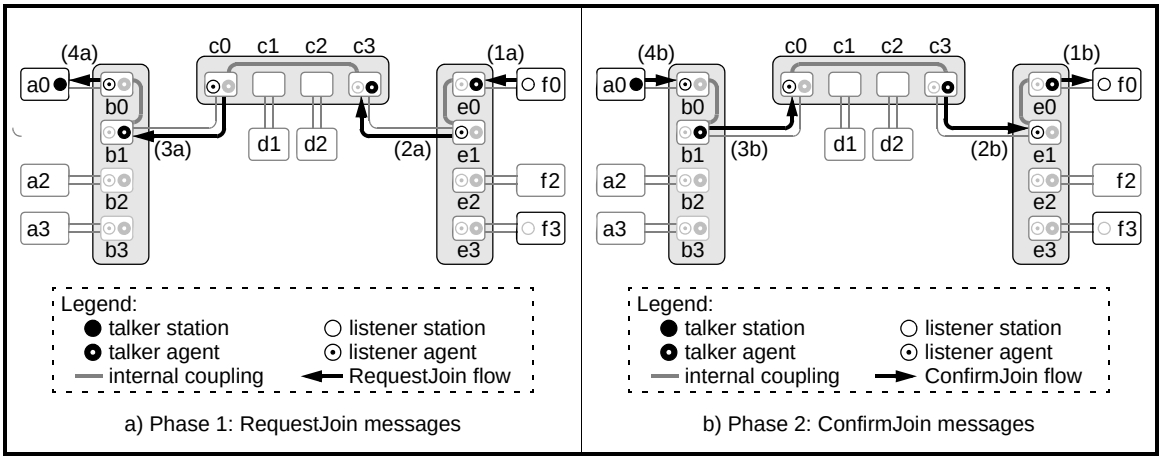


Figure 5.10—Periodic registration messages

In response to the received RequestJoin message (1a), bridgeE reserves talker-agent and listener-agent registration table entries in ports *e0* and *e1* respectively. A cascaded RequestJoin message (2a) is then sent towards talker station *a0*.

The cascaded forwarding continues through bridgeC. In response to the received RequestJoin message (2a), bridge C reserves talker-agent and listener-agent registration table entries in ports *c3* and *c0* respectively. A cascaded RequestJoin message (3a) is then sent towards talker station *a0*.

The cascaded forwarding continues through bridgeB. In response to the received RequestJoin message (3a), bridge B reserves talker-agent and listener-agent registration table entries in ports *b1* and *b0* respectively. A cascaded RequestJoin message (4a) is then sent towards talker station *a0*.

Referring now to Figure 5.10-b, the talker and talker agents are responsible for providing confirming ConfirmJoin messages, to confirm their continued presence. In this example, the RequestJoin messages {1a,2a,3a,4a} of Figure 5.10-a are continually confirmed by the ConfirmJoin messages {1b,2b,3b,4b} of Figure 5.10-b), respectively. In the continued absence of the expected ConfirmJoin messages, the talker (or talker-agent) assumes the listener (or listener-agent) is absent or has been deactivated.

Another timeouts is associated with the absence of periodic RequestJoin messages. In the continued absence of these expected messages, the talker assumes the listener is absent or has been deactivated. Based on this assumption, the associated talker (station or agent) registration resources are released.

5.4.7 Secondary listener registrations

A second listener registers by sending a RequestJoin message towards the talker, as illustrated by the dark-arrow path in Figure 5.11-a. When an established registration is discovered, the bridge (not the talker) processes the message. Thus, the registration is expanded to include a new-listener side path, as illustrated in Figure 5.11-b.

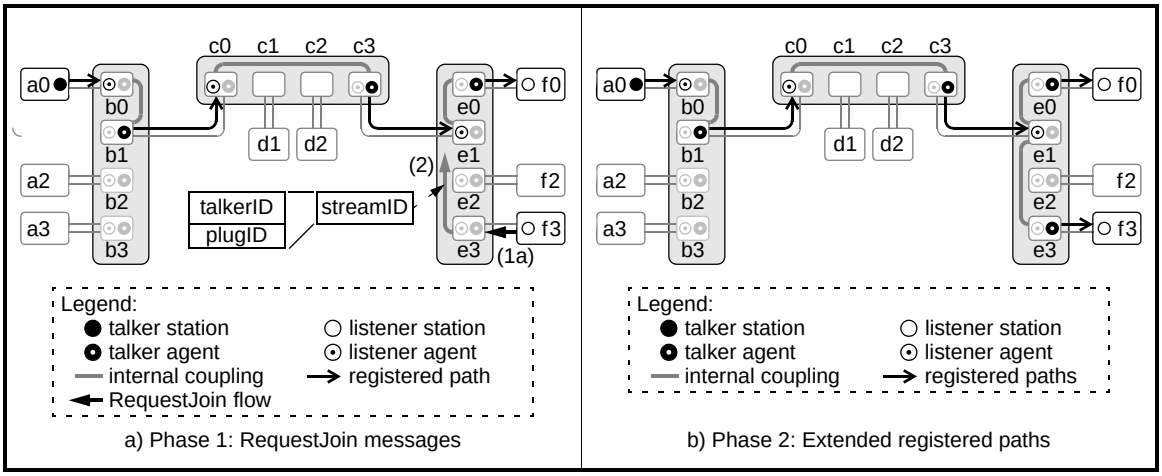


Figure 5.11—Secondary registrations

Each talker and listener agent maintains separate registration state, so that only active paths are registered. Maintaining distinct registrations also allows the bridge to detect when the last listener disconnects, so that its previously shared upstream span can be deregistered appropriately.

Each path is uniquely identified by its associated streamID. The streamID consists of a {*talkerId*, *plugID*} information that uniquely identifies the associated talker resource), as illustrated by the rectangle inserts within Figure 5.11-a. The talkerID represents the MAC address of the talker and the plugID distinguishes between possible streaming sources within the talker.

The multicast address used to route the classA multicast frames, as well as the allocated classA bandwidth, are returned to the listeners within ResponseForm messages.

5.4.8 Secondary listener deregistration

A retiring secondary listener normally leaves an established registration by sending a RequestLeave message towards the talker. That RequestLeave message (1a) propagates to the nearest merging bridge connection, as illustrated in Figure 5.12-a. When an established/merged registration is discovered, the bridge (not the talker) deregisters the listener, as illustrated by the disappearance of external path e0-to-f0 and internal path e1-to-e0 within Figure 5.12-b.

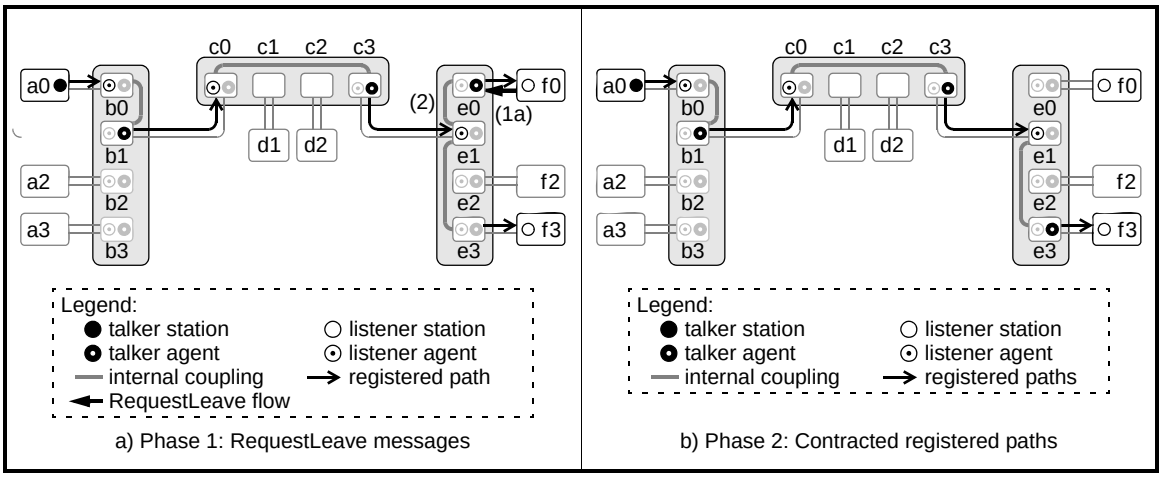


Figure 5.12—Side-path deregistration

5.4.9 Final deregistration

The final retiring listener also sends a RequestLeave message (1a) towards the talker. In this case, variants of that message {2a,3a,4a} eventually propagate to the talker, as illustrated in Figure 5.13-a. No listeners remain registered after this cascaded propagation of RequestLeave messages, as illustrated in Figure 5.13-b.

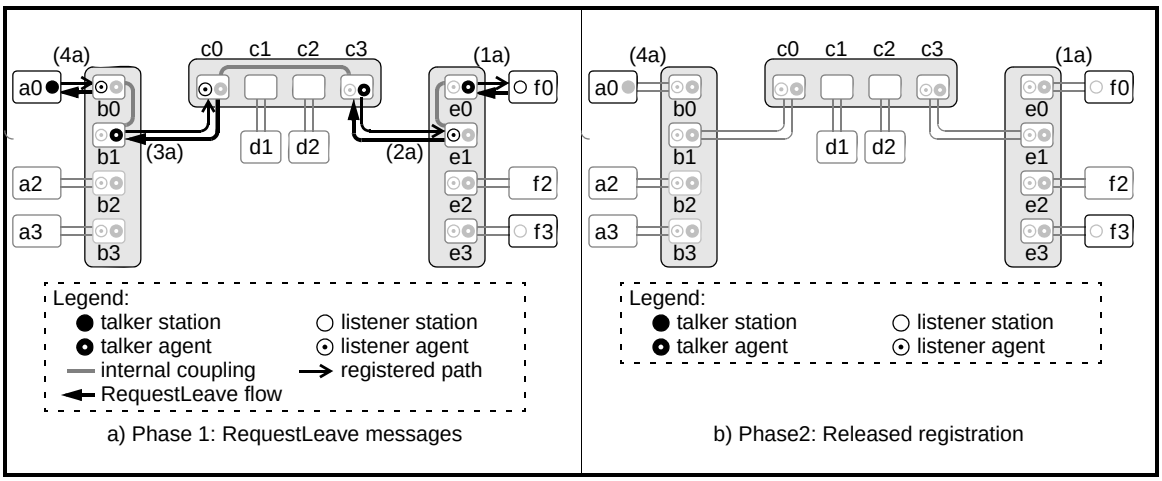


Figure 5.13—Final-path deregistration

Contribution from: dvj@alum.mit.edu.

5.4.10 Stream transmissions

Once listeners are registered (see Figure 5.14-a), a talker communicates critical parameters within the ConfirmPath message (instead of the initial ConfirmJoin messages) and starts its stream transmissions over the registered paths, as illustrated by the arrows in Figure 5.14-b.

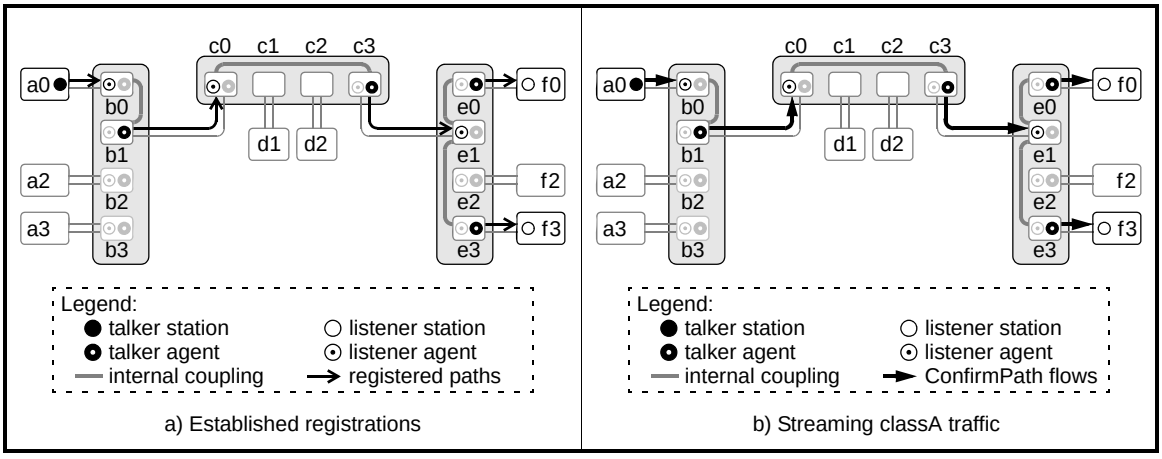


Figure 5.14—Streaming data over registered paths

The ConfirmPath message could be a variant of the ConfirmJoin message with a distinct command-code value. Like the baseline ConfirmJoin message, the ConfirmPath message is also sufficient to sustain the talker's registration. This simplifies the talkers (and talker agents) by eliminating the need to concurrently transmit two distinct periodic registration-sustaining messages.

5.4.11 Insufficient bandwidth conditions

The available link bandwidths can sometimes be insufficient when the talker starts its stream transmissions. For example, bandwidths may be sufficient to sustain listener *f0* but not listener *f3*, as illustrated by the *e0*-to-*f0* and *e3*-to-*f3* paths in Figure 5.15-a, respectively.

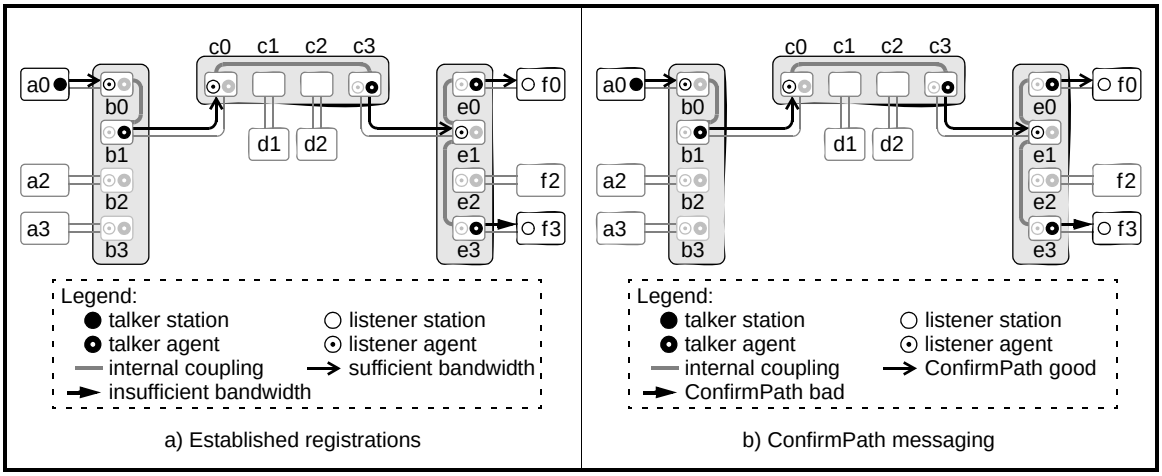


Figure 5.15—Insufficient bandwidth conditions

In this case, listener *f3* does not receive the talker's streaming classA traffic. However, listener *f3* continues to receive its ConfirmJoin messages, each of which contains an error indication code. Listener *f3* is thus informed of the insufficient-bandwidth error condition, allowing corrective/reporting actions to be initiated by higher level protocols.

5.4.12 Errors conditions

Errors may be associated with a variety of failure conditions, including (but not limited to) those listed below.

a) Resources. Insufficient resources are available within the bridge.
(These insufficient-resource errors are handled by GARP specified mechanisms, see TBD.)
   1) Insufficient registration-table storage is available in the bridge's downstream talker agent.
   2) Insufficient registration-table storage is available in the bridge's upstream listener agent.

b) Bandwidth. Insufficient bandwidths are available within the bridge.
(These insufficient-bandwidth errors are handled by ConfirmJoin error codes, see 5.4.11.)
   1) Insufficient bandwidth is available on the link from the talker agent to its adjacent listener.
   2) Insufficient link or memory bandwidth is available with the bridge.

5.4.13 Heartbeat timeouts

Talker agents/stations are responsible for periodically polling locally registered listener agents/stations, to demonstrate their continued presence. In the absence of these polling updates, the listeners assume the talker is absent and deregister the inactive path (or inactive branch from the path). These talker-absent timeouts are performed independently on each span.

Listener agents/stations are responsible for periodically reregistering with locally registered talker agents/stations, to confirm their continued presence. In the absence of these reregistration updates, the talkers assume the listener is absent and deregister the inactive path (or inactive branch from the path). These listener-absent timeouts are performed independently on each span.

These periodic heartbeat-based timeouts handle a variety of error conditions, including the following:

a) A RequestJoin, RequestLeave, ConfirmJoin, or ConfirmPath is (corrupted and) not delivered.
b) The physical topology is changed, causing changes in the paths of streaming classA traffic.
c) A talker or listener is decommissioned and thus is no longer functionally present.
d) A flooded RequestJoin message reaches a non-talker end station or subnet.
e) After the talker's port is learned, a bridge discontinues flooding extraneous RequestJoin messages.

5.4.14 Untended flooding

Registering a new listener normally involves cascaded RequestJoin message sent from the listener *f0* towards the talker *a0*, as illustrated in Figure 5.10-a. In some cases, the talker's address may be unlearned and flooding may be necessary. Thus, BridgeB could sometimes be forced to flood the RequestJoin to stations {*a0*,*a2*,*a3*}, when an unlearned address can't be directed to station *a0*, as illustrated in Figure 5.10-b.

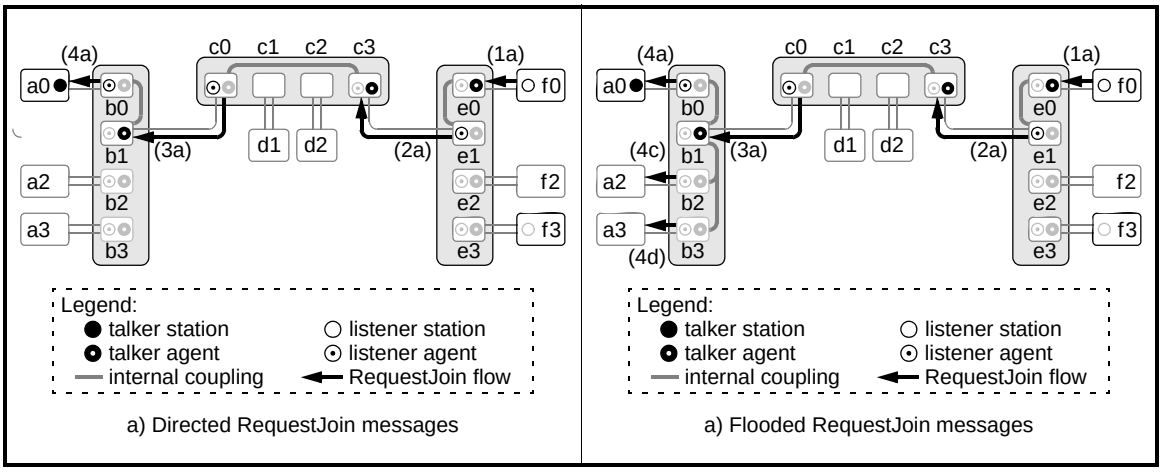


Figure 5.16—Periodic registration messages

In this example, talker *a0* is present and its ConfirmJoin messages will soon propagate back to bridgeB, where the address of talker station *a0* is learned. When this occurs, the flooding to stations {*a2*,*a3*} stops.

Editors' Notes: To be removed prior to final publication.
Additional discussions may be appropriate to discuss what happened when the talker address is absent, as simply summarized below.

As noted previously (see 5.4.13), the talker agent is responsible for providing confirming ResponseJoin messages, so that the absence of a talker station can be readily detected. Allocated registration-table entries within bridges can be released after the talker-station absence is detected. Thus, flooding causes no harm.

5.4.15 GARP primitives

This subclause was intended to clarify the higher level SRP functionality. Thus, names of primitives were chosen form clarity, rather than consistency with the expected GARP messages. For the benefit of experienced GARP users, a sketch of the intended mappings of primitives is provided within this subclause.

The RequestJoin and RequestLeave messages correspond to like-names primitives within GARP. The ConfirmJoin and ConfirmPath messages correspond to variants of the leave-all messages within GARP.

5.5 Synchronized time-of-day clocks

This working paper specifies a protocol to synchronize independent timers running on separate stations of a distributed networked system, based on concepts specified within IEEE Std 1588-2002. Although a high degree of accuracy and precision is specified, the technology is applicable to low-cost consumer devices. The protocols are based on the following design assumptions:

a) Each end station and intermediate bridges provide independent clocks.

b) All clocks are accurate, typically to within ±100PPM.

c) Point-to-point transmit/receive duplex connections are provided.

d) Transmit/receive propagation delays within duplex cables are well matched.

5.6 Formats

5.6.1 Content framing

ClassA content is the client supplied per-cycle classA information, transferred from a talker to one or more listeners. The content within each cycle can be small or large; stereo audio stream transfers involve only approximately 20 bytes per cycle. Uncompressed 32-bits/pixel frame buffers (2 megapixels, 30Hz) would transmit 30 kilobytes per cycle. Framing of this content must be efficient for small sizes and sufficient for large sizes, as illustrated in Figure 5.17.

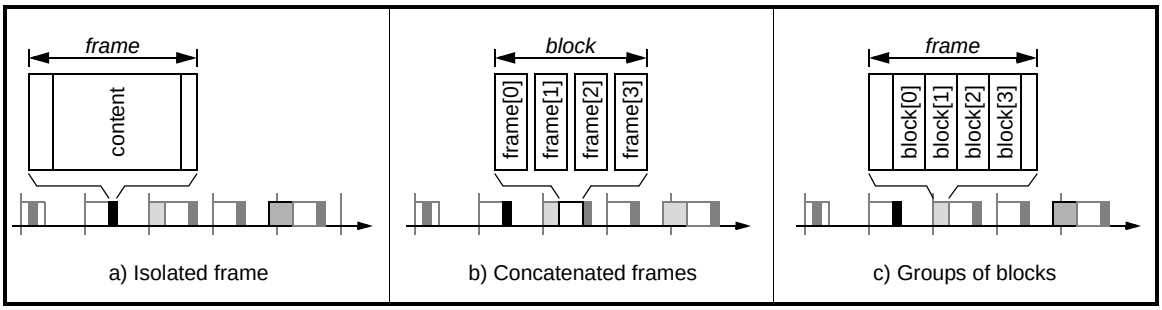


Figure 5.17—Content framing methods

For low bandwidth transmissions, each frame transports distinct classA content, as illustrated in Figure 5.17-a. For high bandwidth transmissions, the content can span multiple frames, as illustrated in Figure 5.17-b (see also C.3.2).

As an alternative improved-efficiency alternative, low bandwidth content could be encapsulated into blocks, where multiple blocks are included within each frame transmission, as illustrated in Figure 5.17-c. This allows the per-frame overhead (the inter-packet gap, header, and trailer fields) to be amortized over multiple blocks. For example, the eight inputs from a guitar may be packed together into the same frame. However, the packing of multichannel content is beyond the scope of this working paper.

Another approach would be to reduce the need for concatenated frames by using the (defacto standard) jumbo-frame sizes, which are approximately 9,000 bytes in size. However, support of the jumbo frame size is not ensured, and (when supported) is considerably less than 2^{16}-byte maximum size of an IEEE 1394 isochronous frame, or the 118 kilobyte size implied by 75% utilization of a 10Gb/s link.

5.6.2 Station plug addressing

Stream addressing is based on the concept of plugs, as illustrated in Figure 5.18. Streams are identified by their 48-bit talker-station identifier concatenated with that talker's 16-bit *plugId*. Each talker station may have up to 2^{16} streams, via logical plugs, in addition to the station's hardwired connections Stations are expected to provide higher level commands for connecting/mixing/amplifying/converting/etc. data between combinations of hardwired and logical plugs. However, the details of such commands are beyond the scope of this working paper.

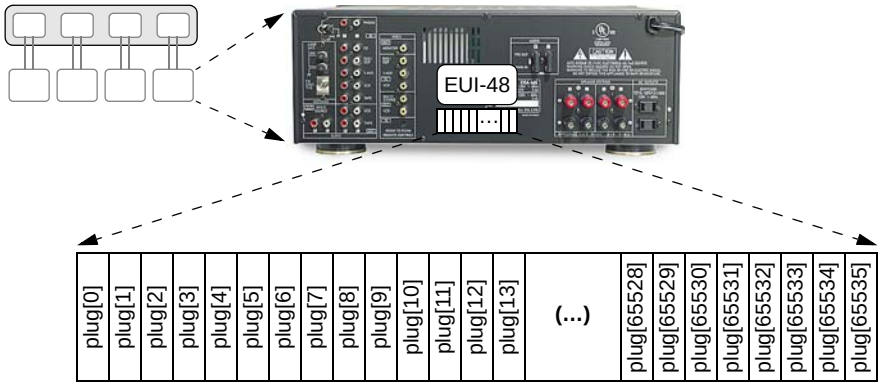


Figure 5.18—Plug addressing

5.6.3 Stream frame formats (alternative 1)

Streaming classA frames are no different than other multicast Ethernet frames. The distinction is that each of these multicast addresses is assumed to have associated *streamID* and bandwidth information saved within each forwarding bridge, as illustrated in Figure 5.19.

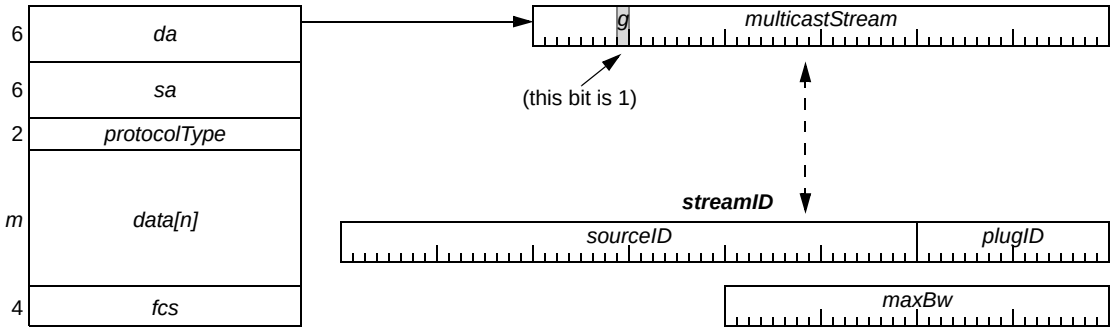


Figure 5.19—ClassA frame format and associated data

The *streamID* consists of two components: *sourceID* and *plugID*. The 48-bit *sourceID* identifies the source and usually equals the *sa* value; the *plugID* identifies the resource within that source. A distinct *maxBw* (maximum bandwidth) field identifies the negotiated maximum for classA bandwidth.

5.6.4 Stream frame formats (alternative 2)

Streaming classA frames are no different than other Ethernet frames. The distinction is that each of these frames supplies a nonzero user_priority field, as illustrated in Figure 5.20.

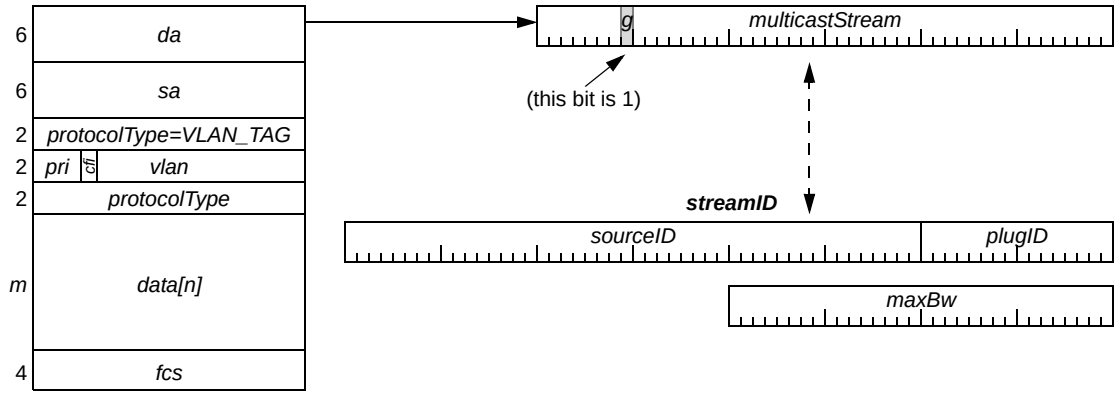


Figure 5.20—ClassA frame format and associated data

The *streamID* consists of two components: *sourceID* and *plugID*. The 48-bit *sourceID* identifies the source and usually equals the *sa* value; the *plugID* identifies the resource within that source. A distinct *maxBw* (maximum bandwidth) field identifies the negotiated maximum for classA bandwidth.

5.6.5 Stream frame formats (alternative 3)

Frames within a stream are no different than other Ethernet frames, with the exception of their distinct *da* (destination address) field, as illustrated in Figure 5.21. The most significant 32-bit portion of the *da* classifies the frame as an classA frame. The less significant 16-bit portion specifies the *plugID* portion of the *streamID* associated with the frame.

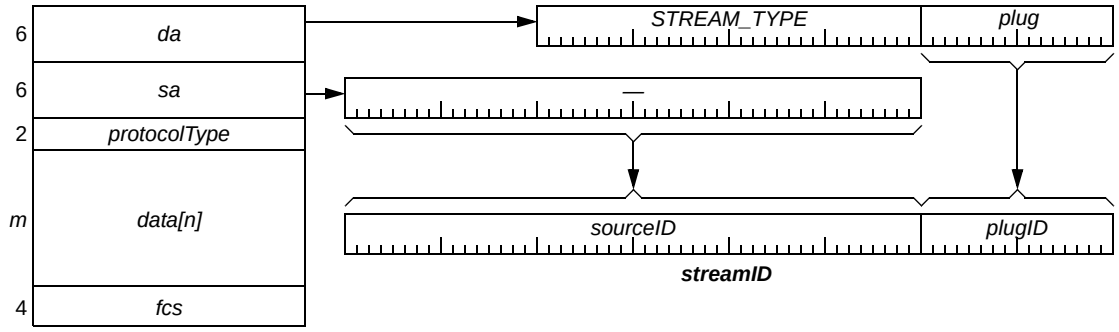


Figure 5.21—ClassA frame formats

5.6.6 Stream frame format alternatives comparison

Quality of service is thus specified by the user_priority field parameter within VLAN-tagged frames, as listed in Table 5.2.

Table 5.2—Tagged priority values

Alternative	Compact	Similar	Multicast server
1: DA-multicast	good	good	poor
2: VLAN-priority	poor	best	poor
3: SA-multicast	good	poor	good

The DA-multicast header is the compact, its forwarding mechanism are similar to those now supported, but a multicast server is required to provide unique multicast-stream addresses.

The VLAN-priority header is the 4 bytes larger, its forwarding mechanism is nearly identical to those now supported, but a multicast server is required to provide unique multicast-stream addresses.

The SA-multicast header is the compact, its forwarding mechanism is quite different than those now supported by bridges, but has the advantage that no multicast server is not required.

5.7 Pacing

5.7.1 Assumptions

This working paper specifies a protocols for pacing classA traffic streams as they pass through multiple bridges. Although a high degree of scalability is implied, the technology is applicable to inexpensive consumer devices. The protocols are based on the following design assumptions:

a) Sizes. The maximum frame size is assumed to be 2 kB, for consistency with established 802.3 frame-extension working group directions.

b) Speeds. Only full-duplex 100 Mb/s, 1 Gb/s, and 100 Gb/s 100-meter links must be supported.

c) Limits. The classA traffic transmissions are limited to 75% of the available link bandwidth.

5.7.2 Objectives

With these assumptions in mind, the time synchronization objectives include the following:

a) Reliable. The worst-case delay between talker and listener stations is small, deterministic, and not effected by operating conditions, including the following:

   1) Loading. Arbitrary talker-station and listener-station traffic patterns can be supported.
   2) Scaling. Any 802.1 supported spanning tree topologies can be supported.

b) Plug-and-play. Manual provisioning of the system is not required.

c) Compatible. The pacing of high-class frames cannot disrupt legacy or lower-class transmissions.

d) Friendly. Some higher-class traffic that cannot be reliably paced, due to legacy sources or bridges; retains precedence over lower-class traffic.

e) Robust. Higher-class traffic never starves the forwarding of lower-class control traffic.

f) Efficient. Unused higher-class bandwidth can be readily reclaimed lower-class traffic.

5.7.3 Strategies

Strategies used to meet these objectives include the following:

a) Buckets. Higher-class traffic is grouped into buckets; buckets are forwarded every 125 μs cycle.

b) Limits. The levels of higher-class traffic are limited to 75% of the link bandwidths.

   1) Excess classA traffic above this 75% limit is discarded.
   2) Excess classB traffic above this 75% limit is temporarily processed as classC traffic.

c) Reuse. Unused higher-precedence bandwidths are reused if not consumed as intended.

   1) Unused classA traffic within the 75% limit is available for classB traffic.
   2) Unused classA/classB traffic within this 75% limit is available for classC traffic.

d) Downgrade. When passing through unsupportive bridges, classA traffic is downgraded to classB. The classB traffic is no longer paced, but retains its precedence over classC traffic.

6. Frame formats

6.1 timeSync frame format

6.1.1 timeSync fields

Clock synchronization (timeSync) frames facilitate the synchronization of neighboring clock span-master and clock span-slave stations. The frame, which is normally sent once each isochronous cycle, includes time-snapshot information and the identity of the network's clock master, as illustrated in 6.1. The gray boxes represent physical layer encapsulation fields that are common across all Ethernet frames.

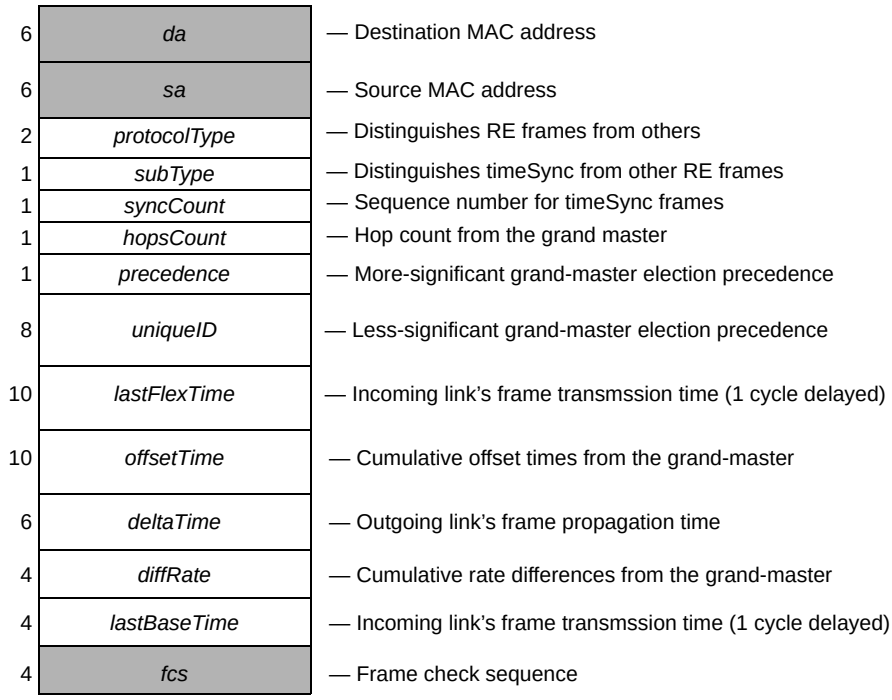


Figure 6.1—timeSync frame format

6.1.1.1 *da*: A 48-bit (destination address) field that specifies the station(s) for which the frame is intended. The *da* field contains either an individual or a group 48-bit MAC address (see 3.11), as specified in 9.2 of IEEE Std 802-2001.

6.1.1.2 *sa*: A 48-bit (source address) field that specifies the local station sending the frame. The *sa* field contains an individual 48-bit MAC address (see 3.11), as specified in 9.2 of IEEE Std 802-2001.

6.1.1.3 *protocolType*: A 16-bit field contained within the payload that identifies the format and function of the following fields.

6.1.1.4 *subType*: A 16-bit field that identifies the format and function of the following fields.

6.1.1.5 *syncCount*: An 8-bit field that is incremented on each timeSync frame transmission.

6.1.1.6 *hopsCount*: An 8-bit field that identifies the maximum number of hops between the talker and associated listeners.

6.1.1.7 *precedence*: An 8-bit field that has highest precedence in the grand-master selection protocols.

6.1.1.8 *uniqueID*: A 64-bit field that specifies the precedence of the grand clock master, specified in 6.1.2.

6.1.1.9 *lastFlexTime*: An 80-bit field that specifies the time within the source station when the previous timeSync frame was transmitted. The format of this field is specified in 6.1.3.

6.1.1.10 *offsetTime*: An 80-bit field that specifies the offset time within the source station. The format of this field is specified in 6.1.3.

6.1.1.11 *deltaTime*: A 48-bit field that specifies the differences between timeSync receive and transmit times, as measured on the opposing link. The format of this field is specified in 6.1.3.

6.1.1.12 *diffRate*: A 32-bit field that specifies the *diffRate* value within the source station.

6.1.1.13 *lastBaseTime*: A 32-bit field that specifies the *timer1* value within the source station when the previous timeSync frame was transmitted.

6.1.1.14 *fcs*: A 32-bit (frame check sequence) field that is a cyclic redundancy check (CRC) of the frame.

6.1.2 *uniqueID* fields

The format of the 64-bit *uniqueID* field is a unique identifier. For stations that have a uniquely assigned 48-bit *macAddress*, the 64-bit *uniqueID* field is derived from the 48-bit MAC address, as illustrated in Figure 6.2.

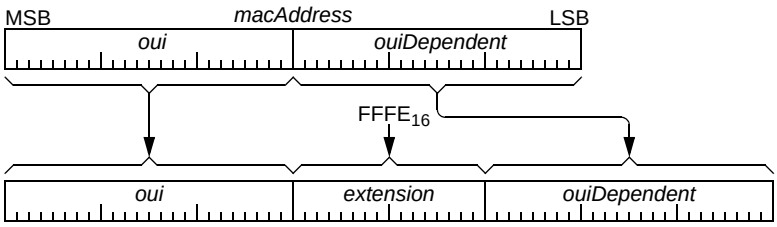


Figure 6.2—*uniqueID* format

6.1.2.1 *oui*: A 24-bit field assigned by the IEEE/RAC (see xx).

6.1.2.2 *extension*: A 16-bit field assigned to encapsulated EUI-48 values.

6.1.2.3 *ouiDependent*: A 24-bit field assigned by the owner of the *oui* field (see xx).

6.1.3 Time field formats

Time-of-day values within a frame are based on 64-bit values, consistent with IETF specified NTP[B8] and SNTP[B9] protocols. The 80-bit *lastFlexTime* and *offsetTime* values consist of three components: a 16-bit *epoch*, a 32-bit *seconds*, and 32-bit *fraction* fields, as illustrated in Figure 6.3.

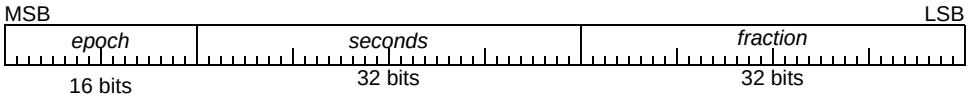


Figure 6.3—Complete seconds timer format

6.1.3.1 *epoch:* A 32-bit field that specifies time in seconds.

6.1.3.2 *seconds:* A 32-bit field that specifies time in seconds.

6.1.3.3 *fraction*: A 32-bit field that specified time offset within the second, in units of 2^{-32} second.

The concatenation of 16-bit *epoch*, 32-bit *seconds* and 32-bit *fraction* field specifies an 80-bit *time* value, as specified by Equation 6.2.

$$time = epoch*2^{32} + seconds + (fraction / 2^{32}) \quad (6.1)$$

Where:

epoch is the most significant component of the time value (see Figure 6.3).
seconds is the more significant component of the time value (see Figure 6.3).
fraction is the less significant component of the time value (see Figure 6.3).

6.1.4 Time-difference field formats

The time-difference value within a frame is based on a 48-bit value, consistent with IETF specified NTP[B8] and SNTP[B9] protocols. This 48-bit *deltaTime* value consists of two components: a 16-bit *seconds* and a 32-bit *fraction* fields, as illustrated in Figure 6.4.

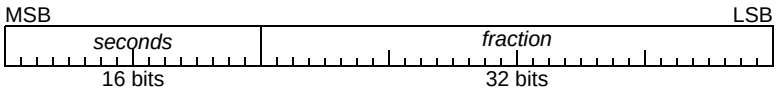


Figure 6.4—Complete seconds timer format

6.1.4.1 *seconds:* A 16-bit field that specifies time in seconds.

6.1.4.2 *fraction*: A 32-bit field that specified time offset within the second, in units of 2^{-32} second.

The concatenation of 16-bit *seconds* and 32-bit *fraction* field specifies a 48-bit *time* value, as specified by Equation 6.2.

$$time = seconds + (fraction / 2^{32}) \quad (6.2)$$

Where:

seconds is the most significant component of the time value (see Figure 6.4).
fraction is the less significant component of the time value (see Figure 6.4).

7. Timer synchronization

7.1 Synchronized time-of-day timers

7.1.1 Assumptions

This working paper specifies a protocol to synchronize independent timers running on separate stations of a distributed networked system, based on concepts specified within IEEE Std 1588-2002. Although a high degree of accuracy and precision is specified, the technology is applicable to low-cost consumer devices. The protocols are based on the following design assumptions:

a) Each end station and intermediate bridges provide independent clocks.

b) All clocks are accurate, typically to within ±100PPM.

c) Point-to-point transmit/receive duplex connections are provided.

d) Transmit/receive propagation delays within duplex cables are well matched.

7.1.2 Objectives

With these assumptions in mind, the time synchronization objectives include the following:

a) Precise. Multiple timers can be synchronized to within 10's of nanoseconds.

b) Inexpensive. For consumer A/V devices, the costs of synchronized timers are minimal. (GPS, atomic clocks, or 1PPM clock accuracies would be inconsistent with this criteria.)

c) Scalable. The protocol is independent of the networking technology. In particular:

   1) Cyclical physical topologies are supported.
   2) Long distance links (up to 2 kM) are allowed.

d) Plug-and-play. The system topology is self-configuring; no system administrator is required.

7.1.3 Strategies

Strategies used to meet these objectives include the following:

a) Precision is achieved by calibrating and adjusting *timeOfDay* clocks.

   1) Offsets. Offset value adjustments eliminate immediate clock-value errors.
   2) Rates. Rate value adjustments reduce long-term clock-drift errors.

b) Simplicity is achieved by the following:

   1) Concurrence. Most configuration and adjustment operations are performed concurrently.
   2) Feed-forward. PLLs are unnecessary within bridges, but possible within applications.
   3) Symmetric. Clock-master/clock-slave computations are similar (only slave results are saved).
   4) Periodic. Messages are sent periodically, rather than in timely response to other requests.
   5) Frequent. Frequent (every 10 ms) interchanges reduces needs for precise clocks.

c) Balanced functionality.

   1) Low-rate. Complex computations are infrequent and can be readily implemented in firmware.
   2) High-rate. Frequent computations are simple and can be readily implemented in hardware.

7.1.4 Timer synchronization services

Clock synchronization involves the transmission and reception of timeSync frames interchanged between adjacent-span stations, using the state machines defined within this clause. When considered as a whole, these provide the following services:

a) Election. The grand clock master is elected from among the grand-clock-master capable stations.

b) Isolation. Timeouts identify the boundaries, beyond which RE services are not supported.

c) Clock-sync. Clock-slave stations are synchronized to the grand master station's time reference.

7.1.5 Grand-master selection

7.1.5.1 Grand-master selection

Clock synchronization involves streaming of clock-synchronization information from a grand-master timer to one or more slave timers. Although primarily intended for non-cyclical physical topologies (see Figure 7.1a), the synchronization protocols also function correctly on cyclical physical topologies (see Figure 7.1b), by activating only a non-cyclical subset of the physical topology.

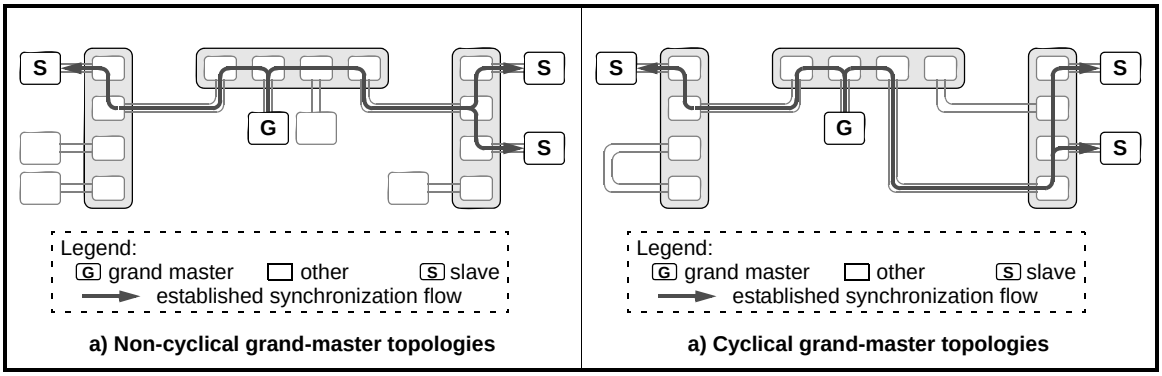


Figure 7.1—Timer synchronization flows

In concept, the clock-synchronization protocol starts with the selection of the reference-timer station, called a grand-master station (oftentimes abbreviated as grand-master). Every RE-capable station is grand-master capable, but only one is selected to become the grand-master station within each network. To assist in the grand-master selection, each station is associated with a distinct preference value; the grand-master is the station with the "best" preference values.

As part of the grand-master selection process, stations forward the best of their observed preference values to neighbor stations, allowing the overall best-preference value to be ultimately selected and known by all. The station whose preference value matches the overall best-preference value ultimately becomes the grand-master.

7.1.5.2 Communicated preference values

The grand-master station observes that its precedence is better than values received from its neighbors, as illustrated in Figure 7.2a. A slave stations observes its precedence to be worse than one of its neighbors and forwards the best-neighbor precedence value to adjacent stations, as illustrated in Figure 7.2b. To avoid cyclical behaviors, a *hopsCount* value is associated with preference values and is incremented before the best-precedence value is communicated to others.

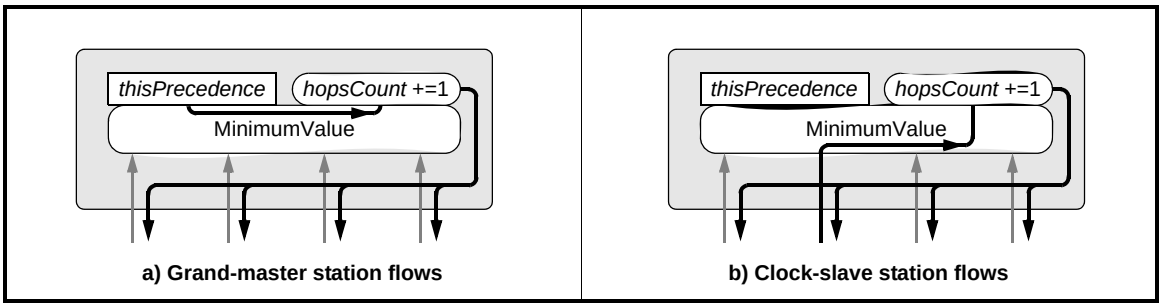


Figure 7.2—Grand-master precedence flows

The grand-master selection precedence includes multiple components, listed and described below (see 7.1.8). The *portTag* value is only needed within a bridge and is therefore not transmitted between stations.

a) *precedence*. A changeable value that is associated with each grand-master capable station. This value can specify grand-master preferences (e.g., a home gateway may be preferred).
b) *uniqueID*. A unique value associated with each station, typically based on its MAC address. This value is used as a tie breaker, when two contenders have identical *precedence* values.
c) *hopsCount*. A value that is incremented when passing through stations. This is the tie breaker, when two ports receive identical *precedence*:*uniqueID* values.
d) *port*. A value that is associated with each port on a grand-master capable station. This is the tie breaker, when two ports receive identical *precedence*:*uniqueID*:*hopsCount* values.

7.1.6 Clock-synchronization agents

Clock-synchronization information conceptually flows from a grand-master station to clock-slave stations, as illustrated in Figure 7.3a. A more detailed illustration shows pairs of synchronized clock-master and clock-slave components, as illustrated in Figure 7.3b.

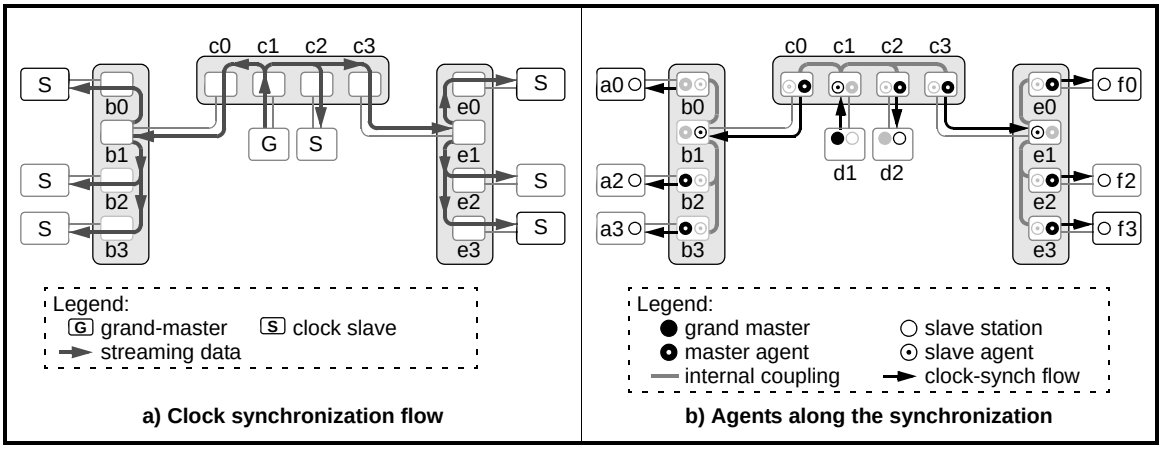


Figure 7.3—Hierarchical flows

7.1.7 Clock-synchronized pairs

Each bridge port provides clock-master and clock-slave agents, although both are never simultaneously active. External communications (see 7.3b) synchronize clock-slaves to clock-masters, as listed in Table 7.1.

Table 7.1—External clock-synchronization pairs

Grand master	Clock master agent	Clock slave agent	Clock slave	Type of synchronization
d1	–	c1	–	Station-to-bridge
–	c0	b1	–	Bridge-to-bridge
–	c3	e1	–	
–	b0	–	a0	Bridge-to-station
–	b2	–	a2	
–	b3	–	a3	
–	c2	–	d2	
–	e0	–	f0	
–	e2	–	f2	
–	e3	–	f3	

Internal communications distribute synchronized time from clock-slave agents b1, c1, and e1 to the other clock-master agents on bridgeB, bridgeC, and bridgeE respectively. However, bridge-internal port-to-port synchronization protocols are implementation-dependent and beyond the scope of this working paper.

Within a clock-slave, precise time synchronization involves adjustments of timer offset and rate values. The adjustments of the timer's offset is called offset synchronization (see 7.1.12); the adjustments of the timer's rate is called rate synchronization (see 7.1.13). Both involve calibration of local clock-master/clock-slave differences and the propagation of cumulative differences in the clock-slave direction, as described by the C code of Annex I.

Time synchronization yields distributed but closely-matched *timeOfDay* values within stations and bridges. No attempt is made to eliminate intermediate jitter with bridge-resident jitter-reducing phase-lock loops (PLLs,) but application-level phase locked loops (not illustrated) are expected to filter high-frequency jitter from the supplied *timeOfDay* values

7.1.8 Grand-master precedence

Grand-master precedence is based on the concatenation of multiple fields, as illustrated in Figure 7.4. The *portTag* value is used within bridges, but is not transmitted between stations.

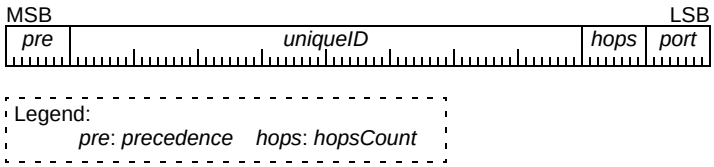


Figure 7.4—Grand-master precedence

This format is similar to the format of the spanning-tree precedence value, but a larger *uniqueID* is provided for compatibility with interconnects based on 64-bit station identifiers.

7.1.9 Synchronization principles

Timer synchronization is based on the concept of free-running local times (*localD*, *localE*, and *localF*) with compensating offset values (*offsetD*, *offsetE*, and *offsetF*), as illustrated in Figure 7.5. Updates involve changes to the offset values, not the free-running local timer values. In this example, we assume that: StationE is synchronized to its adjacent StationD; StationF is synchronized to its adjacent StationE. As a result, StationF is indirectly synchronized to StationD (through StationE).

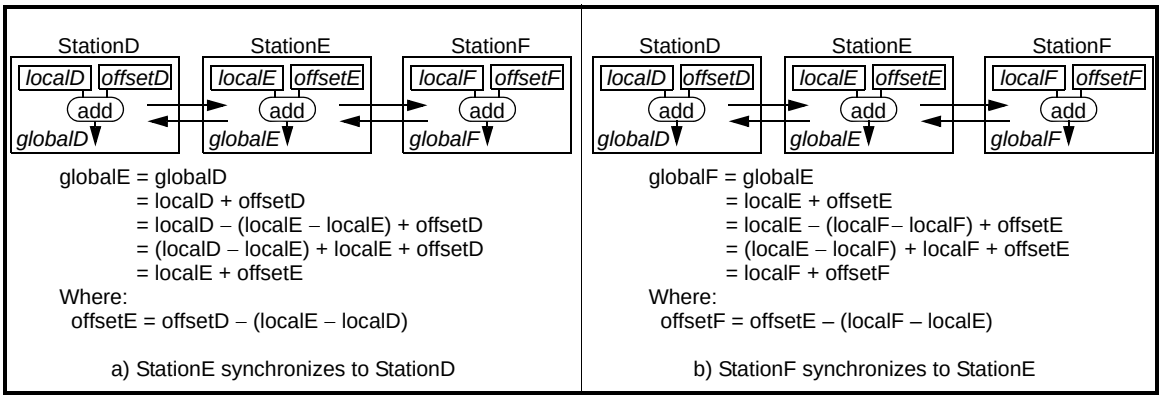


Figure 7.5—Time synchronization principles

The formulation of the *offsetE* value begins the assumption that the *globalE* and *globalD* times are identical. Addition of (*localE–localE*) and regrouping of terms leads to the formulation of the desired *offsetE* value, based on *offsetD* and (*localE–localD*) time difference values, as illustrated in Figure 7.5-a. Synchronization is thus possible using periodic transfers of *offsetD* values and computations of the (*localE–localD*) difference.

The formulation of the *offsetF* value begins the assumption that the *globalF* and *globalE* times are the identical. Addition of (*localF–localF*) and regrouping of terms leads to the formulation of the desired *offsetF* value, based on *offsetE* and (*localF–localE*) time difference values, as illustrated in Figure 7.5-b. Synchronization is thus possible using periodic transfers of *offsetE* values and computations of the (*localF–localE*) difference.

In concept, the *offsetE* value is adjusted first; its adjusted value is then used to compute the desired *offsetF* value. In actuality, the periodic computations of *offsetE* and *offsetF* values are performed concurrently.

7.1.10 Timer snapshot locations

Mandatory jitter-error accuracies are sufficiently loose to allow transmit/receive snapshot circuits to be located with the MAC, as illustrated in Figure 7.6a. Vendors may elect to further reduce timing jitter by latching the receive/transmit times within the PHY, where the uncertain FIFO latencies can be best avoided, as illustrated in Figure 7.6b.

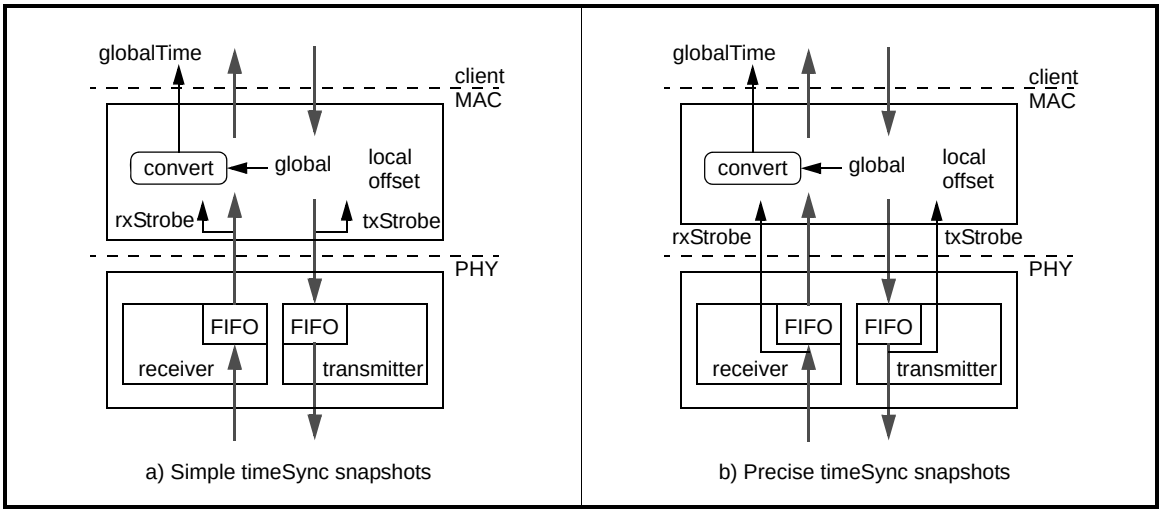


Figure 7.6—Timer snapshot locations

7.1.11 Clock-synchronization updates

7.1.11.1 Clock-synchronization intervals

Clock synchronization involves the processing of periodic events. Three distinct time periods are involved, as listed in Table 7.2. The clock-period events trigger the update of free-running timer values; the period affects the timer-synchronization accuracy and is therefore constrained to be small.

Table 7.2—Clock-synchronization intervals

Name	Time	Description
clock-period	< 20 ns	Time between timer-register value updates
send-period	10 ms	Time between sending of periodic timeSync frames between adjacent stations
slow-period	100 ms	Time between computation of clock-master/clock-slave rate differences

The send-period events trigger the interchange of timeSync frames between adjacent stations. While a smaller period (1 ms or 100 μs) could improve accuracies, the larger value is intended to reduce costs by allowing computations to be executed by inexpensive (but possibly slow) bridge-resident firmware.

The slow-period events trigger the computation of timer-rate differences. The timer-rate differences are computed over two slow-period intervals, but recomputed every slow-period interval. The larger 100 ms (as

opposed to 10 ms) computation interval is intended to reduce errors associated with sampling of clock-period-quantized slow-period-sized time intervals.

7.1.12 Offset synchronization

7.1.12.1 Offset synchronization adjustments

Offset synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.9. Each clock consists of a progressing *timeOfDay* value, whose offset and rate are periodically adjusted. The free-running *flexTimer* timer is never reset; synchronization of stationE (with respect to stationD) is accomplished by adjustments to the *flexOffset* and *flexRate* values within stationE.

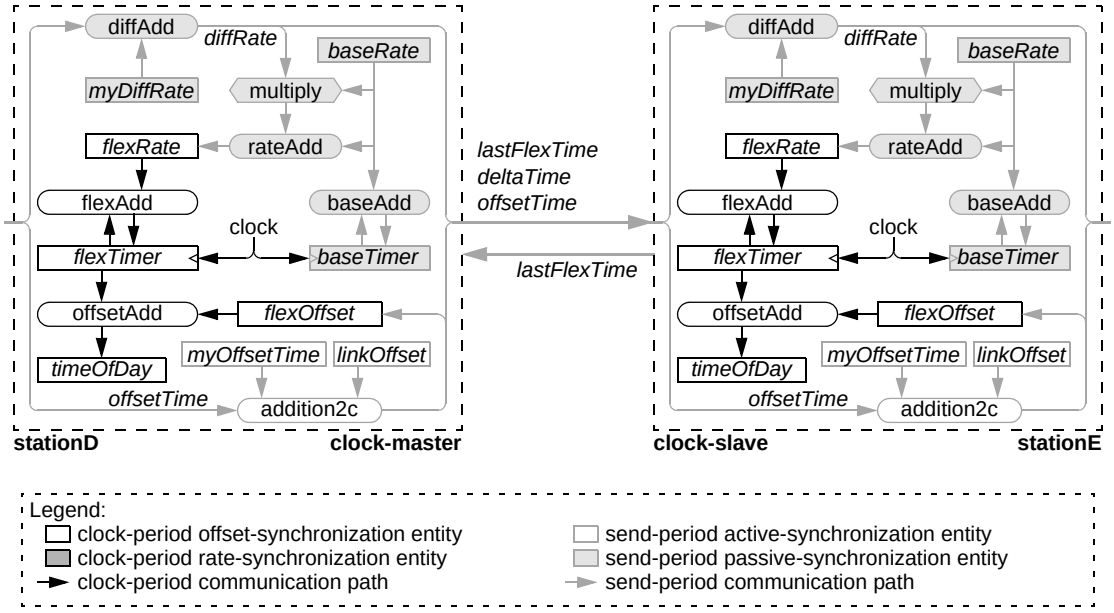


Figure 7.7—Offset synchronization adjustments

The offset-synchronization protocols interchange parameters periodically, possibly every 10 ms. The *lastFlexTime*, *deltaTime*, and *offsetTime* values are sent periodically from the clock-master to the clock-slave. The *lastFlexTime* is sent periodically from the clock-slave to the clock-master, providing information necessary for the clock-master to produce a *deltaTime* value for the clock-slave.

The offset-compensation protocols for stationE adjust its *myOffsetTime* value so that the instantaneous values of *stationE.timeOfDay* and *stationD.timerOfDay* are the same. Computations are performed on clockStrobe reception and clockStrobe transmission.

As an option, an additional *linkOffset* value is available to manually compensate for mismatched transmit-link/receive-link duplex-cable delays on the clock-master side. The *linkOffset* value is expected be manually set when the cable mismatch is known through other mechanisms, such as specialized cable-characterization equipment.

The station's *offsetTime* value is constructed by adding the received *clockStrobe.offsetTime*, local *myOffsetTime*, and local *linkOffset* values. This revised *clockStrobe.offsetTime* value is used within each station and is passed to the downstream neighbor (when such a neighbor is present).

7.1.12.2 Cascaded offsets

The concept of cascaded offset values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.8. The slave-agent in bridgeB is synchronized to its neighbor grand-master via timeSync frames sent on the connecting bidirectional span. Within bridgeB, the clock-slave agent passes the time directly to the clock-master agent. The slave-agent in bridgeC is synchronized to its neighbor clock-master via timeSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the right-most clock-slave station to the left-most grand-master station.

Parameter					
name	grand-master	bridgeB	bridgeC	bridgeD	clock-slave
number	1	2	3	4	5
flexTimer	100	500	-300	200	400
myOffsetTime	10	-400	800	-500	-200
flexOffset	10	-390	410	-90	-290
timeOfDay	110				

Representing:
myOffsetTime[k+1] = flexTimer[k]–flexTimer[k+1];
flexOffset[k+1] = flexOffset[k]+myOffsetTime[k+1];
timeOfDay[k] = flexTimer[k] + flexOffset[k];

Figure 7.8—Cascaded offsets (a possible scenario)

To simplify this illustration, consider only the seconds portion of the *flexTimer* value within each station or bridge. These values may differ dramatically, based (perhaps) on the power-cycling or topology formation sequence. Thus, the grand-master could have a *flexTimer* value of 100 while its bridgeB neighbor has a *flexTimer* value of 500.

The *myOffsetTime* value within bridgeB will converges to the value of –400, representing the differences between grand-master and bridgeB *flexTimer* values. The *flexOffset* value received from the grand-master is added to this *myOffsetTime* value, so that bridgeB's *flexOffset* becomes –390. The *flexTimer* and *flexOffset* values are added, to yield a resultant bridgeB *timeOfDay* value of 110, properly synchronized to the identical grand-master's value.

Similarly, bridgeC is synchronized to bridgeB, bridgeD to bridgeC, and the clock-slave to bridgeD.

7.1.13 Rate synchronization

7.1.13.1 Rate synchronization adjustments

Rate synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.9. The free-running *baseTimer* timer facilitate the determination of rate differences between the clock-master and clock-slave stations.

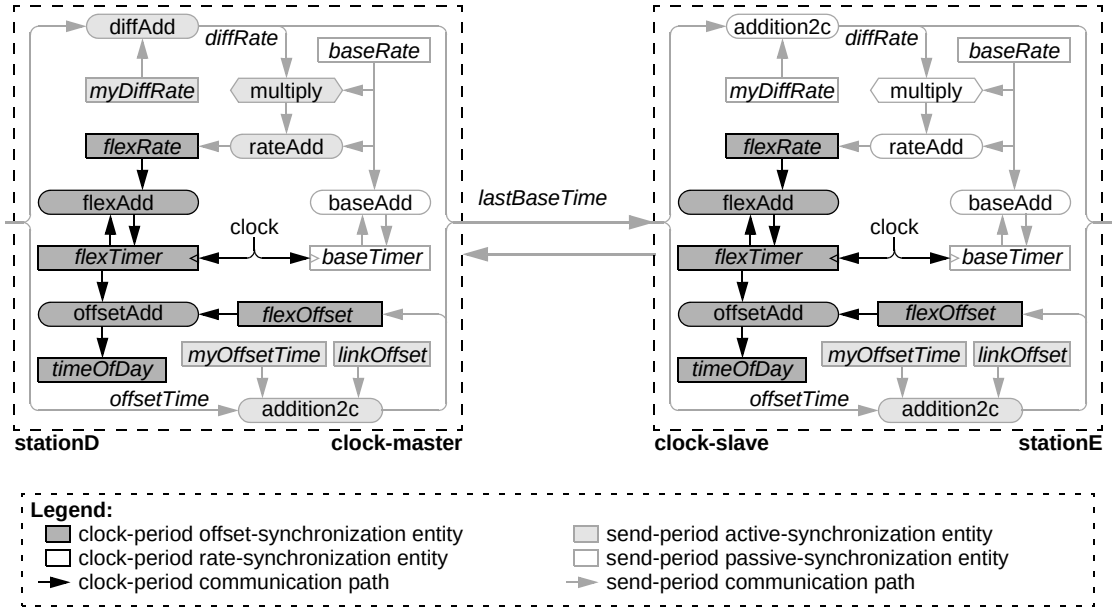


Figure 7.9—Rate synchronization adjustments

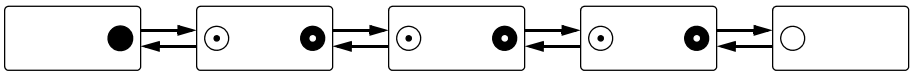
The rate-synchronization protocols interchange parameters periodically, but less frequently than the offset-synchronization protocols, possibly every 100 ms. The *lastBaseTime* value is sent periodically from the clock-master to the clock-slave. Nothing is returned from the clock-slave station.

The rate-compensation protocols for stationE adjust its *myDiffRate* value to accommodate for differences between the *stationD.baseTimer* and *stationE.baseTimer* rates. Computations are performed on clockStrobe reception and clockStrobe transmission.

The station's *diffRate* value is constructed by adding the received *clockStrobe.diffRate* and local *myDiffRate* values. This revised *clockStrobe.diffRate* value is used within each station and is passed to the clock-slave side neighboring station (if present).

7.1.13.2 Cascaded rate differences

The concept of cascaded rate values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.10. Within this figure, the *myDiffRateN* and *diffRateN* represent parts-per-million (PPM) normalized values of *myDiffRate* and *diffRate* respectively.

Parameter					
name	grand-master	bridgeB	bridgeC	bridgeD	clock-slave
number	1	2	3	4	5
crystal deviation	+10 PPM	+100 PPM	–100 PPM	–75 PPM	+75 PPM
myDiffRateN	0 PPM	–90 PPM	200 PPM	–25 PPM	–150 PPM
diffRateN	0 PPM	–90 PPM	110 PPM	+85 PPM	–65 PPM
flexTimer deviation	10 PPM				

Representing:
myDiffRateN[k+1] = flexRate[k]–flexRate[k+1];
diffRate[k+1] = diffRate[k]+myDiffRate[k+1];
flexTimerDeviation[k] = crystalDeviation[k] + diffRate[k];

Figure 7.10—Cascaded rate differences (a possible scenario)

The slave-agent in bridgeB is synchronized to its neighbor grand-master via timeSync frames sent on the connecting bidirectional span. Within bridgeB, the clock-slave agent passes the time directly to the clock-master agent. The slave-agent in bridgeC is synchronized to its neighbor clock-master via timeSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the right-most clock-slave station to the left-most grand-master station.

To simplify this illustration, consider only the parts-per-million (PPM) normalized rate values within each station or bridge. These values may differ significant, based (perhaps) on the nominal value or ambient temperature. Thus, the grand-master could have a crystal deviation of +10 while its bridgeB neighbor has a crystal deviation of +100.

The *myDiffRate* value within bridgeB will converges to the value of –90 PPM, representing the differences between grand-master and bridgeB crystal accuracies. The *diffRate* value received from the grand-master is added to the *myDiffRate* value, so that bridgeB's *diffRate* becomes –90 PPM. The *diffRate* and crystal deviation values are additive, yielding a resultant bridgeB *flexTimer* deviation of 10 PPM, properly synchronized to the identical grand-master's value.

Similarly, the rate of bridgeC is synchronized to bridgeB, bridgeD to bridgeC, and the clock-slave to bridgeD.

7.1.14 Rate-difference effects

If the absence of rate adjustments, significant *timeOfDay* errors can accumulate between send-period updates, as illustrated on the left side of Figure 7.11. The 2 ms deviation is due to the cumulative effect of clock drift, over the 10 ms send-period interval, assuming clock-master and clock-slave crystal deviations of –100 PPM and +100 PPM respectively.

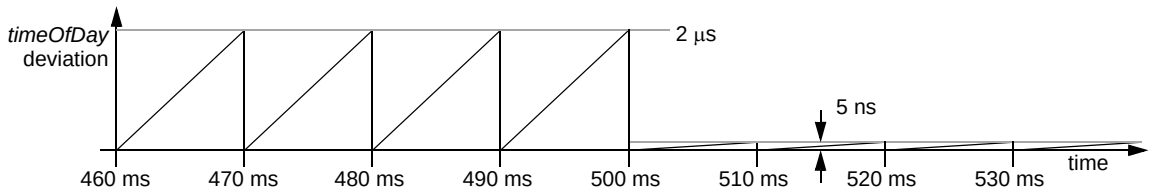


Figure 7.11—Rate-adjustment effects

While this regular sawtooth is illustrated as a highly regular (and thus perhaps easily filtered) function, irregularities could be introduced by changes in the relative ordering of clock-master and clock-slave transmissions, or transmission delays invoked by asynchronous frame transmissions. Tracking peaks/valleys or filtering such irregular functions are thought unlikely to yield similar *timeOfDay* deviation reductions.

The differences in rates could easily be reduced to less than 1 PPM, assuming a 200 ms measurement interval (based on a 100 ms slow-period interval) and a 100 ns arrival/departure sampling error. A clock-rate adjustment at time 100 ms could thus reduce the clock-drift related errors to less than 5 ns. At this point, the timer-offset measurement errors (not clock-drift induced errors) dominate the clock-synchronization error contributions.

7.1.15 *baseTimer* functionality

The external formats within timeSync frames assumes the presence of *baseTimer*-related values. A direct-mapped hardware implementation involves a clocked *baseTimer* register and a precision adder, as illustrated in Figure 7.12a. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

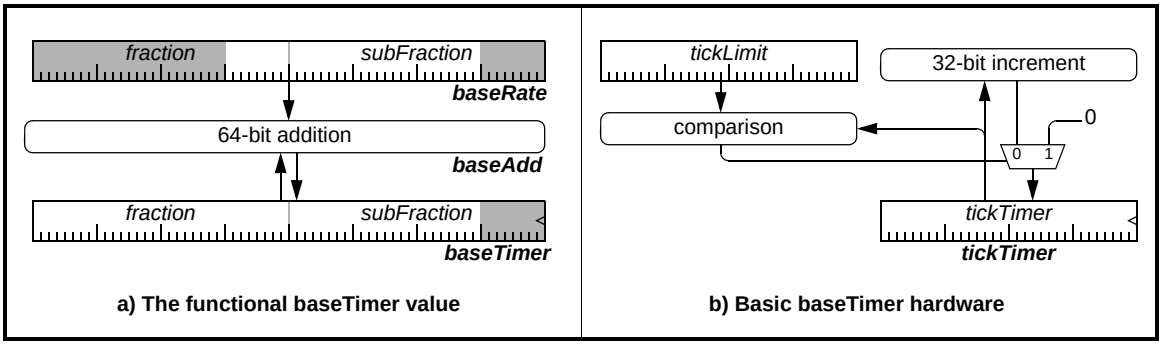


Figure 7.12—*baseTimer* implementation examples

A simpler hardware implementations is to periodically increment a tick timer at an implementation-dependent rate, as illustrated in Figure 7.12a. Although this timer is improperly scaled and insufficiently large, firmware can perform the multiplications and additions necessary to provide the normalized external values.

7.1.16 *flexTimer* functionality

The selection of the best time-of-day format is oftentimes complicated by the desire to equate the clock format granularity with the granularity of the implementation's 'natural' clock frequency. Unfortunately, the 'natural' frequency within a multimodal {1394, 802-100Mb/s, 802.3 1Gb/s} implementation is uncertain, and may vary based on vendors and/or implementation technologies.

The difficulties of selecting a 'natural' clock-frequency can be avoided by realizing that any clock with sufficiently fine resolution is acceptable. Flexibility involves using the most-convenient clock-tick value, but adjusting the timer advance rate associated with each clock-tick occurrence.

The same mechanism easily supports both near-arbitrary clocking rates and fine-grained rate-adjustments, needed to support timer-synchronization protocols, as illustrated in Figure 7.13. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

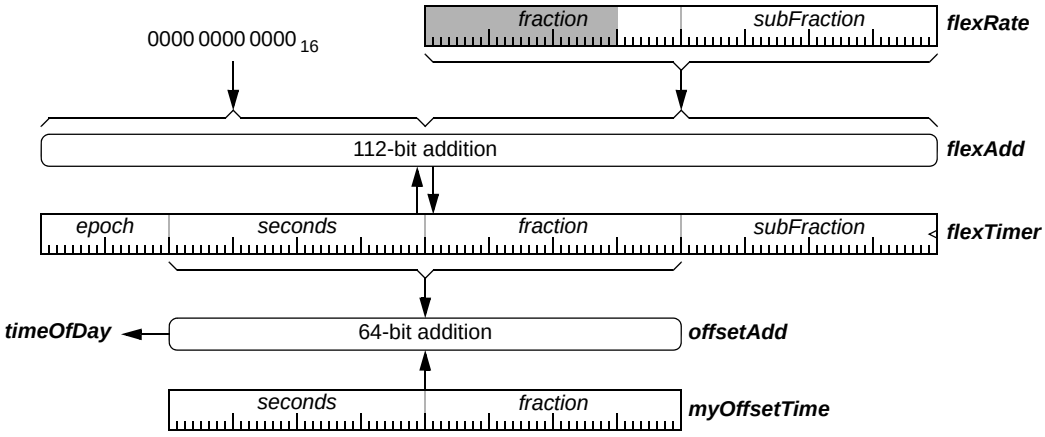


Figure 7.13—*flexTimer* implementation example

This illustration is not intended to constrain implementations, but to illustrate how the system's clock and timer formats can be optimized independently. This allows the *timeOfDay* timer format to be based on arithmetic convenience, timing precision, and years-before-overflow characteristics (see Annex E).

Although a direct hardware implementation is possible, a simpler solution is to utilize a single *tickTimer* register (see 7.1.15). Since the external communication rates are low, firmware conversions between *tickTimer* and *flexTimer* values are expected to be feasible.

7.2 Terminology and variables

7.2.1 Common state machine definitions

The following state machine inputs are used multiple times within this clause.

NULL
: Indicates the absence of a value and (by design) cannot be confused with a valid value.

queue values
: Enumerated values used to specify shared queue structures.
  - Q_RX_FRAME—Identifies the queue that supplies received frames.
  - Q_RX_SYNC—Identifies the queue where received timeSync frames are saved.
  - Q_RX_ELSE—Identifies the queue where received non-timeSync frames are saved.
  - Q_TX_FRAME—The identifier associated with the transmitted timeSync frames.

SCALE
: A multiplicative scalar for converting between the internal *core.diffRate* fraction and the external *frame.diffRate* integer values.
  Value: 2^{44}

7.2.2 Common state machine variables

One instance of each variable specified in this clause exists in each port, unless otherwise noted.

currentTime
: A value representing the current time.

core
: The state associated with the common portion of a bridge:
  - *clockDeviation*—Nominal frequency deviation of crystal-stabilized clock, in PPM.
    Maximim: +100
    Minimum: –100
  - *clockFrequency*—The nominal frequency of the synchronized timer.
    clockFrequency > 50 MHz
  - *diffRate*—The cumulative rate difference from the grand-master station.
  - *flexOffset*—The cumulative value difference from the grand-master station.
  - *linkOffset*—The specified delay differences between transmit and receive links.
  - *myDiffRate*—The measured rate difference between the local and remote ports.
  - *myOffsetTime*—The measured time-offset difference between the local and remote ports.
  - *counter*—A running count that is incremented approximately once every 10 ms.
  - *offsetTicks*—A *tickTimer* snapshot taken at the start of each 10 ms interval.
  - *precedence*—The currently observed grand-master precedence value.
  - *rateCount*—A running count that is incremented approximately once every 100 ms.
  - *slaveCount*—A snapshot of *offsetCount*, received from a grand-master slave port.
  - *slavePort*—The slave port identifier associated with an upstream grand-master station.
  - *precedence*—The most-significant portion of this station’s grand-master precedence.
  - *tickExtra*—The software-maintained seconds-overflow value associated with *core.tickTimer.*
  - *tickLimit*—The maximum value of *core.tickTimer* before a seconds overflow occurs.
  - *tickOffset*—A scalar parameter for the conversion between *tickTimer* and *flexTimer* values.
  - *tickRate*—A scalar parameter for the conversion between *tickTimer* and *flexTimer* values.
  - *tickTime*—A snapshot of the *currentTime* value at the beginning of each *clockPeriod* interval.
  - *tickTimer*—A running count that is incremented at the end of each *clockPeriod* interval.
  - *uniqueID*—The more-significant portion of this station’s grand-master precedence

frame
The contents of a timeSync frame, including the following components (see 6.1.1):
da, *sa*, *protocolType*, *subType*, *hopsCount*, *syncCount*, *cycleCount*, *precedence*, *uniqueID*, *lastFlexTime*, *deltaTime*, *offsetTime*, *diffRate*, *lastBaseTime*, *fcs*.
See 6.1.1.
rxtickTimer—The value of *core.tickTimer*, sampled when this frame was received.
port
The state associated with each of the ports on a bridge:
counter—An accounting value whose comparison to *core.counter* triggers processing.
portPri—A priority differentiator for the port.
portID—A unique identifier for the port.
propTime—An average cable propagation delay measurement for the attached link.
rateCount—An accounting value whose comparison to *core.rateCount* triggers processing.
rxFlexTime0—The current value of *flexTime* when the frame is received.
rxFlexTime—The current of *rxFlexTime0* when the frame is received.
rxTickTime0—The current value of *tickTime* when the frame is received.
rxTickTime —The current value of *rxTickTime0* when the frame is received.
syncCount—The last observed *frame.syncCount* value, saved for consistency checks.
timeSyncAllowed—Indicates when a timeSync frame is enabled for transmission.
txtickTimer—The saved *core.tickTimer* value, when timeSync was last transmitted.
tickTimer
See 7.2.2.

7.2.3 Common state machine routines

Best(*test*, *base*)
Indicates whether the *test* is smaller than the *base* precedence, as defined by Equation 7.1.

```
(test.hi < base.hi || (test.hi == base.hi && test.lo <= base.lo))
```
(7.1)

Dequeue(*queue*)
Returns the next available frame from the specified queue.
frame—The next available frame.
NULL—No frame available.
Enqueue(*queue*, *frame*)
Places the frame at the tail of the specified queue.
FlexTimer(*tickTime*, *tickRate*, *tickOffset*)
Computes the effective *flexTimer* value based on the provided inputs, as defined by Equation xx.

```
(tickTime * tickRate * (1.0 + tickDiff) + tickOffset)
```
(7.2)

Min(*value1*, *value2*)
Returns the numerically smaller of two values.
Precedence(*pre*, *uid*, *hops*, *port*)
Forms a 88-bit precedence value from its component fields, as defined by Equation 7.1.

```
Precedence(pre, uid, hops, port)
{
    doubleInt value;

    value.hi = (sys << 16) | (uid >> 48);
    value.lo = (uid << 16) | (hops << 8) | port;
    return(value);
}
```
(7.3)

QueueEmpty(*queue*)

Indicates when the queue has emptied.

TRUE—The queue has emptied.

FALSE—(Otherwise.)

7.3 Clock synchronization state machines

7.3.1 TickTimer state machine

7.3.1.1 TickTimer state machine definitions

The following definitions are used within this subclause:

NULL
: A constant that indicates the absence of a slave-port identifier.

TICKS_PER_SEC
: The nominal number of tick periods in every cycle.

7.3.1.2 TickTimer state machine variables

Only one instance of each state-machine variable exists in each bridge.

core

currentTime
: See 7.2.2.

precedence
: A computed grand-master precedence value, based on this station's parameters.

7.3.1.3 TickTimer state machine routines

The following routines are used within this subclause:

Best(*test*, *base*)

Precedence(*pre*, *uid*, *hops*, *port*)
: See 7.2.3.

7.3.1.4 TickTimer state table

Each bridge supplies only one TickTimer state machine.

The TickTimer state machine provides the timer values for other state machines, as specified in Table 7.3. Only the START-1, BUMP-1, and BUMP-2 rows (shaded gray) are expected to require hardware support; the remaining rows are expected to be easily mapped to firmware. The notation used in the state table is described in 3.4.

Table 7.3—TickTimer state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	delta = core.clockDeviation / 1000000, (currentTime – core.tickTime) >= 1.0 / (core.clockFrequency * (1.0 + delta))	1	core.tickTime = currentTime;	BUMP

	(core.tickTimer – core.offsetTicks) >= 0.010 * TICKS_PER_SEC	2	core.offsetTicks = core.tickTimer;
			core.counter += 1;
			preference = Precedence(core.precedence, core.uniqueID, 0, 0);
				PLUS

	—	3	—	START
BUMP	core.tickTimer >= core.tickLimit	1	core.tickTimer = 0;	START
	—	2	core.tickTimer += 1;	
PLUS	core.tickTimer < core.lastTimer	1	core.tickExtra += core.tickLimit + 1; core.lastTimer = core.tickTimer;	NEXT
	—	2	—	
NEXT	(core.counter – core.slaveCount) >= 5	1	core.slavePort = NULL;	NEAR
	—	2	—	

NEAR	Best(preference, core.precedence) || core.slavePort == NULL	1	core.slavePort = NULL;
			core.precedence = precedence;
			core.slaveCount = core.counter;
				FINAL

	—	2	—	

Row START-1: The *tickTimer* register is incremented once every *clockPeriod* interval.
(The *tickTimer* register is the timer used to snapshot all arrival and departure times.)
Row START-2: The *counter* register is incremented approximately once every 10 ms interval.
(Changes in the *counter* register triggers transmissions of periodic *timerSync* frames.)
Row START-3: Otherwise, no updates are performed.

Row BUMP-1: The *tickTimer* register overflows after the core.tickLimit value is reached.
Row BUMP-2: Otherwise, the *tickTimer* register is incremented.

Row PLUS-1: The *tickTimer* register seconds-carry eventually propagates into the *core.tickExtra* field.
Row PLUS-2: Otherwise, no seconds-carry is required.

Row NEXT-1: If the slave port receives no heartbeats, the slave-port identifier is released.
Row NEXT-2: Otherwise, no updates are performed.

Row NEAR-1: If this station has the best preference, or no slave port exists, this station has precedence.
Row NEAR-2: Otherwise, the cumulative preference value remains unchanged.

7.3.2 TimerRxLatch state machine

7.3.2.1 TimerRxLatch state machine definitions

The following definitions are used within this subclause:

Q_RX_ELSE
Q_RX_FRAME
Q_RX_SYNC
See 7.2.1.

7.3.2.2 TimerRxLatch state machine variables

The following variables are used within this subclause:

core
frame
See 7.2.2.

7.3.2.3 TimerRxLatch state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)
Enqueue(*queue*, *frame*)
See 7.2.3.

7.3.2.4 TimerRxLatch state table

The TimerRxLatch state machine associates a time stamp within incoming frames, as specified in Table 7.4. The notation used in the state table is described in 3.4.

Table 7.4—TimerRxLatch state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_RX_FRAME)) != NULL	1	—	FINAL
	—	2	—	START
FINAL	frame.protocolType == TIME_SYNC	1	frame.rxTickTimer = core.tickTimer; Enqueue(Q_RX_SYNC, frame);	START
	—	2	Enqueue(Q_RX_ELSE, frame);	

Row START-1: A new frame has arrived and is available for processing.
Row START-2: Wait for a new frame to arrive.

Row FINAL-1: The timeSync frames are immediately time-stamped and enqueued for special processing.
Row FINAL-2: Other frames are processed in their normal fashion.

7.3.3 TimerTxLatch state machine

7.3.4 TimerRxLatch state machine

7.3.4.1 TimerRxLatch state machine definitions

The following definitions are used within this subclause:

Q_TX_FRAME
Q_TX_SYNC
See 7.2.1.

7.3.4.2 TimerRxLatch state machine variables

The following variables are used within this subclause:

core
frame
port
See 7.2.2.

7.3.4.3 TimerTxLatch state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)
Enqueue(*queue*, *frame*)
QueueEmpty(*queue*)
See 7.2.3.

7.3.4.4 TimerTxLatch state table

The TimerTxLatch state machine associates a time stamp within incoming frames, as specified in Table 7.5. The notation used in the state table is described in 3.4.

Table 7.5—TimerTxLatch state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(QueueEmpty(Q_TX_FRAME)) && port.timeSyncAllowed	1	—	FINAL
	—	2	—	START
FINAL	(frame = Dequeue(Q_TX_SYNC)) != NULL	1	port.txTickTimer = core.tickTimer; Enqueue(Q_TX_FRAME, frame);	START
	—	2	—	

Row START-1: A new frame has arrived and is available for processing.
Row START-2: Wait for a new frame to arrive.

Row FINAL-1: The timeSync frames are immediately time-stamped and saved for special processing.
Row FINAL-2: Other frames are processed in their normal fashion.

7.3.5 TimerRxCompute state machine

7.3.5.1 TimerRxCompute state machine definitions

The following definitions are used within this subclause:

Q_TX_SYNC
: See 7.2.1.

7.3.5.2 TimerRxCompute state machine variables

The following variables are used within this subclause:

core
: See 7.2.2.

count
: A saved copy of the last received frame's *syncCount* field.

diffRate
: The copy of the *core*.*diffRate* value.

diffRate0
: The value of *core*.*diffRate*, before the decay computation is performed.

diffRate1
: The value of *core*.*diffRate*, after the decay computation is performed.

formed
: A computed grand-master precedence value, based on the frame-supplied parameters.

frame
: See 7.2.2.

myTickDelta
: The difference between *tickTime* arrival times within sampled frames.

pastCount
: A saved copy of the previous *count* value.

port
: See 7.2.2.

rxBaseDelta
: The difference between *lastBaseTime* values within sampled frames.

rxBaseTime1
: The *lastBaseTime* value observed in a previously sampled frame.

rxPrecedence
: A copy of the grand-master precedence from the last received frame.

rxTickDelta
: A temporary value representing time-snapshot differences on the receiving link.

rxTickTime1
: A *tickTime* value observed in a previously sampled frame.

tickTimer
: A copy of the frame-supplied *rxtickTimer* field.

txDeltaTime
: A temporary value representing time-snapshot differences on the transmitting link.

7.3.5.3 TimerRxCompute state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)
FlexTime(*tickTime, tickRate, tickDiff, tickOffset*)
See 7.2.3.

7.3.5.4 TimerRxCompute state table

The TimerRxCompute state machine processes received timeSync frames while participating in the grand-master selection protocol, as specified in Table 7.6. The notation used in the state table is described in 3.4.

Table 7.6—TimerRxCompute state table

Current		Row	Next	
state	**condition**		**action**	**state**

START	(frame = Dequeue(Q_RX_SYNC)) != NULL && port.timeSyncAllowed	1	pastCount = port.syncCount;
			port.syncCount = count = frame.syncCount;
			tickTime = frame.rxTickTimer + core.tickExtra;
				TIME

	—	2	—	START
TIME	frame.rxTickTimer < core.lastTimer	1	tickTime += core.tickLimit;	PREP
	—	2	—	

PREP	—	1	port.rxFlexTime = port.rxFlexTime0;
			port.rxFlexTime0 = FlexTime(tickTimer, core.tickRate, core.diffRate, core.tickOffset);
			
			rxTickTime = port.rxTickTime0;
			port.rxTickTime0 = tickTimer;
				FIRST

FIRST	count != (pastCount + 1) % 256;	1	—	FINAL

	—	2	port.rxDeltaTime = port.rxFlexTime1 – frame.lastFlexTime;
			txDeltaTime = frame.deltaTime;
			
			port.propTime = (txDeltaTime + port.rxDeltaTime)/2;
			
			rxPrecedence = Precedence(frame.precedence, frame.uniqueID, frame.hopsCount, port.portID);
				FAST

Table 7.6—TimerRxCompute state table

Current		Row	Next	
state	**condition**		**action**	**state**
FAST	**!**Best(rxPrecedence, core.precedence) && core.slavePort != port.portID	1	—	START

	—	2	core.slavePort = port.portID;
			core.precedence = core.compare;
			
			core.myOffsetTime =
			(port.propTime – port.rxDeltaTime);
			core.flexOffset = core.linkOffset +
			frame.offsetTime + core.myOffsetTime;
				NEAR

SLOW	port.counter >= core.counter + 50	1	—	FINAL
	port.counter >= core.counter + 10	2	rxBaseDelta = (frame.lastBaseTime – port.rxBaseTime1); rxTickDelta = (rxTickTime = port.rxTickTime1); myTickDelta = FlexRate(rxTickTime, core.baseRate, 0); diffRate0 = core.myDiffRate; diffRate = (rxTickDelta – myTickDelta) / myTickDelta; core.myDiffRate = diffRate = Clip(diffRate, 1.0 / 4096); diffRate1 = diffRate + (frame.diffRate / SCALE); core.diffRate = diffRate1 = Clip(diffRate1, 1.0 / 4096); core.tickOffset –= FlexTime(core.tickTimer, core.tickRate, diffRate1 – diffRate0, 0);	
	—	3	—	START

FINAL	—	1	port.counter = core.counter;
			
			port.rxBaseTime1 = frame.lastBaseTime;
			port.rxTickTime1 = rxTickTime;
				START

Row START-1: A new frame has arrived for processing.
Save the frame's *syncCount* field, to facilitate detection of missing timeSync frames.
Row START-2: Wait for a new timeSync frame to arrive.

Row TIME-1: If the sampled time has overflowed, an additional overflow amount is added.
Row TIME-2: Otherwise, only the cumulative overflow amount is added.

Row PREP-1: If the sampled time has overflowed, an additional overflow amount is added.
Save the current *flexTime* value, to facilitate computation of:
myOffsetTime—The time difference between neighbor and local *flexTimer* values.

linkOffset—The time difference attributed to link propagation delay.
Save the current *tickTime* value, to facilitate computation of *myDiffRate* values

Row FIRST-1: Discard non-consecutive timeSync frames, since previously saved values are incorrect.
Row FIRST-2: Process consecutive timeSync frames to maintain time synchronization:
Compute *rxDeltaTime* and *txDeltaTime* values.
Compute the cable propagation delays, based on *rxDeltaTime* and *txDeltaTime* values.
Compute the grand-master precedence of the frame, based on frame-supplied values.

Row FAST-1: Skip further processing if this is the primary clock-slave receive port, based on:
The frame-supplied precedence is less than this station's observed precedence.
This port has not been identified as the clock-slave port.
Row FAST-2: The clock-slave port updates appropriate clock-synchronization values.
This port is identified as the clock-slave port and the frame's precedence is saved.
The residual rate differential (if any) decays towards zero with an approximate 1-second time constant.
The core's *flexTimer* offset values are computed and updated.

Row SLOW-1: An excessive lower-rate interval is processed as an error.
Row SLOW-2: Further computations are performed at the lower-rate interval.
The current lower-rate interval index is saved, so that processing only occurs once per interval.
The neighbor's *baseTime* difference is computed.
The station's *baseTime* difference is computed and normalized to the interchange format.
Compute the local (from the neighbor) and cumulative (from the grand-master) rate differences.
This station's timer is adjusted to operate at the observed rate of the grand master.
Row SLOW-3: Otherwise, no rate differences are calculated.

Row FINAL-1: The lower-rate interval is concluded.
Update the counter to facilitate measurement of the next lower-rate interval duration.
Save the observed time snapshot values.

7.3.6 TimerTxCompute state machine

7.3.6.1 TimerTxCompute state machine definitions

The following definitions are used within this subclause:

NULL
: A constant that indicates the absence of a slave-port identifier.

Q_TX_FRAME
: See 7.2.1.

7.3.6.2 TimerTxCompute state machine variables

The following variables are used within this subclause:

core

diffRate0
: The value of core.diffRate, before the decay computation is performed.

diffRate1
: The value of core.diffRate, after the decay computation is performed.

frame

port
: See 7.2.2.

tickTime
: A extended version of core.timerTicks that accounts for per-second overflows.

7.3.6.3 TimerTxCompute state machine routines

The following routines are used within this subclause:

Enqueue(*queue, frame*)

FlexTime(*tickTime, tickRate, tickDiff, tickOffset*)

QueueEmpty(*queue*)
: See 7.2.3.

7.3.6.4 TimerTxCompute state table

The TimerTxCompute state machine provides the arguments for the timeSync frame, as specified in Table 7.7. The notation used in the state table is described in 3.4.

Table 7.7—TimerTxCompute state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(QueueEmpty(Q_TX_FRAME)) && port.counter != core.counter	1	port.counter = core.counter; tickTime = port.txTickTime +core.tickExtra;	TIME
	—	2	—	START
TIME	port.tickTimer < core.lastTimer	1	tickTime += core.tickLimit;	PREP
	—	2	—	

PREP	core.slavePort == NULL	1	diffRate0 = core.diffRate;
			diffRate1 = diffRate0 – (diffRate0 / 256);
			core.diffRate = diffRate1;
			
			core.tickOffset –=
			FlexTime(tickTime, core.tickRate,
			newDiffRate – oldDiffRate, 0);
			
			frame.offsetTime = core.myOffsetTime;
				FINAL

	—	2	frame.offsetTime = core.flexOffset;	

FINAL	—	1	frame.lastFlexTime =
			FlexTime(tickTime, core.tickRate,
			core.diffRate, core.tickOffset);
			
			frame.deltaTime = port.rxDeltaTime;
			
			frame.diffRate = core.diffRate * SCALE;
			
			frame.lastBaseTime =
			FlexTime(tickTime, core.tickRate, 0, 0);
			Enqueue(Q_TX_FRAME, frame)
				SEND

Row START-1: When space is available and the time has arrived, transmit the next timeSync frame.
Row START-2: Wait until the next timeSync transmission time.

Row TIME-1: If the sampled time has overflowed, an additional overflow amount is added.
Row TIME-2: Otherwise, only the cumulative overflow amount is added.

Row PREP-1: The grand-master station processing is distinct.
The inherited rate differences decays towards zero.
Adjust the *tickOffset* value to compensate for rate-change induced offsets.
Transmit an *offsetTime* value that corresponds to the core's present value.
Row PREP-2: Transmit an *offsetTime* value that corresponds to the cumulative value.

Row FINAL-1: When space is available and the time has arrived, transmit the next timeSync frame.
The frame's *lastFlexFrame* field is based on the previously sampled *tickTime* value.
The frame's *deltaTime* field is based on the port's receiver-computed time-difference value.
The frame's *diffRate* field is based on the receiver-computed rate-difference value.
The frame's *lastBaseTime* field is based on the previously sampled *tickTime* value.

7.4 Protocol comparison

A comparison of the Residential Ethernet Study Group (RE-SG) and IEEE 1588 proposals is summarized in Table 7.8.

Table 7.8—Protocol comparison

Properties		Row	Descriptinos	
state			RE-SG	1588
timeSync MTU <= Ethernet MTU		1	yes	
No cascaded PLL whiplash		2	yes	
Number of frame types		3	1	> 1
Phaseless initialization sequencing		4	yes	no
Topology		5	duplex links	general
Grand-master precedence parameters		6	spanning-tree like	special
Rogue-frame settling time, per hop		7	10 ms	1 s
Arithmetic complexity	numbers	8	64-bit binary	2 x 32-bit binary
	negatives	9	2's complement	signed
Master transfer discontinuities	rate	10	gradual change	
	offset limitations	11	duplex-cable match sampling error	
Firmware friendly	no delay constraints	12	yes	
	n-1 cycle sampling	13	yes	
Time-of-day value precision	offset resolution	14	233 ps	
	overflow interval	15	136 years	

Row 1: The size of a timeSync frame should be no larger than an Ethernet MTU, to minimize overhead.
RE-SG: The size of a timeSync frame is an Ethernet MTU.
1588: The size of a timeSync frame is (to be provided).

Row 2: Cascaded phase-lock loops (PLLs) can yield undesirable whiplash responses to transients.
RE-SG: There are no cascaded phase-lock loops.
1588: There are multiple initialization phases (to be provided).

Row 3: There number of frame types should be small, to reduce decoding and processing complexities.
RE-SG: Only one form of timeSync frame is used.
1588: Multiple forms of timeSync frames are used (to be provided).

Row 4: Multiple initialization phases adds complexity, since miss-synchronized phases must be managed.
RE-SG: There are no distinct initialization phases.
1588: There are multiple initialization phases (to be provided).

Row 5: Arbitrary interconnect topologies should be supported.
RE-SG: Topologies are constrained to point-to-point full-duplex cabling.
1588: Supported topologies include broadcast interconnects.

Row 6: Grand-master selection precedence should be software configurable, like spanning-tree parameters.
RE-SG: Grand-master selection parameters are based on spanning-tree parameter formats.
1588: Grand-master selection parameters are (to be provided).

Row 7: The lifetime of rogue frames should be minimized, to avoid long initialization sequences.
RE-SG: Rogue frame lifetimes are limited by the 10 ms per-hop update latencies.
1588: Rogue frame lifetimes are limited by (to be provided).

Row 8: The time-of-day formats should be convenient for hardware/firmware processing.
RE-SG: The time-of-day format is a 64-bit binary number.
1588: The time-of-day format is a (to be provided).

Row 9: The time-of-day negative-number formats should be convenient for hardware/firmware processing.
RE-SG: The time-of-day format is a 2's complement binary number.
1588: The time-of-day format is a (to be provided).

Row 10: The rate discontinuities caused by grand-master selection changes should be minimal.
RE-SG: Smooth rate-change transitions with a 2.5 second time constant is provided.
1588: (To be provided).

Row 11: The time-of-day discontinuities caused by grand-master selection changes should be minimal.
RE-SG: Maximum time-of-day errors are limited by cable-length asymmetry and time-snapshot errors.
1588: (To be provided).

Row 12: Firmware friendly designs should not rely on fast response-time processing.
RE-SG: Response processing time have no significant effect on time-synchronization accuracies.
1588: (To be provided).

Row 13: Firmware friendly designs should not rely on immediate or precomputed snapshot times.
RE-SG: Snapshot times are never used within the current cycle, but saved for next-cycle transmission.
1588: (To be provided).

Row 14: The fine-grained time-of-day resolution should be small, to faciliate accurate synchronization.
RE-SG: The 64-bit time-of-day timer resolution is 233 ps, less than expected snapshot accuracies.
1588: (To be provided).

Row 15: The time-of-day extent should be suffiently large to avoid overflows within one's lifetime.
RE-SG: The 64-bit time-of-day timer overflows once every 136 years.
1588: (To be provided).

8. Subscription state machines

NOTE—This clause has been removed.
The reader should track presentations by Felix Feng.

Subscription state machines are responsible for performing talker-agent and listener-agent duties.

9. Endpoint shaping state machines (proposal 1)

NOTE—Multiple bunch-avoiding pacing protocols are presented for consideration:
a) Clause 9 (this clause) presents an end-point shaper model, with no bridge shaping assumed.
b) Clause 10 presents a pseudo-synchronous transmission model.
c) Clause 11 presents a cross-flow shaper transmission model, an optional bridge Clause 9 enhancement.

9.1 Rate-based scheduling overview

The clause describes a rate-based scheduling technique. The rate-based scheduling concepts are similar to those within rate monotonic scheduling protocols, commonly used within real-time systems. Objectives associated with time-sensitive forwarding alternatives include the following:

a) Multiple time-sensitive transmission rates are supported, including:
   1) Highest rate 8 kHz traffic, such as the traffic generated by simple bridges between RE and existing IEEE 1394[B6] A/V devices.
   2) Lower rate traffic, such as voice over internet protocol (VOIP) traffic, without forcing this traffic to be reblocked into smaller (and therefore less efficient) frame sizes.

b) Frame forwarding should not be dependent on successful time-of-day synchronization between the bridge and adjacent stations. Frame forwarding should succeed before the grand clock-master station has been selected, or when the selected grand-master clock station changes.

c) Frame-forwarding protocols should leverage existing bridge queue and service models, although specification of abstract rate shaper details is expected.

d) Each traffic class has guaranteed latency, when the cumulative traffic is constrained to less than the link capacity. The latency guarantee is approximately an MTU more than an inter-arrival period.

Rate-based scheduling involves associating a priority with frame transmissions, where the priority is a monotonic function of the frame transmission frequency, as illustrated in Figure 11.1.

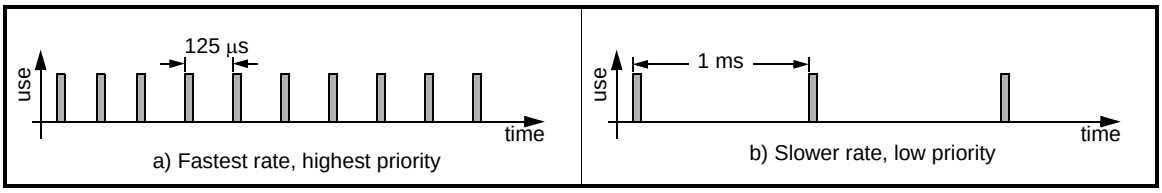


Figure 9.1—Rate-based priorities

An application could desire to send traffic once in every 4 ms interval, while attempting to constrain the worst-case latency to a smaller 1 ms value, as illustrated in Figure 11.2a. This would be done by subscribing for a 1 ms interval, while only sending the traffic in larger 4 ms intervals, as illustrated in Figure 11.2b. The lower latency is not achieved without costs: the effective subscription bandwidth allocated to this application is 4 times larger than necessary.

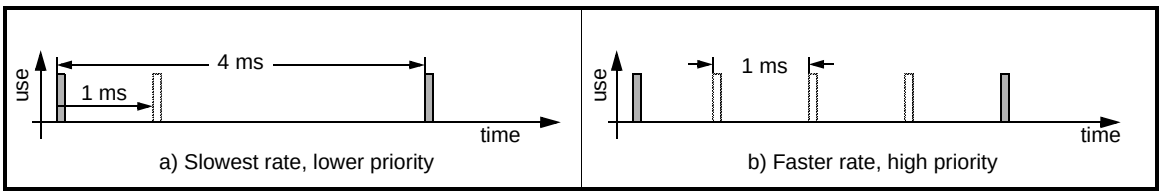


Figure 9.2—Rate-based priorities

9.1.1 Rate-based priorities

Quality of service is based on the availability of user_priority field parameter associated within transmitted time-sensitive frames, as listed in Table 11.1.

Table 9.1—Tagged priority values

Code	Interval (ms)	Name	Description
0	n/a	CLASS_C	Best effort, with minimal guaranteed BW
1	n/a	CLASS_B	Preferred, with minimal guaranteed BW
2-3	—	—	Used for other purposes
4	1.0	CLASS_A1	Guaranteed BW over short interval
5	0.125	CLASS_A0	Guaranteed BW over shorter interval
6-7	—	—	Network management

9.1.2 Transmit shaping and pacing

Transmit ports utilize a shaper and pacer. The purpose of these is to ensure forward progress of best-effort control traffic. In concept, this involves a two-step bandwidth partitioning mechanism:

a) The shaperA limits the cumulative classA and primary classB traffic to 75% of the link bandwidth. The intent is to ensure that 25% residual bandwidth remains available for lower-class traffic.

b) The pacerB partitions the residual 25% traffic equally between classB and classC traffic.
This ensures that classB traffic is never starved, in the presence of 75% classA traffic.
This ensures that classC traffic is not starved, in the presence of excess classB traffic.

9.1.3 Credit-based shapers and pacers

9.1.3.1 Credit-history shapers

The transmit shaper that limits levels of classA traffic (see Table 11.3, transitions BEST-1 to BEST-3) selectively enables transmissions based on an accumulated credits value. The shaper's credits are adjusted down or up, as illustrated in Figure 11.8. When frames are transmitted, the decrement value represents the size of a transmitted frame. Between transmissions, the credits increase based on the maximum allowed transmission rate.

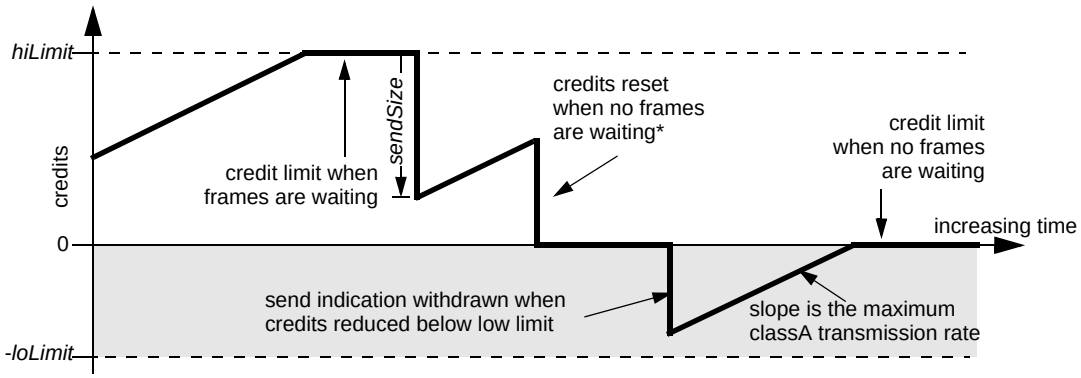


Figure 9.3—Credit-based shapers

Crossing below the zero threshold generates a rate-limiting indication, so that offered traffic can stop. By design, the credit value never goes below the *–loLimit* extreme. To bound the burst traffic after inactivity intervals, when no frames are ready for transmission, credits are reduced to zero (if currently higher than zero) and can accumulate to no more than the zero-value limit.

The *hiLimit* threshold limits the positive credits, to avoid overflow. When frames are ready for transmission (and are being blocked by transit traffic), credits can accumulate to no more than this *hiLimit* value.

In concept, the shaper consist of a token bucket. The number of credits in a token bucket is decremented by the size of each transmitted frame. The credits in the token bucket are continually increasing over time. A frame is only transmitted when the credits are positive.

9.1.3.2 Credit-based pacers

Although multiple pacers are specified within this working paper, the behavior of most pacers can be characterized by a common algorithm and instance-specific parameters (as done within RPR[B5]). The pacer's credits are adjusted down or up, as illustrated in Figure 11.9. The decrement and increment values typically represent sizes of debit and credit frames in each update interval, respectively.

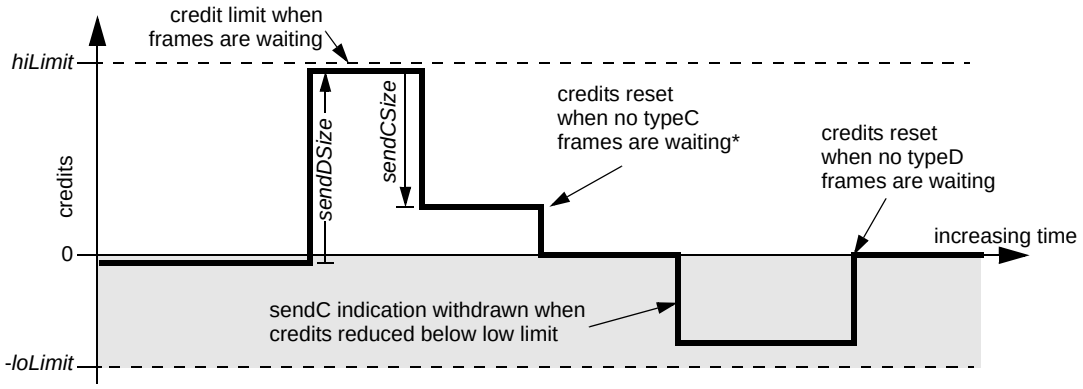


Figure 9.4—Pacer credit adjustments over time

Crossing below the zero threshold generates a rate-limiting indication, so that offered traffic can stop. By design, the credit value never goes below the *–loLimit* extreme. To bound the burst traffic after inactivity intervals, when no frames are ready for transmission, credits are reduced to zero (if currently higher than zero) and can accumulate to no more than the zero-value limit.

The *hiLimit* threshold limits the positive credits, to avoid overflow. When frames are ready for transmission (and are being blocked by transit traffic), credits can accumulate to no more than this *hiLimit* value.

In concept, the pacer consists of a token bucket. The credits in the token bucket are incremented by the size of each transmitted debit-frame. The number of credits in a token bucket is decremented by the size of each transmitted credit-frame. A credit-frame is only transmitted when the credits are positive; a debit-frame is only transmitted when the credits are negative.

9.2 Terminology and variables

9.2.1 Common state machine definitions

The following state machine inputs are used multiple times within this clause.

queue values
: Enumerated values used to specify shared queue structures.
  - QP_TX_PUSH—The transmit port's internal queue, where received frames are placed.
  - QP_TX_A0—The first of the output port's classA buffers.
  - QP_TX_A1—The second of the output port's classA buffers.
  - QP_TX_BP—The output port's classB queue.
  - QP_TX_CP—The output port's classC queue.
  - QP_TX_LINK—The output port's transmit-PHY queue.

9.2.2 Common state machine variables

One instance of each variable specified in this clause exists in each port, unless otherwise noted.

currentTime
: A value representing the current time.

framed
: The contents of a received frame, with supplemental information, as follows:
  - *frame*—The contents of a frame.
  - *sourcePort*—The source port that received the frame.
  - *txTime*—A time-stamp value representing the intended (bunching delayed) transmission time.

9.2.3 Common state machine routines

Max(*value1*, *value2*)
: Returns the numerically larger of two values.

9.2.4 Variables and routines defined in other clauses

This clause references the following variables and routines defined in Clause 7:

currentTime
: See 7.2.2.

Dequeue(*queue*)

Enqueue(*queue, frame*)

Min(*value1*, *value2*)
: See 7.2.3.

9.3 Pacing state machines

9.3.1 TransmitTx state machine

The TransmitTx state machine is responsible for pacing/shaping classA traffic and shaping classB traffic destined for 1 Gb/s links. An intent is to support projected MTU-sized transfers and interleaved lower-class traffic, without exceeding the 1-cycle delay inherent with cycle-synchronous bridge-forwarding protocols.

The following subclauses describe parameters used within the context of this state machine.

9.3.1.1 TransmitTx state machine definitions

BPS
: The nominal link transmission rate, in bytes per second.

MTU
: The maximum frame size, in bytes.

queue values
: Enumerated values used to specify shared queue structures.
  QP_TX_A0, QP_TX_A1
  QP_TX_BP, QP_TX_CP
  QP_TX_LINK
  See 11.2.1.

TICK
: The amount of time between shaper updates.
  Range: [1 bytes transmit time, 16-bit transmit time]
  Default: 1 byte transmit time

9.3.1.2 TransmitTx state machine variables

best
: A value that represents the weight and identify of the next-best classA queue.
  goodness—The smallest *weight*×*wait* value associated with alternate classA transmissions.
  queue—The queue associated with the best futuristic encapsulated frame.

creditA0
: A shaper credit whose positive value enables classA0 transmissions.

creditA1
: A shaper credit whose positive value enables classA1 transmissions.

creditA
: A shaper credit whose positive value enables primary classB transmissions.

creditB
: A shaper credit value whose positive and negative values enable secondary classB and classC transmissions respectively.

currentTime
: See 11.2.4.

frame
: The contents of a to-be-transmitted frame.

framed
: See 11.2.2.

hiLimitA
: A value that limits the cumulative *creditA* credits.
  Value: MTU.

hiLimitB
: A value that limits the cumulative *creditB* credits.
  Value: MTU.

limit
A value that limits the amount of transmitted primary classA/classB bandwidth.
loLimitA0
A value that limits the cumulative *creditA0* debits.
Value: MTU.
loLimitA1
A value that limits the cumulative *creditA1* debits.
Value: MTU.
loLimitA
A value that limits the cumulative *creditA* debits.
Value: MTU.
loLimitB
A value that limits the cumulative *creditB* debits.
Value: MTU.
tickTime
A value that defines when the time-tick interval ends.

9.3.1.3 TransmitTx state machine routines

Dequeue(*queue*)
See 11.2.4.
Unqueue(*queue*)
Dequeues and returns the next frame from the specified *queue*.
framed—The oldest of the overdue frame.
NULL—No frame available.
Enqueue(*queue, frame*)
See 11.2.4.
Size(*frame*)
Returns the size of the specified frame.

9.3.1.4 TransmitTx state table

The TransmitTx state machine is specified in Table 11.3. In the case of any ambiguity between the text and the state machine, the state machine shall take precedence. The notation used in the state table is described in 3.4.

Table 9.2—TransmitTx state table

Current		Row	Next	
state	**condition**		**action**	**state**

START	(currentTime – tickTime) >= TICK;	1	creditA = Min(hiLimitA, creditA + 0.75 * TICK * BPS);
			creditA0 = Min(hiLimitA0, creditA0 + rateA0 * TICK * BPS);
			creditA = Min(hiLimitA1, creditA1 + rateA1 * TICK * BPS);
			tickTime = currentTime ;
				START

	TransmissionInProgress()	2	—	
	—	3	—	LOOP
LOOP	creditA0 > 0 && (framed = Unqueue(queue= QP_TX_A0)) != NULL	1	creditA0 = Min(loLimitA0, creditA0 – Size(framed));	NEXT
	creditA1 > 0	2	countA0 = 0;	LOOP
	creditA1 >= 0 && (framed = Unqueue(queue= QP_TX_A0)) != NULL	3	creditA1 = Min(loLimitA1, creditA1 – Size(framed));	NEXT
	creditA1 > 0	4	countA1 = 0;	LOOP
	creditA >= 0 && (framed = Unqueue(queue= QP_TX_BP)) != NULL	5	creditA = Min(loLimitA1, creditA – Size(framed));	FINAL
	creditA > 0	6	countA = 0;	LOOP
	creditB >= 0 && (framed = Dequeue(QP_TX_BP)) != NULL	7	creditB = creditB – Size(framed);	FINAL
	creditB <= 0 && (framed = Dequeue(QP_TX_CP)) != NULL	8	creditB = creditB + Size(framed);	
	(framed = Dequeue(QP_TX_BP)) != NULL	9	creditB = 0;	
	(framed = Dequeue(QP_TX_CP)) != NULL	10		
	—	11	creditB = 0;	START
NEXT	—	1	creditA = Min(loLimitA, creditA – Size(framed));	FINAL
FINAL	—	1	Enqueue(QP_TX_LINK, framed.frame);	START

START-1: Update the classA credits after each tick interval.
START-2: Wait for the queue to be emptied, so that something can be transmitted.
START-3: Check for possible frame transmissions.

LOOP-1: In the presence of classA0 credits, transmit a classA0 frame.
LOOP-2: In the absence of classA0 frames, clear classA0 credits.
LOOP-3: In the presence of classA1 credits, transmit a classA1 frame.
LOOP-4: In the absence of classA1 frames, clear classA1 credits.
LOOP-5: In the presence of classA credits, transmit a classB frame.
LOOP-6: In the absence of classA frames, clear classA credits.
LOOP-7: If enabled and available, a classB frame is transmitted.
The *creditB* values is decremented by the transmitted frame size, to avoid classC starvation.
LOOP-8: If enabled and available, a classC frame is transmitted.
The *creditB* values is incremented by the transmitted frame size, to avoid classB starvation.
LOOP-9: If available, a classB frame is transmitted.
LOOP-10: If available, a classC frame is transmitted.
LOOP-11: Otherwise, no frame is transmitted.

NEXT-1: Either of classA0/classA1 transmissions also update classA credits.

FINAL-1: The next frame is transmitted and credits are updated accordingly.

10. Transmit state machines (proposal 2)

NOTE—Multiple bunch-avoiding pacing protocols are presented for consideration:
a) Clause 9 (this clause) presents an end-point shaper model, with no bridge shaping assumed.
b) Clause 10 presents a pseudo-synchronous transmission model.
c) Clause 11 presents a cross-flow shaper transmission model, an optional bridge Clause 9 enhancement.

10.1 Pacing overview

10.1.1 Delays

The preferred topologies consists entirely of paced bridges, as illustrated in Figure 10.1a. Within such topologies, a frame transmitted by station a0 in cycle[*n*] incurs fixed nominal delays while passing through bridges. Thus, this frame nominally departs bridgeB in cycle[*n*+2], bridge C in cycle[*n*+4], and bridgeE in cycle[*n*+6].

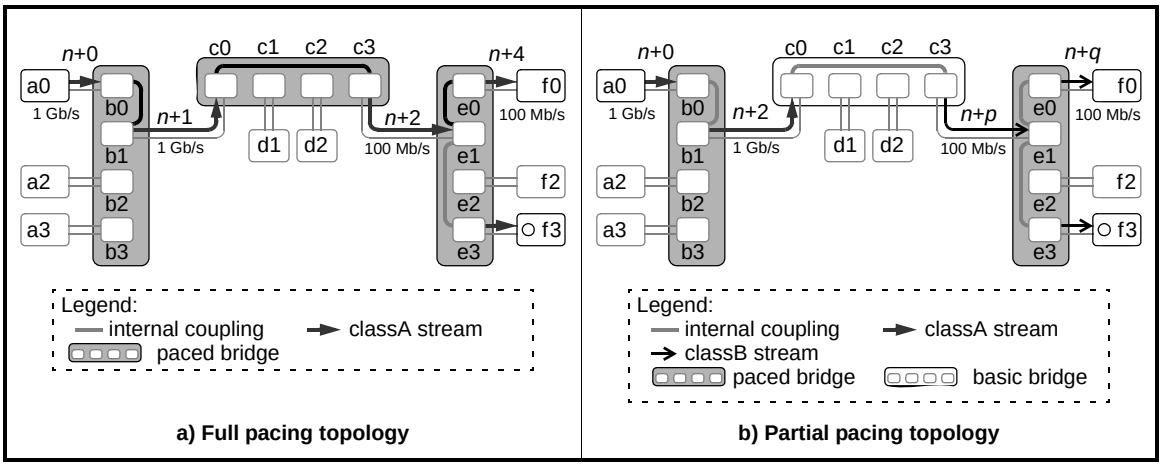


Figure 10.1—Topology-dependent pacing delays

Within Figure 10.1a, the actual transmission times can vary from their nominal targets, due to contention with other traffic. Each bridge compensates for early and late arrivals, so that the extent of deviations from nominal on link b1-to-c0 are the same as those on link e0-to-f0.

Within Figure 10.1b, an intermediate basic bridge is assumed. Output from bridgeC is therefore downgraded from classA to classB, to avoid degradation of well-paced traffic. Thus, the fully-paced properties of bridgeE still apply to possible f3-to-f0 traffic (not illustrated).

The uncertainty of cycle *p* and *q* cycle delays in Figure 10.1b are due to passing through the non-paced bridgeC. Although much of this traffic would arrive earlier, some of the traffic could be delayed up to the nominal delays of Figure 10.1a. In more complex topologies, such delays could exceed the nominal delays through paced bridges, due to bunching effects (see Annex F).

To support such topology, this working paper mandates that compliant end stations provide larger elasticity buffers (see TBD) than required within fully paced topologies. However, defining topology restrictions to ensure elasticity-buffer sufficiency is beyond the scope of this working paper.

10.1.2 Paced 1 Gb/s classA flows

Pacing involves sending accumulated classA traffic once every isochronous cycle, rather than allowing larger (typically an MTU) frames to be accumulated. After each cycle's classA traffic has been sent, the remaining time is available for sending classB/classC traffic. This provides low-jitter bandwidth guarantees, as does time division multiplexing (TDM), while allowing unused classA bandwidths to be utilized by classB/classC traffic.

A pacing bridge maintains this pacing behavior, thus avoiding problems normally associated with bunching (see Annex F). For a bridge between 1 Gb/s link1 and 1 Gb/s link2 (see Figure 10.2a), paced frames can be forwarded with a nominal 1-cycle delay (see Figure 10.2b). The 1-cycle delay is necessary to account for offset migration and store-and-forward processing delays.

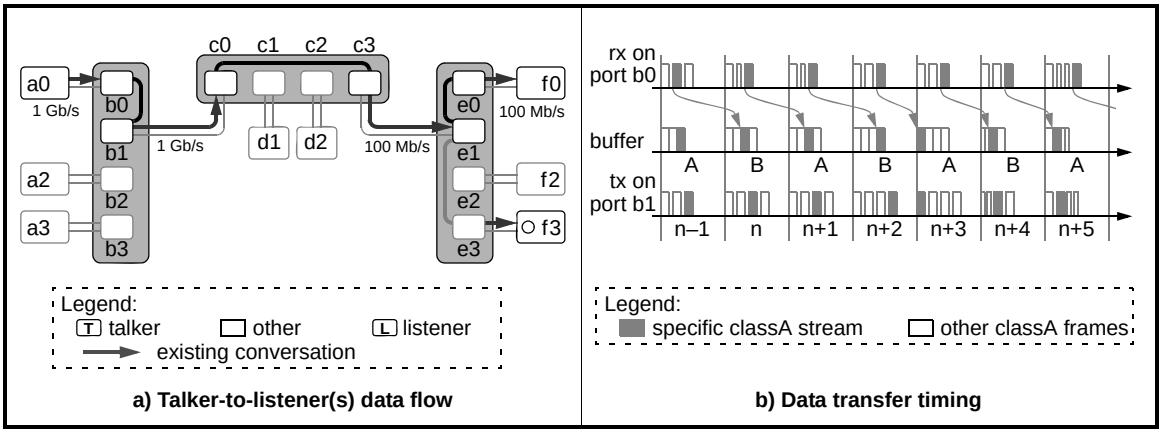


Figure 10.2—Paced 1 Gb/s classA flows

Offset migration refers to changes in a classA frame's within-cycle placement on (for example) link1 and link2. Depending on the timing of unrelated events, the offset of the classA-data frame within the cycle can migrate over time, as other conversations are started, ended, advanced, delayed, joined, or routed elsewhere.

A possible implementation could utilize double output buffers, processed as follows:

cycle[n+2×k+0]: classA traffic is saved in buffer[A] and transmitted from buffer[B].
cycle[n+2×k+1]: classA traffic is saved in buffer[B] and transmitted from buffer[A].

The boundaries between cycles are marked by a distinct set of cycleSync markers (not illustrated), rather than relying on precise time-synchronization and deadbands to imply their temporal placement.

The classA transmissions within each cycle are shaped, to allow for unrelated asynchronous frame transmissions. The shaper allows a higher-than 75% transmission rate, to ensure transmission completion well before before the next cycle begins, even in the presence of conflicting non-classA transmissions.

To better understand the minimal buffer requirements, consider frame transfers that are momentarily disrupted by an MTU-sized classC transmission, started near the end of link1's classA transmissions. For the receive-side slippage scenario of 10.3a, data[*n*] arrives in cycle[*n*] and fills buffer[A]. Since buffer A is not destined for transmission until cycle[*n*+1], conflicts are avoided.

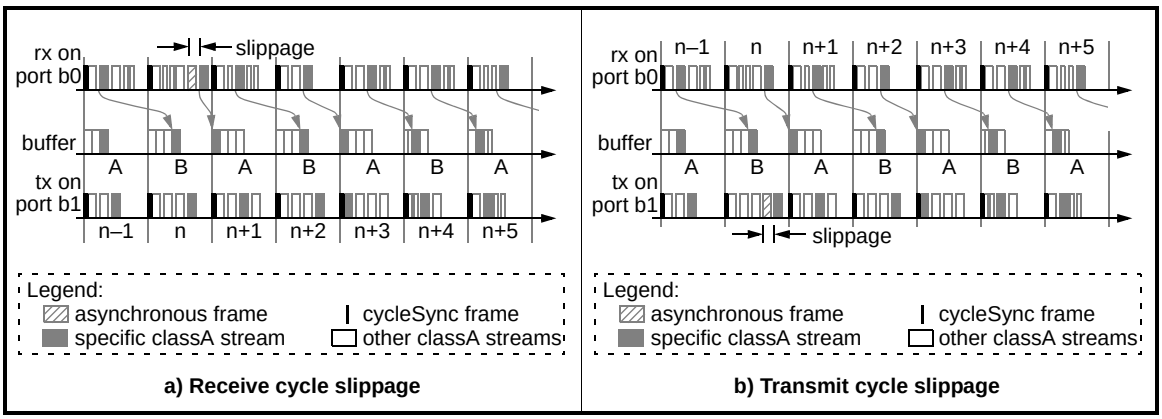


Figure 10.3—Cycle slippage

For the transmit-side slippage scenario of Figure 10.3b, buffer[B] is fully emptied in cycle[*n*]. Since buffer[B] is not destined for filling until cycle *n*+1, conflicts are avoided.

10.1.3 Paced 100 Mb/s flows

Editors' Notes: To be removed prior to final publication.
A two-cycle delay is illustrated, although the protocols can be simplified by assuming a three cycle delay. The tradeoff between protocol simplicity and a passthrough latency has not been carefully reviewed.

A 100 Mb/s pacing bridge also maintains this pacing behavior, thus avoiding problems normally associated with bunching (see Annex F). For a bridge between 100 Mb/s link3 and 100 Mb/s link4 (see Figure 10.4a), paced frames can be forwarded with a nominal 2-cycle delay (see Figure 10.4b).

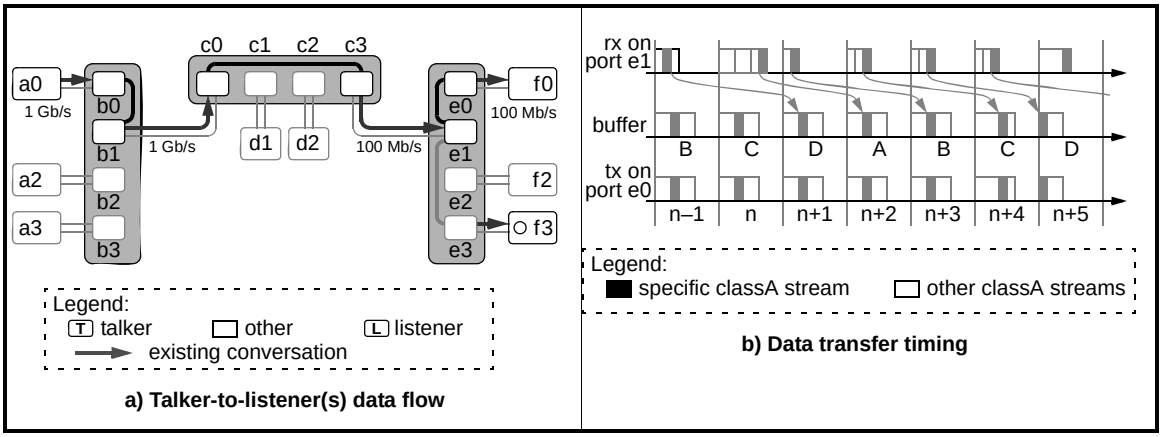


Figure 10.4—Paced 100 Mb/s classA flows

A possible implementation would involved six output buffers, processed as follows:

cycle[n+4×k+0]: classA traffic is saved in buffer[A] and transmitted from buffer[C].
cycle[n+4×k+1]: classA traffic is saved in buffer[B] and transmitted from buffer[D].
cycle[n+4×k+2]: classA traffic is saved in buffer[C] and transmitted from buffer[A].
cycle[n+4×k+3]: classA traffic is saved in buffer[D] and transmitted from buffer[B].

To better understand the minimal buffer requirements, consider frame transfers that are momentarily disrupted by an MTU-sized classC transmission, started near the end of link3 classA transmissions. For the receive-side slippage scenario of Figure 10.5a, data[n] arrives in cycle[n+1] and fills buffer[A]. Since buffer A is not destined for transmission until cycle[n+2], conflicts are avoided.

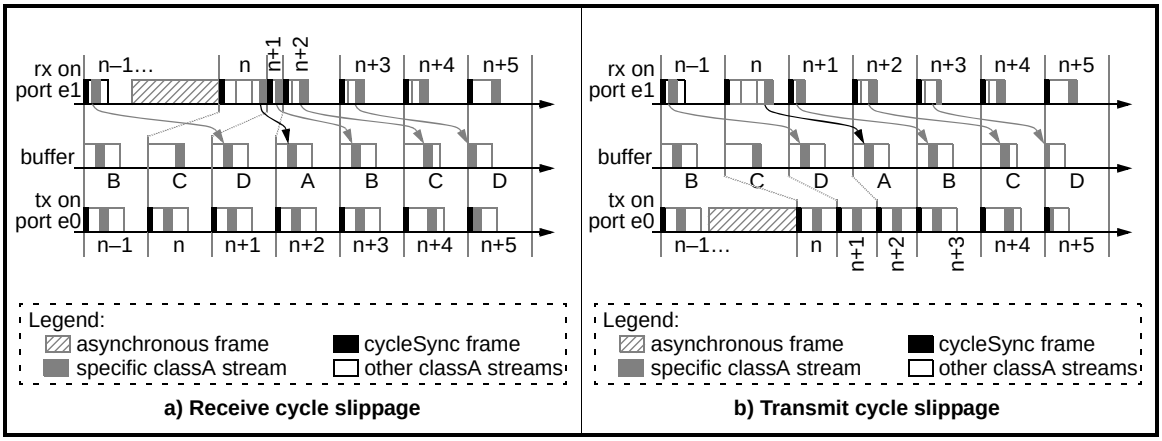


Figure 10.5—Cycle slippage

For the transmit-side slippage scenario of Figure 10.5b, buffer[D] is fully emptied in cycle[n+1] and in cycle[n+2]. Since buffer[D] is not destined for filling until cycle n+3, conflicts are avoided.

To achieve a robust 2-cycle latency objective, restrictions are placed on non-classA transmissions. These restrictions are as follows:

a) An MTU (or sequence of frames not exceeding an MTU) may be appended to the last classA frame within any cycle whose cycleSync frame transmission was not delayed.

b) Within any cycle, any non-classA frame may be transmitted after the last classA frame, but only if this frame transmission would not delay the transmission of the next cycleSync frame.

Condition (a) is sufficient to ensure that all transmissions occur within the intended or following cycle, assuming a 100 Mb/s span, 2000 byte MTU, 125 μs cycle, and 75% classA loading. With these assumptions, the worst-case delay from the start of the intended cycle, as specified by Equation 10.1, is well within the 2-cycle 250 μs constraint.

$$delay \geq (\text{MTU} - 0.25 \times \text{cycle}) + 0.75 \times \text{cycle} \quad (10.1)$$

$$delay \geq 2\,000 \times ((8 \text{ bits/byte}) \times (1 \text{ second})/(100 \text{ Mb/s})) + 0.50 \times (125\ \mu\text{s})$$

$$delay \geq (160\ \mu\text{s}) + (62.5\ \mu\text{s})$$

$$delay \geq 222.5\ \mu\text{s}$$

10.1.4 Transmit port structure

An end station and bridge have functionally distinct transmit queues for classA, classB, and classC traffic, allowing each to be managed separately, as illustrated in Figure 10.6. The transmit port is responsible for pacing classA/classB traffic and shaping classB/classC traffic, so as to limit the high-class traffic to 75% of the link bandwidth. The transmit-port structure is slightly different for 100 Mb/s and 1 Gb/s transmit ports, due to the distinct times associated with an MTU transmission.

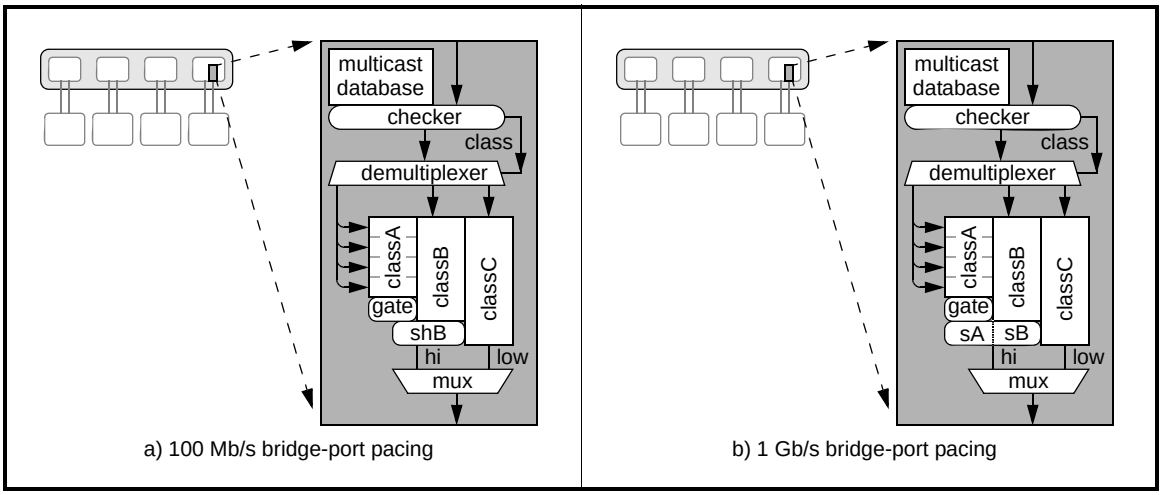


Figure 10.6—Transmit-port structure

Although classA frames have the highest priority, the classA frames are gated to prevent their early departure. Gating involves blocking classA frames that arrived with *sourceCycle*=*n*, until the start of cycle *n*+*p*. After the start of cycle *n*+*p*, the transmitter waits for the completion of preceding non-classA frames (or residual cycle *n*+*p*–1 classA frames), then transmits these arrived-in-cycle-*n* frames with *sourceCycle*=*n*+*p*. As noted previously, *p* is a design-dependent integer constant, preferably no more than 4 cycles (see 5.1.2 and 5.1.3).

A bridge has to cope with frame-reception uncertainties (due to preceding frame-transmission uncertainties), in addition to its own frame-transmission uncertainties. As such, the values of *p* are expected to be slightly larger in bridges than in talker-station or listener-station designs.

Within bridges, the distinction between service classes is based on the multicast addresses within frames. These multicast addresses are checked against the multicast database, which supplies *class* information in addition to the normal multicast routing (forward or not-forward) information. This *class* information controls the demultiplexer, which routes to the appropriate classA, classB, or classC output queues.

The cycle slippage on a 100 Mb/s link mandates the use of four 3/4-cycle output buffers, which incur a 2-cycle pass-through delay. The classA traffic is gated to avoid wrong-cycle transmissions and excessive consumption, but is not otherwise not shaped. The overlapping shB shaper of Figure 10.6a is intended to illustrate the use of classA transmission counts and the classB shaper, not the shaping of classA traffic.

On such 1 Gb/s transmitter ports, the classA traffic is shaped to reduce lower-class blockage, as well as gated to avoid wrong-cycle transmissions and excessive consumption. The adjacency of shA/shB shapers in Figure 10.6b is intended to illustrate distinct classA/classB shaping functions, but sharing of classA transmission counts between shapers.

Achievable delays through a bridge depend only on the speed of the input-link speed, as summarized in Table 10.1. These numbers are slightly misleading, since transmissions on a 100 Mb/s link have implied additional delays incurred when passing through its adjacent 100 Mb/s receiver.

Table 10.1—ClockPort state table

Link type		Delay	
Input	**Output**	**Cycles**	**Time**
100 Mb/s	—	2	250 μs
1 Gb/s	—	1	125 μs

10.1.5 Pacing at 1 Gb/s

Pacing at 1 Gb/s, as illustrated in Figure 10.7. For ontime cycles, a residual amount of classB/classC traffic is allowed throughout the cycle, as illustrated in Figure 10.7a. For slipped cycles, a residual amount of classB/classC traffic becomes available after the delay effects have been overcome, as illustrated in Figure 10.7b.

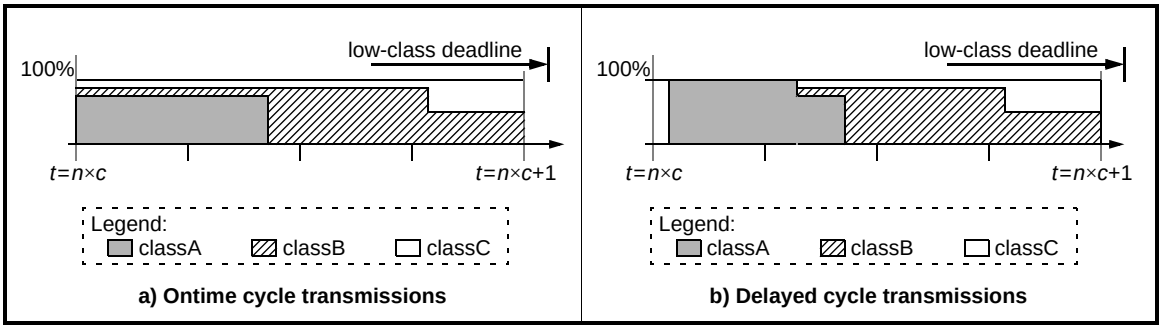


Figure 10.7—Pacing at 1 Gb/s

10.1.6 Pacing at 100 Mb/s

Pacing at 100 Mb/s, as illustrated in Figure 10.8. For ontime cycles, a residual amount of classB/classC traffic is allowed throughout the cycle, as illustrated in Figure 10.8a. For delayed cycles, a residual amount of classB/classC traffic becomes available after the delay effects have been overcome, as illustrated in Figure 10.8b.

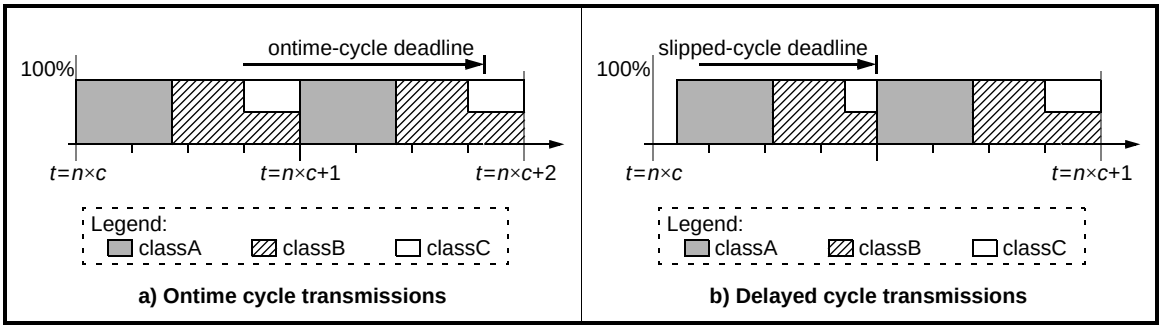


Figure 10.8—Pacing at 100 Mb/s

10.1.7 Shaper behavior

Although multiple shaper are specified within this working paper, the behavior of most shapers can be characterized by a common algorithm and instance-specific parameters (as done within RPR[B5]). The shapers' credits are adjusted down or up, as illustrated in Figure 10.9. The decrement and increment values typically represent sizes of a transmitted frame and of credit increments in each update interval, respectively.

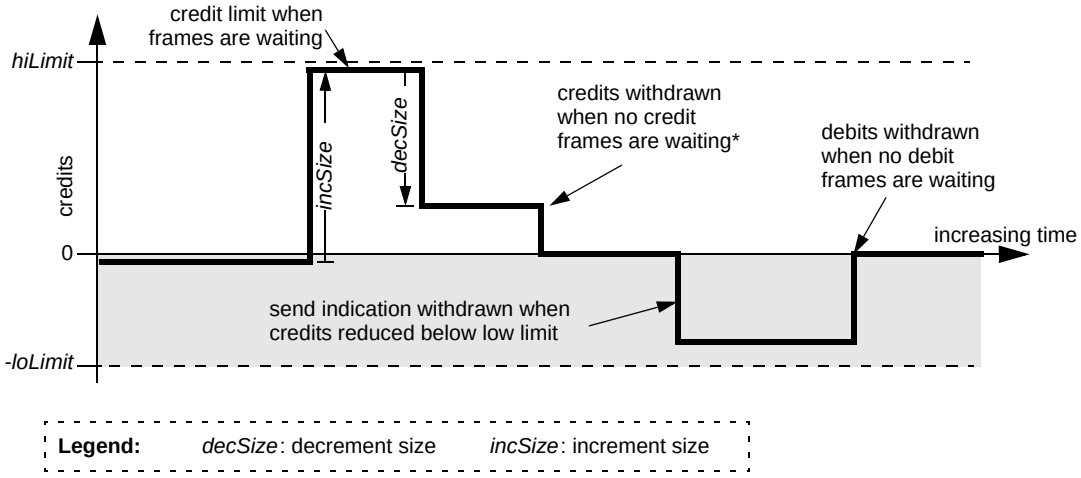


Figure 10.9—Credit adjustments over time

Crossing below the zero threshold generates a rate-limiting indication (the removal of a send indication), so that offered traffic can stop. By design, the credit value never goes below the *–loLimit* extreme. To bound the burst traffic after inactivity intervals, when no frames are ready for transmission, credits are reduced to zero (if currently higher than zero) and can accumulate to no more than the zero-value limit.

The *hiLimit* threshold limits the positive credits, to avoid overflow. When frames are ready for transmission (and are being blocked by transit traffic), credits can accumulate to no more than this *hiLimit* value.

In concept, the shapers consist of a token bucket. The credits in the token bucket are incremented by the size of each debit-frame when it is being transmitted. The number of credits in a token bucket is decremented by the size of each credit-frame when it is being transmitted. When a credit-frame is waiting, it is transmitted only if the number of credits in the token bucket is positive; When a debit-frame is waiting, it is transmitted only if the number of credits in the token bucket is negative.

10.2 Terminology and variables

10.2.1 Common state machine definitions

The following state machine inputs are used multiple times within this clause.

queue values
: Enumerated values used to specify shared queue structures.
  - QP_TX_PUSH—The input port's receive-from-ports queue.
  - QP_TX_CA—The first of the output port's classA buffers.
  - QP_TX_CB—The output port's classB queue.
  - QP_TX_CC—The output port's classC queue.
  - QP_TX_LINK—The output port's transmit-PHY queue.
  - QP_TX_SYNC—The port's queue that provides timeSync frames.

10.2.2 Common state machine variables

One instance of each variable specified in this clause exists in each port, unless otherwise noted.

currentTime
: A value representing the current time.

mtuSize
: The size of the maximum transfer unit (MTU).
  Value: 2000 bytes

NOTE—The specified *mtuSize* is larger than currently supported by IEEE Std 802.3, but consistent with expected near-term frame-extension revisions of this standard.

speedIs100Mbs
: A value that communicates the operating speed of the link.
  - TRUE—The port is operating at a speed of 100 Mb/s.
  - FALSE—The port is operating at speeds of 1 Gb/s or above.

thisCycle
: A cycle counter derived from *thisTime*, as defined by Equation 10.2.

```
Floor(thisTime * 8000);                                    (10.2)
```

thisTime
: A normalized time-of-day counter derived from *timeOfDay*, as defined by Equation 10.3.

```
(timeOfDay / (4.0 * (1<<30)))                              (10.3)
```

10.2.3 Common state machine routines

–none–

10.2.4 Routines defined in other clauses

This clause references the following routines defined in Clause 7:

Dequeue(*queue*)
Enqueue(*queue*, *frame*)
Min(*value1*, *value2*)
See 7.2.3.

10.3 Pacing state machines

10.3.1 ReceiveRx state machine

The ReceiveRx state machine is responsible for receiving pacing classA traffic, shaped classB traffic, and best-effort classC traffic. An intent is to transfer each to the appropriate output queue.

The following subclauses describe parameters used within the context of this state machine.

10.3.1.1 ReceiveRx state machine definitions

CYCLE_SYNC
An assigned *subType* value that distinguishes a timeSync from other Residential Ethernet frames.
GROUP_BIT
A constant value derived from IEEE Std 802-2001 and specified by Equation 10.4.

```
((macAddress & GROUP_BIT) != 0)
```
(10.4)

queue values
Enumerated values used to specify shared queue structures.
QP_TX_CA, QP_TX_CB, QP_TX_CC
QP_TX_PUSH
See 10.2.2.
RES_ETHER
The *protocolType* code value assigned to Residential Ethernet.

10.3.1.2 ReceiveRx state machine variables

class
A value that represents the results of a forwarding database search.
delta
A value that represents the difference between frame-signaled and computed cycle values.
frame
The contents of a received frame.
myCycle
The two least-significant bits of the *thisCycle* value.
queueA
The selected classA queue identifier, based on *delta*-selected locations.
speedIs100Mbs
thisCycle

thisTime
See 10.2.2.

10.3.1.3 ReceiveRx state machine routines

DataBaseClass(*macAddress*, *port*)
Provides a forwarding database indication of how the *macAddress* is routed to the specified *port*.
CLASS_A—The associated multicast frame is forwarded as classA traffic.
CLASS_B—The associated multicast frame is forwarded as classB traffic.
CLASS_C—The associated multicast frame is forwarded as classC traffic.
BLOCKED—The associated multicast frame is not forwarded.
Dequeue(*queue*)
See 10.2.4.
EnqueuePort(*port*, *queue*, *frame*)
Places the *frame* at the tail of the specified *queue* within the specified *port*.
ForwardUnicast(*frame*)
Forwards a unicast frame to the selected output port, if any.
This routine mimics existing standards, which remain unaffected by this working paper.
Multicast(*macAddress*)
Indicates whether the supplied address is a multicast (or broadcast) address, as specified by Equation 10.5.
TRUE—The address is a multicast (or broadcast) address.
FALSE—(Otherwise.)

```
((macAddress & GROUP_BIT) != 0)
```
(10.5)

10.3.1.4 ReceiveRx state table

The ReceiveRx state machine is specified in Table 10.2. In the case of any ambiguity between the text and the state machine, the state machine shall take precedence. The notation used in the state table is described in 3.4.

Table 10.2—ReceiveRx state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(QP_TX_PUSH)) != NULL	1	—	FIRST
	—	2	myCycle = (thisCycle % 4); delta = (4 + myCycle – rxCycle) % 4;	PLACE
PLACE	delta == 3	1	delta = 0;	PLUS
	—	2	—	
PLUS	speedIs100Mbs	1	queueA = QP_TX_CA + (4 + 2 – delta) % 4;	START
	—	2	queueA = QP_TX_CA + (4 + 1 – delta) % 4;	
FIRST	frame.protocolType==RES_ETHER && frame.subType == CYCLE_SYNC	1	rxCycle = (frame.cycleCount % 4);	START
	Multicast(frame.da)	2	class = DataBaseClass(frame.da, port);	CAST
	—	3	ForwardUnicast(frame)	START
PUSH	class == CLASS_A	1	EnqueuePort(port, queueA, frame);	START
	class == CLASS_B	2	EnqueuePort(port, QP_TX_CB, frame);	
	class == CLASS_C	3	EnqueuePort(port, QP_TX_CC, frame);	
	—	4	—	

Row START-1: If a frame has arrived, process that frame.
Row START-2: Otherwise, compute the cycle offset for later classA queue placement.

Row PLACE-1: Frames that arrive early are processed as though they arrived within this cycle.
Row PLACE-2: Otherwise, the difference between labeled and actual cycles determines frame placement.

Row PLUS-1: Frames arriving from a 100 Mb/s link are placed 2-cycles ahead, to allow for cycle slips.
Row PLUS-2: Frames arriving from a 1 Gb/s link are placed 1-cycle ahead, since cycle slips are avoided.

Row FIRST-1: The cycleSync frames identify the cycle number, despite cycle-slip possibilities.
Row FIRST-2: Multicast frames are sent to all enabled ports.
Row FIRST-3: Unicast frames are processed normally.

Row PUSH-1: Multicast classA frames are forwarded to the appropriate cycle-sensitive classA queue.
Row PUSH-2: Multicast classB frames are forwarded to the classB queue.

Row PUSH-3: Multicast classC frames are forwarded to the classC queue.
Row PUSH-4: If no class is specified, multicast frames are not routed through this port.

10.3.2 TransmitTx state machine

The TransmitTx state machine is responsible for pacing/shaping classA traffic and shaping classB traffic destined for 1 Gb/s links. An intent is to support projected MTU-sized transfers and interleaved lower-class traffic, without exceeding the 1-cycle delay inherent with cycle-synchronous bridge-forwarding protocols.

The following subclauses describe parameters used within the context of this state machine.

10.3.2.1 TransmitTx state machine definitions

BPS
Represents a bound on the number of transmitted bytes per second, as defined by Equation 10.6.

```
(speedIs100Mbs ? 12500000 : 125000000)
```
(10.6)

CAP
Represents a bound on the number of transmitted bytes, as defined by Equation 10.7.

```
((speedIs100Mbs && phase != MORE) ?
  ((cycle + 1) * 8000. - thisTime) * BPS : MTU)
```
(10.7)

queue values
Enumerated values used to specify shared queue structures.
QP_TX_CA
QP_TX_CB
QP_TX_CC
QP_TX_LINK
QP_TX_SYNC
See 10.2.2.

10.3.2.2 TransmitTx state machine variables

creditA
A shaper credit whose positive value enables classA/classB primary transmissions.
creditB
A shaper credit whose positive/negative values enable secondary classB/classC transmissions.
cycle
The cycle whose classA data is being transmitted.
cycleSize
The number of bytes included within a 125 μs cycle.
Value:
1562.5—for 100 Mb/s links
15625—for 1 Gb/s links
frame
The contents of a to-be-transmitted frame.
hiLimitB
A value that limits the cumulative *creditB* credits.
Value: MTU.
limit
A value that limits the amount of transmitted primary classA/classB bandwidth.
loLimitB
A value that limits the cumulative *creditB* debits.
Value: MTU.
phase
An indication of what remains to be transferred within the cycle.

HEAD—The cycleSync frame are to be sent.
MORE—Other classA/classB frames are to be sent.
DONE—All classA frames have been sent.

queue
A variable that identifies the appropriate classA queue for this cycle's transmissions.

speedIs100Mbs
See 10.2.2.

thisCycle

thisTime
See 10.2.2.

10.3.2.3 TransmitTx state machine routines

Cap(*speedIs100Mbs*, *phase*, *creditA*, *cycle*, *thisTime*)
Provides a cap on the lengths of classB and classC transmissions.

```
if (speedIs100Mbs) {                                        (10.8)
  if (phase == MORE)
    return(0);
  near = (cycle + 1.0) * 8000;
  safe = (cycle + 0.8) * 8000;
  return(thisTime <= safe ? MTU : near * BPS);
} else {
  if (phase == MORE)
    return(–creditsA/16);
  near = (cycle + 1.05) * 8000;
  return((near - thisTime) * BPS);
}
```

Dequeue(*queue*)
See 10.2.4.

DequeueSize(*queue, size*)
Returns the next available frame from the specified queue, from frames no larger than *size*.

Enqueue(*queue, frame*)
See 10.2.4.

QueueEmpty(*queue*)
Returns the an indication of whether the queue is empty.
0—The specified queue is not empty.
1—The specified queue is empty.

Size(*frame*)
Returns the size of the specified frame.

10.3.2.4 TransmitTx state table

The TransmitTx state machine is specified in Table 10.3. The link-speed independent rows are white; the link-speed dependent rows are shaded. In the case of any ambiguity between the text and the state machine, the state machine shall take precedence. The notation used in the state table is described in 3.4.

Table 10.3—TransmitTx state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	cycle > (thisCycle + 1)	1	cycle = thisCycle; phase = HEAD;	PREP
	cycle < (thisCycle – 1)	2		
	cycle < thisCycle && phase == DONE	3	cycle += 1; phase = HEAD;	
	!QueueEmpty(QP_TX_LINK)	4	—	START

	phase == HEAD	5	queue = QP_TX_CA + (cycle % 4);
			frame = Dequeue(QP_TX_SYNC);
			limit = 0.75 * cycleSize;
			creditA=
			16 * BPS * (thisTime – cycle*8000.);
			phase = MORE;
				POST

	—	6	cap = Cap(speedIs100Mbs, phase, creditA, cycle, thisTime);	PLUS
PLUS	creditB >= 0 && (frame = DequeueSize(QP_TX_CB), cap) != NULL	1	creditB = Max(loLimitB, creditB – Size(frame)); creditA += 16 * Size(frame);	FINAL
	creditB <= 0 && (frame = DequeueSize(QP_TX_CC, cap)) != NULL	2	creditB = Min(hiLimitB, creditB +Size(frame)); creditA += 16 * Size(frame);	
	(frame = DequeueSize(QP_TX_CB, cap)) != NULL	3	creditA += 16 * Size(frame);	
	(frame = DequeueSize(QP_TX_CC, cap)) != NULL	4		
	phase != MORE	5	—	START
	(frame = DequeueSize(queue, limit)) != NULL	6	—	POST
	(frame = Dequeue(queue)) != NULL	7	—	START
	(frame = DequeueSize(QP_TX_CB, limit)) != NULL	8	limit –= Size(frame);	FINAL
	—	9	phase = DONE; creditB= Min(hiLimitB, limit+creditB);	START
POST	—	1	limit –= Size(frame); creditA –= Size(frame);	FINAL
FINAL	—	1	Enqueue(QP_TX_LINK, frame);	START

Row START-1: If *cycle* has advanced two-beyond *thisCycle*, something is in error.
(The *cycle* value can advance one-beyond *thisCycle*, due to small *timeOfDay* update discontinuities.)
Row START-2: If *cycle* has dropped two-behind *thisCycle*, something is in error.
(Large *timeOfDay* update discontinuities can cause cycle to advance or retreat beyond normal bounds.)
|Row START-3: The phase is initialized to HEAD at the start of each cycle.
Row START-4: Wait for the queue to be emptied, so something can be transmitted.
Row START-5: When the next cycle starts, a timeSync frame is transmitted.
The *limit* value is set to limit classA transmissions to no more than 75% of the link bandwidth.
The *creditA* value initialized to account for cycleSync frame slippage (for 1 Gb/s ports only).
Row START-6: Set caps on the maximum transmission size of classB/classC transmissions.

Row PLUS-1: If enabled and available, a classB frame is transmitted.
The *creditB* values is decremented by the transmitted frame size, to effect a classB shaper.
Row PLUS-2: If enabled and available, a classC frame is transmitted.
The *creditB* values is incremented by the transmitted frame size, to effect a classB shaper.
Row PLUS-3: If available, a classB frame is transmitted.
Row PLUS-4: If available, a classC frame is transmitted.
Row PLUS-5: Otherwise, no frame is transmitted.
Row PLUS-6: An enabled, available, and properly sized classA frame is readied for transmission.
Row PLUS-7: An enabled, available, and improperly sized classA frame is discarded.
Row PLUS-8: An enabled, available, and properly sized classB frame is readied for transmission.
Row PLUS-9: If enabled but unavailable, this cycle's primary frame transmissions have completed.

Row POST-1: The shaper's creditA value is decremented to lightly throttle primary transmissions.
The limit value is also decremented, to enforce the 75% cycle classA/classB transmission limitation.

Row FINAL-1: Transmission is affected by placing the frame in the port's transmit queue.

11. Transmit state machines (proposal 3)

NOTE—Multiple bunch-avoiding pacing protocols are presented for consideration:
a) Clause 9 (this clause) presents an end-point shaper model, with no bridge shaping assumed.
b) Clause 10 presents a pseudo-synchronous transmission model.
c) Clause 11 presents a cross-flow shaper transmission model, an optional bridge Clause 9 enhancement.

11.1 Rate-based scheduling overview

The clause describes a rate-based scheduling technique. The rate-based scheduling concepts are similar to those within rate monotonic scheduling protocols, commonly used within real-time systems. Objectives associated with time-sensitive forwarding alternatives include the following:

a) Multiple time-sensitive transmission rates are supported, including:
   1) Highest rate 8 kHz traffic, such as the traffic generated by simple bridges between RE and existing IEEE 1394[B6] A/V devices.
   2) Lower rate traffic, such as voice over internet protocol (VOIP) traffic, without forcing this traffic to be reblocked into smaller (and therefore less efficient) frame sizes.

b) Frame forwarding should not be dependent on successful time-of-day synchronization between the bridge and adjacent stations. Frame forwarding should succeed before the grand clock-master station has been selected, or when the selected grand-master clock station changes.

c) Frame-forwarding protocols should leverage existing bridge queue and service models, although specification of abstract rate shaper details is expected.

d) Each traffic class has guaranteed latency, when the cumulative traffic is constrained to less than the link capacity. The latency guarantee is approximately an MTU more than an inter-arrival period.

Rate-based scheduling involves associating a priority with frame transmissions, where the priority is a monotonic function of the frame transmission frequency, as illustrated in Figure 11.1.

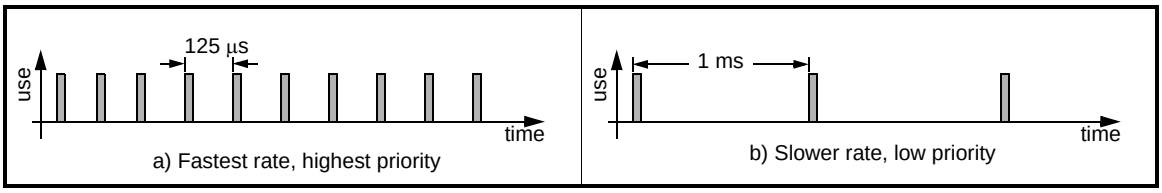


Figure 11.1—Rate-based priorities

An application could desire to send traffic once in every 4 ms interval, while attempting to constrain the worst-case latency to a smaller 1 ms value, as illustrated in Figure 11.2a. This would be done by subscribing for a 1 ms interval, while only sending the traffic in larger 4 ms intervals, as illustrated in Figure 11.2b. The lower latency is not achieved without costs: the effective subscription bandwidth allocated to this application is 4 times larger than necessary.

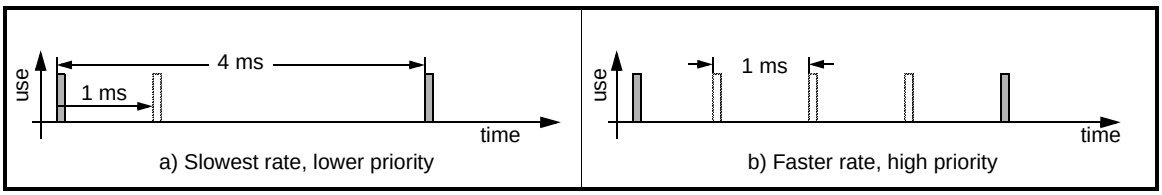


Figure 11.2—Rate-based priorities

11.1.1 Rate-based priorities

Quality of service is based on the availability of user_priority field parameter associated within transmitted time-sensitive frames, as listed in Table 11.1.

Table 11.1—Tagged priority values

Code	Interval (ms)	Name	Description
0	n/a	CLASS_C	Best effort, with minimal guaranteed BW
1	n/a	CLASS_B	Preferred, with minimal guaranteed BW
2-3	—	—	Used for other purposes
4	0.5	CLASS_A1	Guaranteed BW over short interval
5	0.125	CLASS_A0	Guaranteed BW over shorter interval
6-7	—	—	Network management

11.1.2 Port-to-port reshaping, without bunching accumulation

The concept of rate-based scheduling assumes shaped talkers and reshaped talker agents within bridges, as illustrated in Figure 11.3 (only the components associated with specific flows are illustrated). From a functional perspective, multiple shapers are required per class, where each class is illustrated as a distinct layer. From a practical perspective, one shaper instance with multiple contexts is sufficient.

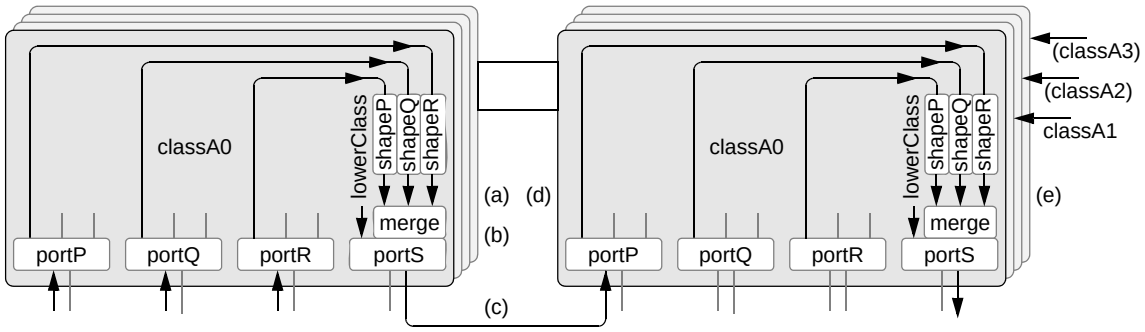


Figure 11.3—Reshaped bridge-traffic topology, with bunching control

In Figure 11.3, classA0 traffic flows between points (a, b, c, d, e), exhibits bunched and reshaped behaviors, as illustrated in Figure 11.4.

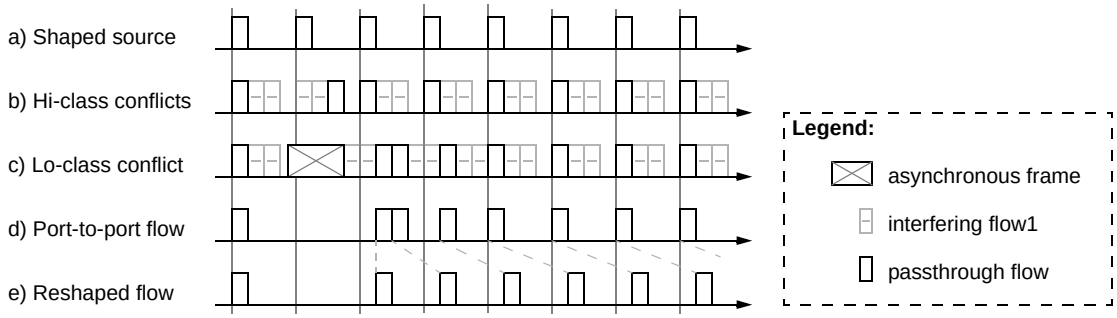


Figure 11.4—Reshaped bridge-traffic timing

The (a) through (e) time lines represent the flow of frames from within one talker-or-bridge into another bridge-or-listener, described as follows:

a) A properly shaped source stream is originally generated within a talker, or a port-to-port flow (consisting of multiple streams) within a bridge.

b) Forwarding of multiple sources to a shared transmission link can produce jitter, due to slight differences in frame-to-frame spacings.

c) Forwarding of multiple sources to a shared transmission link can produce additional jitter, when higher-class traffic waits for the completion of previously initiated lower-class transmissions.

d) Bunching becomes apparent in the port-to-port flow, representing the portion of the received (c) traffic that is forwarded to a specific transmitter port.

e) A shaper delays the forwarding of bunched frames, so that the port-to-port flow is properly shaped. Delays can be invoked by time stamping frames with an in-the-future transmission time.

The reshaped flow (e) retains the properly shaped properties of the preceding flow (a), while incurring a maximum delay *d* through the bridge. These properties ensure a linear maximum delay of $n \times d$, for streams that flow through N bridges.

11.1.3 Port-to-port reshaping, with bunching control

The complexity of managing traffic classes can be reduced by eliminating shapers, so that receiver inputs are merged before being shaped, as illustrated in Figure 11.5 (only the components associated with specific flows are illustrated). In this illustration, classA0 traffic flows between points (a, b, c, d, e), exhibits bunched and reshaped behaviors, as illustrated in Figure 11.4.

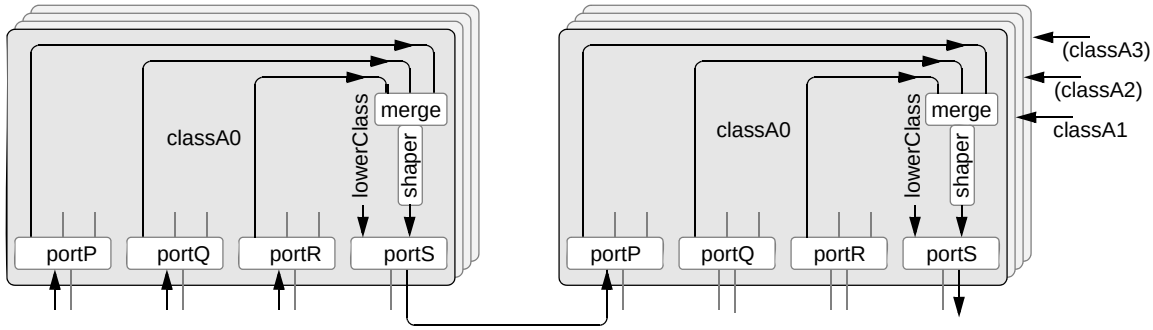


Figure 11.5—Reshaped bridge-traffic topology, without bunching control

This cost-reduced approach solves the bursting problem, without addressing the bunching problem. Within constrained topologies this can sometimes be sufficient to meet worst-case latency requirements.

11.1.4 Transmit ports

11.1.4.1 Transmit port structure

The transmit port is responsible for shaping classA traffic (to avoid bunching) and pacing classB/classC traffic (to avoid classC traffic starvation). Pacing and shaping algorithms assume functionally distinct queues within each transmit port, as illustrated in Figure 11.6.

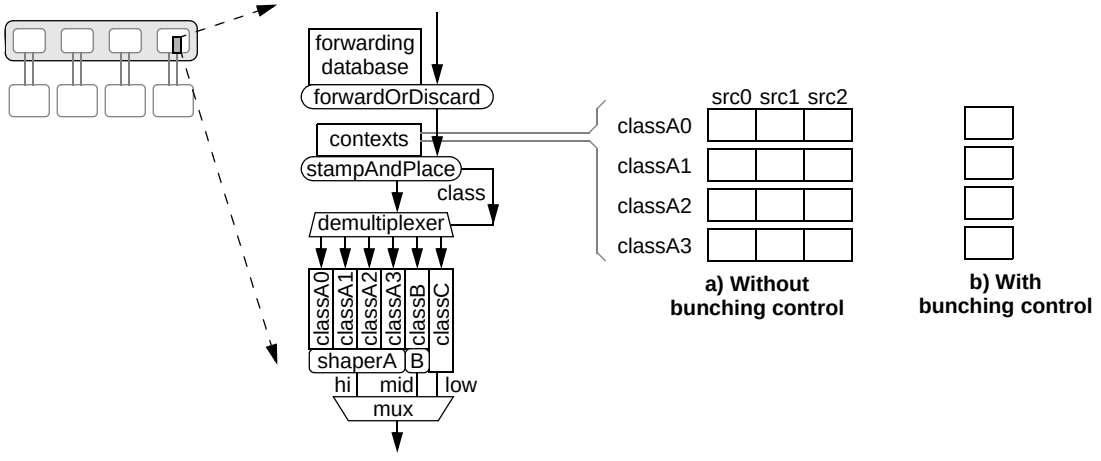


Figure 11.6—Transmit-queue structure

The intent of per-class shapers is to avoid priority inversions, wherein higher-class frames are delayed by the presence of concurrent lower-class traffic. Independent per-class shapers and queues allow enqueued higher-class and lower-class frames to be forwarded independently, thus avoiding priority inversions within queues.

The intent of per-source shapers is to avoid increasingly large cumulative bunching delays. The per-source reshaping eliminates bunches before merging, so that the pass-through bunching severity for 1-bridge and *n*-bridge flows are the same.

11.1.4.2 Enqueue reshaping contexts

The desired per-class latencies could not be guaranteed in the presence of classA traffic bunching. To avoid bunching, frames are shaped before being placed into classified transmit queues.

A shaper is responsible for attaching a time-stamp label to frames. With bursting control, a time-stamp shaper is logically associated with each source port and each classA traffic subclass (classA0, classA1, classA2, classA3). E.g, a four-port switch (which has three possible source ports) would have 12 time-stamp shapers on each transmit port. Without bursting control, a time-stamp shaper is logically associated with each classA traffic subclass (classA0, classA1, classA2, classA3). E.g, a N-port switch (which has N-1 possible source ports) would have 4 time-stamp shapers on each transmit port.

The purpose of a time-stamp shaper is to associate a time-stamp label with each queued frame. The time-stamp label represents a time in the future; the frame's transmission is deferred until the current time reaches the frame's time-stamp value. This facilitates the delayed forwarding of successive frames within each bunch, thus suppressing the bunching effects found on receive-link transmission.

The context for each time-stamp shaper is based on the frame's receive port and traffic class. While the context is considerably larger than that associated with strict per-port shapers, only one shaper (within each port) is ever active. Thus, context-switching per-port shaper instances represent a viable implementation technology.

11.1.4.3 Dequeue shaping and pacing

Transmit ports utilize a shaper and pacer, as illustrated as shaperA and B components within Figure 11.6. The purpose of these is to ensure forward progress of best-effort control traffic. In concept, this involves a two-step bandwidth partitioning mechanism:

a) The shaperA limits the cumulative classA and primary classB traffic to 75% of the link bandwidth.
   The intent is to ensure that 25% residual bandwidth remains available for lower-class traffic.
b) Pacer B partitions the residual 25% traffic equally between classB and classC traffic.
   This ensures that classB traffic is never starved, in the presence of 75% classA traffic.
   This ensures that classC traffic is not starved, in the presence of excess classB traffic.

11.1.5 Credit-based shapers and pacers

11.1.5.1 Credit-predictive shapers

The transmit shaper that schedules classA traffic (see Table 11.7, transition SHAPE-1) prioritizes frames based on their target transmission times. The shaper's credits are adjusted down or up, as illustrated in Figure 11.8. When frames are transmitted, the decrement value represents the size of a transmitted frame. Between transmissions, the credits increase based on the allowed per-class transmission rate.

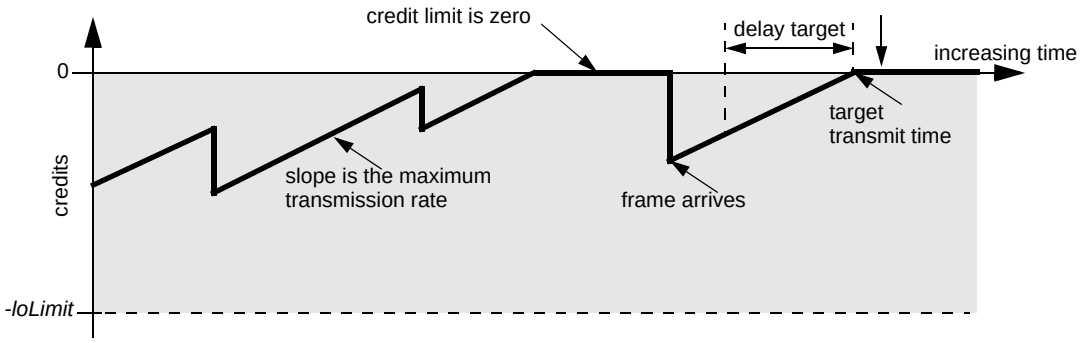


Figure 11.7—Credit-predictive shapers

Crossing above the zero threshold is disallowed, since this would encourage frame-transmission bunching. In normal operation, the credit value never goes below the *–loLimit* extreme.

When a frame arrives, the credits are evaluated to determine a target time for the frame transmission, based on credit losses from this and recent transmissions, divided by the allowed transmission rate. This provides a delay target time; the enqueued frame with the nearest delay-target time is selected for transmission.

In concept, the shaper consist of a token bucket. The number of credits in a token bucket is decremented by the size of each transmitted frame. The credits in the token bucket are incremented at the end of every credit update interval. A transmit time is targeted for when the debits (negative credits) are eliminated.

11.1.5.2 Credit-history shapers

The transmit shaper that limits levels of classA traffic (see Table 11.3, transitions BEST-1 to BEST-3) selectively enables transmissions based on an accumulated credits value. The shaper's credits are adjusted down or up, as illustrated in Figure 11.8. When frames are transmitted, the decrement value represents the size of a transmitted frame. Between transmissions, the credits increase based on the maximum allowed transmission rate.

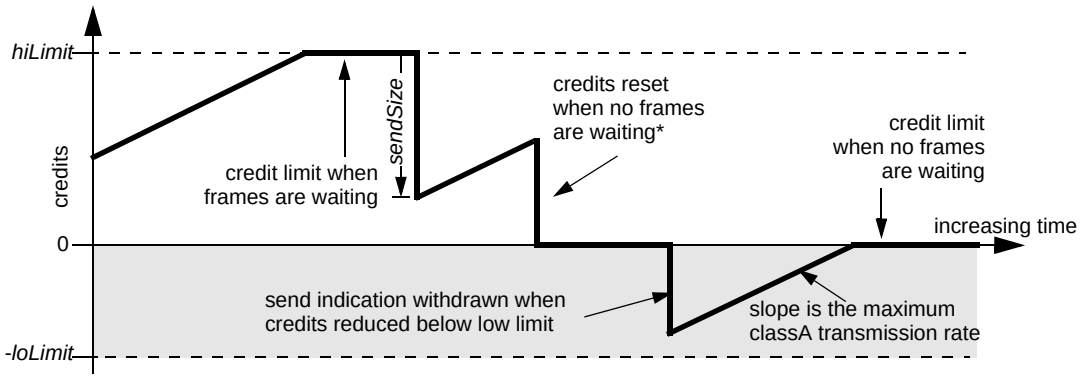


Figure 11.8—Credit-based shapers

Crossing below the zero threshold generates a rate-limiting indication, so that offered traffic can stop. By design, the credit value never goes below the *–loLimit* extreme. To bound the burst traffic after inactivity intervals, when no frames are ready for transmission, credits are reduced to zero (if currently higher than zero) and can accumulate to no more than the zero-value limit.

The *hiLimit* threshold limits the positive credits, to avoid overflow. When frames are ready for transmission (and are being blocked by transit traffic), credits can accumulate to no more than this *hiLimit* value.

In concept, the shaper consist of a token bucket. The number of credits in a token bucket is decremented by the size of each transmitted frame. The credits in the token bucket are continually increasing over time. A frame is only transmitted when the credits are positive.

11.1.5.3 Credit-based pacers

Although multiple pacers are specified within this working paper, the behavior of most pacers can be characterized by a common algorithm and instance-specific parameters (as done within RPR[B5]). The pacer's credits are adjusted down or up, as illustrated in Figure 11.9. The decrement and increment values typically represent sizes of debit and credit frames in each update interval, respectively.

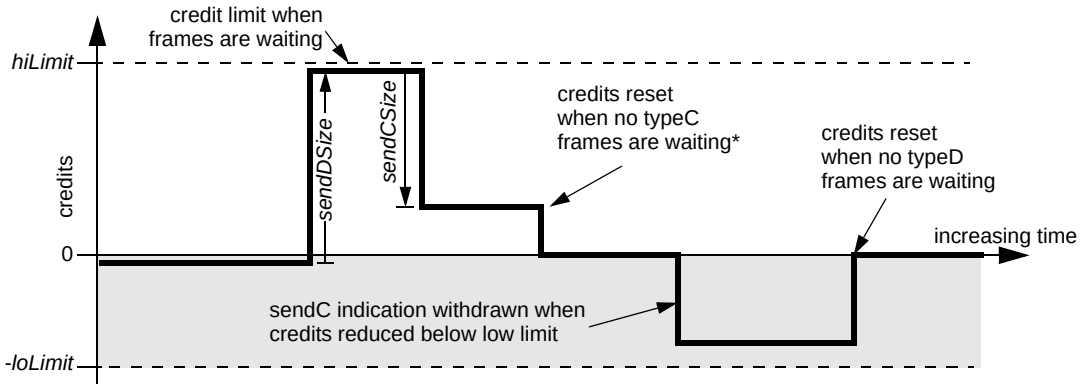


Figure 11.9—Pacer credit adjustments over time

Crossing below the zero threshold generates a rate-limiting indication, so that offered traffic can stop. By design, the credit value never goes below the *–loLimit* extreme. To bound the burst traffic after inactivity intervals, when no frames are ready for transmission, credits are reduced to zero (if currently higher than zero) and can accumulate to no more than the zero-value limit.

The *hiLimit* threshold limits the positive credits, to avoid overflow. When frames are ready for transmission (and are being blocked by transit traffic), credits can accumulate to no more than this *hiLimit* value.

In concept, the pacer consists of a token bucket. The credits in the token bucket are incremented by the size of each transmitted debit-frame. The number of credits in a token bucket is decremented by the size of each transmitted credit-frame. A credit-frame is only transmitted when the credits are positive; a debit-frame is only transmitted when the credits are negative.

11.2 Terminology and variables

11.2.1 Common state machine definitions

The following state machine inputs are used multiple times within this clause.

queue values
: Enumerated values used to specify shared queue structures.
  - QP_TX_PUSH—The transmit port's internal queue, where received frames are placed.
  - QP_TX_A0—The first of the output port's classA buffers.
  - QP_TX_A1—The second of the output port's classA buffers.
  - QP_TX_BP—The output port's classB queue.
  - QP_TX_CP—The output port's classC queue.
  - QP_TX_LINK—The output port's transmit-PHY queue.

11.2.2 Common state machine variables

One instance of each variable specified in this clause exists in each port, unless otherwise noted.

currentTime
: A value representing the current time.

framed
: The contents of a received frame, with supplemental information, as follows:
  - *frame*—The contents of a frame.
  - *sourcePort*—The source port that received the frame.
  - *txTime*—A time-stamp value representing the intended (bunching delayed) transmission time.

11.2.3 Common state machine routines

Max(*value1*, *value2*)
: Returns the numerically larger of two values.

11.2.4 Variables and routines defined in other clauses

This clause references the following variables and routines defined in Clause 7:

currentTime
: See 7.2.2.

Dequeue(*queue*)

Enqueue(*queue, frame*)

Min(*value1*, *value2*)
: See 7.2.3.

11.3 Pacing state machines

11.3.1 TransmitRx state machine

The TransmitRx state machine is responsible for enqueuing traffic (received on other ports and broadcast to all possible transmitter ports) for possible forwarding. An intent is to transfer each to the appropriate output queue.

The following subclauses describe parameters used within the context of this state machine.

11.3.1.1 TransmitRx state machine definitions

queue values
Enumerated values used to specify shared queue structures.
QP_TX_A0, QP_TX_A1
QP_TX_BP, QP_TX_CP
QP_TX_PUSH
See 11.2.1.

11.3.1.2 TransmitRx state machine variables

credits
A value that corresponds to accumulated credits since the last frame transmission.
class
A value that represents the frame's priority class.
currentTime
See 11.2.4.
delay
A value that represents the time delay assigned by the frame's shaper.
framed
See 11.2.2.
sPtr
Represents a pointer to shaper values.

11.3.1.3 TransmitRx state machine routines

ContextCheck(*sourcePort, class*)
Returns a pointer to the associated pacer context, with the following fields:
credit—The cumulative credit from past pacer activities.
lastTime—The last time the pacer was invoked.
loLimit—The low limit for shaper credits.
rate—The highest allowed rate of the paced traffic, in bytes-per-second.
Dequeue(*queue*)
See 11.2.4.
Enqueue(*queue, frame*)
Places the *frame* at the tail of the specified *queue* within the assumed port.
ForwardClass(*framed*)
The forwarding database is checked. If forwarding is enabled, the priority class is returned.
Otherwise, a NULL class value is returned. The following enumerated values are returned:
CLASS_A0—The associated multicast frame is forwarded as classA traffic.
CLASS_A1—The associated multicast frame is forwarded as classA traffic.
CLASS_A2—The associated multicast frame is forwarded as classA traffic.
CLASS_A3—The associated multicast frame is forwarded as classA traffic.
CLASS_B—The associated multicast frame is forwarded as classB traffic.
CLASS_C—The associated multicast frame is forwarded as classC traffic.
Max(*value1, value2*)
See 11.2.3.
Min(*value1, value2*)
See 11.2.4.

11.3.1.4 TransmitRx state table

The TransmitRx state machine is specified in Table 11.2. In the case of any ambiguity between the text and the state machine, the state machine shall take precedence. The notation used in the state table is described in 3.4.

Table 11.2—TransmitRx state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(framed = Dequeue(QP_TX_PUSH))!=NULL	1	—	FIRST
	—	2	—	START
FIRST	(class = ForwardClass(framed)) != NULL	1	sPtr = ContextCheck(framed.sourcePort, class);	NEXT
	—	2	—	START
NEXT	class == CLASS_A0	1	queue = QP_TX_A0;	SHAPE
	class == CLASS_A1	2	queue = QP_TX_A1;	
	class == CLASS_B	3	queue = QP_TX_BP;	FINAL
	—	4	queue = QP_TX_CP;	
SHAPE	—	1	credits = sPtr–>rate * (currentTime – sPtr–>lastTime); sPtr–>lastTime = currentTime; sPtr–>credit = Max(0, Min(sPtr–>loLimit, sPtr–>credit + backpay – Size(frame))); framed.txTime = currentTime – sPtr–>credit / sptr–>rate;	FINAL
FINAL	—	1	Enqueue(queue, frame);	START

START-1: If a frame has arrived, process that frame.
START-2: Otherwise, wait for the next frame to arrive.

FIRST-1: When forwarded frames, the shaper context is based on the source port and class.
FIRST-2: The non-forwarded frames are discarded.

NEXT-1: The classA0 frames are forwarded to the appropriate time-sensitive classA0 queue.
NEXT-2: The classA1 frames are forwarded to the appropriate time-sensitive classA1 queue.
NEXT-3: The classB frames are forwarded to the appropriate time-sensitive classB queue.
NEXT-4: The classC frames are forwarded to the appropriate time-sensitive classC queue.

SHAPE-1: ClassA frames are time stamped by shapers.
Shaper pacer parameters for each distinct {*class*, *source*} pair constrains bunching within each class.
Shaper parameters are decremented on transmissions and incremented by *backpay*, where *backpay* corresponds to the credits accumulated since the last applicable frame transmission (a specified in 11.1.5.2).
The last-updated time is then updated, to account for the incremented credits.
High and low limits are applied to the updated credits, thus avoiding burst-related positive credits.
Negative credits correspond to delayed transmission times, affiliated with output-port-queue frames.

FINAL-1: The received frames are placed into the appropriate queue.

11.3.2 TransmitTx state machine

The TransmitTx state machine is responsible for pacing/shaping classA traffic and shaping classB traffic destined for 1 Gb/s links. An intent is to support projected MTU-sized transfers and interleaved lower-class traffic, without exceeding the 1-cycle delay inherent with cycle-synchronous bridge-forwarding protocols.

The following subclauses describe parameters used within the context of this state machine.

11.3.2.1 TransmitTx state machine definitions

BPS
: The nominal link transmission rate, in bytes per second.

MTU
: The maximum frame size, in bytes.

queue values
: Enumerated values used to specify shared queue structures.
  QP_TX_A0, QP_TX_A1
  QP_TX_BP, QP_TX_CP
  QP_TX_LINK
  See 11.2.1.

TICK
: The amount of time between shaper updates.
  Range: [1 bytes transmit time, 16-bit transmit time]
  Default: 1 byte transmit time

11.3.2.2 TransmitTx state machine variables

best
: A value that represents the weight and identify of the next-best classA queue.
  goodness—The smallest *weight*×*wait* value associated with alternate classA transmissions.
  queue—The queue associated with the best futuristic encapsulated frame.

countA
: A speculative value of creditA, used only when the frame is qualitified for transmission.

countB
: A speculative value of creditB, used only when the frame is qualitified for transmission.

creditA
: A shaper credit whose positive value enables classA/classB primary transmissions.

creditB
: A shaper credit value whose positive and negative values enable secondary classB and classC transmissions respectively.

currentTime
: See 11.2.4.

frame
: The contents of a to-be-transmitted frame.

framed
: See 11.2.2.

hiLimitA
: A value that limits the cumulative *creditA* credits.
  Value: MTU.

hiLimitB
: A value that limits the cumulative *creditB* credits.
  Value: MTU.

limit

A value that limits the amount of transmitted primary classA/classB bandwidth.

loLimitA

A value that limits the cumulative *creditA* debits.

Value: MTU.

loLimitB

A value that limits the cumulative *creditB* debits.

Value: MTU.

tickTime

A value that defines when the time-tick interval ends.

11.3.2.3 TransmitTx state machine routines

Dequeue(*queue*)

See 11.2.4.

Unqueue(*queue*, *weight*, *&best*, *currentTime*)

Dequeues and returns the most overdue frame from the specified *queue*, excluding those frames whose scheduled transmission time is after the specified *currentTime* value.

framed—The oldest of the overdue frame.

NULL—No frame available.

In the presence of only futuristic frames, a *test*= *weight*×(*txTime*-*currentTime*) value is computed.

If *best.queue* is NULL or *test*<*best.goodness*, the *best.queue* and *best.goodness* components are updated to reflect the best alternate classA transmission queue.

Enqueue(*queue*, *frame*)

See 11.2.4.

Size(*frame*)

Returns the size of the specified frame.

StaleFrame(*frame*, *queue*)

Indicates whether the specified frame is stale and discardable, as specified by Equation 11.1.

0—The specified frame is not stale.

1—(Otherwise.)

```
// The value of "internal" depends on the class, as specified in Table 11.1.   (11.1)
(index = (QP_TX_A0 - queue),
(currentTime - framed.txTime) > (2 * (MTU + interval[index] * BPS)))
```

11.3.2.4 TransmitTx state table

The TransmitTx state machine is specified in Table 11.3. In the case of any ambiguity between the text and the state machine, the state machine shall take precedence. The notation used in the state table is described in 3.4.

Table 11.3—TransmitTx state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(currentTime – tickTime) >= TICK;	1	creditA = Min(hiLimitA, creditA + 0.75 * TICK * BPS); tickTime = currentTime ;	START
	TransmissionInProgress()	2	—	
	creditA < 0	3	—	FAIR
	—	4	best.queue = NULL;	BEST
BEST	(framed = Unqueue(queue= QP_TX_A0, 32, &best, currentTime)) != NULL	1	countA = Min(loLimitA, creditA – Size(framed));	NEAR
	(framed = Unqueue(queue= QP_TX_A1, 16, &best, currentTime)) != NULL	2		
	best.queue **!**= NULL && (framed = Dequeue(queue= best.queue)) != NULL	3		
	(framed = Dequeue(QP_TX_BP)) != NULL	4	creditA = Min(loLimitA, creditA – Size(framed));	FINAL
	—	5	creditA = 0;	START
FAIR	creditB >= 0 && (framed = Dequeue(QP_TX_BP)) != NULL	1	creditB = creditB – Size(framed);	FINAL
	creditB <= 0 && (framed = Dequeue(QP_TX_CP)) != NULL	2	creditB = creditB + Size(framed);	
	(framed = Dequeue(QP_TX_BP)) != NULL	3	creditB = 0;	
	(framed = Dequeue(QP_TX_CP)) != NULL	4		
	—	5	creditB = 0;	START
NEAR	StaleFrame(framed, queue)	1	—	START
	—	2	creditA = countA;	FINAL
FINAL	—	1	Enqueue(QP_TX_LINK, framed.frame);	START

START-1: Update the classA credits after each tick interval.
START-2: Wait for the queue to be emptied, so that something can be transmitted.
START-3: In the absence of classA credits, fairly transmit enqueued classB and classC frames.
START-4: Fairly service classB/classC when the classA/classB transmissions are disallowed.

BEST-1: If enabled and available, a classA0 frame is transmitted.
BEST-2: If enabled and available, a classA1frame is transmitted.
BEST-3: If available, a scheduled-for-the-future classA frame is transmitted.
BEST-4: If enabled and available, a classB frame is transmitted.
BEST-5: Since nothing is ready to be sent, the classA credits are cleared.

FAIR-1: If enabled and available, a classB frame is transmitted.
The *creditB* values is decremented by the transmitted frame size, to avoid classC starvation.
FAIR-2: If enabled and available, a classC frame is transmitted.
The *creditB* values is incremented by the transmitted frame size, to avoid classB starvation.
FAIR-3: If available, a classB frame is transmitted.
FAIR-4: If available, a classC frame is transmitted.
FAIR-5: Otherwise, no frame is transmitted.

NEAR-1: Stale frames, whose delivery times cannot be guaranteed, are discarded.
NEAR-2: Non-stale frames are not discarded.

FINAL-1: The next frame is transmitted and credits are updated accordingly.

Annexes

Annex A

(informative)

Bibliography

NOTE—This clause should be skipped on the first reading (continue with Annex B).
Although not finalized, this bibliography provides useful material for understanding this working paper.

[B1] IEEE 100, The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition.[1]

[B2] IEEE Std 802-2002, IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture.

[B3] IEEE Std 801-2001, IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.

[B4] IEEE Std 802.1D-2004, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges.

[B5] IEEE Std 802.17-2004, IEEE Standard for Local and Metropolitan Area Networks: Resilient packet ring (RPR) access method and physical layer specifications.

[B6] IEEE Std 1394-1995, High performance serial bus.

[B7] IEEE Std 1588-2002, IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.

[B8] IETF RFC 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis, David L. Mills, March 1992[2]

[B9] IETF RFC 2030: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, D. Mills, October 1996.

[B10] IETF RFC 2205: Resource Reservation Protocol (RSVP), R. Braden, L. Zhang, S. Berson, and S. Herzog, S. Jamin, October 1996.

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

[2]IETF publications are available via the World Wide Web at http://www.ietf.org.

Annex B

(informative)

Background material

B.1 Related standards

B.1.1 IEEE 1394 Serial Bus

As background, real-time features of an existing (and widely adopted on PCs) serial interface standard are summarized in this subclause: IEEE 1394-1995 High Performance Serial Bus. To avoid confusion with other serial buses (serial ATA, etc.), the term “SerialBus” is used within this annex to refer to this specific IEEE standard.

B.1.1.1 SerialBus topologies

Since its conception, SerialBus evolved from being a shared bus (like Ethernet) to a collection of point-to-point duplex links, as illustrated in Figure B.1. Arbitrary hierarchical topologies can be supported, but dotted-line redundant looping connections are only allowed in recent upgrades of the standard.

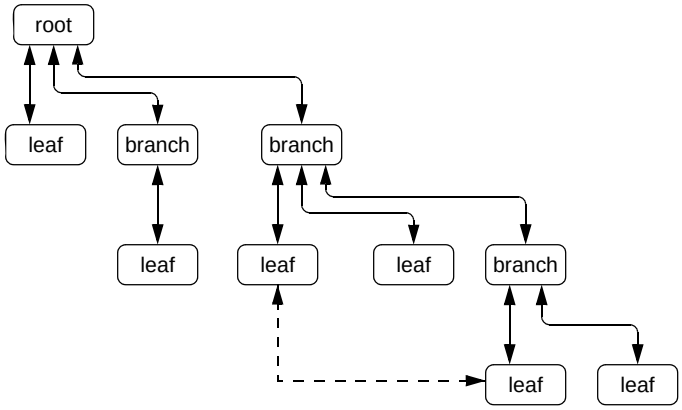


Figure B.1—SerialBus topologies

This physical duplex-link topology could, in concept, support concurrent non-overlapping data transfers. SerialBus only partially utilizes these capabilities (arbitration and data transfers can be overlapped), because its arbitration protocols were inherited from its initial conception as an arbitrated shared broadcast bus.

B.1.1.2 Isochronous data transfers

SerialBus isochronous traffic is transmitted at a 8 kHz rate, as illustrated by the 125 μs cycles within Figure B.2.

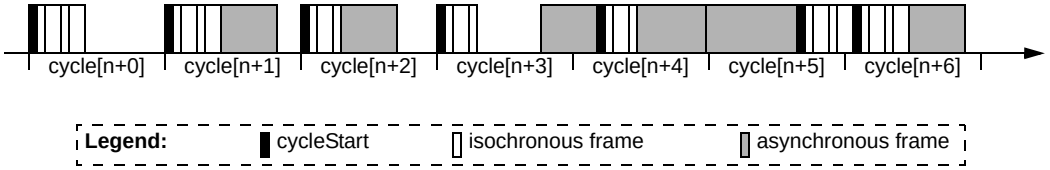


Figure B.2—Isochronous data transfer timing

In the absence of conflicting traffic, an 8kHz cycle starts with the transmission of a cycleStart frame, as illustrated in cycle[*n*+0]. The cycleStart frame triggers the sending of the isochronous frames that have been queued for cycle[*n*+0] transmission; these continue until all isochronous traffic has been sent.

After a cycle's isochronous traffic has been sent, one or more asynchronous transmissions are allowed, as illustrated in cycle[*n*+1].

Devices can be paused, compression rates can be variable, and connections can fail. For such reasons, the amounts of isochronous traffic within each cycle can vary below its scheduled limits, as illustrated in cycle[*n*+2].

The asynchronous traffic is not constrained to start at the end of a cycle, but can start at anytime that the frame is available and isochronous transfers are idle, as illustrated near the end of cycle[*n*+3]. If started near the end of a cycle, the isochronous transfer can be forced to start within the following cycle[*n*+4].

A large late-starting asynchronous frame can extend the start of isochronous transfers, so that spill-over into the next cycle is possible, as illustrated in cycle[*n*+5]. Since isochronous transfers have priority, the delay in the next isochronous cycle is reduced, and the isochronous traffic completes within the boundaries of cycle[*n*+6].

B.1.1.3 Isochronous reservations

Even the best of isochronous transfers fails when the offered load exceeds the link capacity. To eliminate this possibility, isochronous bandwidth is reserved before being consumed. On a single bus (of up to 64 stations), reservations are controlled through access to compare&swap register, which all isochronous stations provide, although only one is selected to be used (based on the largest populated device address).

On a multiple bus topology (buses interconnected through bridges), reservations management is more complex. In this case, frames are passed from the source to its desired-to-be-connected destination(s), reserving reservations along the data-transmission path. As is true on a single bus, reservation requests are rejected when insufficient bandwidth capacity remains. This is not described in the baseline 1394 specification, but is described in a follow-on P1394.1 draft (currently progressing through Sponsor ballot).

B.1.1.4 SerialBus experiences

Experiences, as follows:

a) Cycle slip. Cycle-slip reduces design complexity, permits transmissions of large asynchronous frames, and improves asynchronous traffic throughput. Transmission precision is unnecessary: error in the cycleStart transmission time is encoded within that frame, allowing clock-slave devices to accurately adjust their phase-lock-loops, regardless of observed cycleStart transmission times.

b) Cycle time. An 8 kHz cycle rate represents a good trade-off between efficiency (the overhead is less, when cycle times are longer) and latency (the latency is less, when cycle times are longer).

c) Pseudo frames. The SerialBus isochronous frames have a distinct (6-bit channel number) addressing scheme. In hindsight, using a standard frame header (destination address and source address) would have many benefits, including the simplification of bridges between segments.

d) Service classes. SerialBus has evolved to support three classes of traffic: isochronous, prioritized asynchronous, and baseline asynchronous. These are roughly equivalent to the classA, classB, and classC service classes defined for RPR (see B.1.2).

B.1.2 Resilient packet ring (RPR)

As background, the time-sensitive capabilities associated with IEEE P802.17 Resilient packet ring (RPR) are summarized in this subannex. RPR is a metropolitan area network (MAN) that can be transparently bridged to Ethernet.

B.1.2.1 RPR rings

RPR employs a ring structure using unidirectional, counter-rotating ringlets. Each ringlet is made up of links with data flow in the same direction. The ringlets are identified as ringlet0 and ringlet1, as shown in Figure B.3.

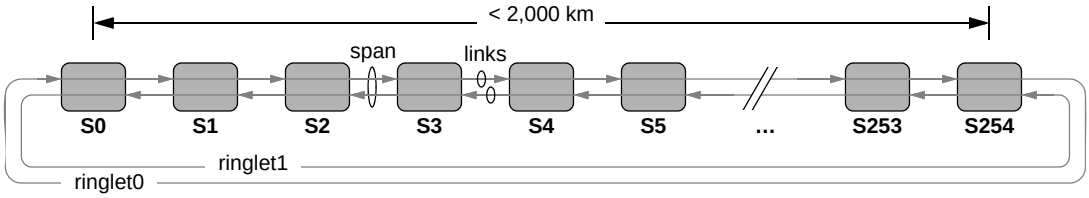


Figure B.3—RPR rings

Stations on the ring are identified by an IEEE 802 48-bit MAC address. All links on the ring operate at the same data rate, but may exhibit different delay properties. Ring circumference of less than 2,000 kilometers. are assumed.

The portion of a ring bounded by adjacent stations is called a span. A span is composed of unidirectional links transmitting in opposite directions.

B.1.2.2 RPR resilience

RPR stations are resilient, in that communications can continue in that operations continue in the presence of single-point failures, as illustrated in Figure B.4. Resilient features can recover from failed links by bypassing the frame-manipulation portions of a partially failed station (see Figure B.4-b), thus avoiding a failed station (see Figure B.4-c and Figure B.4-d) or a failed span (see Figure B.4-e and Figure B.4-f).

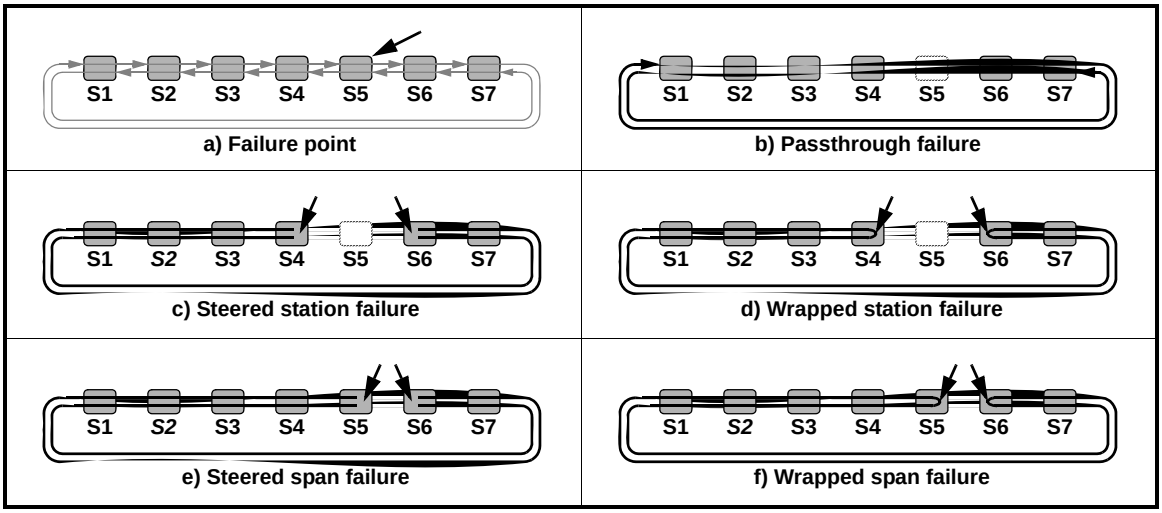


Figure B.4—RPR resilience

B.1.2.3 RPR spatial reuse

RPR efficiently strips local unicast frames at their destination, so that bandwidth on unaffected links is available for other frame transfers, as illustrated in Figure B.5-a. A unicast frame is added by the source station, and is stripped at the destination station. The frame is normally copied at the destination station for delivery to the local MAC client or MAC control entity. If ringlet selection is based on shortest hop-count, a response frame is likely to take an opposing ringlet path, as illustrated in Figure B.5-b.

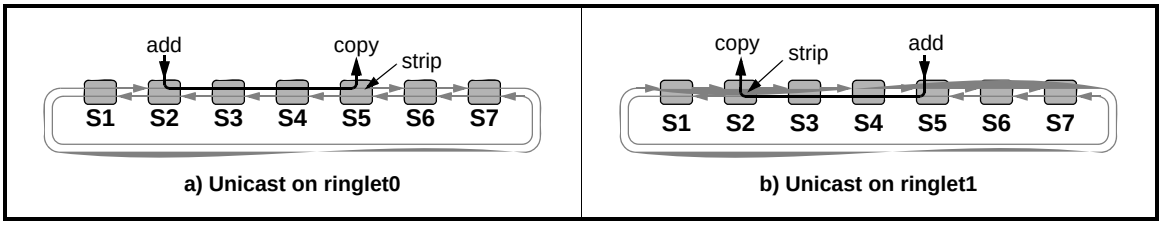


Figure B.5—RPR destination stripping

The RPR frame transmissions on one link are largely independent of frame transmissions on other link. This allows per-link bandwidths to be utilized beyond that possible with IEEE Std 802.5-1998 Token Ring or ANSI FDDI ring based LAN technologies. Spatial reuse is illustrated in Figure B.6.

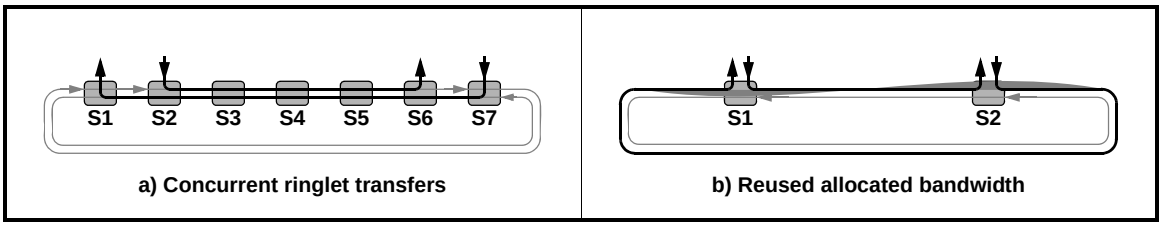


Figure B.6—RPR spatial reuse

Concurrent per-ringlet transmissions (see Figure B.6-a) allow stations bandwidths to exceed individual link capacities. The effective bandwidths of non-overlapping transfers (see Figure B.6-b) are similarly improved.

B.1.2.4 RPR service classes

RPR provides transit queues, which allow received traffic to be queued during a station's frame transmission, as illustrated in Figure B.7. The highest priority frames are classA and have their own bypass buffer; the lower priority frames are classB and classC, and share the use of a distinct bypass buffer. To minimize the classA latencies, servicing of the classA buffer has precedence over servicing of the classB/classC buffer.

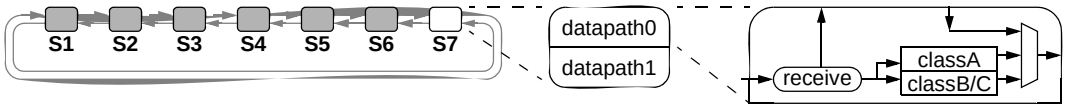


Figure B.7—RPR service classes

During the initial phases of investigation, techniques for allowing newly-arrived classA traffic to preempt an active classB/classC frame transmission were considered. While such techniques are practical, the metropolitan area networks (MANs) environments limits the effectiveness of such techniques; at these longer distances, the link delays can often exceed the retransmission-blocked delays within individual stations.

Annex C

(informative)

Encapsulated IEEE 1394 frames

To illustrate the sufficiency and viability of the RE isochronous services, the transformation of IEEE 1394 packets is illustrated. A connection between an IEEE 1394 talker, IEEE 1394 adapter, intermediate Ethernet links, IEEE 1394 adapter, and an IEEE 1394 listener is assumed.

C.1 Hybrid network topologies

C.1.1 Supported IEEE 1394 network topologies

This annex focuses on the use of RE to bridge between IEEE 1394 domains, as illustrated in Figure C.1. The boundary between domains is illustrated by a dotted line, which passes through a SerialBus adapter station.

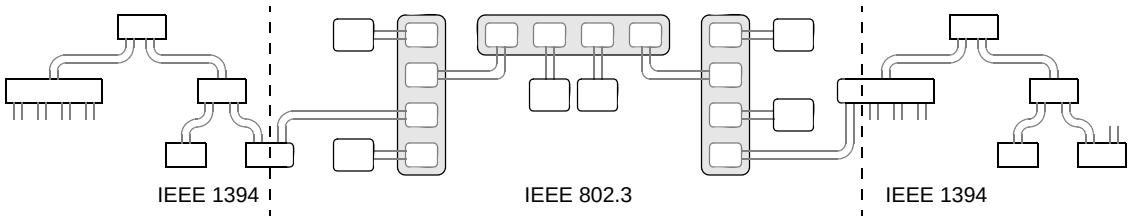


Figure C.1—IEEE 1394 leaf domains

C.1.2 Unsupported IEEE 1394 network topologies

Another approach would be to use IEEE 1394 to bridge between IEEE 802.3 domains, as illustrated in Figure C.2. While not explicitly prohibited, architectural features of the topology-supportive adapters and encapsulated-frame formats are beyond the scope of this working paper.

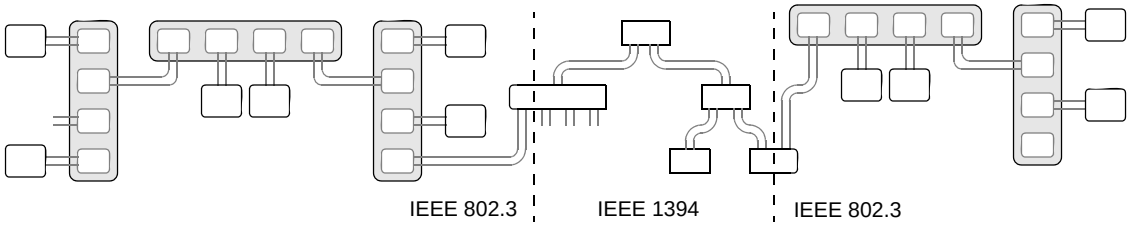


Figure C.2—IEEE 802.3 leaf domains

C.2 1394 isochronous frame formats

C.2.1 1394 isochronous frame formats

An IEEE 1394 isochronous frame contains header and payload components, as illustrated by Figure C.3. While all components could be encapsulated into an Ethernet frame, some of these fields would be redundant (with fields in the encapsulating frame) or unnecessary.

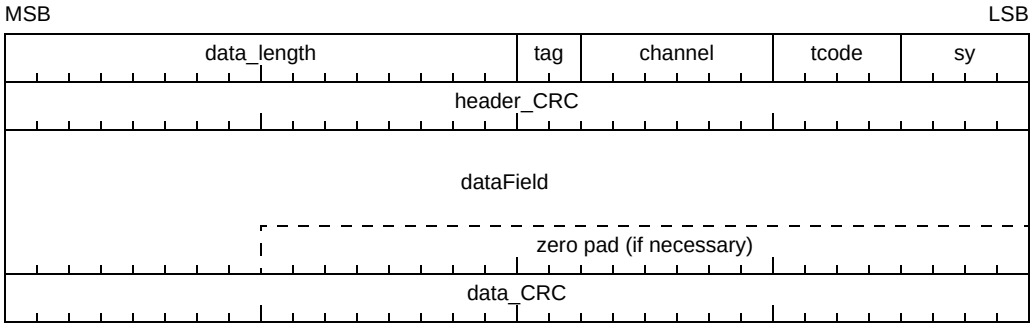


Figure C.3—IEEE 1394 isochronous packet format

C.2.2 Encapsulated IEEE 1394 frame payload

For uniframe groups, the IEEE 1394 isochronous frames are modified slightly and placed within an Ethernet *serivceDataUnit*. The format of this *serviceDataUnit* is illustrated by Figure C.4.

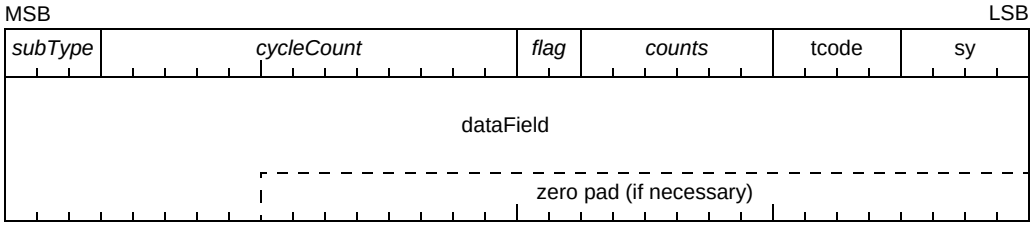


Figure C.4—Encapsulated IEEE 1394 frame payload

C.2.2.1 *subType*: A 3-bit field that distinguishes encapsulated 1394 frames from other formats with the same *protocolType* specifier.

C.2.2.2 *cycleCount*: A 13-bit field that identifies the isochronous cycle during which this frame was transmitted. For the first frame within any group, this information is needed to perform CIP header updates (see C.4). These fields also provide error-detecting consistency checks.

C.2.2.3 *flag*: A 2-bit field that distinctively identifies the frame type, as specified in Table C.1.

Table C.1—*flag* field values

Value	Name	Description
0	ONLY	Only frame within a uniframe group
1	LAST	Final frame within a multiframe group
2	CORE	Intermediate frame within an multiframe group
3	LEAD	First frame within a multiframe group

C.2.2.4 *counts*: A 6-bit field that identifies additional frame-group parameters, as specified in Table C.2. When interpreted as a *partCount* value, this effectively identifies the number of zero-pad bytes. When interpreted as a *frameCount* value, the values of {*n*-1,*n*-2,…,1} label the first through next-to-last frames of an *n*-frame multiframe group.

Table C.2—*counts* field values

flag	Name	Description
ONLY	*partCount*	The LSBs of the residual data_length field.
LAST		
CORE	*frameCount*	A sequence identifier for frames within the group
LEAD		

C.2.2.5 *dataField*: For a uniframe group, the contents of the SerialBus ‘data field’ bytes.

C.3 Frame mappings

C.3.1 Synchronous frame mappings

Adapters are required to manage differences between IEEE 1394 isochronous packets and RE frames, as illustrated in Figure C.5.

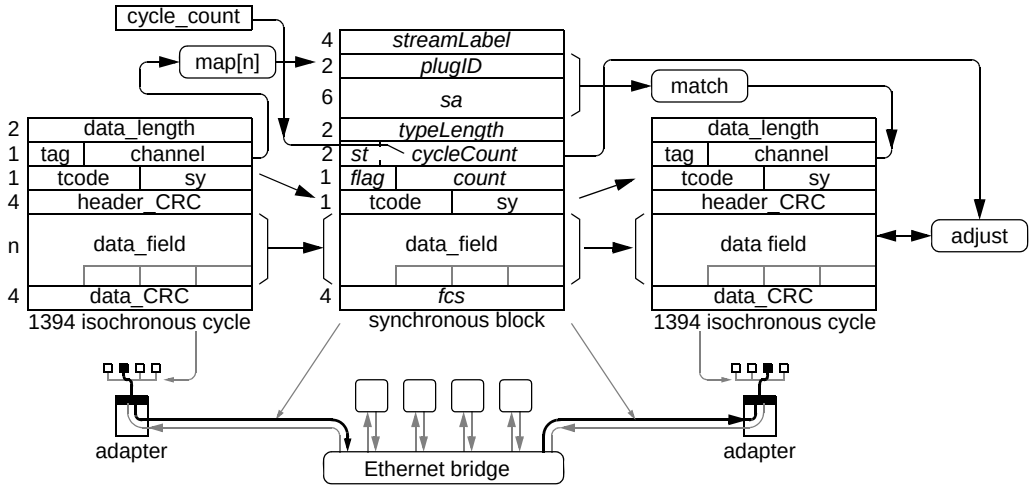


Figure C.5—Conversions between IEEE 1394 packets and RE frames

The IEEE 1394 to Ethernet frame translation involves the following:

a) The IEEE 1394 data_length field is discarded
(The data_length information can be reconstructed from the length of the received frame.)

b) The IEEE 1394 tag field is ignored (this connection context is known to higher layer software).

c) The IEEE 1394 channel field becomes an index into an array of communication contexts.
The selected context provides the *plugID* value, the least-significant portion of the Ethernet *da*.

d) The IEEE 1394 isochronous transmission cycle number is copied to the Ethernet *cycleCount* field.
(The cycle number is the cycle_time_data.cycle_count field from the preceding cycle-start packet.)

e) The IEEE 1394 *tcode* and *sy* fields are copied to the corresponding Ethernet fields.

f) The data_length, header_CRC, and data_CRC fields are checked; if any are found to be inconsistent, no RE frame is created (the presumed to be corrupted frame is dropped).

NOTE — Unlike IEEE 1394, no synchronous frame transformations are required when passing through bridges. This is consistent with 802.3 specifications, which leave frames unmodified when passing through bridges.

The Ethernet to IEEE 1394 frame translation involves the following:

a) Invalid Ethernet frames (multicast *sa* address, too-short or too-long, or bad *fcs*) are discarded.

b) The IEEE 1394 data_length field is derived from the Ethernet frame length.

c) The context with the matching *streamId* (*sa* concatenated with *plug*) values is selected.
This context provides the provides the channel field value.

d) The IEEE 1394 tag and tcode fields are set to identify isochronous IEEE 1394 packets.

e) The IEEE 1394 tcode and sy fields are copied from the Ethernet frame.

f) The IEEE 1394 data_field is directly mapped to the RE content field.
(IEC61883-type content may have its synchronization fields updated as needed, see C.4.)

g) The IEEE 1394 header_CRC and data_CRC fields are computed.

C.3.2 Multiframe groups

To avoid exceeding the maximum Ethernet frame size, large frames are decomposed into multiframe groups. The initial frames within the multiframe group are distinctively identified by their *counts* values, as illustrated in Figure C.6.

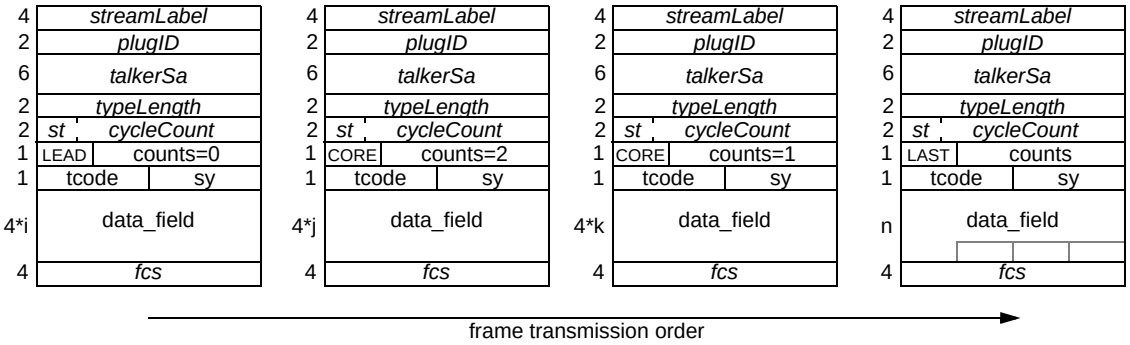


Figure C.6—Multiframe groups

The final frame within the group is identified by its distinctive *flag*=LAST identifier. For this frame, the *counts* field specifies the number of data bytes within the frame, modulo 64.

C.4 CIP payload modifications

Isochronous 1394 data packets may conform to a common isochronous packet (CIP) format, as defined by IEC 61883/FIS. The presence of a CIP format is indicated by a tag=1 bit in the Serial Bus isochronous packet header, as illustrated in Figure C.7. The white shading identifies those fields (when present and valid) are modified when passing through a RE-to-1394 adapter.

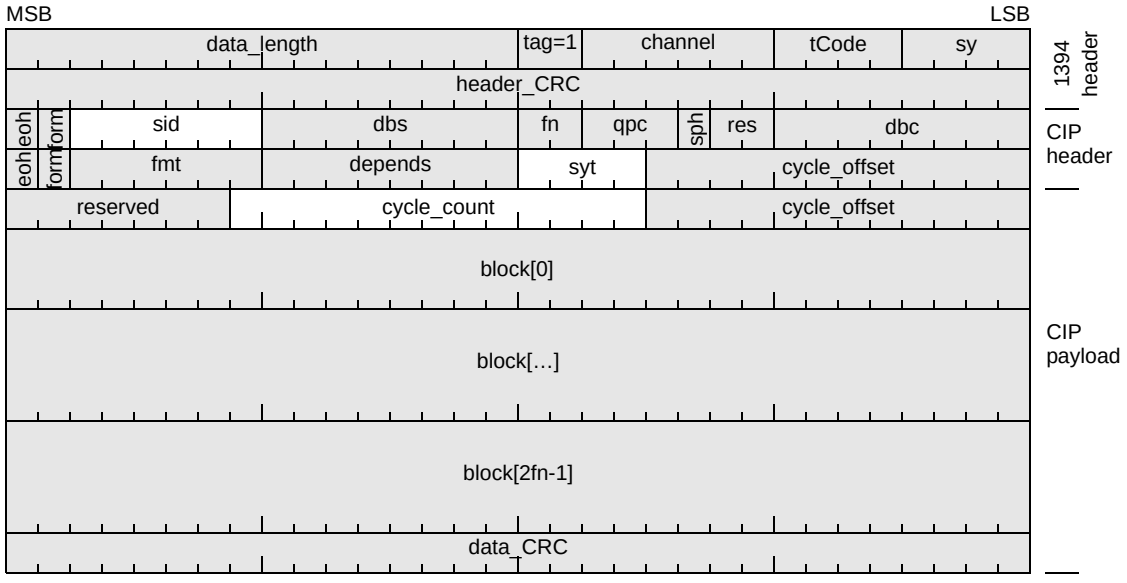


Figure C.7—Isochronous 1394 CIP packet format

The *sid* field must be set to the physical ID of the talking portal. This allows the listener to identify the bridge's talker portal.

Two-quadlet CIP headers may also contain absolute time stamp information or indicate its presence elsewhere in the packet's data payload. Absolute time stamps may be found in one or more places in isochronous:

— the *syt* field of the second quadlet of the CIP header if the *fmt* field in that quadlet has a value between zero and $1F_{16}$, inclusive; and
— the *cycle_count* and *cycle_offset* fields of all of the source packet headers (SPH) within the isochronous subaction.

Both of these time stamps are specified as absolute values that specify a future cycle time. Since isochronous subactions experience delays when routed over RE, these time stamps must be adjusted by the difference in cycle times between the talker and the RE-to-1394 bridge. The delay, in units of cycles, is the difference between the talker and 1394 adapter's transmission times, as specified in Equation 3.2.

```
latency= (adapter.sendCycle - syncBock.talkerCycle);                    (3.1)
```

When the *syt* or cycle_count fields are present, their adjustments are specified by Equation 3.2. Because IEEE 1394 constrains cycle_count to the range zero to 7999, inclusive, the time stamp adjustments must be performed modulus 8000

```
transmitted.syt = (received.syt + latency) % 8000;                      (3.2)
transmitted.cycle_count = (received.cycle_count + latency) % 8000;      (3.3)
```

C.4.1 Time-of-day format conversions

The difference between RE and IEEE 1394 time-of-day formats is expected to require conversions within the RE-to-1394 adapter. Although multiplies are involved in such conversions, multiplications by constants are simpler than multiplications by variables. For example, a conversion between RE and IEEE 1394 involves no more than two 32-bit additions and one 16-bit addition, as illustrated in Figure C.8.

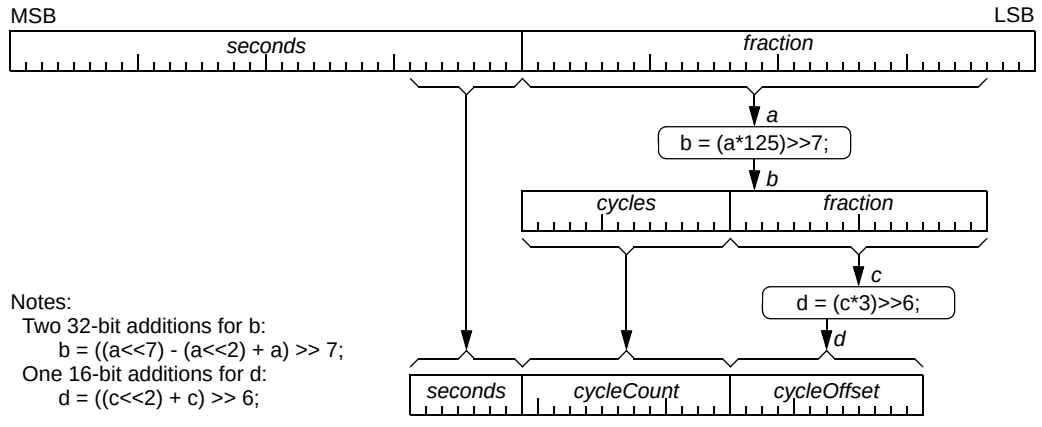


Figure C.8—Time-of-day format conversions

C.4.2 Grand-master precedence mappings

Compatible formats allow either an IEEE 1394 or IEEE 802.3 stations to become the network's grand-master station. While difference in format are present, each format can be readily mapped to the other, as illustrated in Figure C.9:

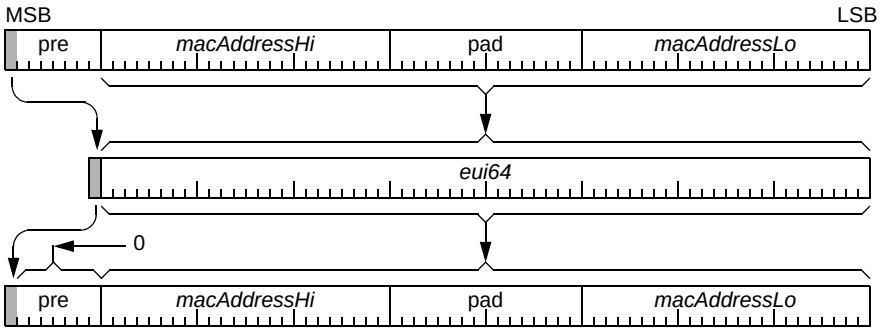


Figure C.9—Grand-master precedence mapping

Annex D

(informative)

Review of possible alternatives

D.1 Clock-synchronization alternatives

D.1.1 Statistical averaging

Wide-area network based protocols distribute time by enclosing time-stamp values in specialized calibration frames. Higher level frame-processing protocols are responsible for determining the average transmission delays through the interconnect, so that calibration-frames can be used for accurate time-synchronization purposes.

The frame transmission latency is highly variable, based on delays incurred when waiting behind other previously-queue frames. Long-term averaging is typically used to cope with nonrandom delays, whether they be periodic, biased, or time-of-day dependent.

The use of long-time averages has limited applicability within the home, where small numbers of streams can exhibit very non-random statistical behaviors. Furthermore, long-term averaging intervals restricts transient-event response times, such as the insertion or removal of associated clock-synchronized devices.

D.1.2 Phase-locked synchronization

Local-area network based protocols, such as IEEE Std 1588, specify communication protocols for communicating timer-difference errors from a local clock-master station to its neighboring clock-slave station. However, this standard does not define how the clock-slave station compensates its values to track the time reference of the neighboring clock-master station.

The most common method of synchronizing clock-master and clock-slave devices involves phase-lock-loop (PLL) circuits. Such circuits integrate sensed differences between the clock-master and clock-slave devices, using these integrated values to adjust the clock-slave operating frequency.

The clock-slave resident PLLs are useful for reducing the transmission-induced timing-error jitters. However, the response time of a cascaded set of PLLs degrades as the number of cascaded devices increases. Also, the dynamics of more-responsive (gain peaking) cascaded PLL can be undesirable, causing the deviations of later stages to exponentially increase with their distance from the source, a characteristic commonly called the whip-lash effect.

D.1.3 Offset-locked synchronization

Another possible IEEE 1588 synchronization technique involves adding an offset value to the clock-slave device, where the value of that offset is based on the time differences sensed between the clock-master and clock-slave stations.

Constantly updated offsets ensures tracking of the clock-slave to the clock-master, without the response-time and whiplash effects normally associated with PLLs. However, since the clock rates remain unchanged, clock drifts can cause significant forward or backward jumps of the synchronized clock-slave

timer. These discontinuities and transmit-time uncertainties can limit the accuracies of the slave-resident timer values.

D.2 Pacing alternatives

D.2.1 Higher level flow control

Higher layer protocols (such as the flow-control mechanisms of TCP) throttle the source to the bandwidth capabilities of the destination or intermediate interconnect. With the appropriate excess-traffic discards and rate-limiting recovery, such higher layer protocols can be effective in fairly distributing available bandwidth.

For real-time applications, however, the goal is to limit the number of talkers (so they can each have sufficient bandwidth), not to distribute the insufficient bandwidth fairly.

D.2.2 Over-provisioning

Over-provisioning involves using only a small portion of the available bandwidth, so that the cumulative bandwidth of multiple applications rarely exceeds that of the interconnect. This technique works well when frame losses are expected (voice over IP delays and gaps are similar to satellite-connected long distance phone calls) or when large levels of cumulative bandwidth ensure a tight statistical bound for maximum bandwidth utilization.

For most streaming applications within the home, however, frame losses are viewed as equipment defects (stutters in video or audio streams), which correspond to eventual loss of brand name values. Also, the existing kinds of transfers in a home (disk-to-disk, memory-to-display, tuner-to-display, multi-station games, etc.) do not (nor should not) have bandwidth limits.

D.2.3 Strict priorities

Existing networks can assign priority levels to different classes of traffic, effectively ensuring delivery of one before delivery of the other. One could provide the highest priority to the video traffic (with large bandwidth requirements), a high priority to the audio traffic (lower bandwidth, but critical), and the lowest priority level to file transfers. A typical number of priorities is eight.

Strict priority protocols are deficient in that the priorities are statically assigned, and the assignments (based on traffic class) often do not correspond to the desires of the consumer (my PBS show, rather than my teenager’s games, perhaps). For example, priorities could result in transmission of two video streams, but not the audio associated with either.

Strict priority protocols usually assign fixed application-dependent priorities, assigning one priority to video and another to audio, for example. Mixed traffic (such as video streams with encapsulated audio) are not easily classified in this manner.

D.3 IEEE 1394 alternative

Isochronous data transfers are well supported by the IEEE 1394 Serial Bus family of standards. This IEEE standards family (also called FireWire and iLink) is herein referred to simply as IEEE 1394.

Existing consumer equipment (digital camcorders, current generation high-definition televisions (HDTVs), digital video cassette recorders (DVCRs), digital video disk (DVD) recorders, set top boxes (STBs), and computer equipment intended for media authoring) support the IEEE 1394 interconnect. While some versions limit cable lengths to 4.5 meters, other physical layers support considerably longer lengths. A hub-like connection of IEEE 1394 devices supports seamless real-time services.

Although IEEE 1394 supports longer-reach physical layers, not all devices are compatible with these physical layers, or the distinct connectors associated with distinct physical layers. The RE protocols are based on Ethernet connections, a vast majority of which are based on 100 meter cables and the RJ-45 connector.

The IEEE 1394 isochronous packet addressing was designed with single-bus topologies in mind, which complicates the design of such bus bridges. The RE synchronous frames are designed with multiple stations and bridges in mind.

IEEE 1394 packets are differentiated by bus-local channel identifier, which must be allocated from a central per-bus resources and updated when isochronous packets pass through bridges. Mechanism must therefore be defined to agree upon the central per-bus resource, from among multiple available resources, and to renegotiate that agreement when any of the current central per-bus resources are removed.

Furthermore, absolute time stamps within some IEEE 1394 isochronous packets must be adjusted when passing through bridges. Such data-format dependent adjustments complicate bridge designs; their data-format dependent nature would most likely inhibit their successful adoption within an Ethernet bridge standard.

Annex E

(informative)

Time-of-day format considerations

To better understand the rationale behind the ‘extended binary’ timer format, other formats are evaluated and compared within this annex.

E.1 Possible time-of-day formats

E.1.1 Extended binary timer formats

The extended-binary timer format is used within this working paper and summarized herein. The 64-bit timer value consist of two components: a 32-bit *seconds* and 32-bit *fraction* fields, as illustrated in Figure 5.1.

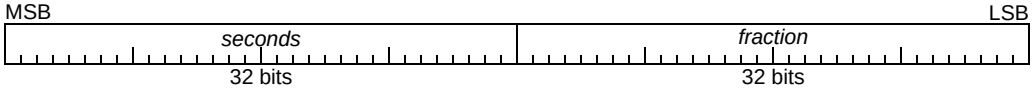


Figure 5.1—Complete seconds timer format

The concatenation of 32-bit *seconds* and 32-bit *fraction* field specifies a 64-bit *time* value, as specified by Equation E.1.

$$time = seconds + (fraction / 2^{32}) \quad \text{(E.1)}$$

Where:

seconds is the most significant component of the time value (see Figure 5.1).
fraction is the less significant component of the time value (see Figure 5.1).

E.1.2 IEEE 1394 timer format

An alternate “1394 timer” format consists of *secondCount*, *cycleCount*, and *cycleOffset* fields, as illustrated in Figure E.2. For such fields, the 12-bit *cycleOffset* field is updated at a 24.576MHz rate. The *cycleOffset* field goes to zero after 3171 is reached, thus cycling at an 8kHz rate. The 13-bit *cycleCount* field is incremented whenever *cycleOffset* goes to zero. The *cycleCount* field goes to zero after 7999 is reached, thus restarting at a 1Hz rate. The remaining 7-bit *secondCount* field is incremented whenever *cycleCount* goes to zero.

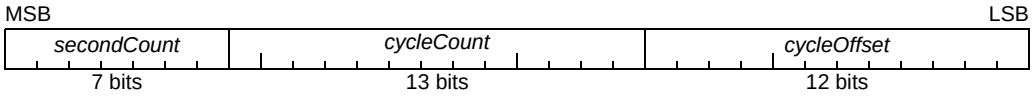


Figure E.2—IEEE 1394 timer format

E.1.3 IEEE 1588 timer format

IEEE 1588 timer format consists of seconds and nanoseconds fields components, as illustrated in Figure E.3. The nanoseconds field must be less than 10^9; a distinct *sign* bit indicates whether the time represents before or after the epoch duration.

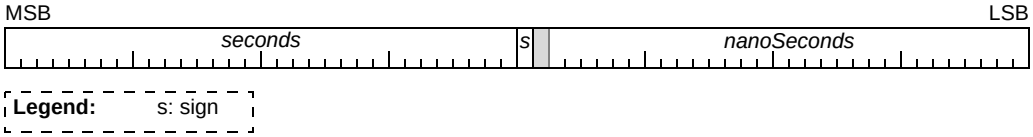


Figure E.3—IEEE 1588 timer format

E.1.4 EPON timer format

The IEEE 802.3 EPON timer format consists of a 32-bit scaled nanosecond value, as illustrated in Figure E.4. This clock is logically incremented once each 16 ns interval.

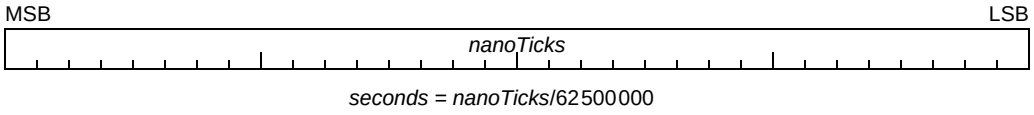


Figure E.4—EPON timer format

E.1.5 Compact seconds timer format

An alternate “compact seconds” format could consist of 8-bit *seconds* and 24-bit *fraction* fields, as illustrated in Figure E.5. This would provided similar resolutions to the IEEE 1394 timer format, without the complexities associated with its binary coded decimal (BCD) like encoding.

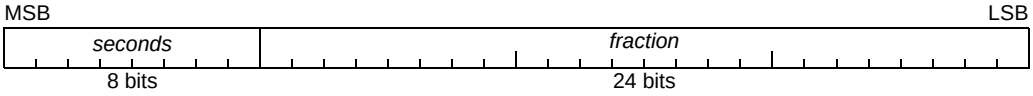


Figure E.5—Compact seconds timer format

E.1.6 Nanosecond timer format

An alternate “nanosecond” format could consists of 2-bit *seconds* and 30-bit *nanoSeconds* fields, as illustrated in Figure E.6.

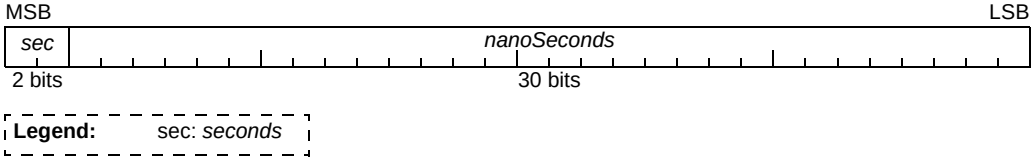


Figure E.6—Nanosecond timer format

E.2 Time format comparisons

To better understand the relative benefits of different time formats, the relevant properties are summarized in Table E.1. Counter complexity is not included in the comparison, since the digital logic complexity (see 7.1.16) is comparable for all formats.

Table E.1—Time format comparison

Name	Subclause	Range	Precision	Arithmetic	Seconds	Defined standards
Column	—	1	2	3	4	5
extended binary	TBD	136 years	232 ps	Good	Good	RFC 1305 NTP, RFC 2030 SNTPv4
IEEE 1394	E.1.2	128 s	30 ns	Poor	Good	IEEE 1394
IEEE 1588	E.1.3	272 years	1 ns	Fair	Good	IEEE 1588
IEEE 802 (EPON)	E.1.4	69 s	16 ns	Good	Poor	IEEE 802.3
compact seconds	E.1.5	256 s	60 ns	Best	Good	—
nanoseconds	E.1.6	4 s	1 ns	Best	Poor	—

Column 1: A desirable property is the support of a wide range of second values, to eliminate the need for defining/coordinating/implementing auxiliary seconds-synchronization protocols. The 136-year range of the extended binary format is sufficient for this purpose.

Column 2: A desirable property is a fine-grained resolution, sufficient to measure each bit-transmission times. The ‘extened binary’ provides the most precision; exceeds the resolution of expected cost-effective time-capture circuits.

Column 3: Computation of time differences involves the subraction of two timer-snapshot values. Subtraction of ‘extended binary’ numbers involving standard 64-bit binary arithmetic; no special field-overlow compensations are required. Only the less precise ‘compact seconds’ and nanoseconds formats are simpler, due to the reduced 32-bit size of the timer values.

Column 4: Time values must oftentimes be compared to externally provided values (e.g., timers extracted from GPS or stratum-clock sources). For these purposes, the availability of a seconds component is desired. The ‘extended binary’ format provides a seconds component that can be easily extracted or such purposes.

Annex F

(informative)

Bursting and bunching considerations

F.1 Topology scenarios

F.1.1 Bridge design models

The sensitivity of bridges to bursting and bunching is highly dependent on the queue management protocols within the bridge. To better understand these effects, a few bridge design models are evaluated, as illustrated in Figure F.1.

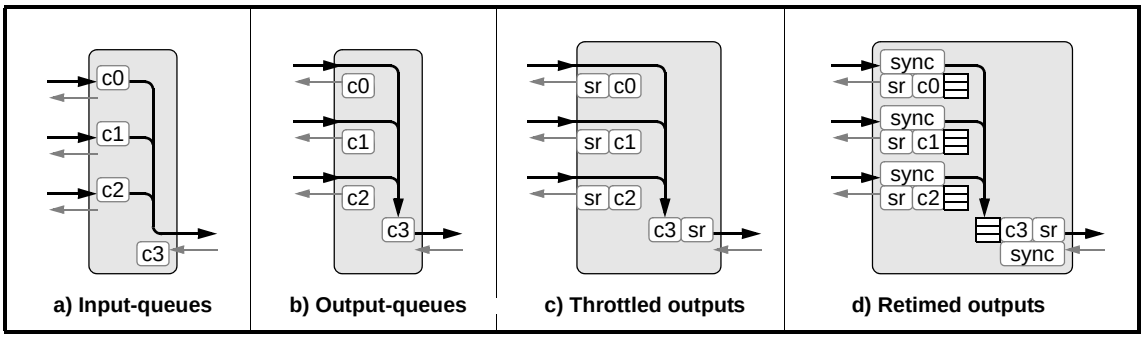


Figure F.1—Bridge design models

The input-queue design (see Figure F.1-a) assumes that frames are queued in receive buffers. The transmitter accepts frames are from the receivers, based on service-class precedence. In the case of a tie (two receivers can provide same-class frames), the lowest numbered receive port has precedence. This model best illustrates nonlinear bunching problems.

The output-queue design (see Figure F.1-b) assumes that received frames are queued in transmit buffers. Within each service class, frames are forwarded in FIFO order. This model best illustrates linear bunching problems (for steady flows), but also exhibits nonlinear bunching (for nonsteady flows).

The throttled-output design (see Figure F.1-c) is an enhanced output-queue model, with an output shaper to limit transmission rates. The purpose of the output shaper is to ensure sufficient nonreserved bandwidth for less time-sensitive control and monitoring purposes. The model illustrates how shapers can worsen the output-queue bridge's bunching behaviors.

The retimed-outputs design (see Figure F.1-d) reduces (and can eliminate) bunching problems by detecting late-arrival frames at the receivers. Several synchronous-cycle buffers are provided at the transmitters, to compensate for transmission delays in the received data.

F.1.2 Hierarchical topologies

A hierarchical topology best illustrate potential problems with bunching, as illustrated in Figure F.2. Traffic from talkers {a0,a1,a2} flows into bridge B. Bridge B concentrates traffic received from three talkers, with the cumulative b3 traffic sent to c3.

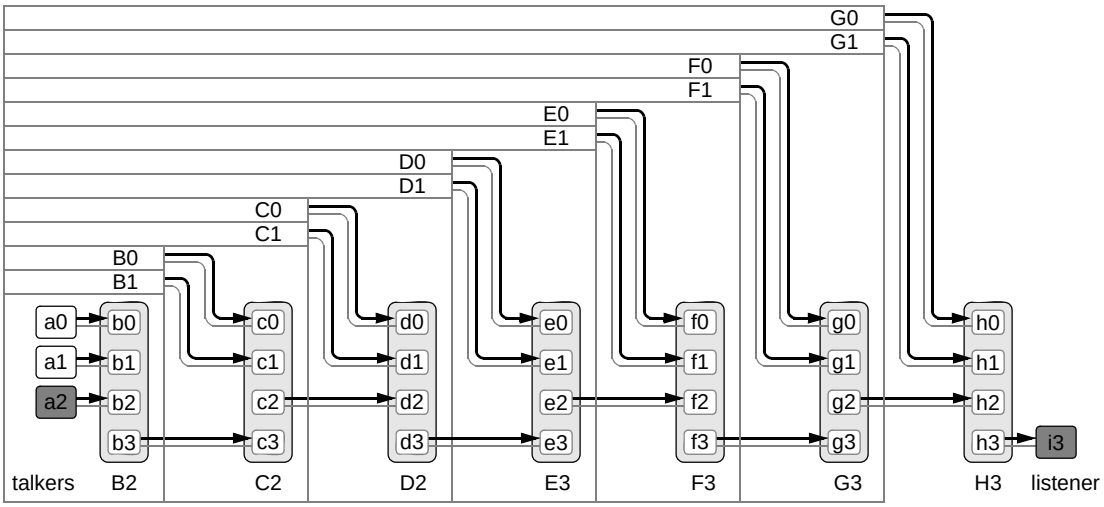


Figure F.2—Three-source hierarchical topology

Traffic from talker subnets {B0,B1,B2} flows into bridge C. Bridge C merges 1/3 of the traffic received from each of three talkers, with the cumulative c2 traffic sent to d2. The most-bunched traffic is always selected to be the forwarded.

Similarly, traffic from talker subnets {C0,C1,C2} flows into bridge D. Bridge D merges 1/3 of the traffic received from three subnets; the cumulative d3 traffic sent to e3. The most-bunched traffic is forwarded.

Similarly, traffic from talker subnets {D0,D1,D2} flows into bridge E. Bridge E merges 1/3 of the traffic received from three subnets; the cumulative e2 traffic sent to f2. The most-bunched traffic is forwarded.

Similarly, traffic from talker subnets {E0,E1,E2} flows into bridge F. Bridge F merges 1/3 of the traffic received from three subnets; with the cumulative f3 traffic sent to g3. The most-bunched traffic is forwarded.

Similarly, traffic from talker subnets {F0,F1,F2} flows into bridge G. Bridge G merges 1/3 of the traffic received from three subnets; the cumulative g2 traffic sent to h2. The most-bunched traffic is forwarded.

Similarly, traffic from talker subnets {F0,F1,F2} flows into bridge G. Bridge G merges 1/3 of the traffic received from three subnets; the cumulative g2 traffic sent to h2. The most-bunched traffic is forwarded.

To simplify illustrations, only the bottom-most flows are illustrated in the following subclauses.

F.1.3 Six-source hierarchical topology

Spreading the traffic over multiple sources, as illustrated in Figure F.3, exasperates bursting and bunching problems. Traffic from talkers {a0,a1,a2,a3,a4,a5} flows into ports on bridge B. Bridge B concentrates traffic received from six talkers, with the cumulative b6 traffic sent to c6. Identical traffic flows are assumed at bridge ports {c0,c1,c3,c3,c4,c6}, although only one of these sources is illustrated. Bridges {C,D,E,F,G,H} behave similarly.

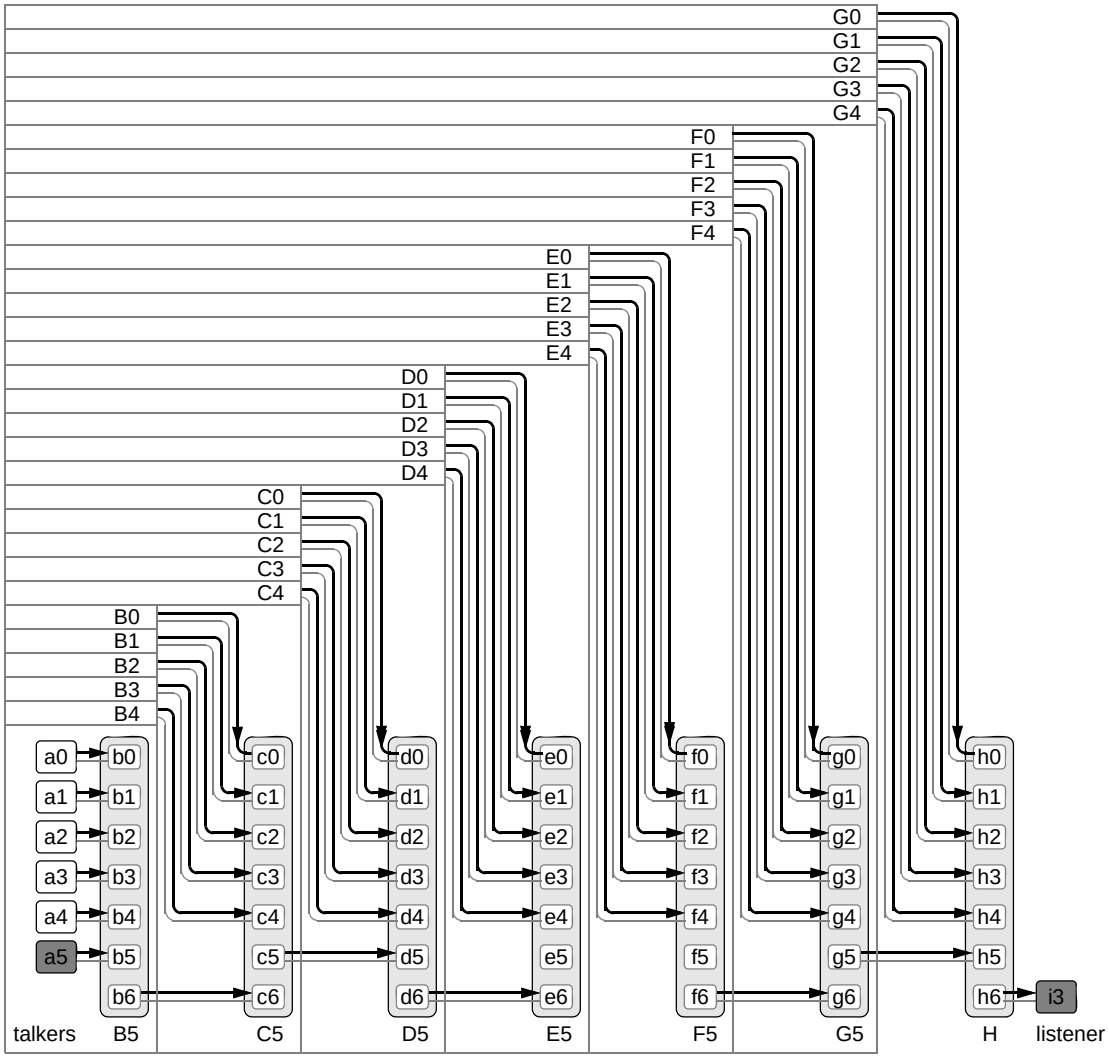


Figure F.3—Six-source topology

F.2 Bursting considerations

F.2.1 Three-source bursting scenario

A troublesome bursting scenario on a 100 Mb/s link can occur when small bandwidth streams coincidentally provide their infrequent 1500 byte frames concurrently, as illustrated in Figure F.4. Even though the cumulative bandwidths are considerably less than the capacity of the 100 Mb/s links, significant delays are incurred when passing through multiple bridges.

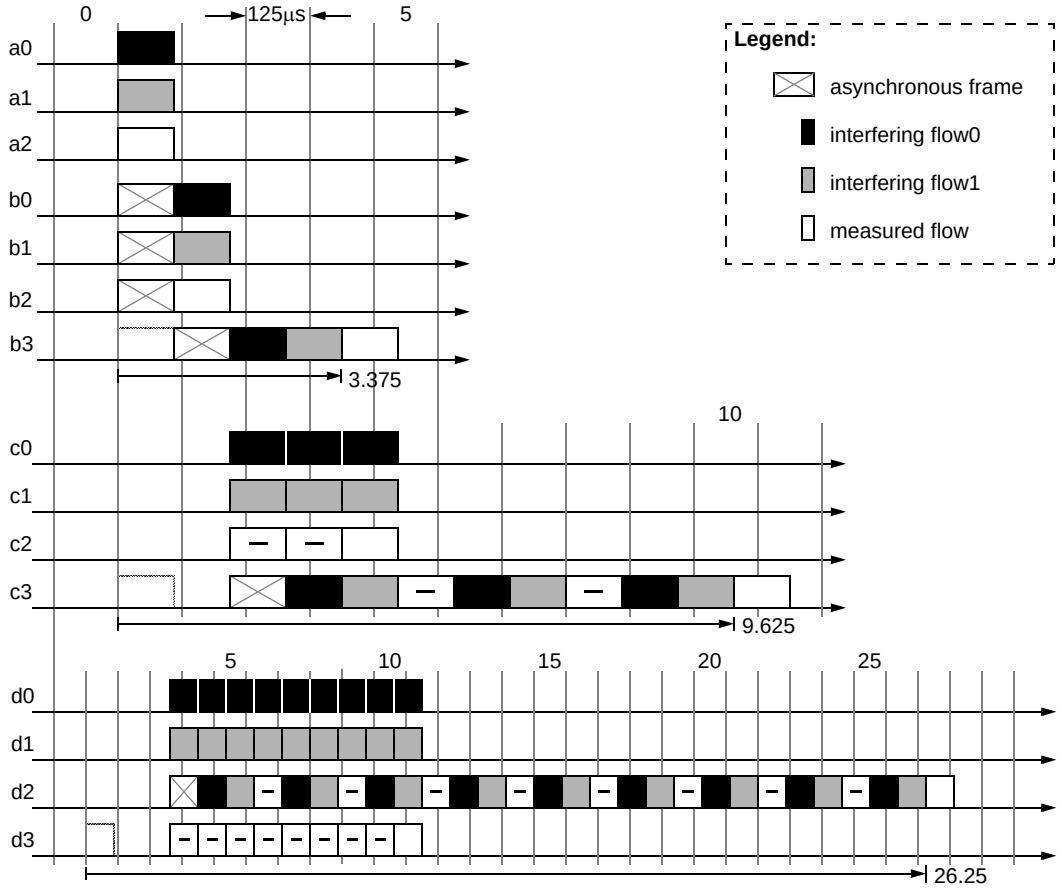


Figure F.4—Three-source bunching timing; input-queue bridges

F.2.1.1 Cumulative bunching latencies

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.1 and plotted in Figure F.5.

Table F.1—Cumulative bursting latencies

Topology	Units	Measurement point							
		A	B	C	D	E	F	G	H
3-source (see F.2.2.1)	mtu	1	4	11	30	85	248	735	2194
	ms	.120	.480	1.32	3.6	10.2	29.6	88.2	263
6-source (see F.2.2.2)	mtu	1	7	38	219	1300	7781	46662	229943
	ms	.120	.840	4.56	26.3	156	934	5600	27600

The values within this table are computed based on Equation F.1.

$$delay[n] = mtu \times (n + p^n) \quad \text{(F.1)}$$

Where:

mtu (maximum transfer unit) is the maximum frame size
n is the number of hops from the source
p is the number of receive ports in each bridge.

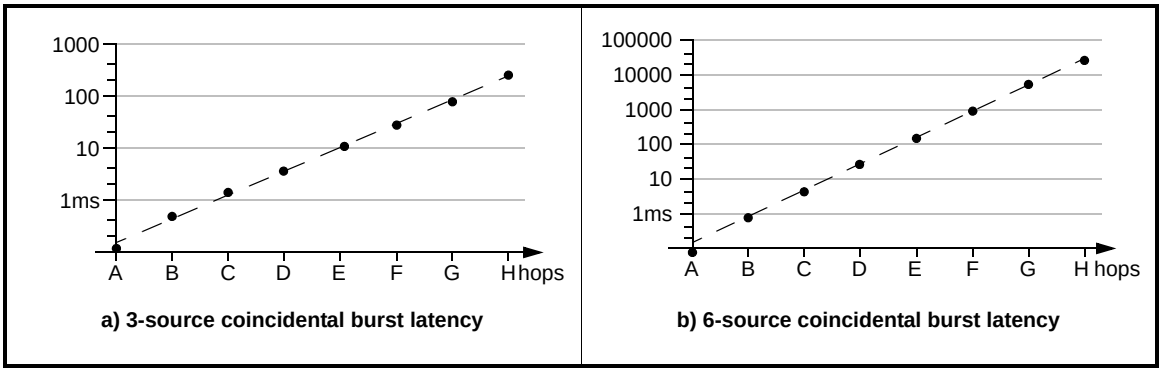


Figure F.5—Cumulative coincidental burst latencies

Conclusion: The classA traffic bandwidths should be enforced over a time interval that is on the order of an MTU size (120 μs), so as to avoid excessive delays caused by coincidental back-to-back large-block transmissions.

F.2.2 Bunching scenarios; input-queue bridges

F.2.2.1 Three-source bunching; input-queue bridges

To illustrate the effects of worst case bunching on input-queue bridges, specific flows are illustrated in Figure F.6. Bridge ports {b0,b1,b2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through b3. Each stream consumes 25% of the link bandwidth; 25% is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c3},…,{e0,e1,e3}, only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

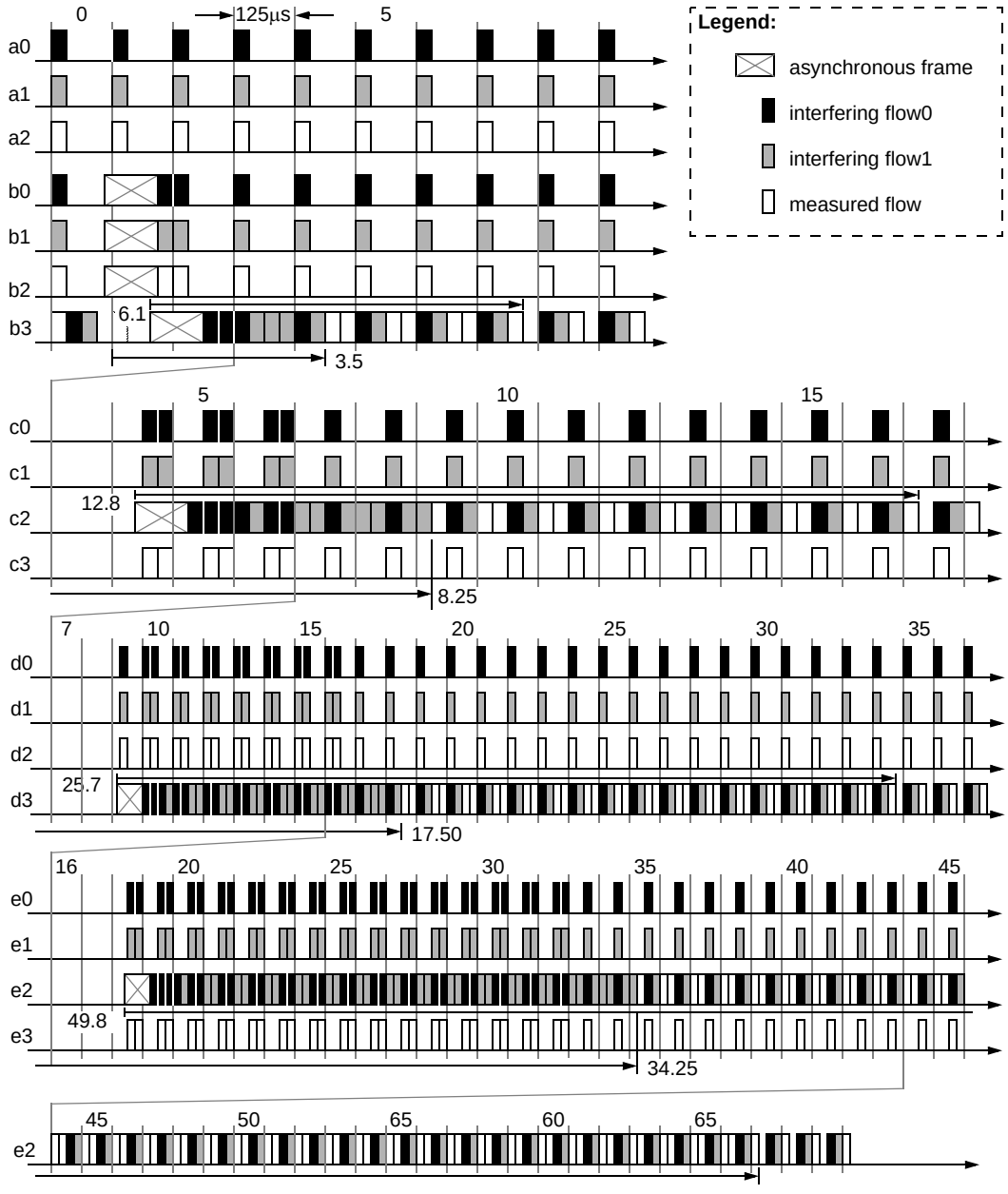


Figure F.6—Three-source bunching; input-queue bridges

F.2.2.2 Six-source bunching; input-queue bridges

To better illustrate the effects of worst case bunching on input-queue bridges, specific flows are illustrated in Figure F.7. Bridge ports {b0,b1,b2,b3,b4,b5} concentrates traffic from three talkers; one sixth of the cumulative traffic is forwarded through b6. Each of six streams consumes 12.5% of the link bandwidth, so that 25% is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c2,c3,c4,c6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

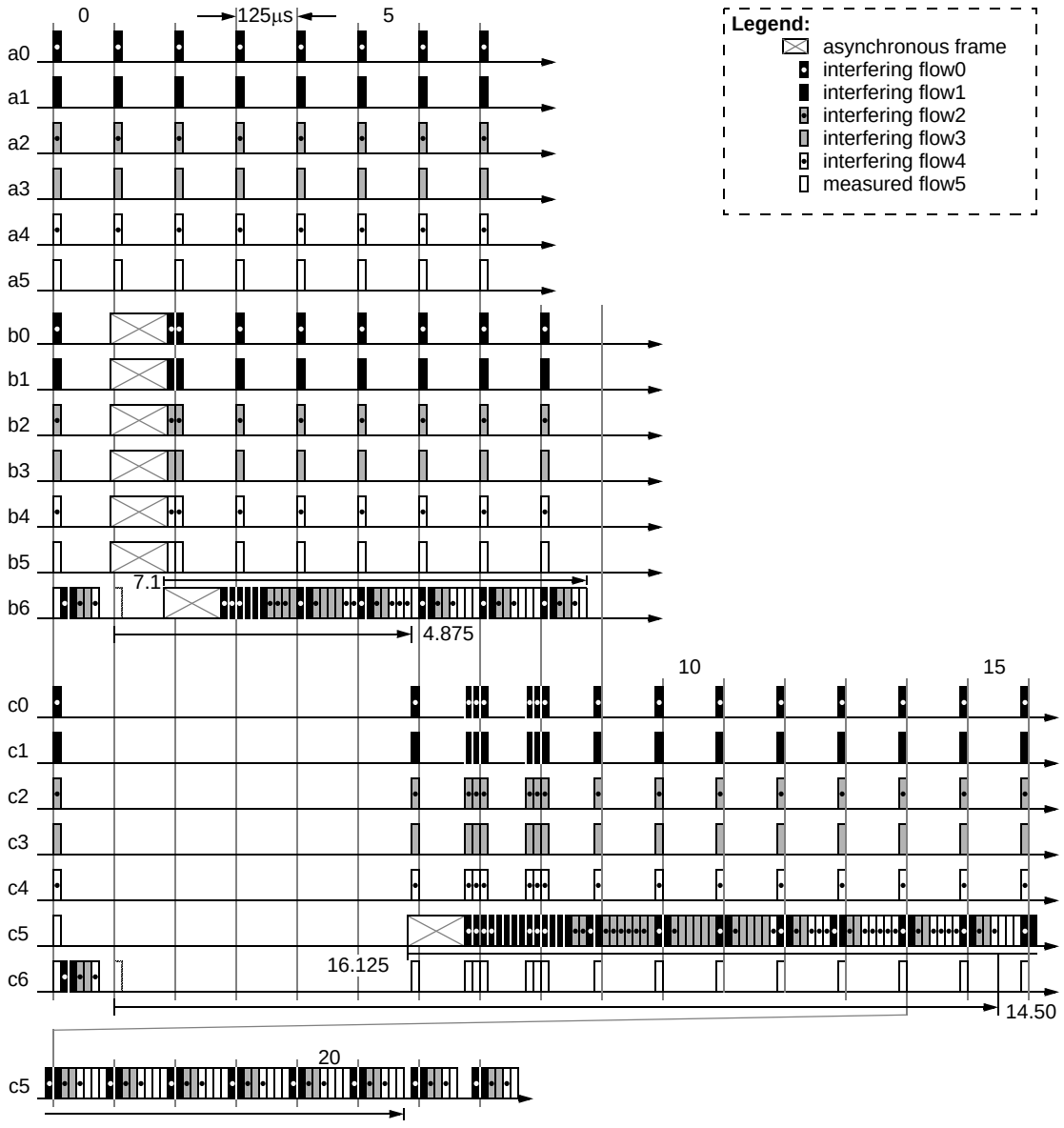


Figure F.7—Six source bunching timing; input-queue bridges

F.2.2.3 Cumulative bunching latencies, input-queue bridge

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.2 and plotted in Figure F.8.

Table F.2—Cumulative bunching latencies; input-queue bridge

Topology	Units	Measurement point							
		A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
3-source (see F.2.2.1)	cycles	0.125	3.5	8.25	17.5	34.25	(70.75)	(143.2)	(288.2)
	ms	0.01	0.44	1.03	2.19	4.28	8.84	17.9	36.0
6-source (see F.2.2.2)	cycles	0.125	4.875	14.50	(39.33)	(107.2)	(288.2)	(771)	2058
	ms	0.01	0.61	1.81	4.92	13.4	36.0	96.4	257

The first few numbers are generated using graphical techniques, as illustrated in Figure F.2.2.2. The following numbers are estimated, based on Equation F.2.

$$delay[n+1] = (mtu + delay[n]) \times (1/(1-0.75\times(p-1)/p)) \quad \text{(F.2)}$$

Where:

mtu (maximum transfer unit) is the maximum frame size
rate is the fraction of the bandwidth reserved for class A traffic, assumed to be 0.75
n is the number of hops from the source
p is the number of receive ports in each bridge.

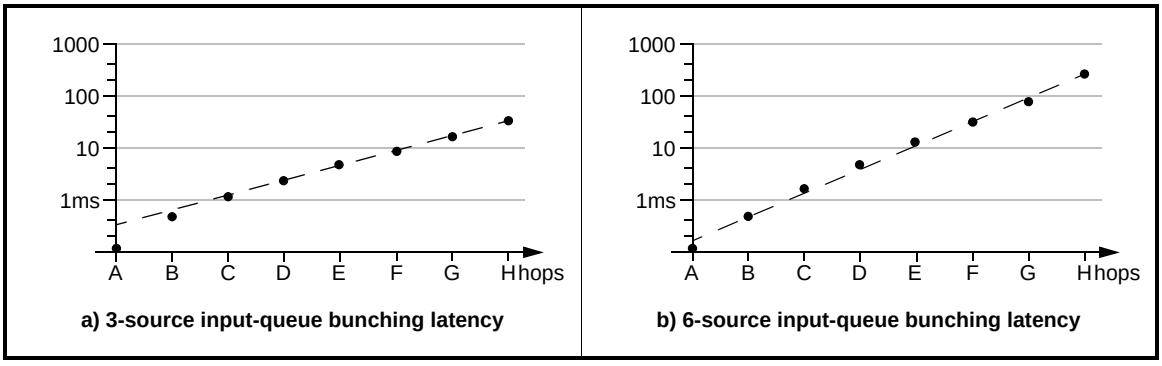


Figure F.8—Cumulative bunching latencies; input-queue bridge

Conclusion: A FIFO based output-queue bridge should be used. Alternatively (if input queuing is used), received frames should be time-stamped to ensure FIFO like forwarding.

F.2.3 Bunching topology scenarios; output-queue bridges

F.2.3.1 Three-source bunching timing; output-queue bridges

To illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.9. Bridge ports {b0,b1,b2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through b3. Each stream consumes 25% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {b0,b1,b2},…,{e0,e1,e3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

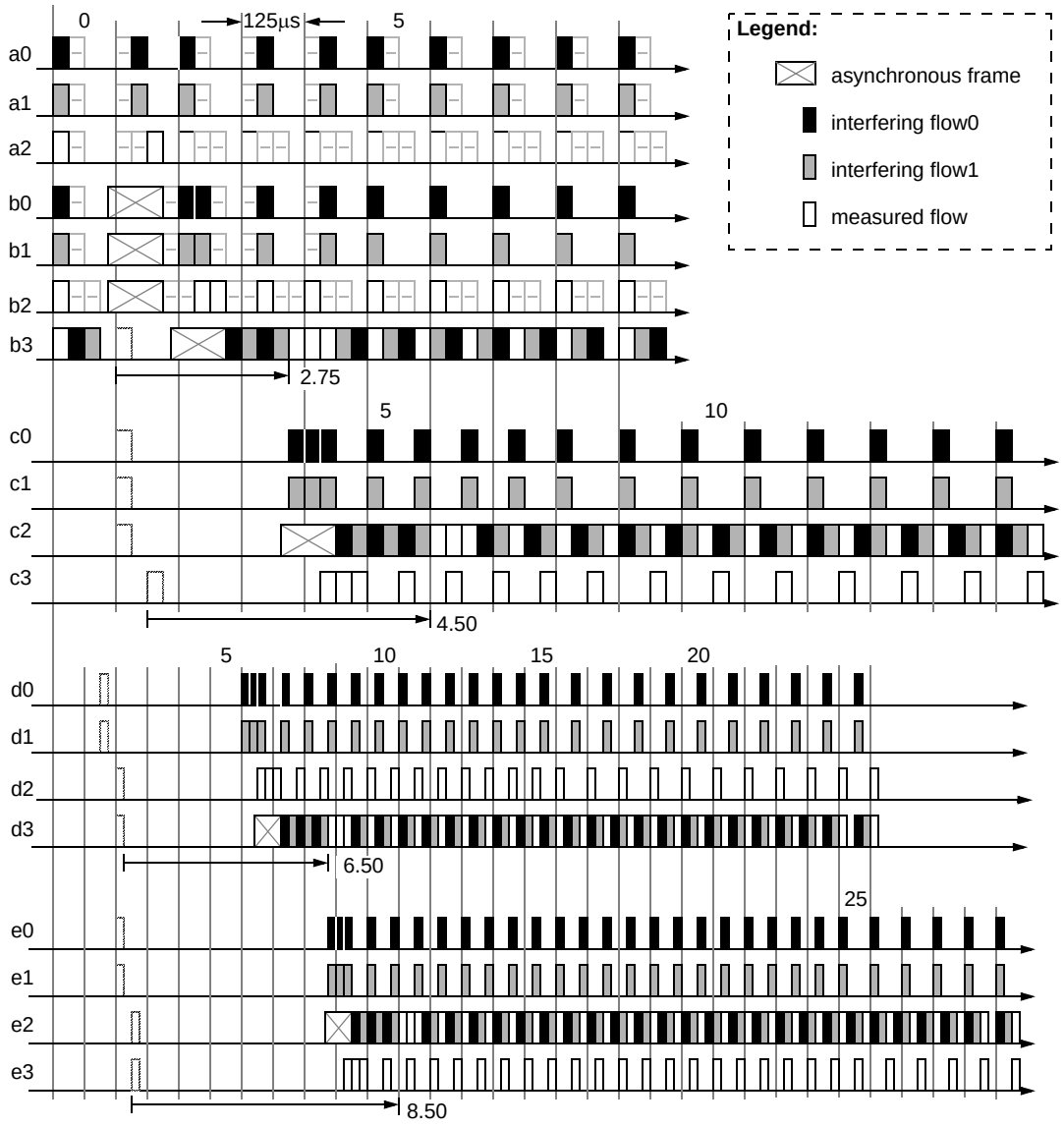


Figure F.9—Three-source bunching; output-queue bridges

F.2.3.2 Six-source bunching; output-queue bridges

To better illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.10. Bridge ports {b0,b1,b2,b3,b4,b5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port b6. Each of six streams consumes 12.5% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c2,c3,c4,c6} and {d0,d1,d2,d3,d4,d5} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

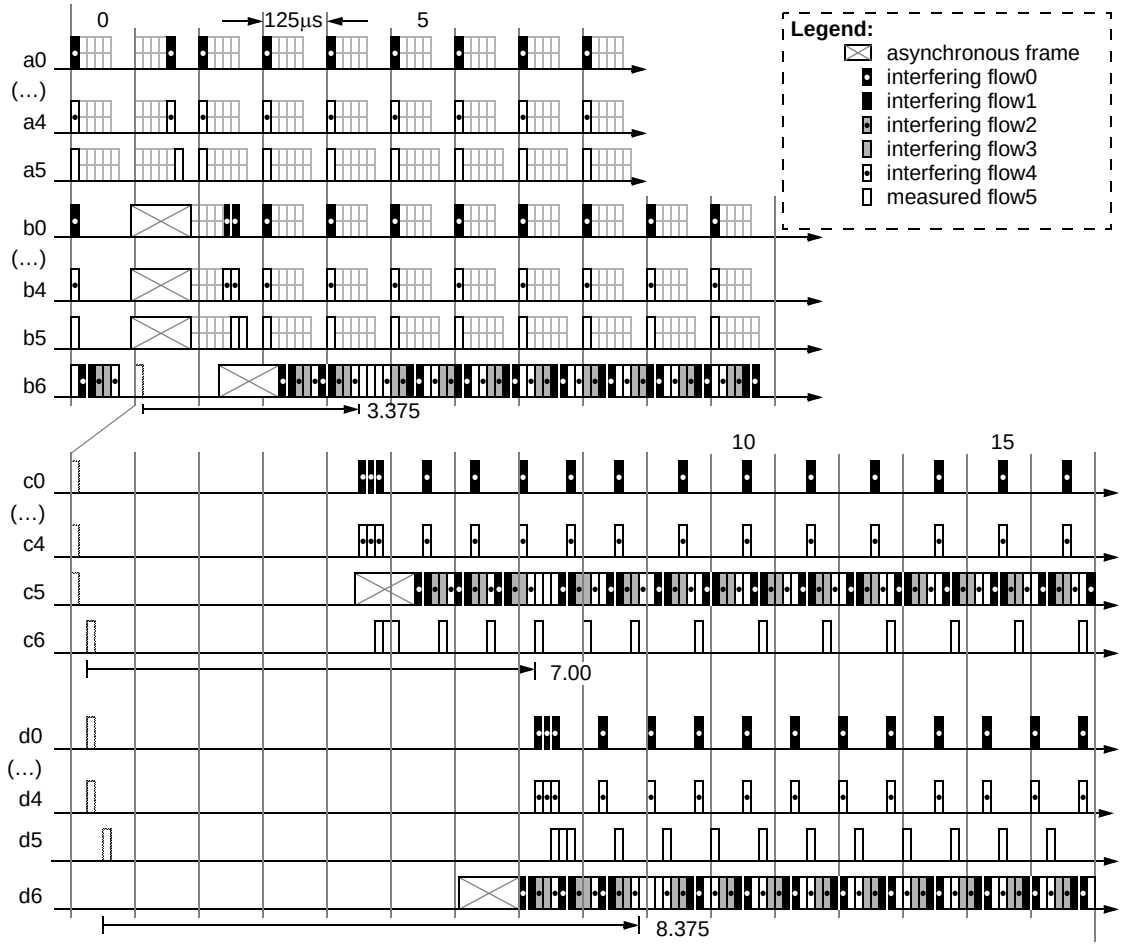


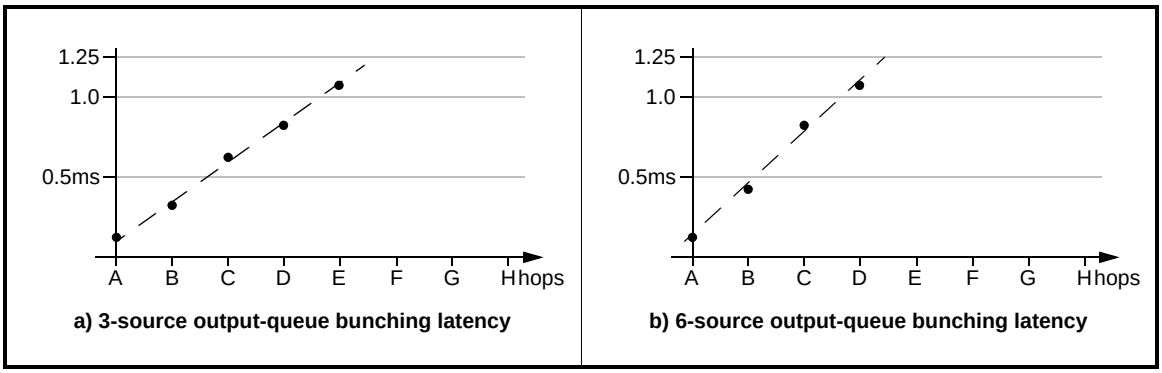
Figure F.10—Six source bunching; output-queue bridges

F.2.3.3 Cumulative bunching latencies; output-queue bridge

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.3 and plotted in Figure F.11.

Table F.3—Cumulative bunching latencies; output-queue bridge

Topology	Units	Measurement point							
		B	C	D	E	F	G	H	I
3-source (see F.2.2.1)	cycles	.875	2.75	4.5	6.5	8.5	–	–	–
	ms	0.10	0.34	0.56	0.81	1.6	–	–	–
6-source (see F.2.2.2)	cycles	.875	3.375	7.00	8.375	–	–	–	–
	ms	0.10	0.42	.875	1.05	–	–	–	–



a) 3-source output-queue bunching latency

b) 6-source output-queue bunching latency

Figure F.11—Cumulative bunching latencies; output-queue bridge

Conclusion: For steady-state classA traffic, acceptably small linear latencies are introduced by output-queue bridges on 75% loaded links. Unfortunately, the nonsteady-state nature of variable-rate traffic makes this conclusion suspect (see F.2.4).

F.2.4 Bunching topology scenarios; variable-rate output-queue bridges

F.2.4.1 Three-source bunching; variable-rate output-queue bridges

To illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.12. Bridge ports {b0,b1,b2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port b3. Each stream consumes 25% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c3},…,{e0,e1,e3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

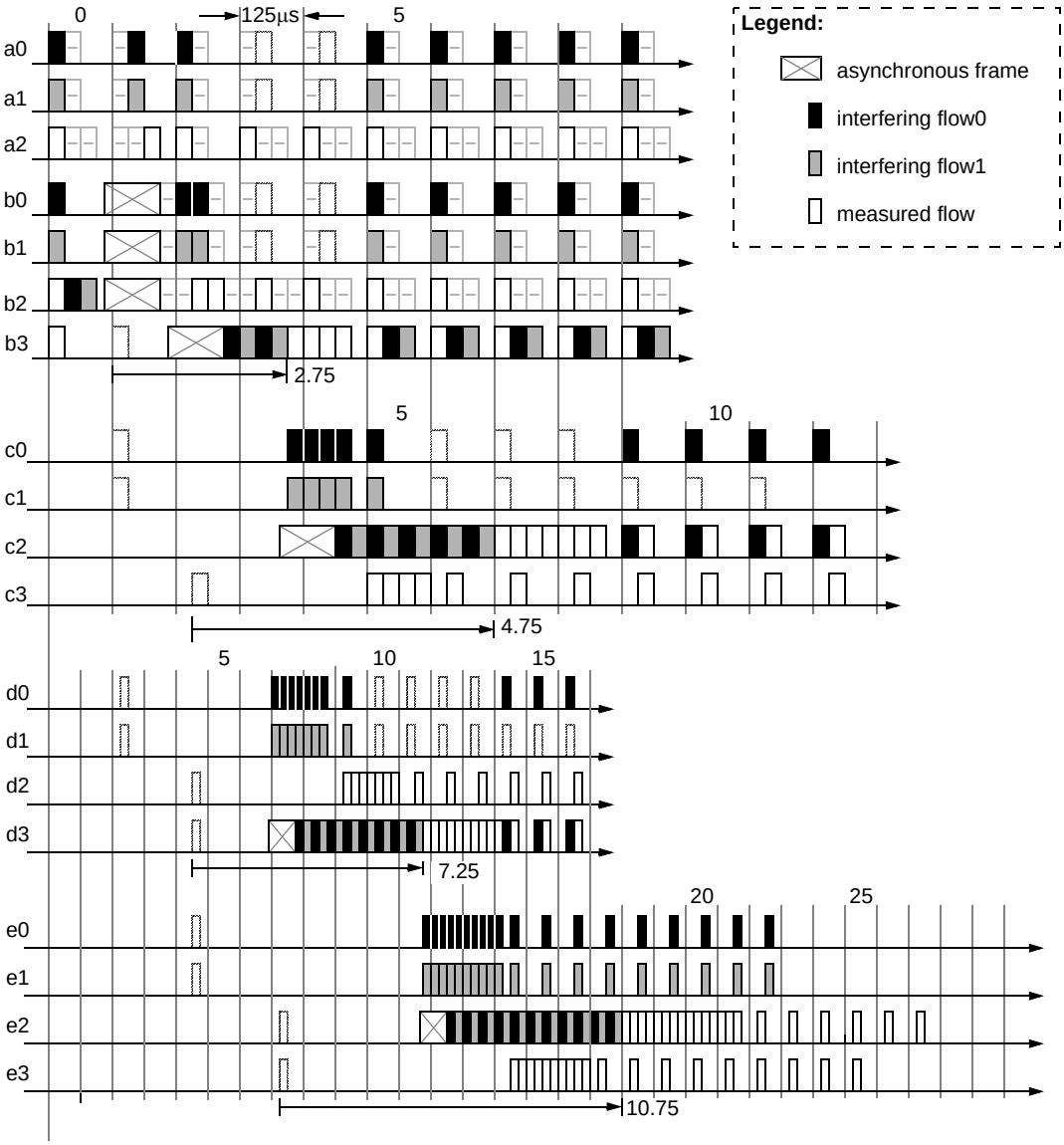


Figure F.12—Three-source bunching; variable-rate output-queue bridges

F.2.4.2 Six-source bunching; variable-rate output-queue bridges

To better illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.13. Bridge ports {b0,b1,b2,b3,b4,b5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port b6. Each of six streams consumes 12.5% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c2,c3,c4,c6}, {d0,d1,d2,d3,d4,d5}, and {e0,e1,e2,e3,e4,e6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

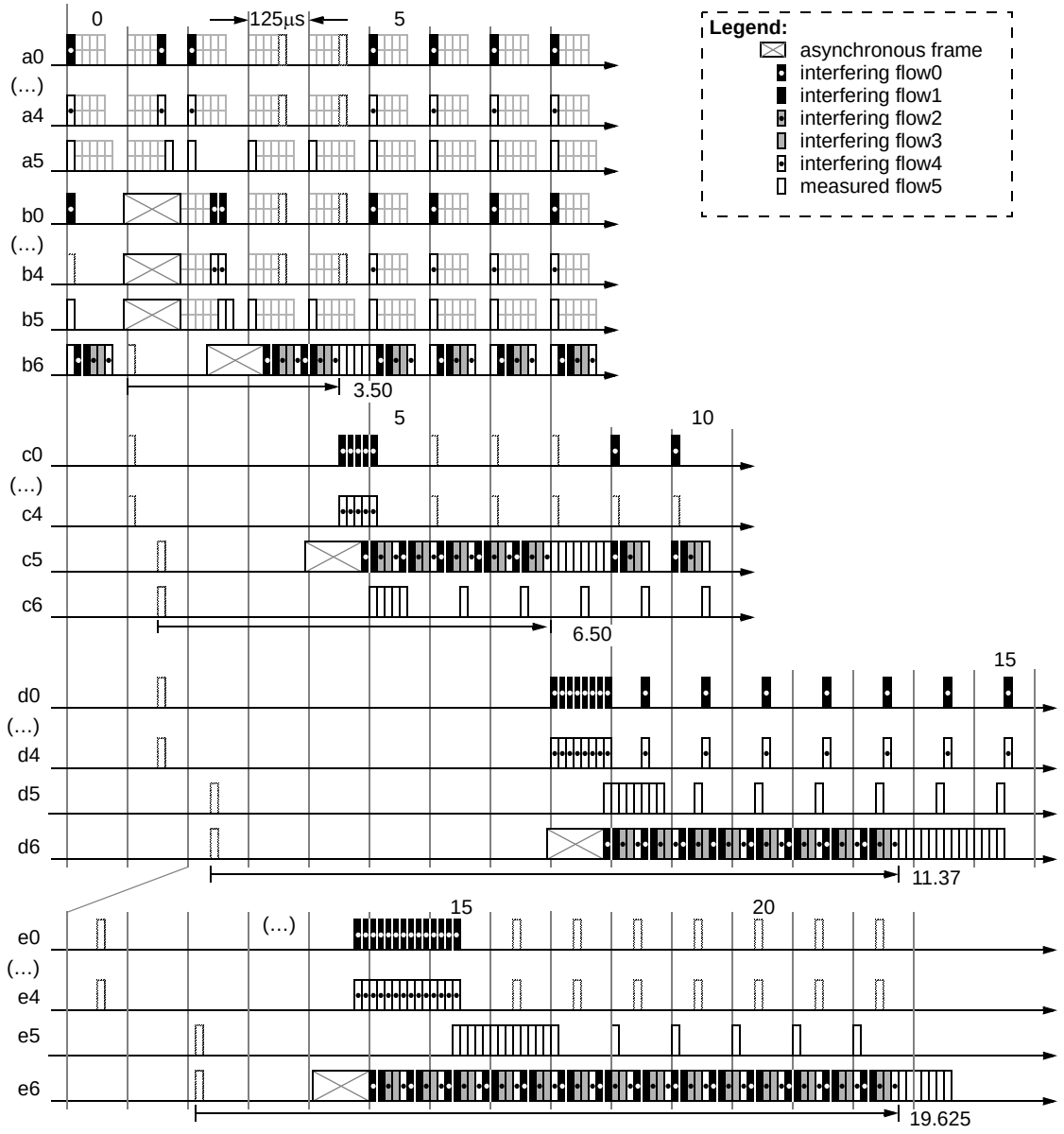


Figure F.13—Six source bunching; variable-rate output-queue bridges

F.2.4.3 Cumulative bunching latencies; variable-rate output-queue bridge

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.4 and plotted in Figure F.14.

Table F.4—Cumulative bunching latencies; variable-rate output-queue bridge

Topology	Units	Measurement point							
		A	B	C	D	E	F	G	H
3-source (see F.2.2.1)	cycles	0.75	2.75	4.75	7.25	10.75	–	–	–
	ms	0.10	0.34	0.59	0.90	1.34	–	–	–
6-source (see F.2.2.2)	cycles	0.75	3.50	6.50	11.38	19.63	–	–	–
	ms	0.10	0.44	0.81	1.42	2.45	–	–	–

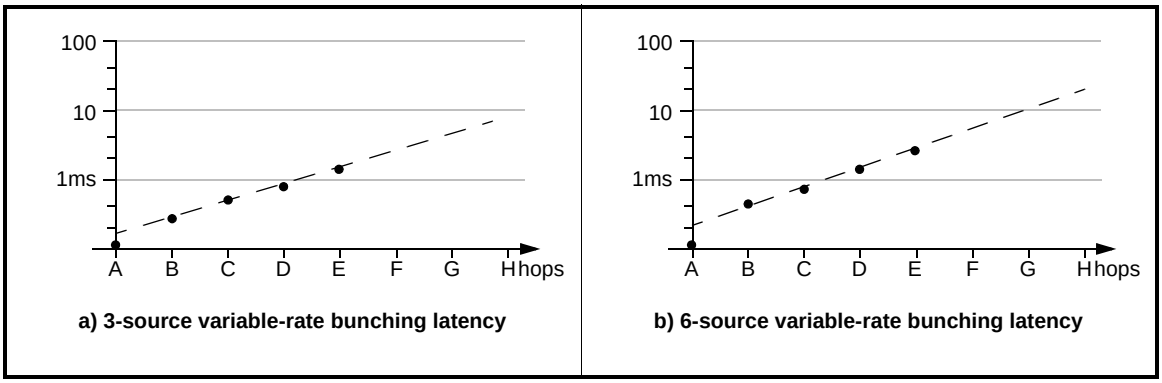


Figure F.14—Cumulative bunching latencies; variable-rate output-queue bridge

Conclusion: For nonsteady-state classA traffic, significant expediential latencies are introduced by output-queue bridges on 75% loaded links. Unfortunately, throttled outputs further exasperates this latency (see F.2.4).

F.2.5 Bunching topology scenarios; throttled-rate output-queue bridges

F.2.5.1 Three-source bunching; throttled-rate output-queue bridges

To illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.15. Bridge ports {b0,b1,b2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port b3. Each stream consumes 25% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c3}, {d0,d1,d2}, and {e0,e1,e3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

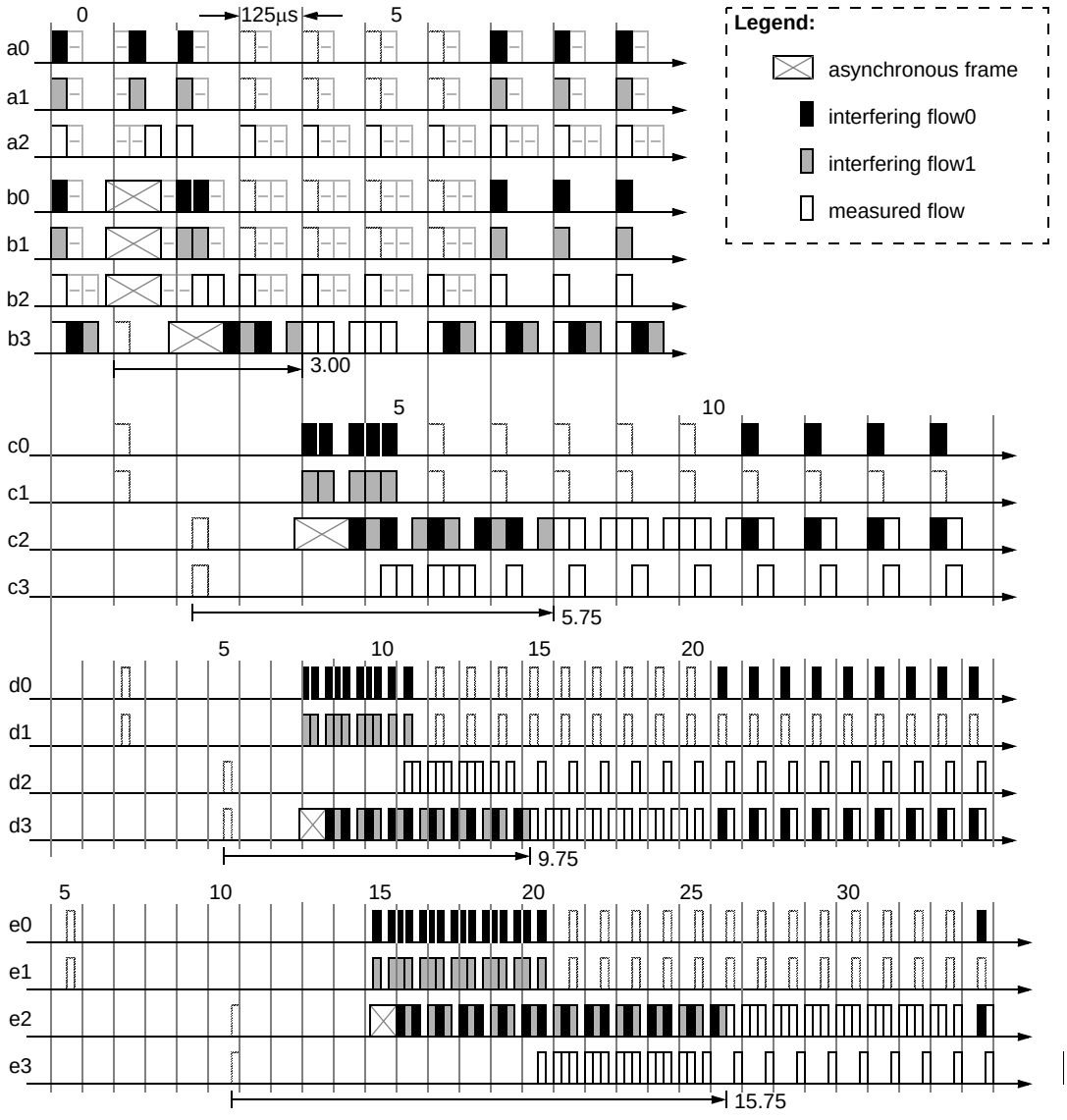


Figure F.15—Three-source bunching; throttled-rate output-queue bridges

F.2.5.2 Six-source bunching; throttled-rate output-queue bridges

To better illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.16. Bridge ports {b0,b1,b2,b3,b4,b5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port b6. Each of six streams consumes 12.5% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c2,c3,c4,c5},…,{e0,e1,e2,e3,e4,e6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

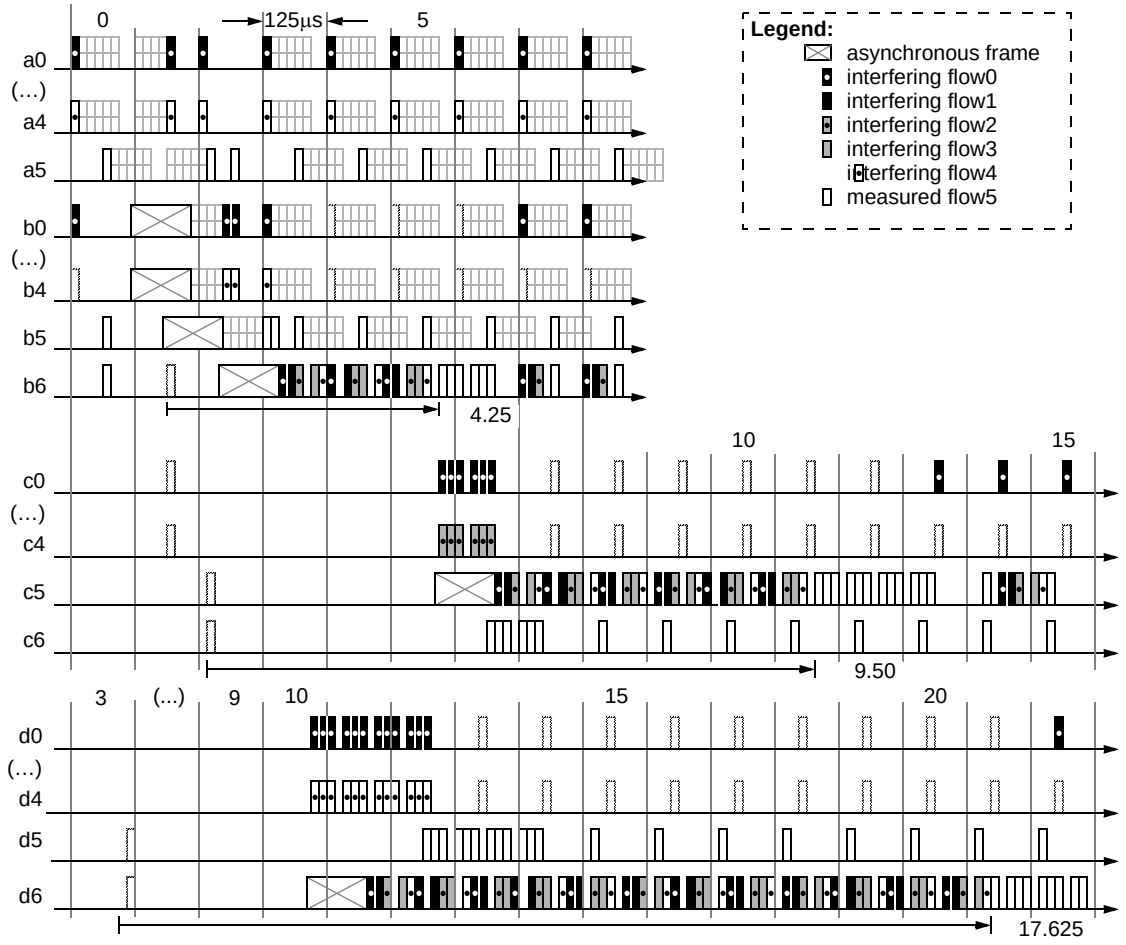


Figure F.16—Six source bunching; throttled-rate output-queue bridges

F.2.5.3 Cumulative bunching latencies; throttled-rate output-queue bridge

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.5 and plotted in Figure F.17.

Table F.5—Cumulative bunching latencies; throttled-rate output-queue bridge

Topology	Units	Measurement point							
		A	B	C	D	E	F	G	H
3-source (see F.2.2.1)	cycles	0.75	3.00	5.75	9.75	15.75	–	–	–
	ms	0.09	0.38	0.73	1.21	1.97	–	–	–
6-source (see F.2.2.2)	cycles	0.75	4.25	9.5	17.63	–	–	–	–
	ms	0.09	0.53	1.19	2.20	–	–	–	–

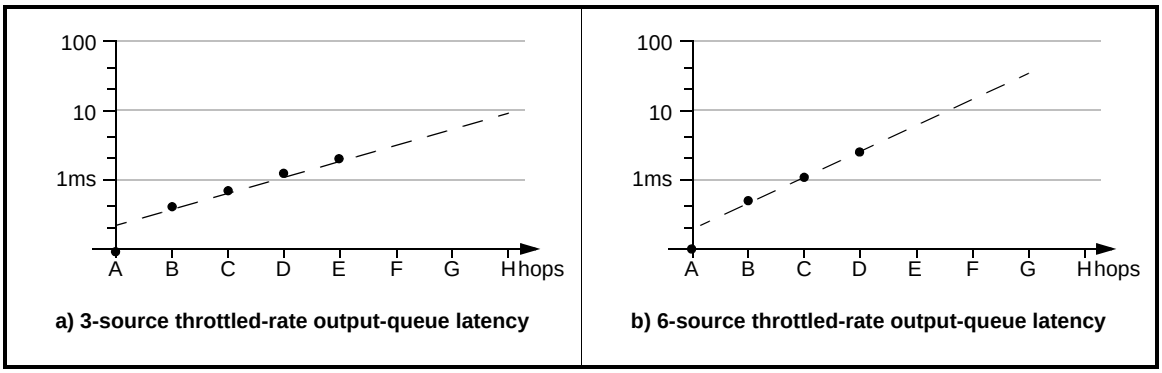


Figure F.17—Cumulative bunching latencies; throttled-rate output-queue bridge

Conclusion: On large topologies, the classA traffic latencies can accumulate beyond acceptable limits. Some form of receiver retiming may therefore be desired.

F.2.6 Bunching topology scenarios; classA throttled-rate output-queue bridges

The extent of bunching extent is worst when large classC frames are present. However, bunching can also occur in the absence of large classC frames, as described in the remainder of this subannex.

F.2.6.1 Three-source bunching; classA throttled-rate output-queue bridges

To illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.18 and Figure F.19. Bridge ports {b0,b1,b2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port b3. Each stream consumes 25% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c3}, {c0,d1,d2}, and {e0,e1,e3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

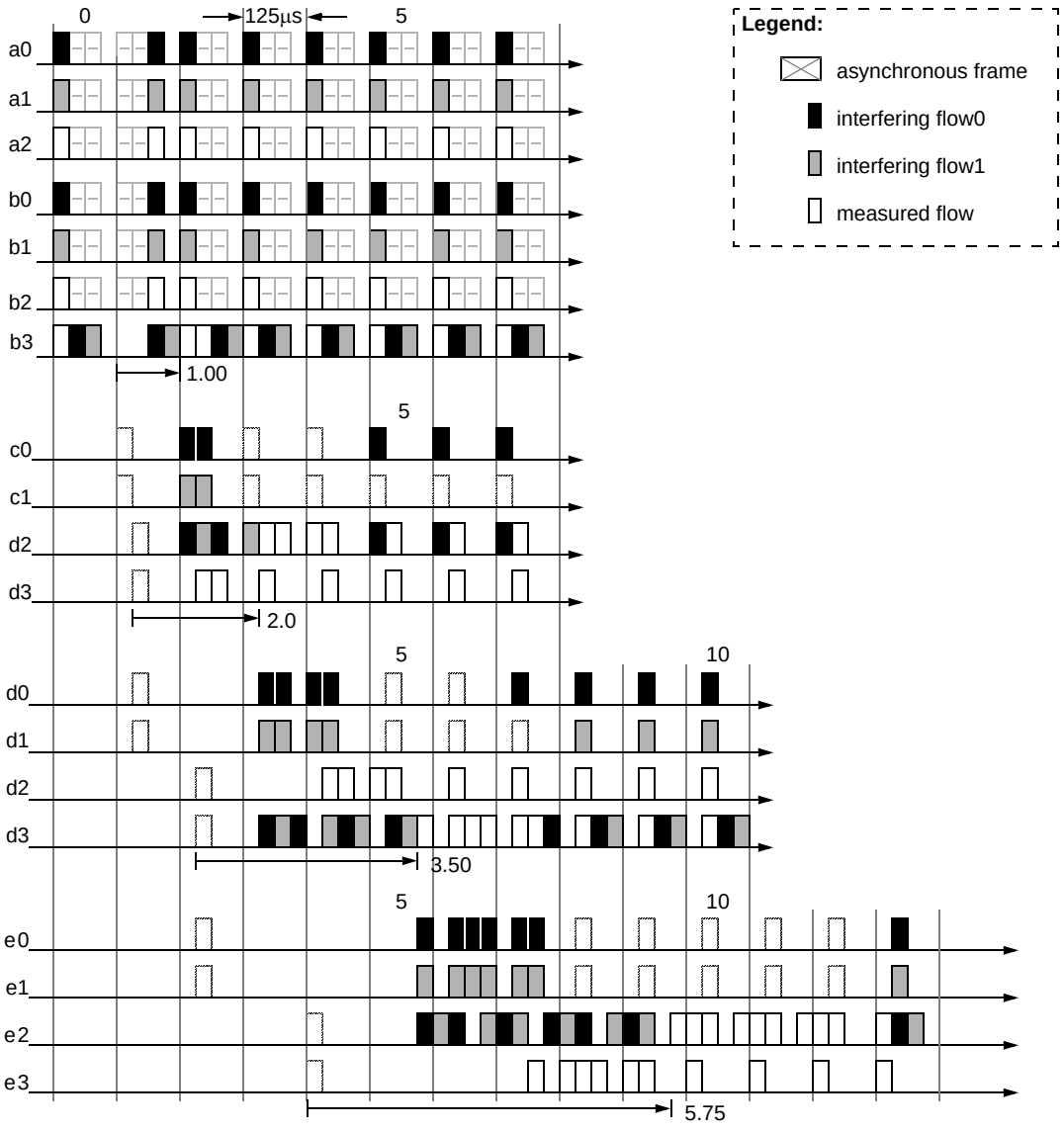


Figure F.18—Three-source bunching; throttled-rate output-queue bridges

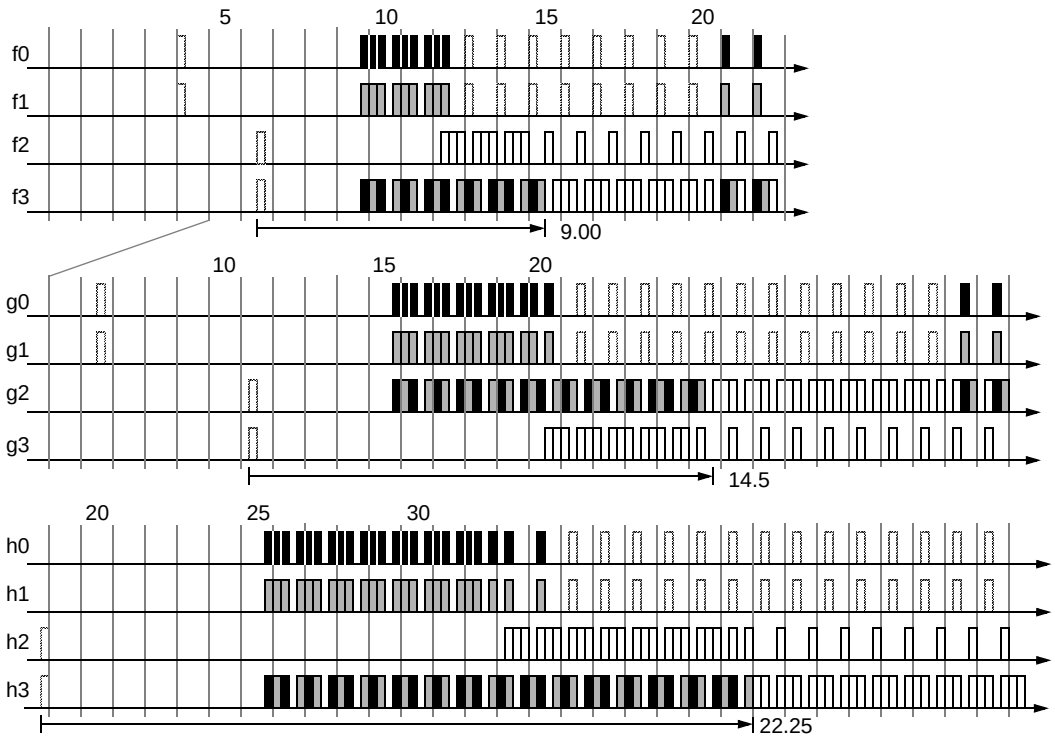


Figure F.19—Three-source bunching; throttled-rate output-queue bridges

F.2.6.2 Six-source bunching; classA throttled-rate output-queue bridges

To better illustrate the effects of worst case bunching, specific flows are illustrated in Figure F.20. Bridge ports {b0,b1,b2,b3,b4,b5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port b6. Each of six streams consumes 12.5% of the link bandwidth; 25% of the link bandwidth is available for asynchronous traffic.

For clarity, the traces for input traffic on ports {c0,c1,c2,c3,c4,c5},…,{d0,d1,d2,d3,d4,d6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

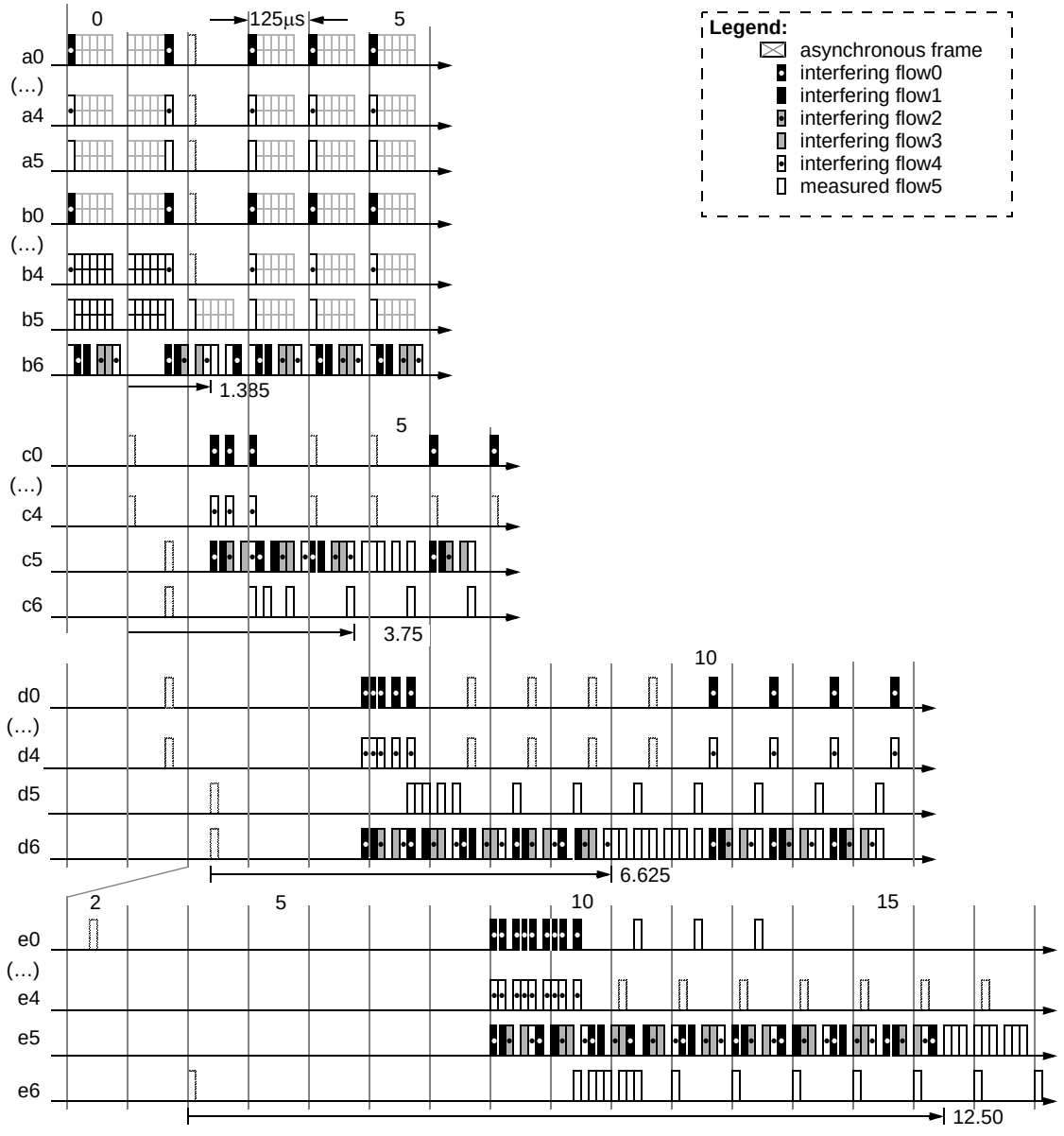


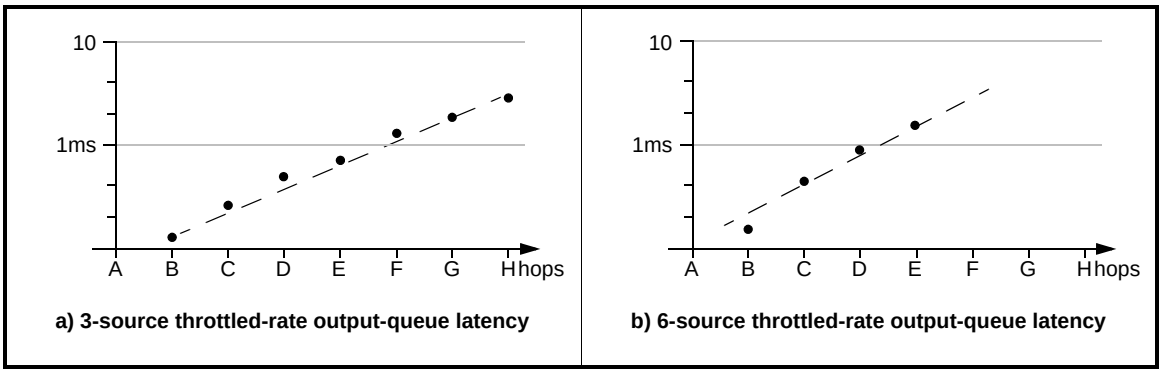
Figure F.20—Six source bunching; classA throttled-rate output-queue bridges

F.2.6.3 Cumulative bunching latencies; classA throttled-rate output-queue bridge

The cumulative worst-case latencies implied by coincidental bursting are listed in Table F.6 and plotted in Figure F.21.

Table F.6—Cumulative bunching latencies; classA throttled-rate output-queue bridge

Topology	Units	Measurement point							
		A	B	C	D	E	F	G	H
3-source (see F.2.2.1)	cycles	–	1.00	2.00	3.5	5.75	9.00	14.5	22.5
	ms	–	0.125	0.25	0.44	0.72	1.13	1.81	2.81
6-source (see F.2.2.2)	cycles	–	1.385	3.75	6.625	12.50	–	–	–
	ms	–	0.17	0.47	0.83	1.56	–	–	–



a) 3-source throttled-rate output-queue latency

b) 6-source throttled-rate output-queue latency

Figure F.21—Cumulative bunching latencies; classA throttled-rate output-queue bridge

Conclusion: On large topologies, the classA traffic latencies can accumulate beyond acceptable limits, even in the absence of conflicting lower-class traffic. Some form of receiver retiming may therefore be desired, even on higher speed links where the size of the MTU (in time) becomes much smaller than an assumed 8 kHz cycle time.

Annex G

(informative)

Denigrated alternatives

G.1 Stream frame formats

NOTE—The following streaming classA frame format options were considered but rejected.
These options are retained for historical purposes and (if opinions change) possible reconsideration.
For these reasons, the perceived advantages and disadvantages of each technique are listed.

G.1.1 VLAN routed frame formats (alternative 4)

Frames within a stream are no different than other Ethernet frames, with the exception of their distinct *da* (destination address) and *control* field values, as illustrated in Figure G.1.

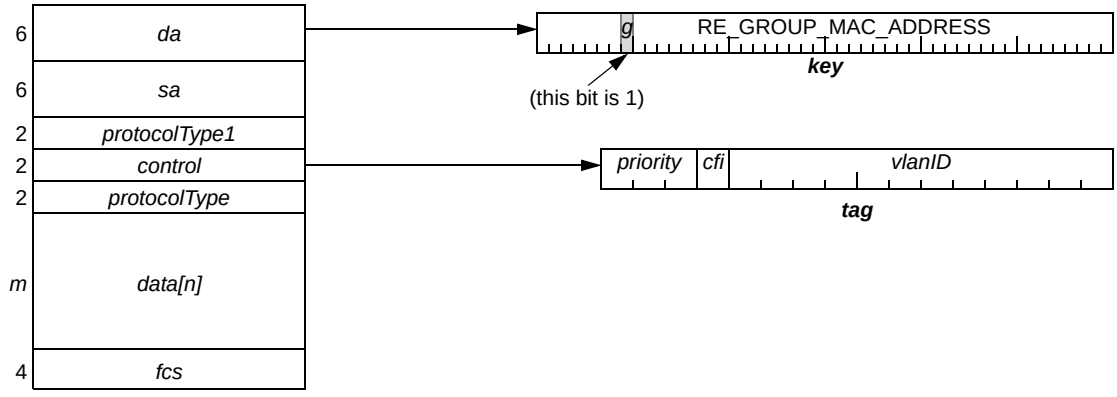


Figure G.1—classA frame formats

A single multicast address (labeled as RE_GROUP_MAC_ADDRESS) identifies the multicast time-sensitive nature of the frame. The following VLAN tag identifies the frame priority and provides a distinct *vlanID* identifier. The *vlanID* identifier is also the *streamID* identifier, allowing each stream to be independently selectively-switched through bridges.

The over-riding disadvantages of this design approach relates to its forwarding through bridges:

a) Overloaded. This novel *vlanID* usage could conflict with existing bridge implementations.

b) VLAN service. A method of generating distinct *vlanID* values would be required.
(Some for of central server or distributed assignment algorithm would be required).

G.1.2 Broadcast routed frame formats (alternative5)

Frames within a stream are no different than other Ethernet frames, with the exception of their distinct fixed multicast *da* (destination address), as illustrated in Figure G.2.

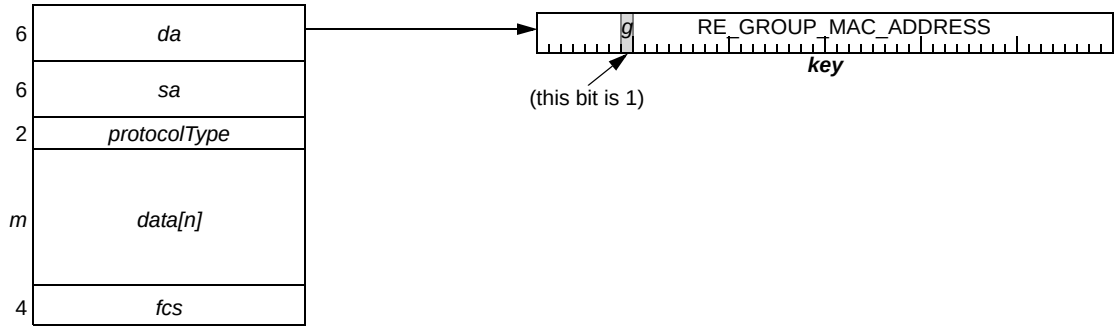


Figure G.2—ClassA frame formats

A single multicast address (labeled as RE_GROUP_MAC_ADDRESS) identifies the multicast time-sensitive nature of the frame.

The over-riding disadvantages of this design approach relates to its forwarding through bridges:

a) Bandwidth. Bandwidth is wasted because frames are broadcast to all potential listeners, rather than only the subscribed listeners.

b) Ambiguous. The *da* field is insufficient to identify the frame, mandating the presence of stream identifier information within the *data*[ ] payload.

Annex H

(informative)

Frequently asked questions (FAQs)

H.1 Unfiltered email sequences

H.1.1 Bandwidth allocation

Question (AM): Is bandwidth allocation really necessary to meet RE requirements? Over-provisioning and best-effort (with class of service) may be adequate. You can get a lot of data through a conventional gigabit switch with very low latencies. The RE traffic can be given a higher priority and so not be held up by less urgent traffic.

Answer (MJT): I think admission control is needed. In an unmanaged layer 2 environment there is no way to *guarantee* that the streaming QoS parameters can be met … you can only say *probably.* With GigE and a fully bridge-based environment with class of service you can get to a pretty good *probably*, but you can't get to the *it will always work* QoS that the wonderful BER of Ethernet promises. On the other hand, a simple admission control system and simple pacing mechanism can get you there, even with an FE-only network.

H.1.2 Best effort

Question (AM): With access control what happens if access is denied? My assumption is that a user connecting to a RE network would prefer best-effort service to no service at all if there is no spare bandwidth to be allocated. If you decide you need to support best-effort as a fallback then you need buffers in your end stations and the reason for using time slots goes away.

Answer (MJT): Your assumption is only correct if the service the consumer is subscribing to *is* a best-effort service. Right now, consumers expect that when they select a channel, or a CD, or a DVD they will get it *perfectly.* Cable companies get lots of calls if a stream is substandard for any reason. The general procedure to select a stream on a CE-oriented network would be something like:

a) Hit the *directory* or *guide* button on your remote control
b) Find the content you want (note that the content entries might be labeled with *not currently available* or *low quality only* or not even present depending on the state of the path to the source).
c) Hit the *play* button.

Once the consumer hits that *play* button, the endpoints and network need to make a contract to deliver the content with the QoS expected by the consumer. So, in the case you describe where there is no guaranteed bandwidth available, you *may* present an alternative method (such as the *low quality* tag). This may be perfectly OK. If, on the other hand, the consumer wants to see the HD movie with full quality, they can yell at their kid to stop watching the movie that is causing the network link of interest to saturate.

Annex I

(informative)

C-code illustrations

NOTE—This annex is provided as a placeholder for illustrative C-code. Locating the C code in one location (as opposed to distributed throughout the working paper) is intended to simplify its review, extraction, compilation, and execution by critical reviewers.
Also, placing this code in a distinct Annex allows the code to be conveniently formatted in 132-character landscape mode. This eliminates the need to truncate variable names and comments, so that the resulting code can be better understood by the reader.

This Annex provides code examples that illustrate the behavior of RE entities. The code in this Annex is purely for informational purposes, and should not be construed as mandating any particular implementation. In the event of a conflict between the contents of this Annex and another normative portion of this standard, the other normative portion shall take precedence.

The syntax used for the following code examples conforms to ANSI X3T9-1995.

Index