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
June 28, 2005

Residential Ethernet (RE)
(a working paper)

Draft 0.124

Contributors:

Alexei Beliaev	Gibson
Dirceu Cavendish	NEC Labs America
George Claseman	Micrel
Jim Haagen-Smit	HP
David V. James	JGG
Michael D. Johas Teener	Broadcom

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.

Alexei Beliaev	Gibson
Dirceu Cavendish	NEC Labs America
George Claseman	Micrel
Jim Haagen-Smit	HP
David V James	JGG
Michael D. Johas Teener	Broadcom

Version history

Version	Data	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 in 6.7.2. – 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.
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 J: Time-synchronization code added
—	—	—	TBDs

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) The concept of cycles and periodic transmissions is used before being introduced (from MJT).
b) Consider whether the cycleStart transmissions should be every cycle or N’th cycle (from MJT), and how the cycle count would be transmitted/implied if these were not every cycle.
c) Better describe the benefits of bridge pacing:
    1) Easy to enforce 75% usage limits.
    2) Easier to detect timeouts by classA traffic absence.
    3) Easier to ensure sufficient classA queue sizes.
d) Better describe the per-cycle clockSync benefits:
    1) Simplified bridge pacing.
    2) Low latency clock synchronization.

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.

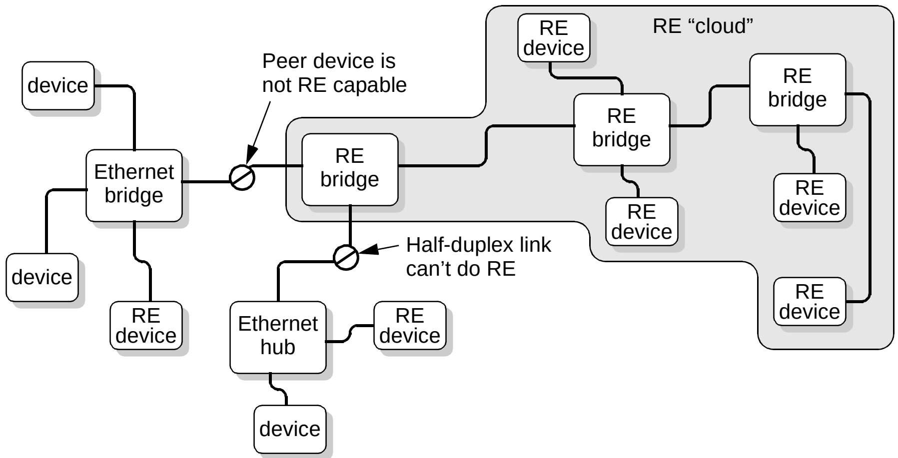


Figure 1.1—Topology and connectivity

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: Clock synchronization
— Clause 8: Subscription state machines
— 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: Comment responses
— Annex J: 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 [B4] 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 packet: A generic term for a PDU associated with a layer-entity above the MAC sublayer.

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

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

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

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

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

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

3.2.30 source station: The station that originates a frame.

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

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

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

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

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

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

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

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

3.2.38 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).

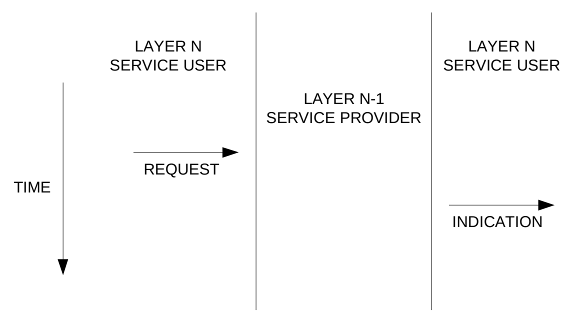


Figure 3.1—Service definitions

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 state		**Row**	**Next state**	
state	**condition**		**action**	**state**
START	sizeOfMacControl > spaceInQueue	1	—	START
	passM == 0	2		
	—	3	TransmitFromControlQueue();	FINAL
FINAL	SelectedTransferCompletes()	4	—	START
	—	5	—	FINAL

Row 3.1-1: Do nothing if the size of the queued MAC control frame is larger than the PTQ space.
Row 3.1-2: Do nothing in the absence of MAC control transmission credits.
Row 3.1-3: Otherwise, transmit a MAC control frame.

Row 3.1-4: When the transmission completes, start over from the initial state (i.e., START).
Row 3.1-5: 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 state		Row	Next state	
state	**condition**		**action**	**state**
START	sizeOfMacControl > spaceInQueue	1	—	FINAL
	passM == 0	2		
	—	3	TransmitFromControlQueue();	RETURN
FINAL	MacTransmitError();	4	errorDefect = TRUE	RETURN
	—	5	—	

Row 3.2-1: The size of the queued MAC control frame is less than the PTQ space.
Row 3.2-2: In the absence of MAC control transmission credits, no action is taken.
Row 3.2-3: MAC control transmissions have precedence over client transmissions.

Row 3.2-4: If the transmission completes with an error, set an error defect indication.
Row 3.2-5: 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

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

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	WRAP_AVOID	Frame is discarded at the wrap point
1	WRAP_ALLOW	Frame passes through wrap points
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.

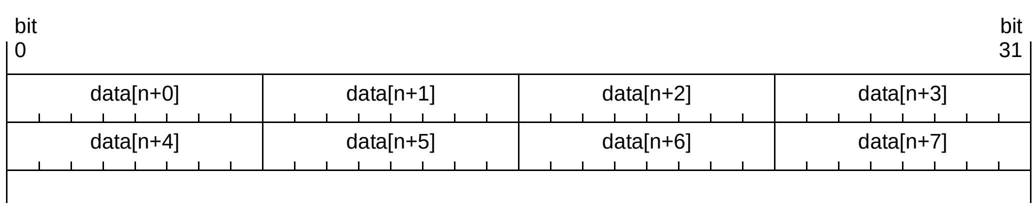


Figure 3.2—Bit numbering and ordering

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.

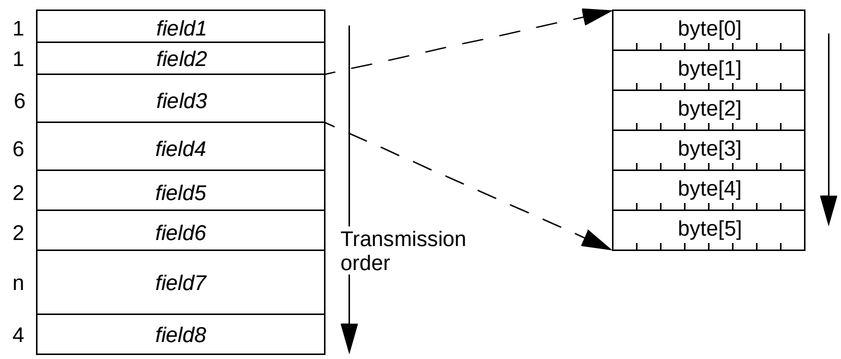


Figure 3.3—Byte sequential field format illustrations

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.

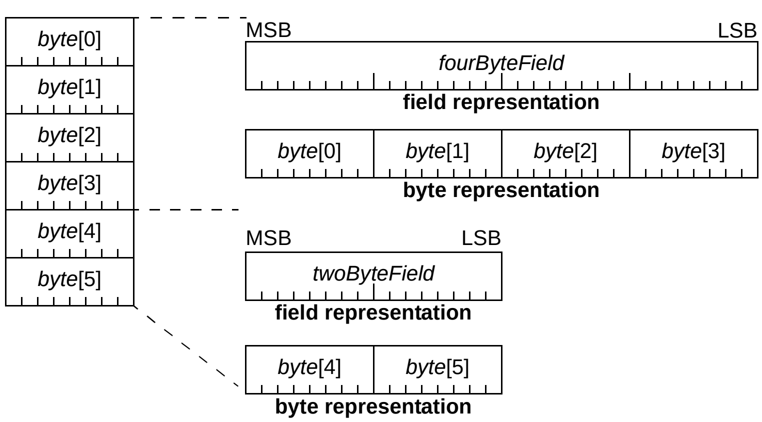


Figure 3.4—Multibyte field illustrations

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.

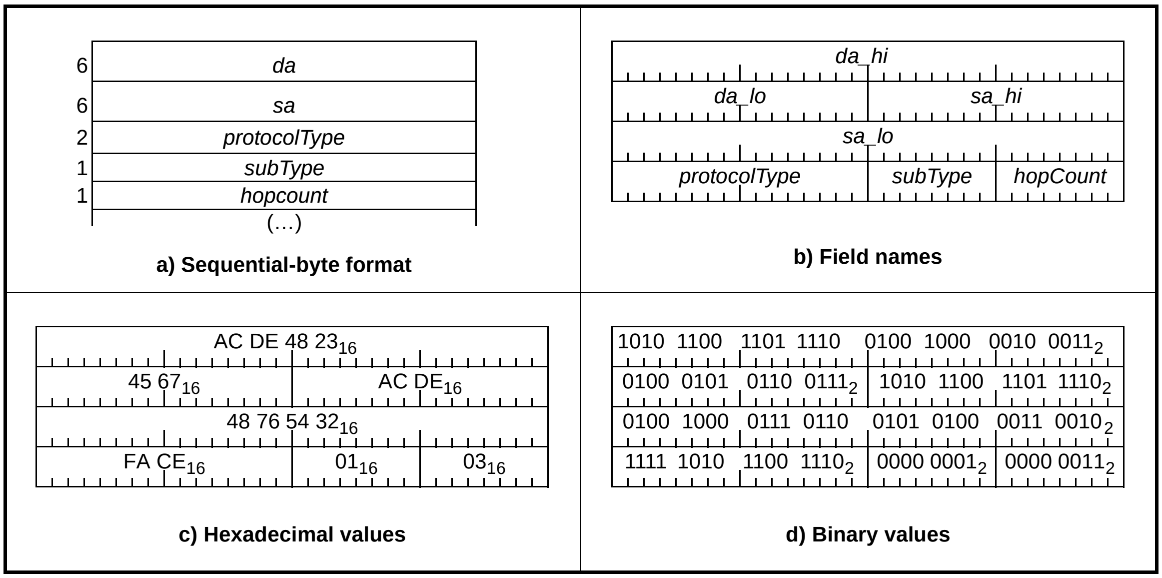


Figure 3.5—Illustration of fairness-frame structure

3.11 MAC address formats

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

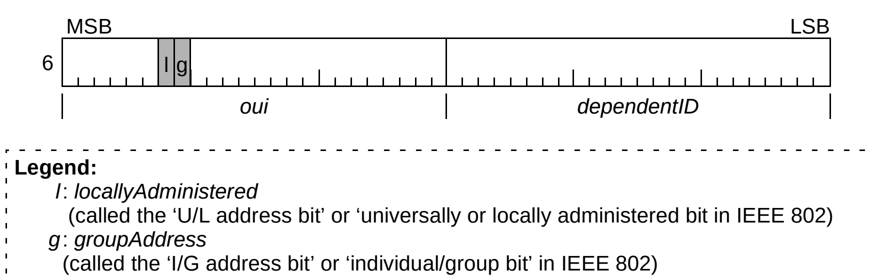


Figure 3.6—MAC address format

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.

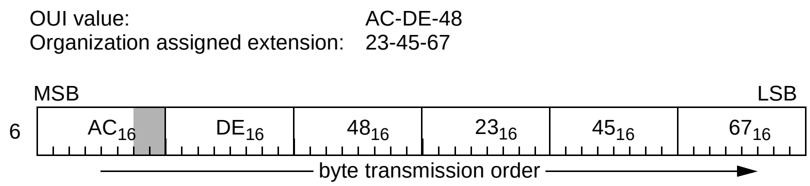


Figure 3.7—48-bit MAC address format

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 J.

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

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.

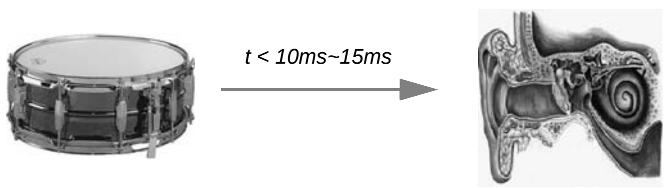


Figure 5.1—Interactive audio delay considerations

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.

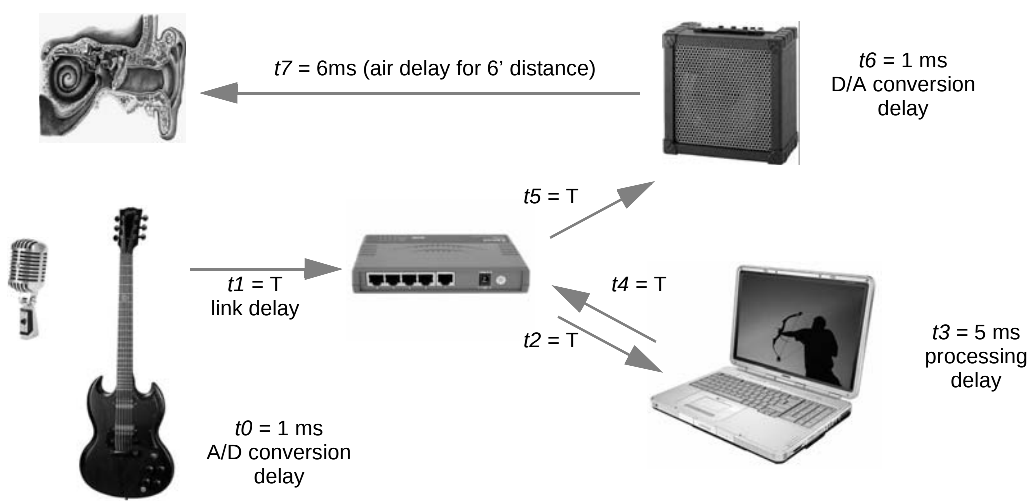


Figure 5.2—Home recording session

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.

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

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.

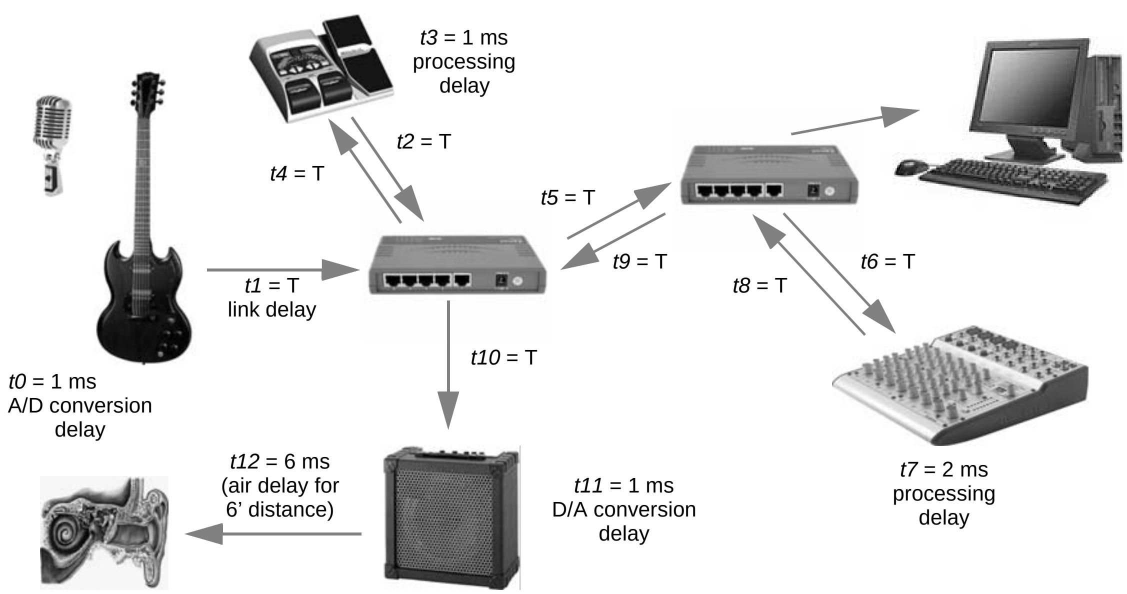


Figure 5.3—Garage jam session

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).

$$t0 + t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + t9 + t10 + t11 + t12 < 15 \text{ ms} \quad (5.2)$$

$$1ms + T + T + 1ms + T + T + T + 2ms + T + T + T + 1ms + 6ms < 15ms$$
$$8 \times T + 11ms < 15ms$$
$$T < 0.5\ ms$$

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.

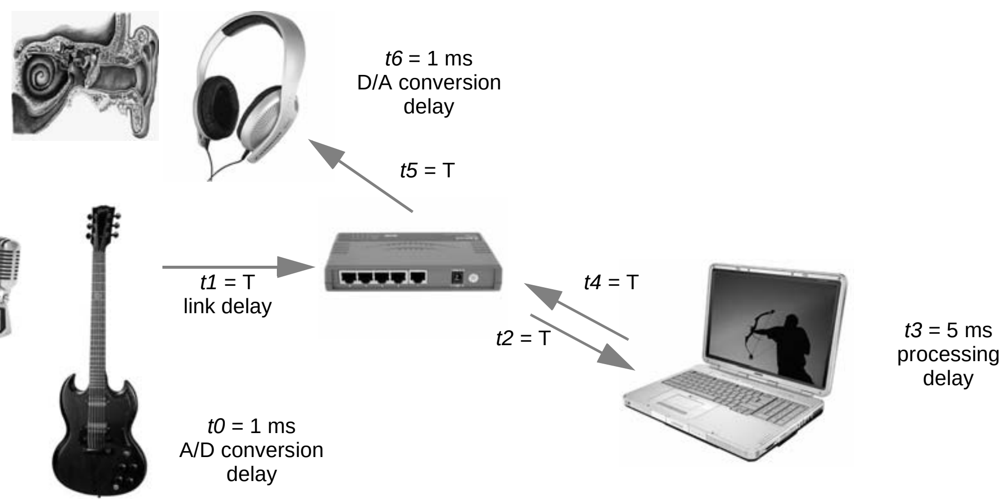


Figure 5.4—Urban recording session

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).

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

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.

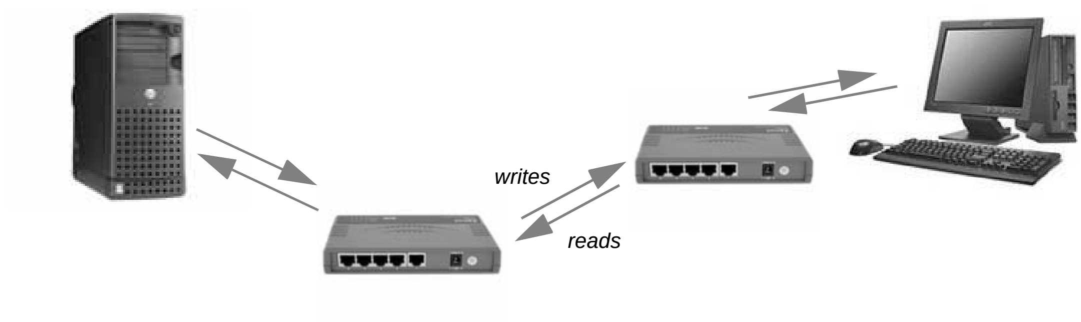


Figure 5.5—Conflicting data transfers

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.

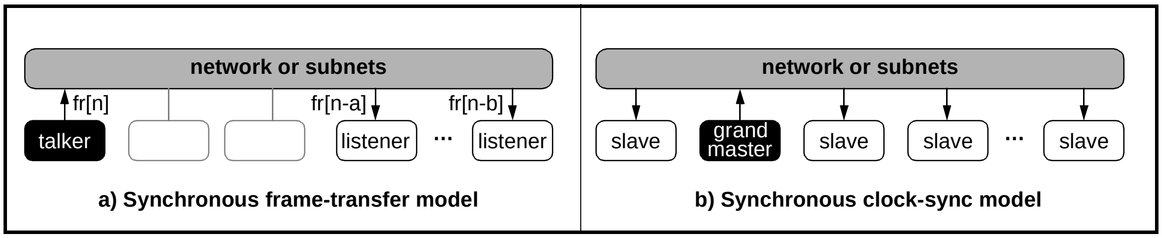


Figure 5.6—Hierarchical control

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.

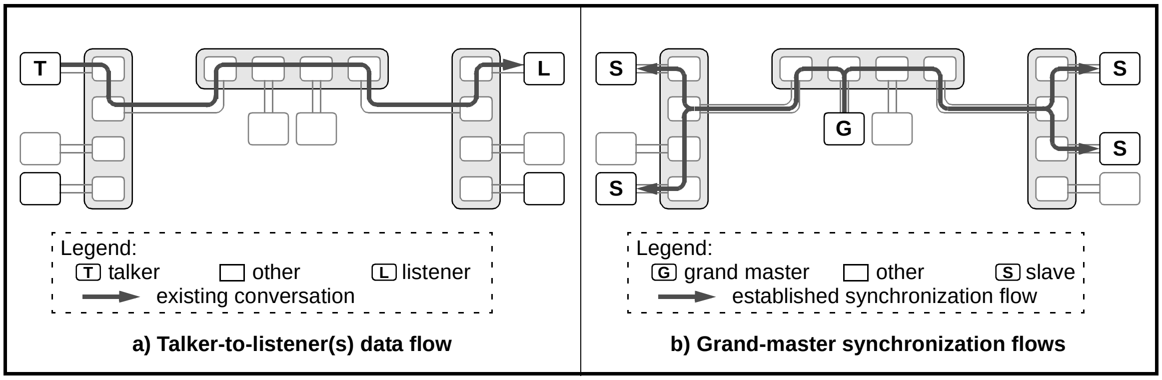


Figure 5.7—Hierarchical flows

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

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.

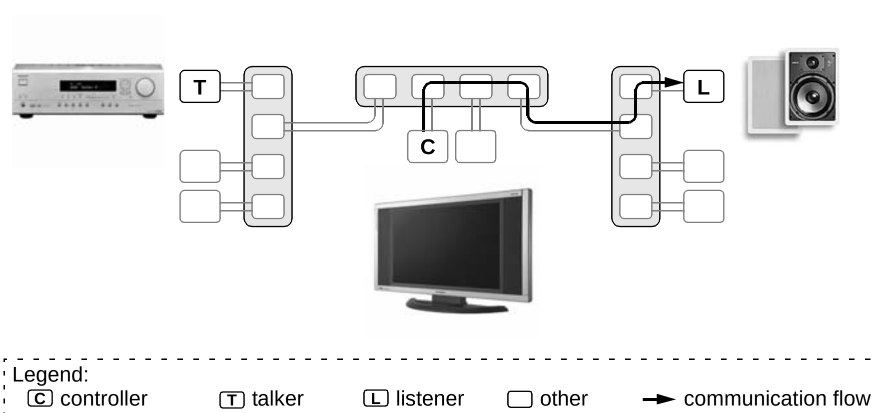


Figure 5.8—Controller activation

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.

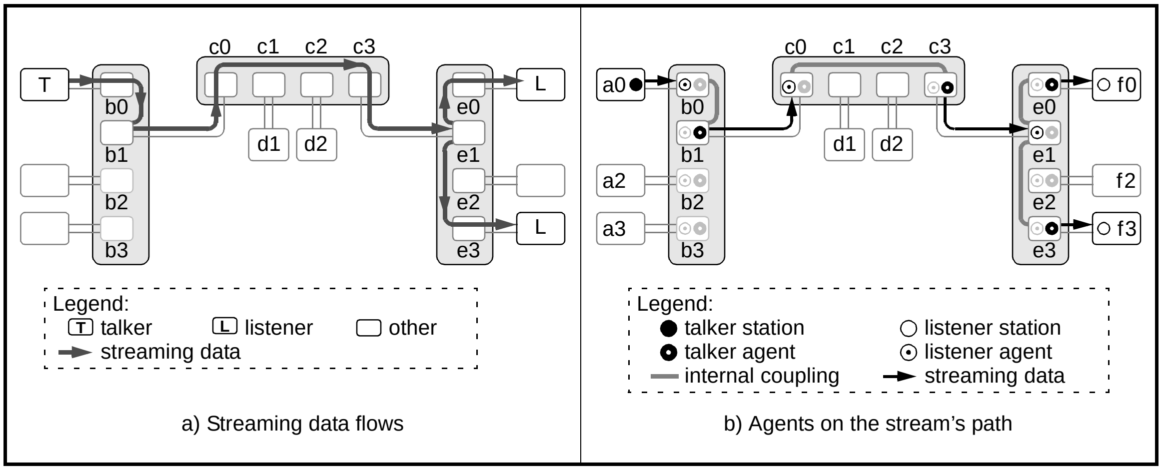


Figure 5.9—Agents on an established path

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.

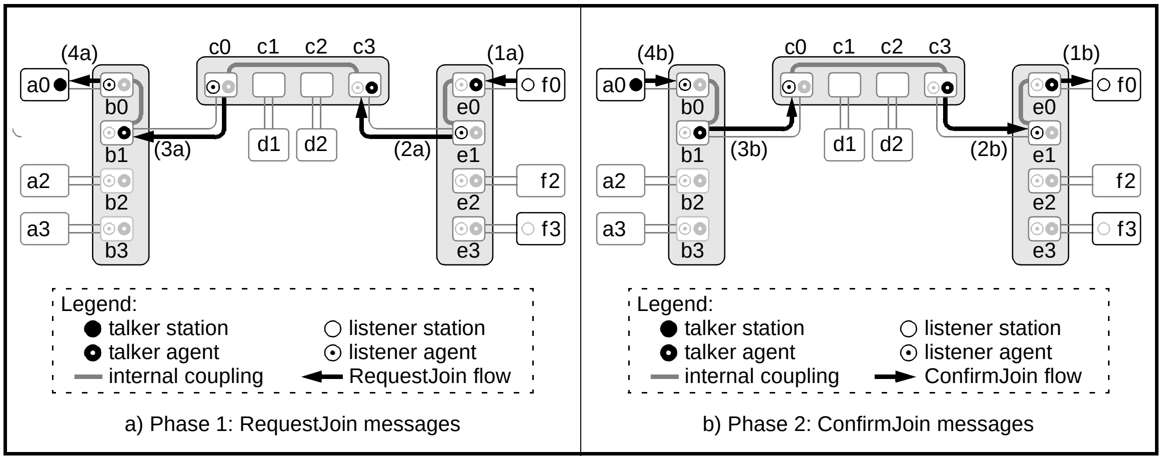


Figure 5.10—Periodic registration messages

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.

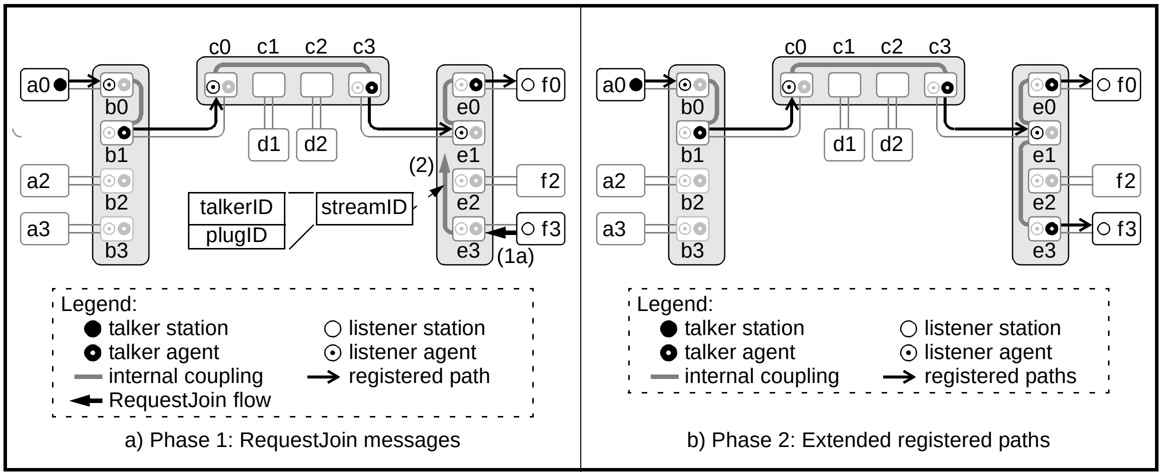


Figure 5.11—Secondary registrations

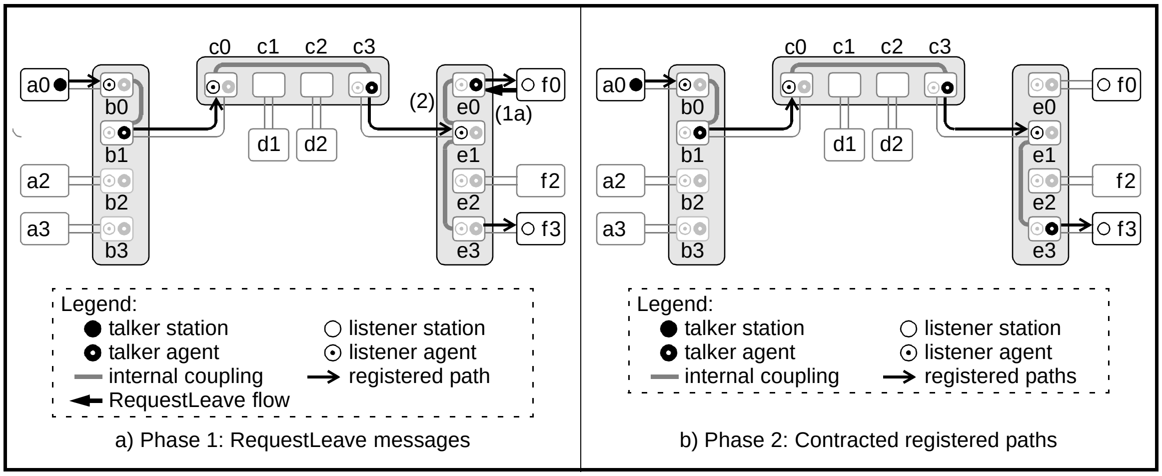
Each talker and listener agent maintains separate registration state, so that only active paths are registered. Maintaining distinct registrations also allows the bridge to detect when the last listener disconnects, so that its previously shared upstream span can be deregistered appropriately.

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.

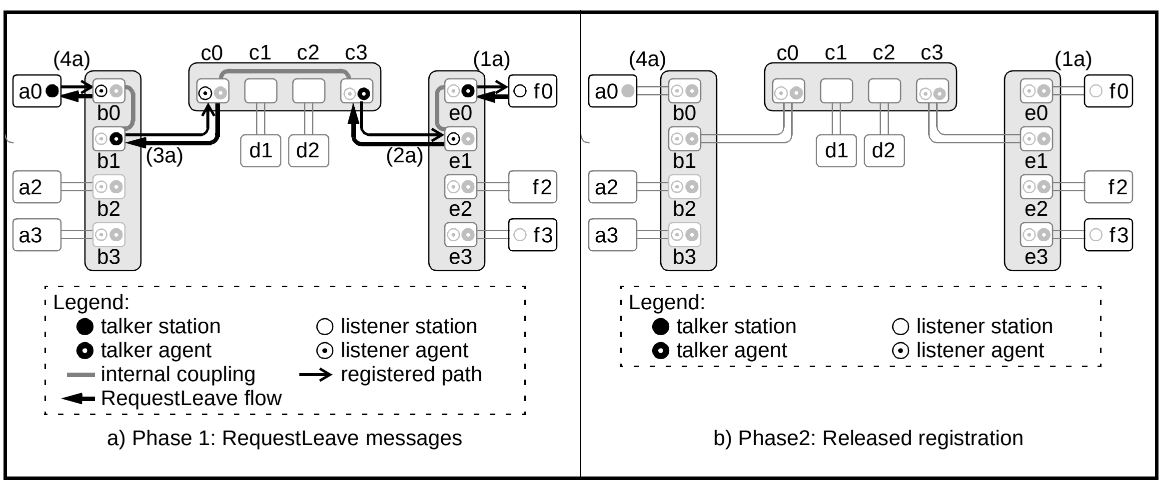


a) Phase 1: RequestLeave messages   b) Phase 2: Contracted registered paths

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.



a) Phase 1: RequestLeave messages   b) Phase2: Released registration

Figure 5.13—Final-path deregistration

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.

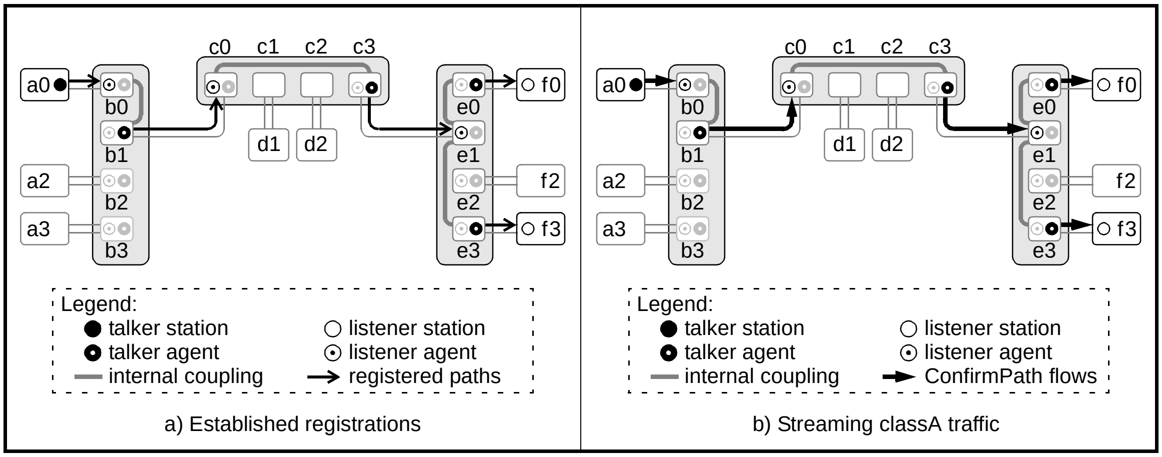


Figure 5.14—Streaming data over registered paths

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.

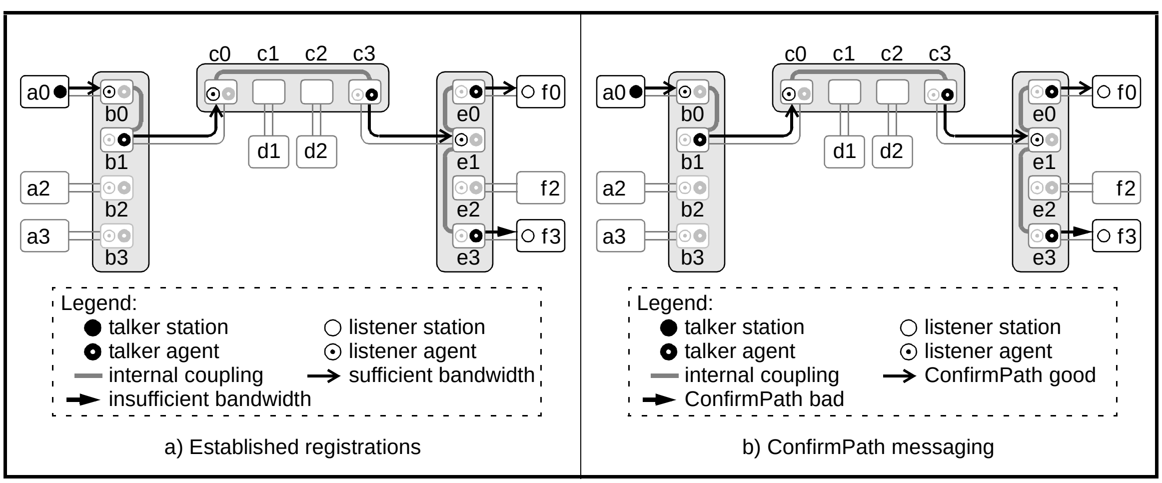


Figure 5.15—Insufficient bandwidth conditions

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.

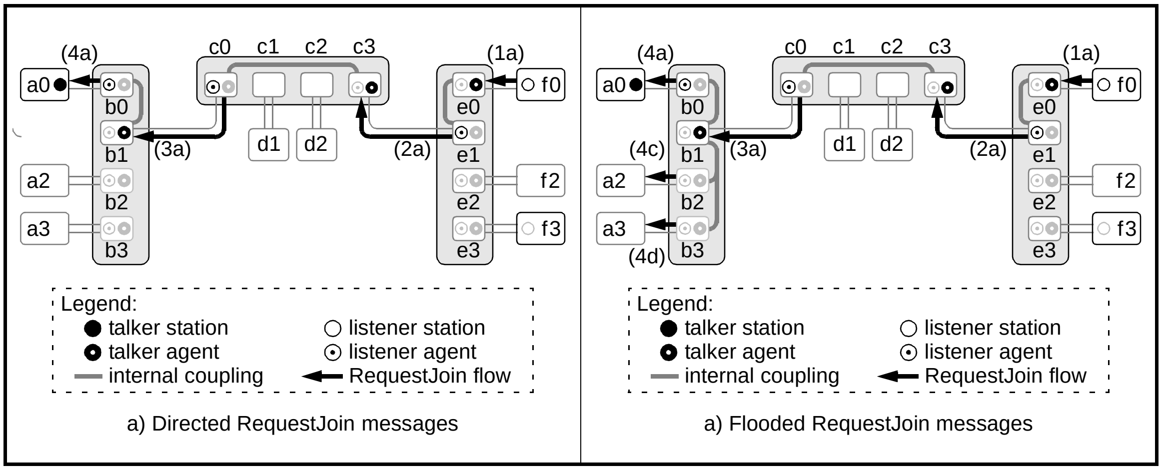


Figure 5.16—Periodic registration messages

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

5.5.1 Limitations of current approaches

5.5.1.1 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.

5.5.1.2 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.

5.5.1.3 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.

5.5.2 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.

5.5.3 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.

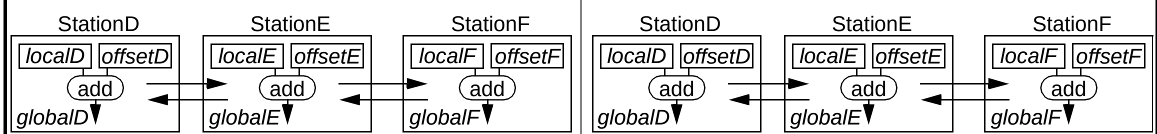
5.5.4 Strategies

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 (typically 1 kHz) 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.

5.5.5 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 5.17. 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).



globalE = globalD
= localD + offsetD
= localD – (localE – localE) + offsetD
= (localD – localE) + localE + offsetD
= localE + offsetE
Where:
offsetE = offsetD – (localE – localD)

a) StationE synchronizes to StationD

globalF = globalE
= localE + offsetE
= localE – (localF– localF) + offsetE
= (localE – localF) + localF + offsetE
= localF + offsetF
Where:
offsetF = offsetE – (localF – localE)

b) StationF synchronizes to StationE

Figure 5.17—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 5.17-a. Synchronization is thus possible using periodic transfers of *offsetD* values and computations of (*localE*–*localD*) timer

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 5.17-b. Synchronization is thus possible using periodic transfers of *offsetE* values and computations of (*localF*–*localE*) timer differences.

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.

5.5.6 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 5.18a. 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.

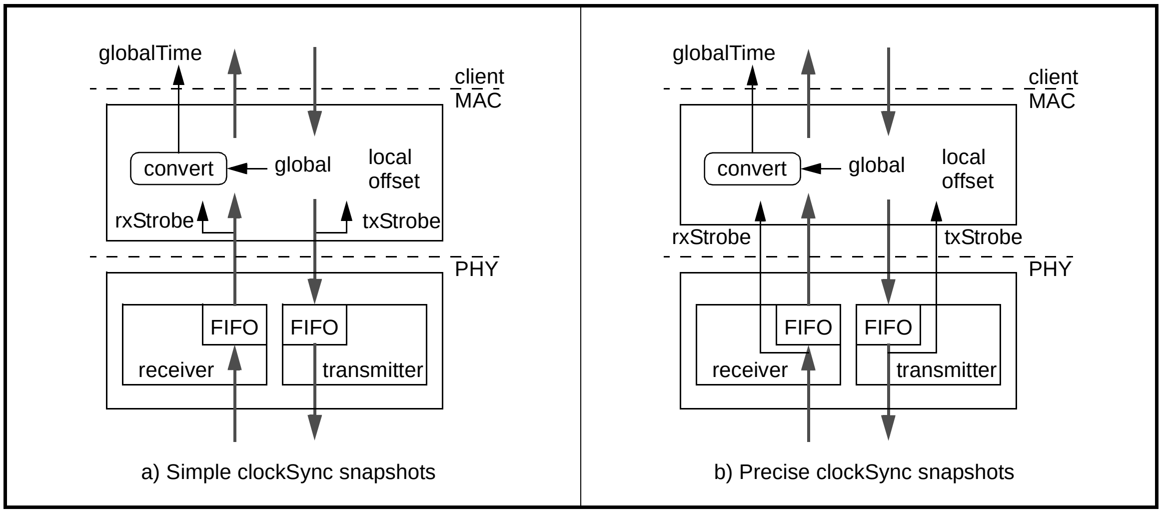


Figure 5.18—Timer snapshot locations

5.5.7 Bridge PLL possibilities

In addition to other valuable properties, the precise low-latency time-of-day synchronization protocols reduce jitter sufficiently to eliminate the needs for PLLs within bridges, as illustrated in Figure 5.19a. Elimination of such PLLs (illustrated in Figure 5.19b) simplifies the bridge design, while allowing each end-point application to independently optimize the effective capture-time and jitter-magnitude requirements of its PLL.

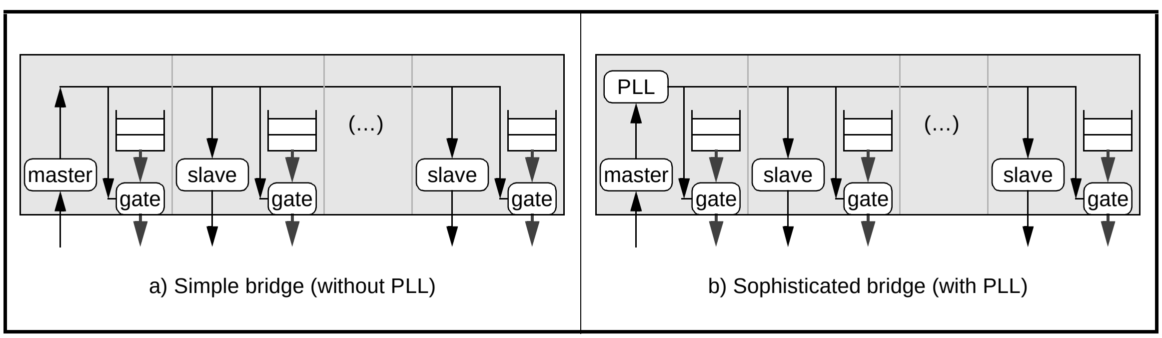


Figure 5.19—Bridge PLL possibilities

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 5.20.

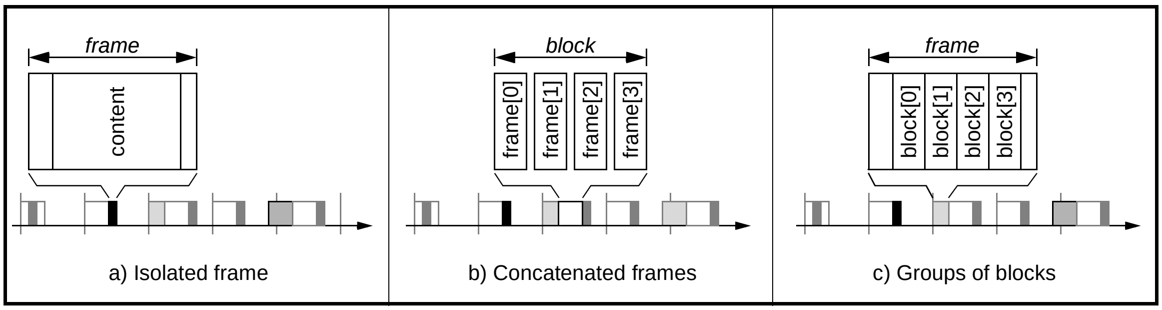


Figure 5.20—Content framing methods

For low bandwidth transmissions, each frame transports distinct classA content, as illustrated in Figure 5.20-a. For high bandwidth transmissions, the content can span multiple frames, as illustrated in Figure 5.20-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 5.20-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 5.21. 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.

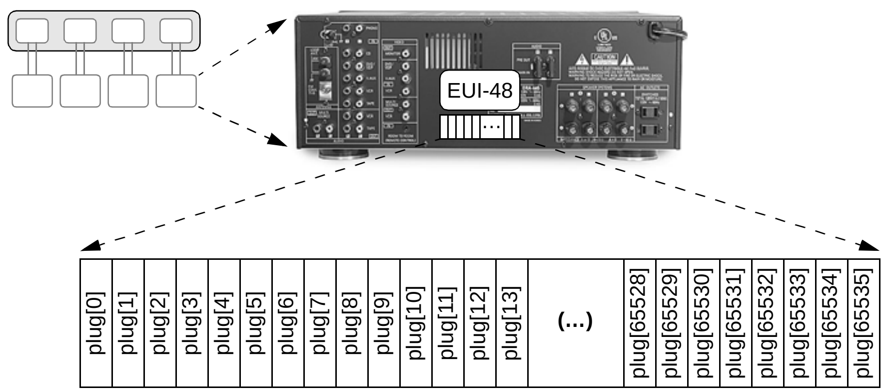


Figure 5.21—Plug addressing

5.6.3 Stream frame formats

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 bridges, as illustrated in Figure 5.22.

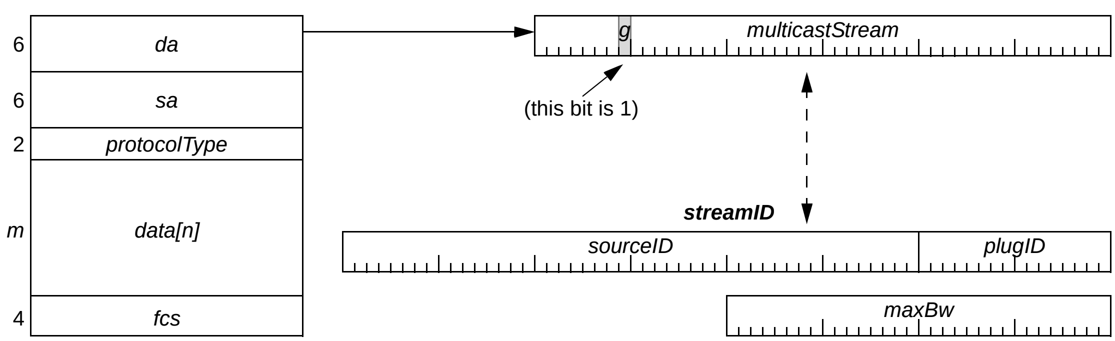


Figure 5.22—ClassA frame format and associated data

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.

This design approach (which relies on the multicast nature of classA streams) has desirable properties:

a) Uniform. Using a multicast *da* is consistent with forwarding database use on existing bridges.

b) Efficient. The inclusion of a *protocolType* field to identify a frame's classA nature is unnecessary. Efficiency reduces the need for bridge-aware multi-block frame formats (see 5.3.3).

c) Structured. The stacking order of *protocolType* values is unaffected by its classA nature.

5.7 Pacing

5.7.1 Pacing

Pacing involves the throttling of classA streams so that their average bandwidth can be guaranteed over small averaging intervals. Such fine-grained pacing has the following advantages:

a) Latency. Talker-to-listener delays are small, deterministic, and link-utilization independent.

b) Jitter. Delay variations between a talker and listeners are bounded and topology independent.

c) Intervals. Short bandwidth averaging intervals have several benefits:

   1) Short intervals simplify the detection/enforcement of maximum classA bandwidths.
   (A goal is to limit classA bandwidths to no more than 75% of the link capacity, see 1.2.3.)

   2) Subscription protocols (see 5.4) can base timeouts on detected talker absent/present conditions.

5.7.2 Talker and bridge pacing

An end station and bridge have similar transmit logic for classA and non-classA frames, as illustrated in Figure 5.23. Functionally distinct transmit queues are provided for classA and non-classA traffic, allowing each to be managed separately.

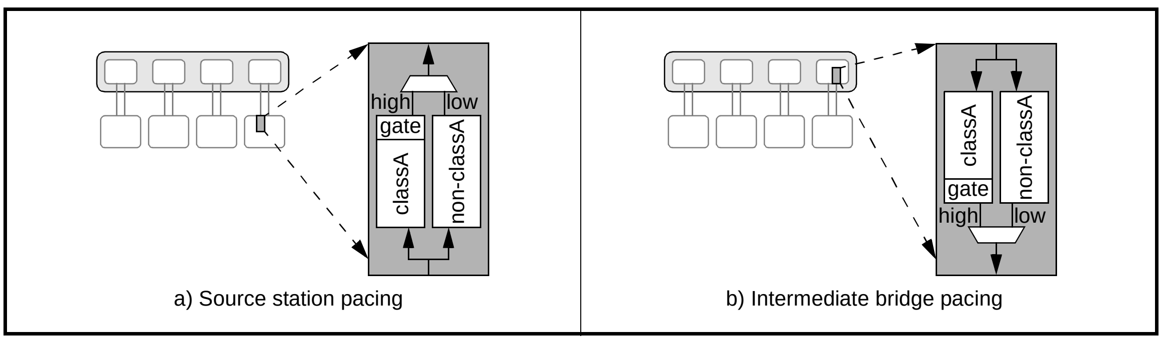


Figure 5.23—ClassA traffic pacing

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 end-station designs.

5.7.3 Quasi-synchronous classA flows

The group of classA frames sent once every cycle is called a group. Each group transports a clockSync frame (that provides cycle-count and clock-synchronization information) and one or more classA data frames. That classA data frame (illustrated in black) incurs fixed nominal delays when passing through bridges, as illustrated in Figure 5.24.

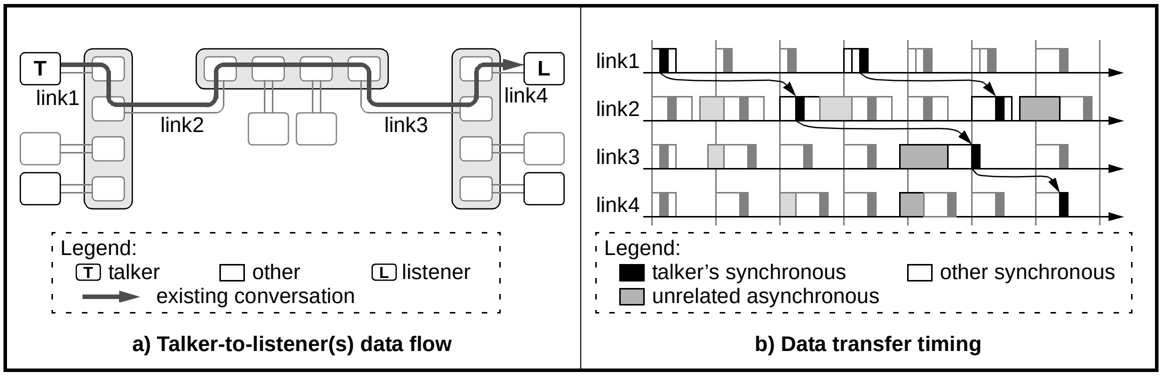


Figure 5.24—Quasi-synchronous classA deliveries: delay and jitter

Depending on the timing of unrelated events, the location of the classA-data frame within the group can migrate over time, as other conversations are started and/or ended, as illustrated by the black rectangle of the link1 timing sequence.

Similarly, the group transmission time within the nominal synchronous cycle may be delayed due to conflicts with other frame transmissions, as illustrated by the shaded rectangles of the link2 timing sequence. On occasion, conflicts with other frame transmissions can delay the classA block transmission into the next cycle, as illustrated near the end of the link3 timing sequence.

5.7.4 Traffic congestion points

Existing networks have multiple potential congestion points with respect to real-time data transmissions, as illustrated in Figure 5.25. ClassA traffic from the *a0* source must share link2 bandwidth with classA sources *a2* and *a3*. Similarly, classA link2 traffic must share link3 bandwidth with non-classA sources *b1* and *b2*. And, although more subtle, classA link3 traffic must share the bridgeC bridge-internal bandwidth from sources *c2* and *c3*.

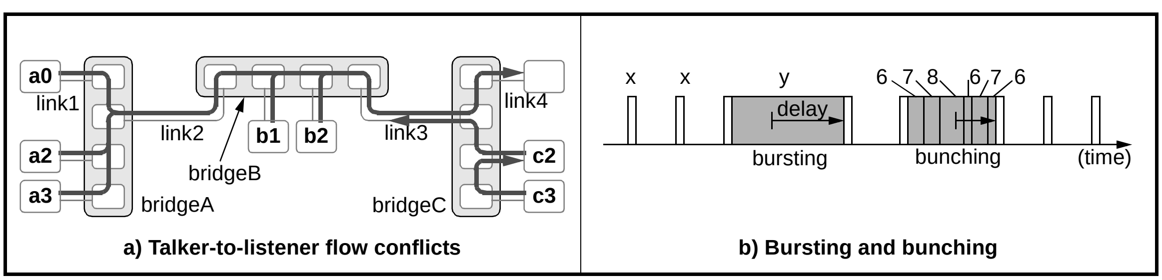


Figure 5.25—ClassA bandwidth considerations

The *a0* classA traffic is guaranteed by limiting the cumulative classA link bandwidths to no more than 75% of the shared link/bridge capacity, and forwarding classA traffic in a preferential manner. Cumulative limits imply bandwidth reservations; bandwidth reservations are expressed in terms of bytes-per-second, but are enforced in terms of bytes-per-cycle, where all stations agree on a common cycle duration.

Bandwidth reservations are sometimes insufficient to ensure expected classA behaviors; bursting and bunching are also potential problems. Bursting involves large packet transmissions, which interfere with the fixed-rate transmission of smaller frames, as illustrated by the y frame in Figure 5.25-b. Bunching involves the near simultaneous arrival of slow and fast arrivals, with the effective behavior of a burst, as illustrated by the cycle[6],cycle[7],cycle[8] arrivals in Figure 5.25-b. See Annex F for worst-case bursting and bunching scenario details.

Dealing with bursting and bunching is similar to designing clocked synchronous systems: data is updated based on a common clock, causing fast and slow computations to flow through pipeline stages with the same fixed delays.

6. Frame formats

6.1 ClassA frames

6.1.1 ClassA frame fields

A classA frame differs from other frames in the format of its multicast *da* (destination address), as illustrated in Figure 6.1.

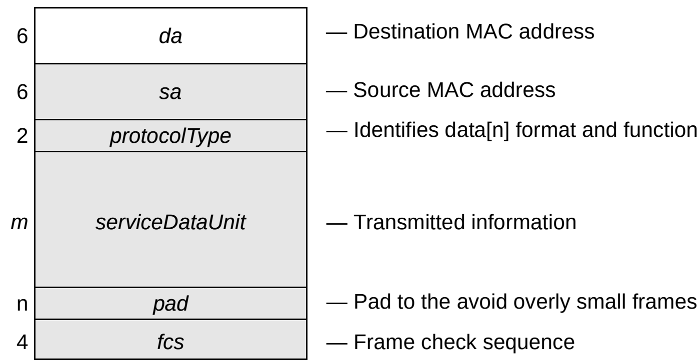


Figure 6.1—ClassA frame formats

6.1.1.1 ***da*****:** A 6-byte (destination address) field that specifies a multicast address associated with the stream.

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. When the value of *protocolType* is greater than or equal to 1536 (600_{16}) the *protocolType* field indicates the nature of the MAC client protocol (type interpretation), selecting from values designated by the IEEE Type Field Register. When less than 1536 (0_{16}–$5FF_{16}$), the *protocolType* is interpreted as the length of the frame (length interpretation). The length and type interpretations of this field are mutually exclusive.

6.1.1.4 ***serviceDataUnit*****:** An m-byte field the contains the service data unit provided by the client.

6.1.1.5 ***pad*****:** If the sum of the other field lengths is less than 64 bytes, then the number of zero-valued *pad* bytes are sufficient to make a 64-byte frame. Otherwise, the *pad* field is not present.

6.1.1.6 ***fcs*****:** A 4-byte (frame check sequence) field whose 32-bit CRC covers the frame’s content.

6.2 clockSync frame format

6.2.1 clockSync fields

Clock synchronization (clockSync) 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 6.2. The gray boxes represent physical layer encapsulation fields that are common across all Ethernet frames.

6	*da*	— Destination MAC address
6	*sa*	— Source MAC address
2	*protocolType*	— Distinguishes RE frames from others (see 6.7.1)
1	*subType*	— Distinguishes clockSync from other RE frames (see 6.7.2)
1	*syncCount*	— Sequence number, for error detection
1	*hopsCount*	— Hop count from the grand master
1	*reserved*	— Hop count from the grand master
4	*systemTag*	— Reserved for revisions&enhancements
8	*uniqueID*	— Precedence for grand master election
8	*lastFlexTime*	— Incoming link's frame transmssion time (1 cycle delayed)
8	*deltaTime*	— Outgoing link's frame propagation time
8	*offsetTime*	— Offset time within the neighbor
4	*diffRate*	— Cumulative rates from the grand-master
4	*lastBaseTime*	— Incoming link's frame transmssion time (1 cycle delayed)
4	*fcs*	— Frame check sequence

Figure 6.2—clockSync frame format

6.2.1.1 *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.

6.2.1.2 *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.

6.2.1.3 *protocolType*: A 16-bit field contained within the payload that identifies the format and function of the following fields (see 6.7.1).

6.2.1.4 *subType*: A 16-bit field that identifies the format and function of the following fields (see 6.7.2).

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

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

6.2.1.7 ***reserved*****:** An 8-bit field that shall be cleared to zero when the frame is generated and ignored when the frame is processed.

6.2.1.8 ***systemTag*****:** A 16-bit field that has highest precedence in the grand-master selection protocols.

6.2.1.9 ***uniqueID*****:** A 64-bit field that specifies the precedence of the grand clock master, specified in 6.2.3.

6.2.1.10 ***lastFlexTime*****:** A 64-bit field that specifies the time within the source station when the previous clockSync frame was transmitted. The format of this field is specified in 6.2.4.

6.2.1.11 ***deltaTime*****:** A 64-bit field that specifies the differences between clockSync receive and transmit times, as measured on the opposing link. The format of this field is specified in 6.2.4.

6.2.1.12 ***offsetTime*****:** A 64-bit field that specifies the offset time within the source station. The format of this field is specified in 6.2.4.

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

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

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

6.2.2 *systemTag* subfields

The format of the 16-bit *systemTag* field is based on the format of the spanning tree protocol precedence value, as illustrated in Figure 6.3.

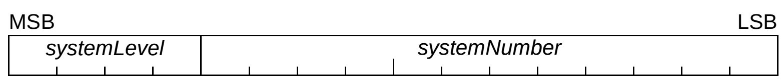


Figure 6.3—*systemTag* subfields

6.2.2.1 ***systemLevel***: A 4-bit field that comprise a settable priority component that permits the relative priority of bridges to be managed.

6.2.2.2 ***systemNumber*****:** A 12-bit field that comprise a locally assigned system identifier extension. (The term *systemID* is equivalent to 'system ID', as specified within IEEE Std 802.1D-2004.)

6.2.3 *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 6.4.

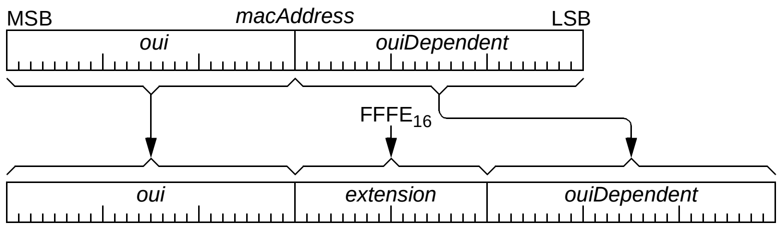


Figure 6.4—*uniqueID* format

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

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

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

6.2.4 Time field formats

Time-of-day values within a frame are specified by 64-bit values, consistent with IETF specified NTP[B7] and SNTP[B8] protocols. These 64-bit values consist of two components: a 32-bit *seconds* and 32-bit *fraction* fields, as illustrated in Figure 6.5.

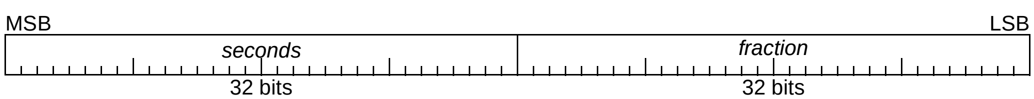


Figure 6.5—Complete seconds timer format

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

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

The concatenation of 32-bit *seconds* and 32-bit *fraction* field specifies a 64-bit *time* value, as specified by Equation 6.1.

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

Where:

seconds is the most significant component of the time value (see Figure 6.5).
fraction is the less significant component of the time value (see Figure 6.5).

6.3 RequestRefresh subscription frame

Editors' Notes: To be removed prior to final publication.
The following format definitions have not been updated to track recent subscription changes and have thus become obsolete.

6.3.1 RequestRefresh fields

RequestRefresh subscription frames contain channel-acquisition information, as illustrated in Figure 6.6.

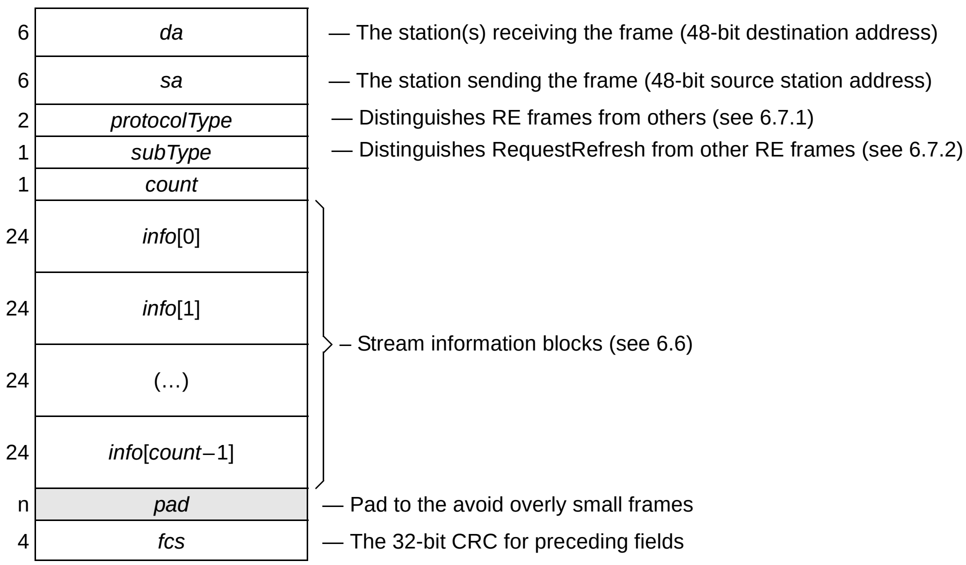


Figure 6.6—RequestRefresh frame format

6.3.1.1 *da*: A 6-byte (destination address) field that normally specifies the destination address for the frame transmission, with unicast and multicast forms. For the RequestRefresh frame, the *da* represents the ultimate destination of the talker.

6.3.1.2 *sa*: A 6-byte (source address) field that normally specifies the source address for the frame transmission. If a bridge is present between the frame and its associated listener, the *sa* value identifies the bridge.

6.3.1.3 *protocolType*: A 2-byte field that normally specifies the frame length, or the format and function of the following fields (excluding the 4-byte *fcs* field). This RE assigned value distinguishes its frame formats from others (see 6.7.1).

6.3.1.4 *subType*: A 1-byte field that distinguishes the ResponseError frame from other frames defined within this working paper.

6.3.1.5 *count*: A 1-byte field that specifies the number of elements within the following *info*-block array.

6.3.1.6 *info*: A 24-byte array element that provides listener subscription information (see 6.6).

6.3.1.7 *pad*: If the sum of the other field lengths is less than 64 bytes, then the number of zero-valued *pad* bytes are sufficient to make a 64-byte frame. Otherwise, the *pad* field is not present.

6.3.1.8 *fcs*: The 4-byte (frame check sequence) field whose 32-bit CRC covers the frame's content. For RE content frames, the standard definition applies.

6.4 RequestLeave subscription frame

The RequestLeave subscription frames contain channel-release information, as illustrated in Figure 6.7.

6	*da*	— The station(s) receiving the frame (48-bit destination address)
6	*sa*	— The station sending the frame (48-bit source station address)
2	*protocolType*	— Distinguishes RE frames from others (see 6.7.1)
1	*subType*	— Distinguishes RequestLeave from other RE frames (see 6.7.2)
1	*reservedA*	— Reserved
24	*info*	— Stream information block (see 6.3.2)
20	*reservedB*	— Pad to the avoid overly small frames
4	*fcs*	— The 32-bit CRC for preceding fields

Figure 6.7—RequestLeave subscription frame format

6.4.1 *da*: A 6-byte (destination address) field that specifies the span-local destination address for the frame transmission. For the RequestRefresh frame, the *da* represents the ultimate destination of the talker.

NOTE—ResponseError frames are only returned to their transmitting source, which could be a bridge's listener agent or the listener station. In the case of a listener agent, the bridge is responsible for forwarding similar messages downstream, based on the databases information contained within each of this stream's associated talker agents.

6.4.2 *sa*: A 6-byte (source address) field that specifies the span-local source address for the frame transmission. If a bridge is present between the frame and its associated listener, the *sa* value identifies the bridge.

6.4.3 *protocolType*: A 2-byte field that normally specifies the frame length, or the format and function of the following fields (excluding the 4-byte *fcs* field). This RE assigned value distinguishes these frame formats from those defined by other standards (see 6.7.1).

6.4.4 *subType*: A 1-byte field that distinguishes the ResponseError frame from other frames defined within this working paper (see 6.7.2).

6.4.5 *reservedA*: A 1-byte zero-valued field that is ignored when the frame is processed.

6.4.5.9 *info*: A 24-byte array element that provides listener subscription information (see 6.6).

6.4.6 *reservedB*: A 2-byte field reserved for future extensions of this working paper.

6.4.7 *fcs*: The 4-byte (frame check sequence) field whose 32-bit CRC covers the frame's content. For RE content frames, the standard definition applies.

6.5 ResponseError subscription frame

The ResponseError subscription frames contain channel-release information, as illustrated in Figure 6.8.

6	*da*	— The station(s) receiving the frame (48-bit destination address)
6	*sa*	— The station sending the frame (48-bit source station address)
2	*protocolType*	— Distinguishes RE frames from others (see 6.7.1)
1	*subType*	— Distinguishes ResponseError from other RE frames (see 6.7.2
1	*errorCode*	— Reserved
24	*info*	— Stream information block (see 6.3.2)
20	*reservedB*	— Pad to the avoid overly small frames
4	*fcs*	— The 32-bit CRC for preceding fields

Figure 6.8—ResponseError subscription frame format

6.5.1 *da*: A 6-byte (destination address) field that specifies the span-local destination address for the frame transmission. If a bridge is present between the frame and its associated listener, this value identifies the bridge.

NOTE—ResponseError frames are only returned to their transmitting source, which could be a bridge's listener agent or the listener station. In the case of a listener agent, the bridge is responsible for forwarding equivalent messages downstream, based on the databases information contained within each of this stream's associated talker agents.

6.5.2 *sa*: A 6-byte (source address) field that specifies the span-local source address for the frame transmission. If a bridge is present between the frame and its associated talker, the *sa* value identifies the bridge.

6.5.3 *protocolType*: A 2-byte field that normally specifies the frame length, or the format and function of the following fields (excluding the 4-byte *fcs* field). This RE assigned value distinguishes these frame formats from those defined by other standards (see 6.7.1).

6.5.4 *subType*: A 1-byte field that distinguishes the ResponseError frame from other frames defined within this working paper (see 6.7.2).

6.5.5 *errorCode*: A 1-byte field that distinguishes between error types.

6.5.5.10 *info*: A 24-byte array element that provides listener subscription information (see 6.6).

6.5.6 *reservedB*: A 24-byte field reserved for future extensions of this working paper.

6.5.7 *fcs*: The 4-byte (frame check sequence) field whose 32-bit CRC covers the frame's content. For RE content frames, the standard definition applies.

6.6 Common *info* field format

Many frame transports an array of one or more *info*[ ] fields, whose content is illustrated in Figure 6.9.

6	*mcastID*	— Multicast destination label
6	*talkerID*	— Multicast talker identifier
2	*plugID*	— Resource within the talker
2	*maxCycles*	— Delay from the talker
4	*maxBw*	— Maximum required bandwidth
4	*reserved*	— Reserved

Figure 6.9—Common *info* field format

6.6.1 *mcastID*: A 6-byte (multicast identifier) field that routes frames betwee the talker and audience.

6.6.2 *talkerID*: A 6-byte field that identifies the stream's talker.

6.6.3 *plugID*: A 16-bit field that specifies the plug identifier within the talker.

The concatenation of the 48-bit *talkerID* and 16-bit *plugID* fields forms a 64-bit *streamID* that uniquely identifies the classA multicast stream.

6.6.4 *maxCycles*: A 2-byte field that is updated by bridges, as the RequestRefresh flows from the talker to the listener, allowing the maximum number of delay cycles between the talker and listener stations to be known to the talker.

6.6.5 *maxBw*: A 4-byte field that specifies the level of negotiated classA bandwidth, measured in bytes of per-cycle content.

6.6.6 *reserved*: A 4-byte zero-valued field that is ignored.

6.7 Unique identifier values

6.7.1 *protocolType* identifier

> NOTE—The following *protocolType*-assignment text will ultimately be updated with assigned values.

The clockSync (see 6.2) and subscription (see 6.3) frames are distinguished from other frames by their 16-bit distinct *protocolType* value, as illustrated in Figure 6.10. The following 1-byte *subType* field further distinguishes between these uses (see 6.7.2).

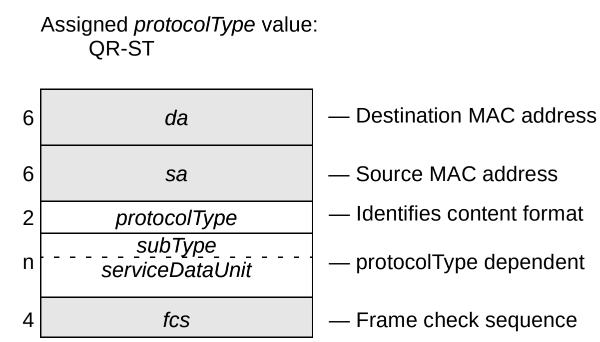


Figure 6.10—*protocolType* field value

6.7.2 *subType* identifier

Distinct *subType* identifiers distinguish between RE frame types, as specified by Table 6.1.

Table 6.1—Assigned *subType* identifiers

Value	Name	Row	See	Description
TBD	CLOCK_SYNC	1	6.2	Demarcates boundaries between isochronous cycles.
TBD	REQ_REFRESH	2	6.3	Subscription resource request.
TBD	REQ_LEAVE	3	6.4	Subscription resource release.
TBD	RES_ERROR	4	6.5	Subscription error response.
192-255	E1394	5	C.2.2	Encapsulated IEEE 1394 packet (or portion of 1394 packet)

7. Clock synchronization

7.1 Clock-synchronization overview

7.1.1 Clock synchronization services

Clock synchronization involves the transmission and reception of clockSync 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.

7.1.2 Clock-synchronization agents

Clock-synchronization information conceptually flows from a grand-master station to clock-slave stations, as illustrated in Figure 7.1a. A more detailed illustration shows pairs of synchronized clock-master and clock-slave components, as illustrated in Figure 7.1b.

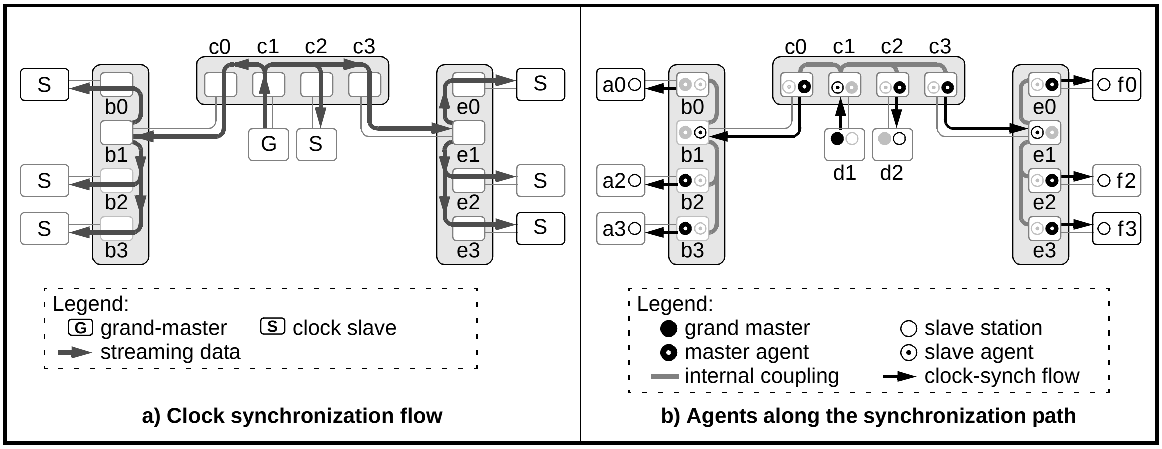


Figure 7.1—Hierarchical flows

7.1.3 Clock-synchronized pairs

Each bridge port provides clock-master and clock-slave agents, although both are never simultaneously active. External communications (see 7.1b) 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 7.1.5); the adjustments of the timer's rate is called rate synchronization (see 7.1.7). 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 J.

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

7.1.4 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 clockSync 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 clockSync 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.

7.1.5 Offset synchronization

Offset synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.4. 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.

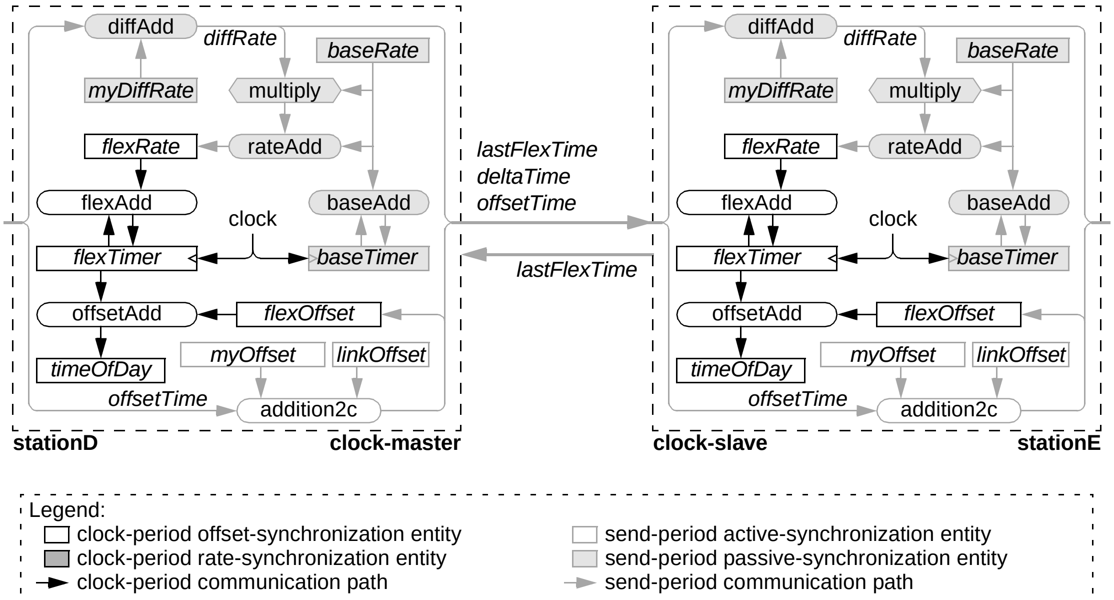


Figure 7.2—Offset synchronization

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.

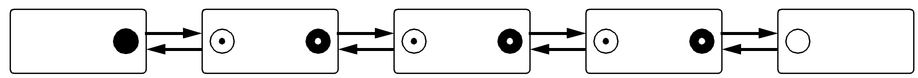
The offset-compensation protocols for stationE adjust its *myOffset* value so that the instantaneous values of *stationE.timeOfDay* and *stationD.timerOfDay* are the same. Computations are performed on clockStrobe reception and clockStrobe transmission.

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 *myOffset*, 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).

7.1.6 Cascaded offsets

The concept of cascaded offset values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.3. The slave-agent in bridgeB is synchronized to its neighbor grand-master via clockSync 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 clockSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the rightmost clock-slave station to the leftmost grand-master station.

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

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

Figure 7.3—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 *myOffset* 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 *myOffset* 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.

7.1.7 Rate synchronization

Rate synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.4. The free-running *baseTimer* timer facilitate the determination of rate differences between the clock-master and clock-slave stations.

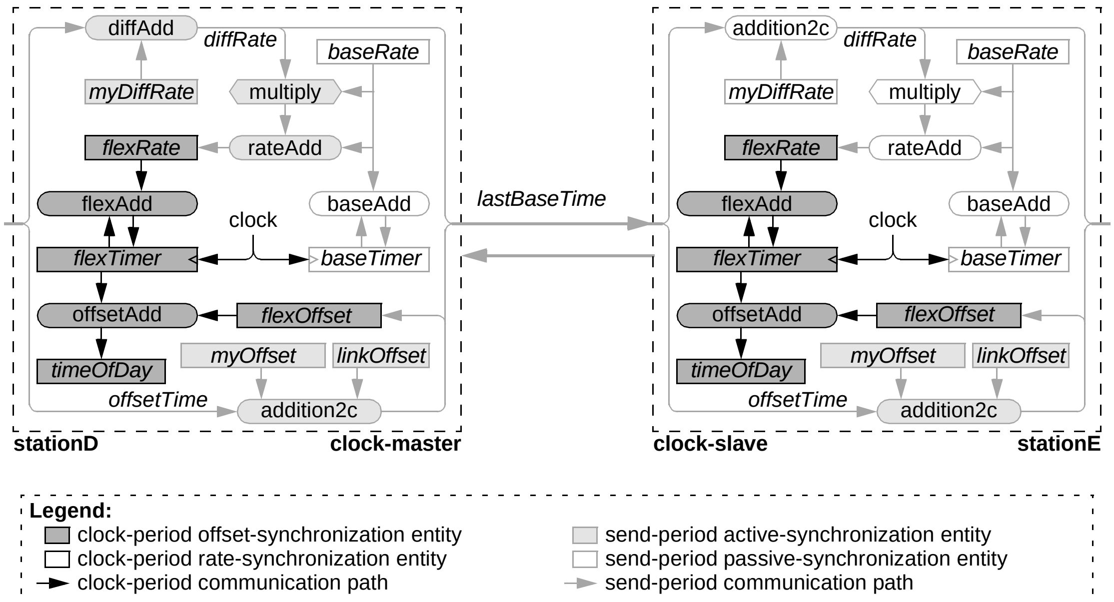


Figure 7.4—Rate synchronization

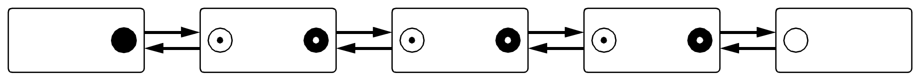
The rate-synchronization protocols interchange parameters periodically, but less frequently than the offset-synchronization protocols, possibly every 100 ms. The *lastBaseTime* value is sent periodically from the clock-master to the clock-slave. Nothing is returned from the clock-slave station.

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).

7.1.8 Cascaded rate differences

The concept of cascaded rate values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.5. 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 7.5—Cascaded rate differences (a possible scenario)

The slave-agent in bridgeB is synchronized to its neighbor grand-master via clockSync 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 clockSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the rightmost clock-slave station to the leftmost 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.

7.1.9 Rate-difference effects

If the absence of rate adjustments, significant *timeOfDay* errors can accumulate between send-period updates, as illustrated on the leftside of Figure 7.6. 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.

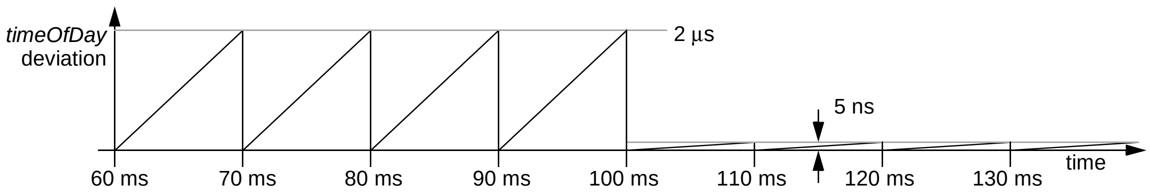


Figure 7.6—Rate-adjustment effects

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.

7.1.10 *flexTimer* implementation example

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 7.7. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

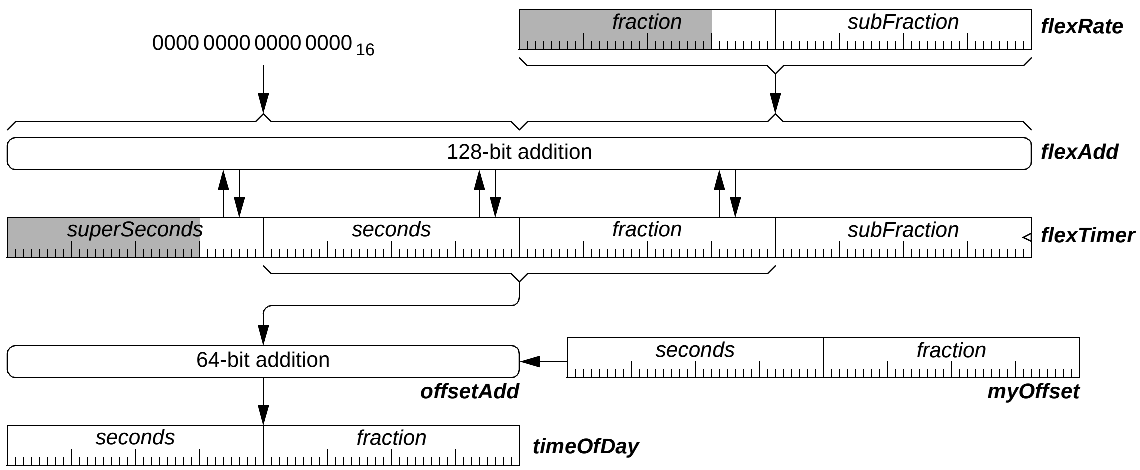


Figure 7.7—*flexTimer* implementation example

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).

7.1.11 An alternative *baseTimer* implementation

An alternative implementation could implement the *baseTimer*-related circuitry in hardware. For such implementations, the associated firmware can be simplified, since the multiplies are eliminated from the most frequently executed loop (see Annex J).

A possible *baseTimer* hardware implementation is much simpler than the fully adjustable timer implementation, due to the absence of offset-compensation, rate-compensation, and seconds-accumulation hardware, as illustrated in Figure 7.8. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

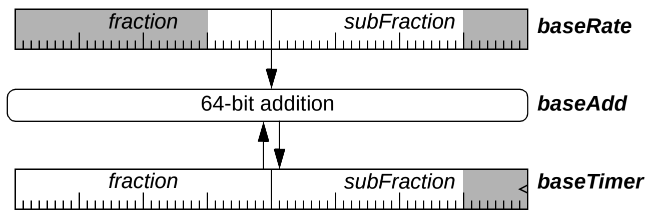


Figure 7.8—*baseTimer* implementation example

7.2 Terminology and variables

NOTE—The remainder of this clause should be skipped on the first reading (go to Annex B). Also, the remaining text has been obsoleted by recent timer-related changes.
Thus, the state tables only provide an indication of possible documentation styles and formats.

7.2.1 Common state machine definitions

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

CYCLES
    The number of isochronous cycles within each second; defined to be 8,000.
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_CRX_SYNC—The identifier associated with the received clockSync frames.
        Q_CTX_SYNC—The identifier associated with the transmitted clockSync frames.
        Q_ARX_REQ*—The identifier associated with the received subscription request frames.
        Q_ATX_REQ*—The identifier associated with the transmitted subscription request frames.
        Q_ATX_RES*—The identifier associated with the transmitted ResponseError frames.
        Q_ARX_STR*—The identifier associated with the talker agent's streaming input.
        Q_ATX_STR*—The identifier associated with the talker agent's streaming output.

NOTE—Those queue identifiers with an '*' are used in other clauses, but are described above. This allows all queue identification values in one location, rather than interleaving their definitions throughout this working paper.

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.

7.2.3 Common state machine routines

ClockSyncArrived(*stationInfoPtr*, *portInfoPtr*)
Snapshots the clockSync frame arrival time, on specified station and port (see Annex J).
ClockSyncDeparted(*stationInfoPtr*, *portInfoPtr*)
Snapshots the clockSync frame departure time, on specified station and port (see Annex J).
ClockSyncTransmit(*stationInfoPtr*, *portInfoPtr*, *clockSyncPtr*)
Forms a clockSync frame for transmission (see Annex J).
ClockSyncReceive(*stationInfoPtr, portInfoPtr, clockSyncPtr, rateAdjust*)
Processes a clockSync frame after reception (see Annex J).
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.
QueueEmpty(*queue*)
Indicates when the queue has emptied.
TRUE—The queue has emptied.
FALSE—(Otherwise.)
TimerTick(*stationInfoPtr*)
Updates *flexTimer* (and *baseTimer*) entities on each clock tick (see Annex J).

7.2.4 Variables and literals defined in other clauses

This clause references the following parameters, literals, and variables defined in Clause TBD:

TBDs

7.3 Clock synchronization state machines

7.3.1 ClockAgent state machine

7.3.1.1 ClockAgent state machine definitions

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

MSEC1
A constant representing the time duration value of 1 ms.
MSEC10
A constant representing the time duration value of 10 ms.
MSEC100
A constant representing the time duration value of 100 ms.

7.3.1.2 ClockAgent state machine variables

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

changing
A bit indicating the presence of a changed *bestPrecedence* selection evaluation.
0—The precedence selection criterial is stable.
1—(Otherwise.)

clockPeriod
The duration of a synchronized timer update interval.
clockPeriod < 20 ns

currentTime
See 7.2.2.

frame
The contents of a clockSync frame.

msCount
A count that is incremented at the end of each 1 millisecond interval.

rateFlag
An indication that triggers a rate-update, after observing a 100 ms rate-update gap.

rateCount
A milliseconds-snapshot taken during the clockSync receive processing.
The *rateCount* value paces the relatively infrequent rate-update computations.

rxClockLast
The previously observed value of *rxClockSync*, used to detect changes in this toggling value.

rxClockSync
An indication whose value is toggled on the PHY-sensed arrival of each clockSync frame.
This value is toggled before a frame can be dequeued from the Q_CRX_SYNC queue.

rxCount
A milliseconds-snapshot taken during clockSync receive processing.
The *rxCount* value paces the detection of clockSync-silence timeouts.

sendCount
A milliseconds-snapshot taken during the clockSync transmission processing.
The *sendCount* value paces the relatively clockSync frame transmissions.

tickTime
A time snapshot taken at the start of each clockPeriod interval.

txClockSync
An indication whose value is toggled on the PHY-sensed departure of each clockSync frame.
This value is toggled shortly after a frame has departed from the Q_CTX_SYNC queue.

txClockLast
The previously observed value of *txClockSync*, used to detect changes in this toggling value.

7.3.1.3 ClockAgent state machine routines

ClockSyncArrived(*stationInfoPtr*, *portInfoPtr*)
ClockSyncDeparted(*stationInfoPtr*, *portInfoPtr*)
ClockSyncTransmit(*stationInfoPtr*, *portInfoPtr*, *clockSyncPtr*)
ClockSyncReceive(*stationInfoPtr*, *portInfoPtr*, *clockSyncPtr*, *rateAdjust*)
ClockSyncTransmit(*stationInfoPtr*, *portInfoPtr*, *clockSyncPtr*)
See 7.2.3.

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

TimerTick(*stationInfoPtr*)
See 7.2.3.

7.3.1.4 ClockAgent state table

The ClockAgent state machine calls other C-code routines, as specified in Table 7.3. A purpose of the ClockAgent state machine is to ensure correctness of the other routines, by ensuring their indivisible executions. The notation used in the state table is described in 3.4.

Table 7.3—ClockAgent state table

Current state		Row	Next state	
state	**condition**		**action**	**state**
START	(currentTime – tickTime) >= clockPeriod	1	TimerTick(siPtr); tickTime = currentTime;	START
	(currentTime – msTime) >= .001	2	msTime = currentTime; msCount += 1;	
	rxClockSync != rxClockLast	3	ClockSyncArrived(siPtr, piPtr); rxClockLast = rxClockSync	
	txClockSync != txClockLast	4	ClockSyncDeparted(siPtr, piPtr); txClockLast = txClockSync	
	(frame = Dequeue(Q_CRX_SYNC)) != NULL	5	rxCount = sendCount;	FINAL

	changing &&
	(msCount – sendCount) >= 1
		6	changing = 0;
			sendCount = msCount;
			ClockSyncTransmit(siPtr, piPtr, &frame);
			Enqueue(Q_CTX_SYNC, frame);
				START

	(msCount – sendCount) >= 10	7		
	(msCount – rateCount) >= 100	8	rateFlag = 1;	
	(msCount – rxCount) >= 100	9	ClockSyncReceive(siPtr, piPtr,NULL, 0);	
	—	10	—	
NEAR	rateFlag	11	ClockSyncReceive(siPtr, piPtr, &frame, 1); rateFlag = 0;	FINAL
	—	12	ClockSyncReceive(siPtr, piPtr, &frame, 0);	

Row 7.3-1: Update the *flexTimer* and *baseTimer* once every *clockPeriod* interval.
Row 7.3-2: Update the millisecond counter once each millisecond interval.
Row 7.3-3: Process the PHY-generated signal to determine when the clockSync frame arrived.
Row 7.3-4: Process the PHY-generated signal to determine when the clockSync frame departed.
Row 7.3-5: When a clock-sync frame arrives, mark its arrival time and process.
Row 7.3-6: Transmit quickly when the grand-master selection is changing.
Row 7.3-7: Transmit routinely when the grand-master selection has stabilized.
Row 7.3-8: Trigger the rate adjustments on approximate 100 ms intervals.
Row 7.3-9: A port timeout occurs in the continued absence of clockSync frame arrivals.
Row 7.3-10: Otherwise, wait for the next event to occur.

Row 7.3-11: Restart the rate timeout condition after the last rate-measurement completion.
Row 7.3-12: Otherwise, wait for the next event to occur.

8. Subscription state machines

NOTE—This clause should be skipped on the first reading (continue with Annex B).
The following state machines were previously highly preliminary and subject to change.
They have not yet been updated to track on recent changes to the SRP, so they are also obsolete.
Thus, the structure and formatting is useful but the details should be ignored.

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

8.1 Terminology and variables

8.1.1 Common state machine definitions

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

NULL
  Indicates the absence of a value and (by design) cannot be confused with a valid value.
subtype specifiers
  ST_ERROR—A control response that provides an SRP refresh-operation error indication.
  ST_FRESH—A control request that provides blocks of SRP refresh parameters.
  ST_LEAVE—A control request that provides a block of SRP leave parameters.

8.1.2 Common state machine variables

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

localTimer
  A 64-bit timer representing the current 64-bit internal free-running time-of-day value.
myMacAddress
  MAC address of the bridge.
refreshFlag
  A variable that is toggled periodically; each change activates refresh interval activities.
srpState
  The information associated with an element of talker-agent state. This includes:
    maxBw—The maximum bandwidth of the associated stream.
    maxCycles—The maximum cycles to the attached listener.
    refreshTime—The time of the last observed RequestRefresh frame.
    srcPortID—The port identifier of the assumed source.
    srcMac—The address of the downstream bridge.
    state—The connectivity state, one of the following:
      IS_JOINING—Stream communications are now using this path.
      IS_LEAVING—Stream communication are no longer using this path.
      IS_FAILED—Stream communications have failed; message must be sent.
      IS_ACTIVE—Stream communications remain active.
      IS_PASSIVE—The SRP state is queued for deletion, behaving as though nonexistent.
    streamTime—The time of the last observed stream flow.
    streamID—The streamID of the associated stream.
    subCode—The error subcode associated with the IS_FAILED state.

8.1.3 Common state machine routines

StateSearch(*streamID*)
Returns the talker-state information associated with the specified stream value.
srpState—matching talker-agent state
NULL—no matching state found

8.1.4 Variables and literals defined in other clauses

This clause references the following parameters, literals, and variables defined in Clause 7

Dequeue(*queue*)
Enqueue(*queue*, *frame*)
localTimer
Q_ARX_REQ
Q_ATX_REQ
Q_ARX_STR
Q_ATX_STR
Q_ATX_RES

8.2 Subscription state machines

8.2.1 AgentAction state machine

The AgentAction state machine controls the sequencing of AgentTalker, AgentTimer, and AgentListener state machines. There are multiple instances of these state machine, one per bridge port, each of which is invoked. A refresh flag is also complemented at a regular interval.

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

8.2.1.1 AgentAction state machine definitions

–none–

8.2.1.2 AgentAction state machine variables

localTimer
refreshFlag
See 8.1.2.
refreshTime
The time when the last refresh was performed.
refreshTimeout
The time interval between successive refresh operations.

8.2.1.3 AgentAction state machine routines

AgentListeners()
A routine that calls all of the AgentListener state machines (one for each bridge port).
AgentTalkers()
A routine that calls all of the AgentTalker state machines (one for each bridge port).
AgentTimers()
A routine that calls all of the AgentTimer state machines (one for each bridge port).

8.2.1.4 AgentAction state table

The AgentAction state machine is specified in Table 8.1.

Table 8.1—AgentAction state table

Current state		Row	Next state	
state	condition		action	state
START	—	1	AgentTalkers(); AgentTimers(); AgentListeners();	LOOP
TIMER	(localTimer – refreshTime) >= refreshTimeout	2	refreshTime = localTimer; refreshFlag ^= 1;	FINAL
	—	3	—	

Row 8.1-1: Execute each of the AgentTalker, AgentTimer, and AgentListener state machines.

Row 8.1-2: Complement the refresh flag at the end of each refresh interval.
Row 8.1-3: Otherwise, wait until the arrival of the next refresh interval.

8.2.2 AgentTalker state machine

The AgentTalker state machine monitors received RequestRefresh and RequestLeave frames. There are multiple AgentTalker state machines per bridge, one for each of the bridge ports.

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

8.2.2.1 AgentTalker state machine definitions

IS_FAILED
IS_JOINING
IS_LEAVING
    See 8.1.2.
NULL
    Indicates the absence of a value and (by design) cannot be confused with a valid value.
Q_ARX_REQ
Q_ARX_STR
Q_ATX_STR
    See 8.1.4.
ST_REFRESH
ST_LEAVE
    See 8.1.1.
subCode field values
    SC_DA_LOST—No route to the specified destination is present.
    SC_DA_MINE—The route to the specified destination loops back.
    SC_BAD_HERE—This port’s SRP state has different parameters than the refresh request.
    SC_BW_LIMIT—The requested stream bandwidth would exceed 75% of the link capacity.
    SC_BAD_THERE—Another port’s SRP state has different parameters than the refresh request.
    SC_UP_FULL—The associated listener port has insufficient space to support the refresh request.

8.2.2.2 AgentTalker state machine variables

block
A data structure representing the contents of a RequestRefresh info block.
frame
The received RequestRefresh or RequestLeave control frame (see 6.3 and 6.4).
linkCapacity
A variable representing the operational bandwidth of the link.
(This can be affected by autonegotiation protocols and capabilities of the span partners.)
localTimer
See 8.1.4.
matching
A variable representing the presence of matching SRP state within another talker-agent port.
myMacAddress
See 8.1.2.
oldState
The information associated with a closely matching element of another talker-agent state.
refreshTime
A variable representing the arrival time of the preceding RequestRefresh message.
srpState
See 8.1.2.
tstState
The information associated with a closely matching element of this talker-agent state.
stream
A variable representing a stream identifier.

8.2.2.3 AgentTalker state machine routines

Dequeue(*queue*)
See 8.1.4.
FullSearch(*srpState*, *info*)
Searches through other talker agents searching for an entry with matching *info* parameters.
The search starts at the *srpState*-specified entry and returns each matching entry at most once.
The search ignores the *srpState* entries with a phase of IS_FAILED or IS_PASSIVE.
tstState—Another talker agent has the same *streamID* and matching state.
NONE—Another talker agent has the same *streamID*, but different state.
NULL—No more other-talker agents have the same *streamID*.
InfoSelect(*frame*, *i*)
Returns the *streamID*-specified information block within the RequestRefresh frame.
info—selected frame parameters
NULL—no matching parameters found
LinkBandwidth()
Returns the cumulative link bandwidth associated with the talker agent.
(This excludes bandwidths associated with entries in the IS_FAILED phase.)
ListenerListing(*srpState*)
Publishes the *srpState* information in the associated listener agent registry.
srpState—Completes sucessfully.
NULL—(Otherwise).
SrcRoute(*da*)
Returns the port identifier passed through when routed to the *da*-specified MAC.
positive—matching *portID* value
negative—no matching port found
StateSearch(*streamID*)
See 8.1.3.

StateForm(*streamID*, *bandwidth*)
Allocates and initializes the talker-state information associated with the argument values.
srpState—matching talker-agent state
NULL—no state-space available

8.2.2.4 AgentTalker state table

The AgentTalker state machine is responsible for establishing and demolishing paths, as specified in Table 8.2. 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 8.2—AgentTalker state table

Current state		Row	Next state	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_ARX_REQ)) != NULL	1	—	PARSE
	—	2	—	RETURN
PARSE	frame.subtype == ST_FRESH	3	info = NULL;	LOOP
	frame.subtype == ST_LEAVE	4	tstState = StateSearch((info.talkerID<<16) | info.portID);	LEAVE
	—	5	—	RETURN
LOOP	(info = InfoSelect(frame, info)) != NULL	6	tstState = StateSearch((info.talkerID<<16) | info.portID);	TEST
	—	7	—	RETURN
TEST	tstState == NULL	8	—	FORM
	tstState.phase == IS_FAILED	9	—	LOOP
	tstState.mcastID != block.mcastID	10	—	FORM
	tstState.maxCycles != block.maxCycles	11		
	tstState.maxBw != block.maxBw	12		
	tstState.phase == IS_LEAVING	13	tstState.phase = IS_ACTIVE	POKE
	—	14	—	
POKE	—	15	tstState.refreshTime = localTimer;	LOOP

FORM	(srpState = StateForm()) != NULL	16	srpState.mcastID = info. mcastID;
			srpState.talkerID = info.talkerID;
			srpState.plugID = info.plugID;
			srpState.maxCycle = info.maxCycles;
			srpState.maxBw = info.maxBw;
			oldState = FullSearch(NULL, info);
				CHECK

	—	17	—	LOOP

Table 8.2—AgentTalker state table

Current state		Row	Next state	
state	**condition**		**action**	**state**
CHECK	tstState != NULL	18	srpState.subCode = SC_BAD_HERE;	NACK
	port < 0	19	srpState.subCode = SC_DA_NONE;	
	port == myPortID	20	srpState.subCode = SC_DA_MINE;	
	LinkBandwidth() > 0.75 * linkCapacity	21	srpState.subCode = SC_BW_LIMIT;	
	oldState == DIFF	22	srpState.subCode = SC_BAD_THERE	
	—	23	srpState.refreshTime = localTimer; srpState.streamTime = localTimer;	PEEK
NACK	—	24	srpState.phase = IS_FAILED	LOOP
PEEK	oldState != NULL	25	srpState.phase = IS_ACTIVE;	TOSS
	ListenerListing(srpState) == NULL	26	srpState.subCode = SC_UP_FULL;	NACK
	—	27	srpState.phase = IS_JOINING;	LOOP
TOSS	oldState.phase == IS_LEAVING	28	oldState.phase == IS_PASSIVE;	LAST
	—	29	—	
LAST	(oldState = FullSearch(oldState, info)) != NULL	30	—	TOSS
	—	31	—	LOOP
LEAVE	tstState == NULL	32	—	RETURN
	tstState.phase == IS_FAILED	33		
	FullSearch(NULL, info) == NULL	34	tstState.phase = IS_LEAVING;	
	—	35	Release(tstState);	

Row 8.2-1: Dequeue a received subscription-request message, if available.
Row 8.2-2: Otherwise, wait for the next subscription-request message.

Row 8.2-3: Process received RequestRefresh messages.
Row 8.2-4: Process received RequestLeave messages.
Row 8.2-5: Discard unrecognized refresh messages.

Row 8.2-6: Find state associated with the selected blocks within the RequestRefresh messages.
Row 8.2-7: Stop processing after the last RequestRefresh block has been processed.

Row 8.2-8: If a matching entry cannot be found, a new one must be formed.
Row 8.2-9: The refresh is ignored while the matching entry is dedicated to error reporting.
Row 8.2-10: If the matching entry has a distinct multicast identifier, the refresh is erroneous.
Row 8.2-11: If the matching entry has a distinct *maxCycles* count, the refresh is erroneous.
Row 8.2-12: If the matching entry has a distinct maximum bandwidth, the refresh is erroneous
Row 8.2-13: If the state was leaving, it changes to active.
Row 8.2-14: Otherwise, the state (joining or active) remains unchanged.

Row 8.2-15: Update the refresh timeout when a matching entry is observed.

Row 8.2-16: If storage is available, update the new state based on the supplied *info* field parameters.
Row 8.2-17: If no storage is available, nothing can be done and the *info* state is discarded.
(A timeout is necessary to detect this discard, since no storage state is available for error reporting purposes.)

Row 8.2-18: With a matching/inconsistent same-port state, the appropriate error-status code is returned.
Row 8.2-19: If no upstream port can be found, the appropriate error-status code is returned.
Row 8.2-20: If the upstream port is one's self, the appropriate error-status code is returned.
Row 8.2-21: If the cumulative bandwidth limit is exceeded, the appropriate error-status code is returned.
Row 8.2-22: With a matching/inconsistent other-port state, the appropriate error-status code is returned.
Row 8.2-23: Otherwise, the timeouts are reset before the refresh is accepted.

Row 8.2-24: The SRP state is marked to communicate the failure condition.

Row 8.2-25: If matching state is found on another talker agent, this port's state is set to active.
Row 8.2-26: Otherwise, this port's state is set to joining.
(This triggers the near-immediate transmission of a limited refresh message, to first establish the stream.)

Row 8.2-28: If an existing entry is marked as leaving, its state is changed to passive to ensure removal.
(This talker agent is joining, so the connection remains and there is no need to announce another's leaving.)
Row 8.2-29: Otherwise, the existing entry is ignored.

Row 8.2-30: Check to confirm the presence an another existing entry.
Row 8.2-31: Or, terminate the search in the absence of another existing entry.

Row 8.2-32: If no matching to the leaving request is found, the leave request is ignored.
Row 8.2-33: If a matching error response is found, the leave request is ignored.
Row 8.2-34: If no other port has an active request, the leave request is accepted.
Row 8.2-35: If another port has an active request, this leave request can be safely ignored.

8.2.3 AgentTimer state machine

The AgentTimer state machine monitors received RequestRefresh and RequestLeave frames. There are multiple AgentTimer state machines per bridge, one for each of the bridge ports.

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

8.2.3.1 AgentTimer state machine definitions

IS_ACTIVE
IS_FAILED
  See 8.1.2.
NULL
  Indicates the absence of a value and (by design) cannot be confused with a valid value.
Q_ATX_RES
Q_ARX_STR
Q_ATX_STR
  See 8.1.4.
ST_ERROR
  See 8.1.1.
A *subtype* specifier that distinguishes the ResponseError frame from other RE frames.

8.2.3.2 AgentTimer state machine variables

frame
The received streaming classA frame or generated SRP ResponseError frame (see 6.1 and 6.5).

info
A data structure representing the contents of a RequestRefresh/RequestLeave info block.

localTimer
See 8.1.4.

myMacAddress
See 8.1.2.

refreshTime
A variable representing the arrival time of the preceding RequestRefresh message.

refreshTimeout
A variable representing a timeout interval for RequestRefresh messages.

srpState
See 8.1.2.

stream
A variable representing a stream identifier.

8.2.3.3 AgentTimer state machine routines

CastSearch(*mcastID*)
Returns the talker-state information associated with the specified multicast identifier.
srpState—matching talker-agent state
NULL—no matching state found

Dequeue(*queue*)
Enqueue(*queue, frame*)
See 8.1.4.

QueueHasSpace(*index*)
Indicates whether space is available for frame transmissions.
TRUE—Space is available.
FALSE—(Otherwise.)

StateSearch(*streamID*)
See 8.1.3.

StateSelect(*index*)
Returns the talker-agent state associated with the specified *index*.
info—matching talker-agent state
NULL—no state-space available

StateToss(*index*)
Discards talker-state information associated with the argument value.

8.2.3.4 AgentTimer state table

The AgentTimer state machine is responsible for reporting timeout and upstream-communicated errors, as specified in Table 8.3. 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 8.3—AgentTimer state table

Current state		Row	Next state	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_ARX_STR)) != NULL	1	srpState = CastSearch(frame.da);	FLOW
	(frame = Dequeue(Q_ARX_RES)) != NULL	2	info = frame.info; tstState = StateSearch((info.talkerID<<16) | info.portID);	SERVE
	—	3	srpState = NULL	LOOP
FLOW	srpState == NULL	4	—	START
	—	5	Enqueue(Q_ATX_STR, frame); srpState.streamTime = localTimer;	
SERVE	tstState != NULL	6	tstState.phase = IS_FAILED; tstState.subCode = frame.subCode;	START
	—	7	—	
LOOP	(srpState = StateSelect(srpState)) != NULL	8	—	TIMES
	—	9	—	RETURN
TIMES	srpState.phase == IS_FAILED	10	—	NEAR
	srpState.phase == IS_JOINING	11	—	LOOP
	srpState.phase == IS_LEAVING	12		
	srpState.phase == IS_PASSIVE	13	StateToss(srpState);	
	(localTimer – srpState.refreshTime) >= refreshTimeout	14		
	(localTimer – srpState.streamTime) >= dataTimeout	15		
	—	16	—	

NEAR	QueueHasSpace(Q_ATX_RES)	17	frame.da = srpState.srcMac;
			frame.sa = myMacAddress;
			frame.subType = ST_ERROR;
			frame.subCode = srpState.subCode;
			frame.streamId = srpState.streamID;
			frame.maxBw = srpState.maxBw;
			frame.cycles = srpState.maxCycles;
			Enqueue(Q_ATX_RES, frame);
			StateToss(srpState);
				LOOP

	—	18	—	

Row 8.3-1: Monitor the received stream flow, as frames pass through.
Row 8.3-2: Process received error messages, when they become available.
Row 8.3-3: Otherwise, aging timeouts are invoked.

Row 8.3-4: Stream flows are not forwarded in the absence of matching state.
Row 8.3-5: Otherwise, stream flows are monitored and flow downstream.

Row 8.3-6: In the presence of matching talker-agent state, the stream passes through.
Row 8.3-7: In the absence of matching talker-agent state, the stream passes through.

Row 8.3-8: Select each talker-state element associated with the port.
Row 8.3-9: Stop when all talker-state elements have been processed.

Row 8.3-10: A failed entry is processed distinctively.
Row 8.3-11: The joining phase indications has no timeout.
Row 8.3-12: The leaving phase indications has no timeout.
Row 8.3-13: The passive phase indication has been effectively discarded, so discard it immediately.
Row 8.3-14: In the absence of sustained refresh messages, the active SRP state is discarded.
Row 8.3-15: In the absence of sustained stream flows, the active SRP state is discarded.
Row 8.3-16: Otherwise, no timeout actions are required.

Row 8.3-17: In the presence of a failed phase indication, a ResponseError is sent downstream.
Row 8.3-18: Otherwise, no action is taken.

8.2.4 AgentListener state machine

The AgentListener state machine generates RequestRefresh and RequestLeave control frames. There are multiple AgentListener state machines on each bridge, one is associated with each of the bridge ports.

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

8.2.4.1 AgentListener state machine definitions

Q_ATX_REQ
  See 8.1.4.
IS_PASSIVE
  See 8.1.2.
NULL
  Indicates the absence of a value and (by design) cannot be confused with a valid value.

8.2.4.2 AgentListener state machine variables

frame
  An SRP control frame.
localTimer
  See 8.1.4.
myMacAddress
  See 8.1.2.
refreshTime
  A variable representing the transmission time of the preceding RequestRefresh message.
refreshTimeout
  A variable representing a timeout interval for RequestRefresh messages.
refreshList
  A list of *srpState* entries prepared for upstream transmission.
srpState
  See 8.1.2.

8.2.4.3 AgentListener state machine routines

Enqueue(*queue*, *frame*)
  See 8.1.4.
EnqueueList(*queue*, *list*)
  Transfers content from the rpState lists into one or more frames.
  Each of these frames is then placed into the specified queue.
JoiningList()
  Forms a list of the joining-phase entries from the listener agent's state array.
JoiningToActive(*list*)
  Within all listed entries, each phase value of IS_JOINING is changed to IS_ACTIVE.
QueueHasSpace(*index*)
  Indicates whether space is available for frame transmissions.
    TRUE—Space is available.
    FALSE—(Otherwise.)
RefreshList()
  Forms a list of the joining-phase and active-phase entries from the listener agent's state array.
ReviseListenerList()
  Revises the listener list entries to ensure consistency with distributed AgentTalker state content.

8.2.4.4 AgentListener state table

The AgentListener state machine is responsible for generating upstream RequestRefresh and RequestLeave frames, as specified in Table 8.4. 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 8.4—AgentListener state table

Current state		Row	Next state	
state	**condition**		**action**	**state**
START	—	1	ReviseListenerList();	FIRST
FIRST	QueueHasSpace(Q_ARX_REQ)	2	—	TIMER
	—	3	—	RETURN
CHECK	localTimer >= (refreshTime + refreshTimeout) && ((refreshList= RefreshList()) != NULL)	4	refreshTime = localTimer;	FRESH

	srpState = QueueHasLeave()	5	frame.da = upstreamAddress;
			frame.sa = myMacAddress;
			frame.info = srpState.info;
			EnqueueFrame(Q_ATX_REQ, frame);
			srpState.phase = IS_PASSIVE;
				START

	(refreshList = JoiningList()) != NULL	6	—	FRESH
	—	7	—	RETURN
FRESH	—	8	EnqueueList(Q_ATX_REQ, refreshList); JoinToActive(refreshList);	START

Row 8.4-1: Refresh the listener list, ensuring consistency with distributed AgentTalker state content.
Row 8.4-2: In the presence of transmission-queue storage, transmissions are enabled.
Row 8.4-3: Otherwise, transmissions are inhibited.

Row 8.4-4: When periodically enabled, the list of joining and active states is sent.
Row 8.4-5: Leave requests are checked; distinct ones cause a RequestListen frame to be sent.
Row 8.4-6: When entries are found, the list of joining states is sent.
Row 8.4-7: Otherwise, no talker-agent refresh/leave messages are transmitted.

Row 8.4-8: Enqueue the refresh-list entries for eventual transmission.
Afterwards, change the phase from joining to active, to inhibit unnecessary future transmissions.

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

[B3] IEEE Std 802.1D-2004, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges.

[B4] IEEE Std 802-2002, IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture.

[B5] IEEE Std 1394-1995, High performance serial bus.

[B6] IEEE Std 1588-2002, IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.

[B7] IETF RFC 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis, David L. Mills, March 1992[2]

[B8] IETF RFC 2030: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, D. Mills, October 1996.

[B9] 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.

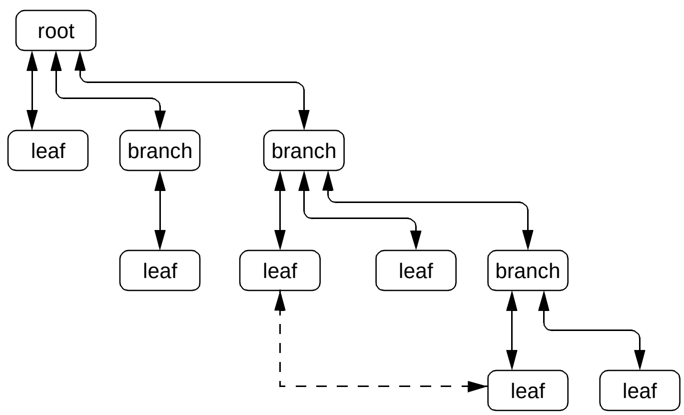


Figure B.1—SerialBus topologies

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.

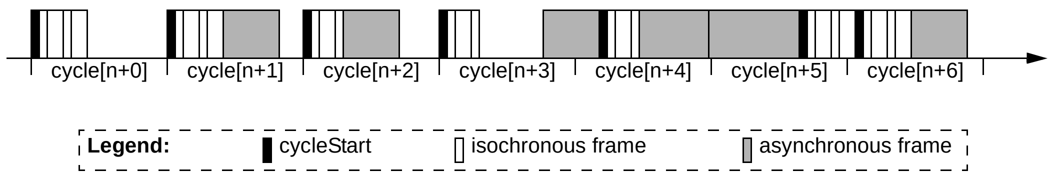


Figure B.2—Isochronous data transfer timing

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.

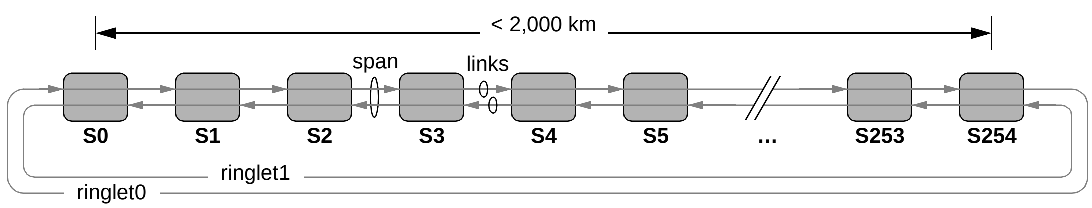


Figure B.3—RPR rings

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).

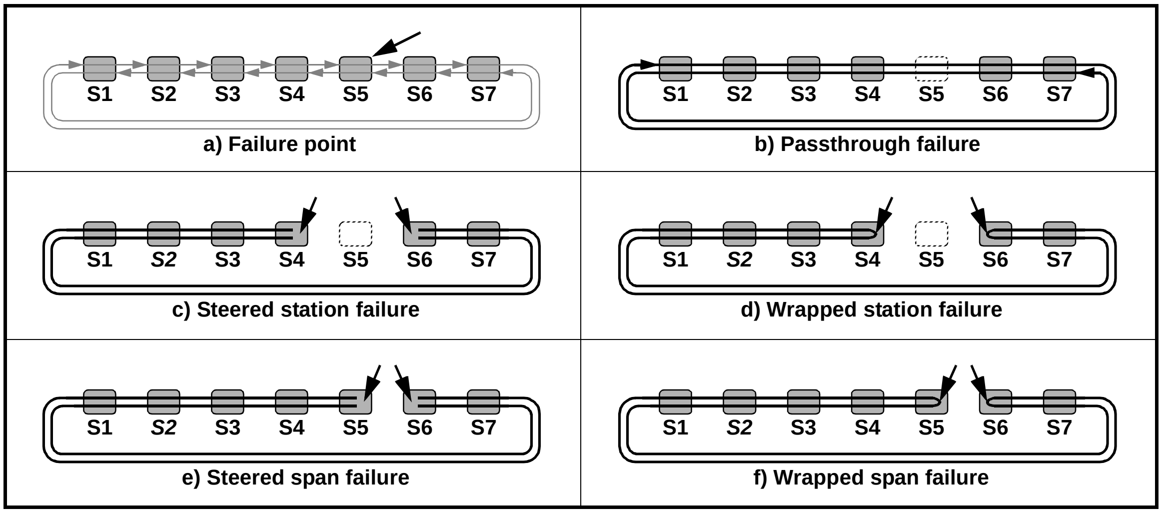


Figure B.4—RPR resilience

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.

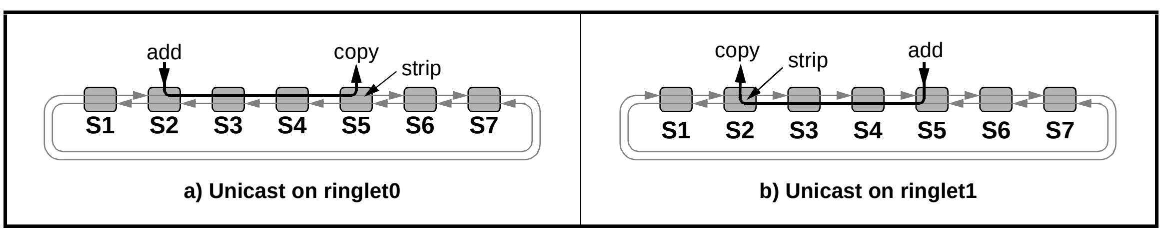


Figure B.5—RPR destination stripping

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.

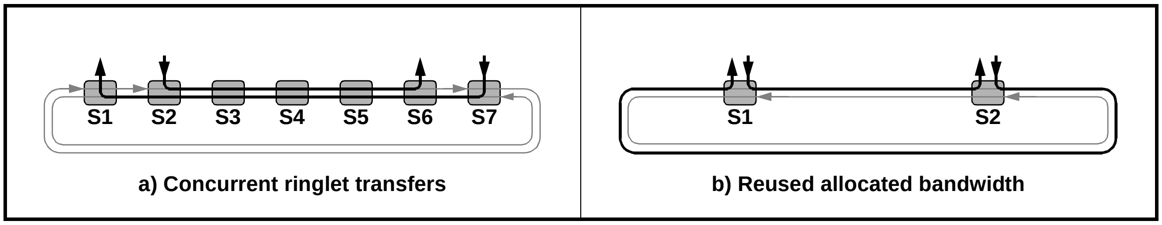


Figure B.6—RPR spatial reuse

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.

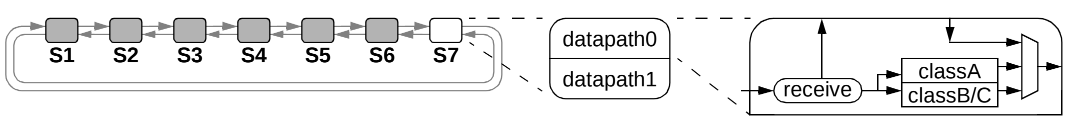


Figure B.7—RPR service classes

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.

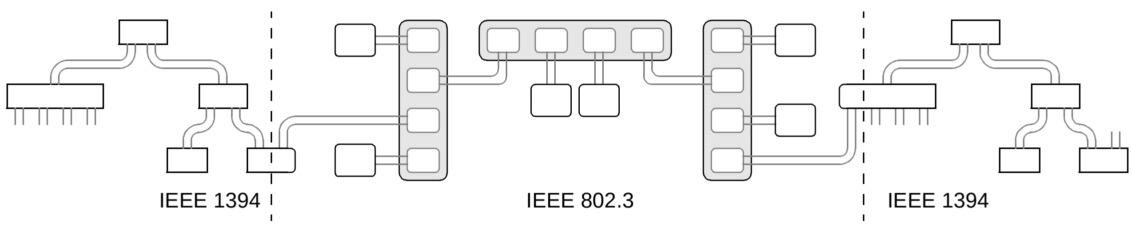


Figure C.1—IEEE 1394 leaf domains

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.

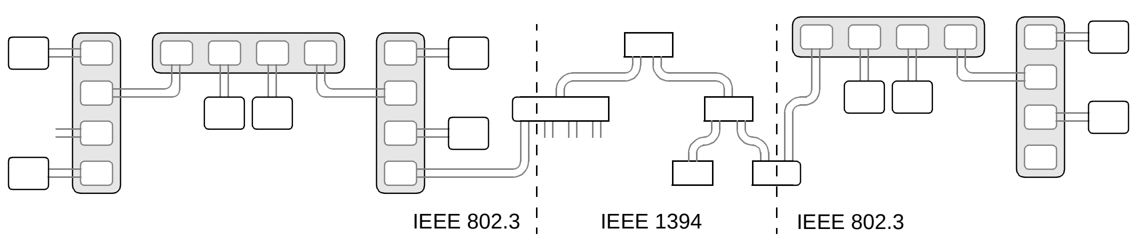


Figure C.2—IEEE 802.3 leaf domains

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.

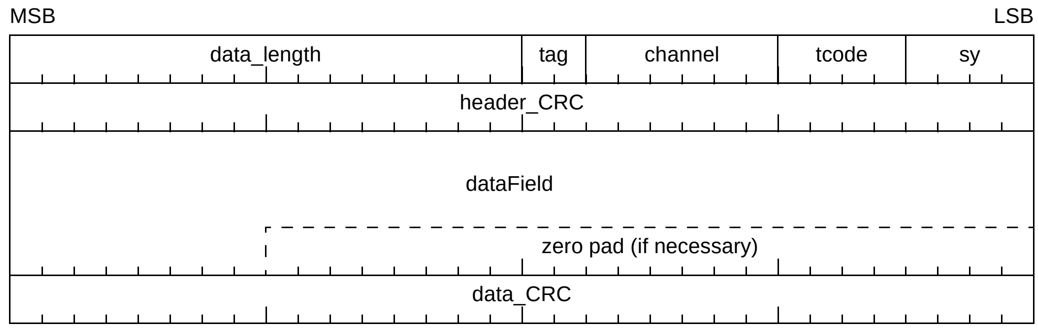


Figure C.3—IEEE 1394 isochronous packet format

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.

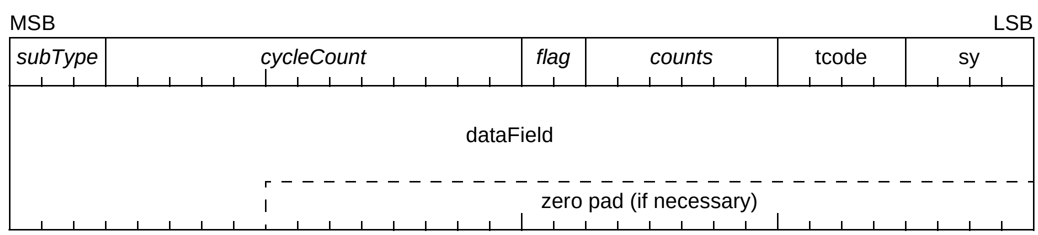


Figure C.4—Encapsulated IEEE 1394 frame payload

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

<table>
<tr><th>flag</th><th>Name</th><th>Description</th></tr>
<tr><td>ONLY</td><td rowspan="2">partCount</td><td rowspan="2">The LSBs of the residual data_length field.</td></tr>
<tr><td>LAST</td></tr>
<tr><td>CORE</td><td rowspan="2">frameCount</td><td rowspan="2">A sequence identifier for frames within the group</td></tr>
<tr><td>LEAD</td></tr>
</table>

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.

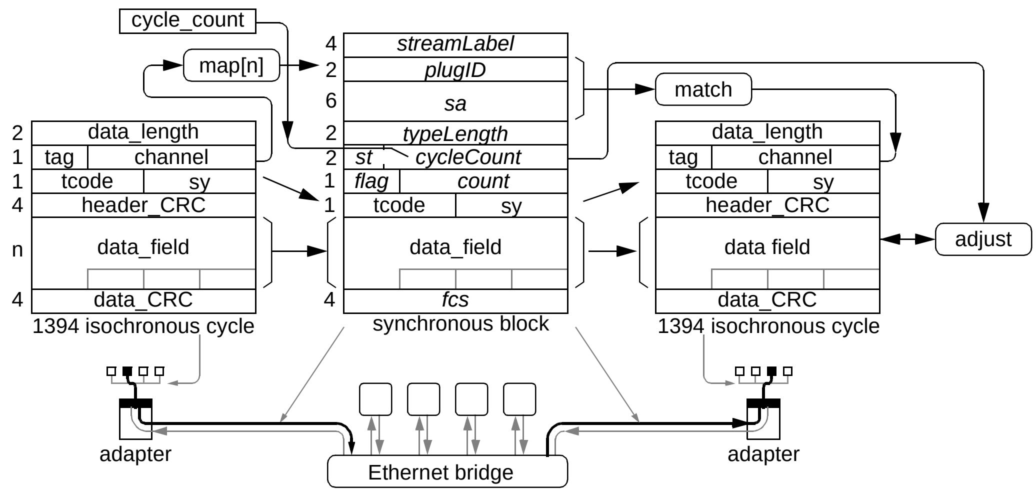


Figure C.5—Conversions between IEEE 1394 packets and RE frames

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.

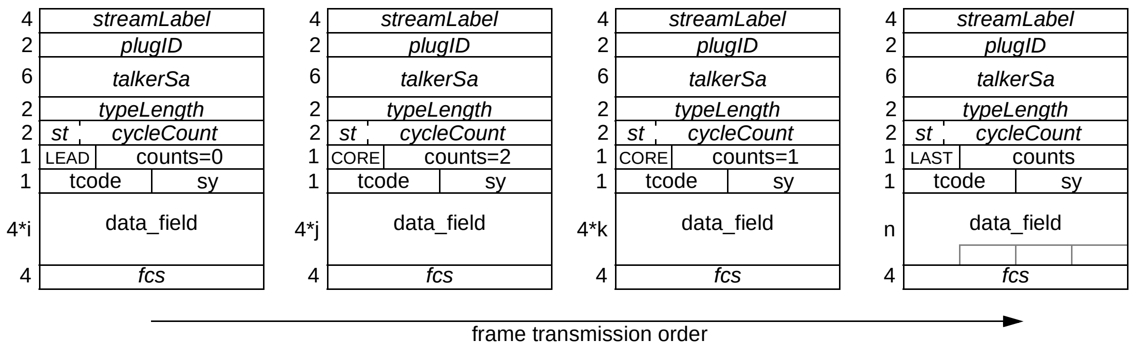


Figure C.6—Multiframe groups

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.

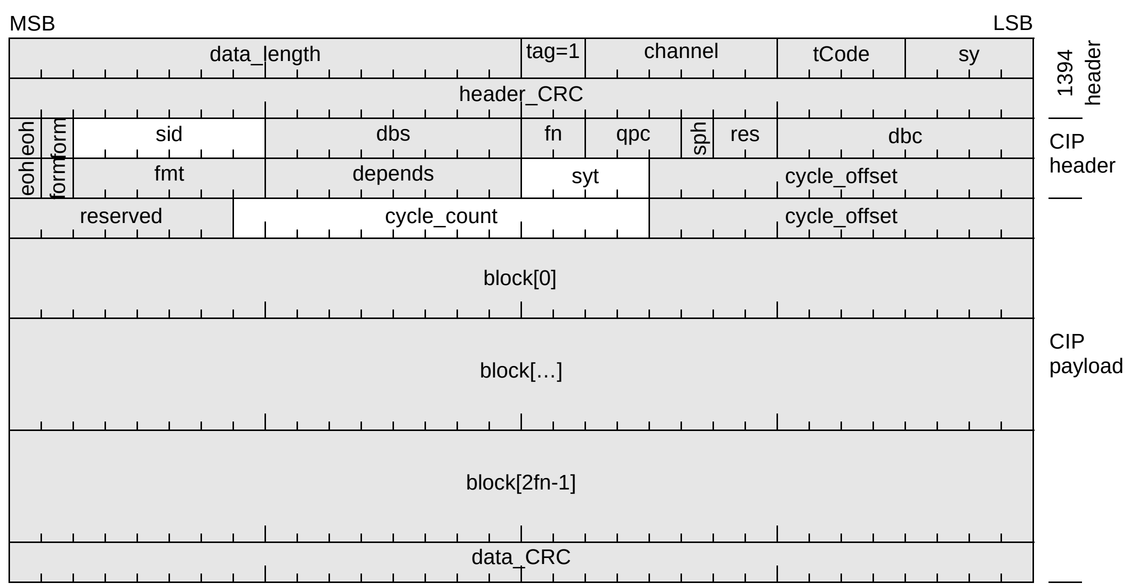


Figure C.7—Isochronous 1394 CIP packet format

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

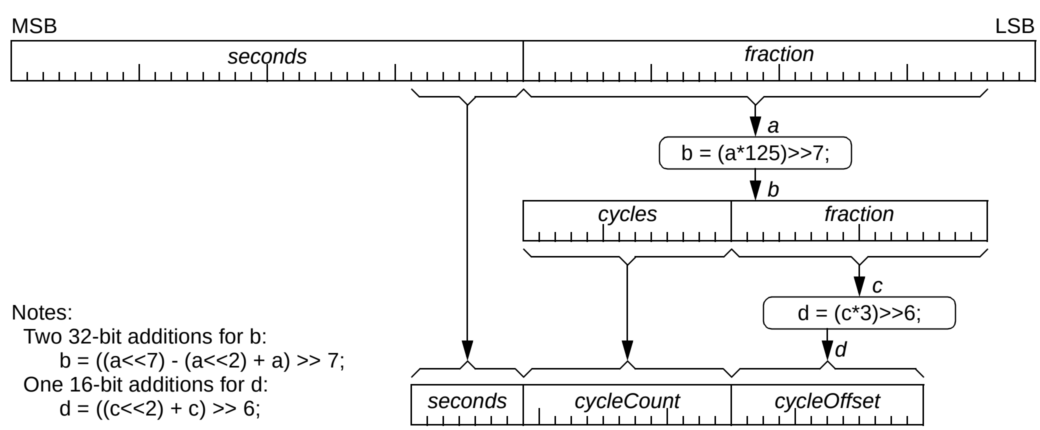


Figure C.8—Time-of-day format conversions

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:

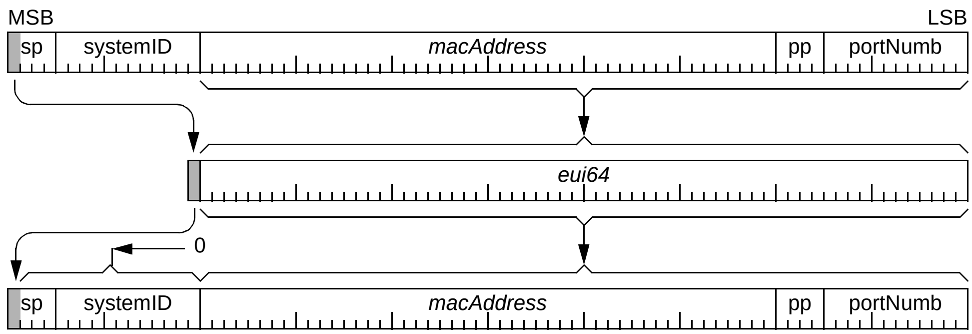


Figure C.9—Grand-master precedence mapping

Annex D

(informative)

Review of possible alternatives

D.1 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 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.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.4 IEEE 1394 alternatives

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.

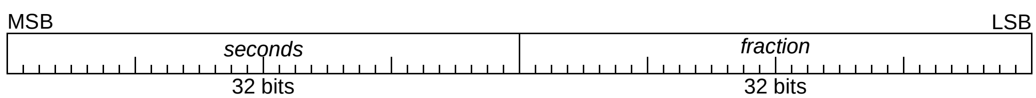


Figure 5.1—Complete seconds timer format

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.

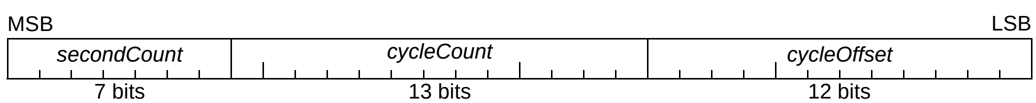


Figure E.2—IEEE 1394 timer format

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.

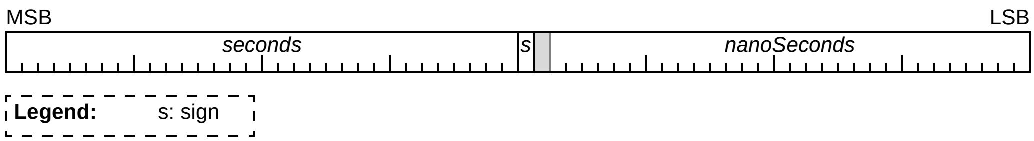
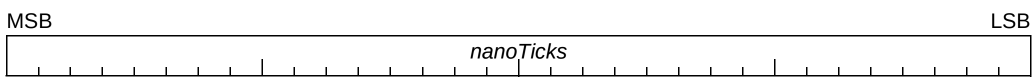


Figure E.3—IEEE 1588 timer format

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.



seconds = *nanoTicks*/62500000

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.

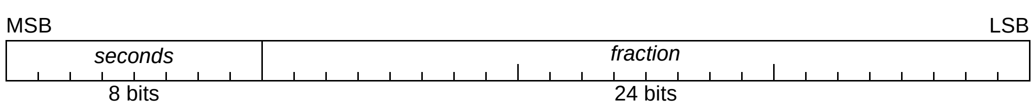


Figure E.5—Compact seconds timer format

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.

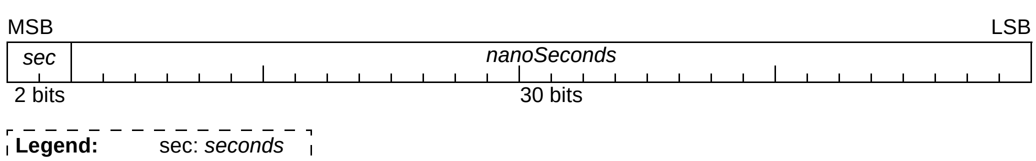


Figure E.6—Nanosecond timer format

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 9.2.4) 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.

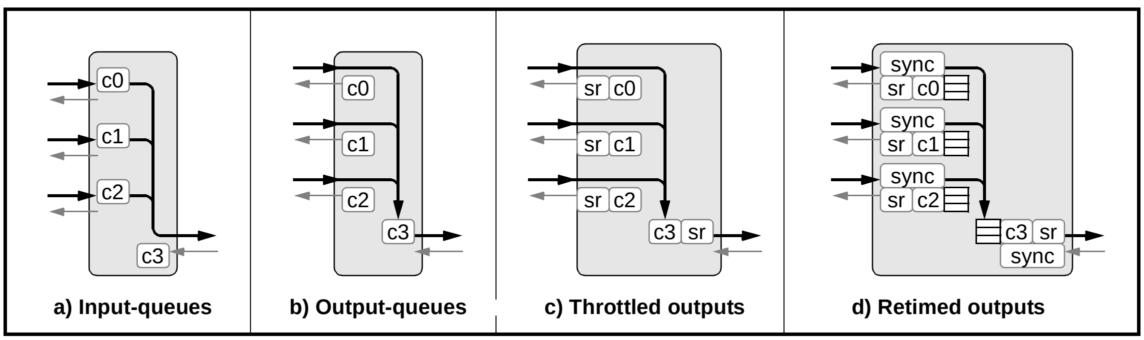


Figure F.1—Bridge design models

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.

TBD—
Should we assume that frames are forwarded using cut-through or store-and-forward? Store-and-forward delays are variable and approximately equal to the frame length (about 120µs, on a 100 Mb/s link). Thus, the difference would be 2-cycle vs. 3-cycle delays.

F.1.2 Three-source hierarchical topology

A hierarchical topology best illustrate potential problems with bunching, as illustrated in Figure F.2. Traffic from sources {a0,a1,a2} is transmitted by talker stations {b0,b1,b2}. Bridge C concentrates traffic received from three talkers, with the cumulative c3 traffic sent to d3. Identical traffic flows are assumed at bridge ports {d0,d1,d3}, although only one of these sources is illustrated. Bridges {C,D,E,F,G,H,I} behave similarly.

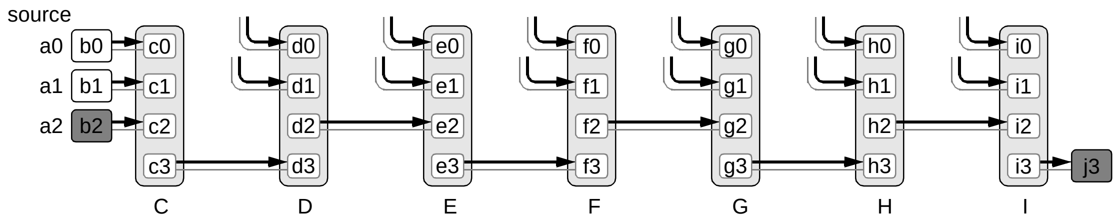


Figure F.2—Three-source topology

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 sources {a0,a1,a2,a3,a4,a5} is transmitted by talker stations {b0,b1,b2,b3,b4,b5}. Bridge C concentrates traffic received from three talkers, with the cumulative c6 traffic sent to d6. Identical traffic flows are assumed at bridge ports {d0,d1,d3,d3,d4,d6}, although only one of these sources is illustrated. Bridges {C,D,E,F,G,H,I} behave similarly.

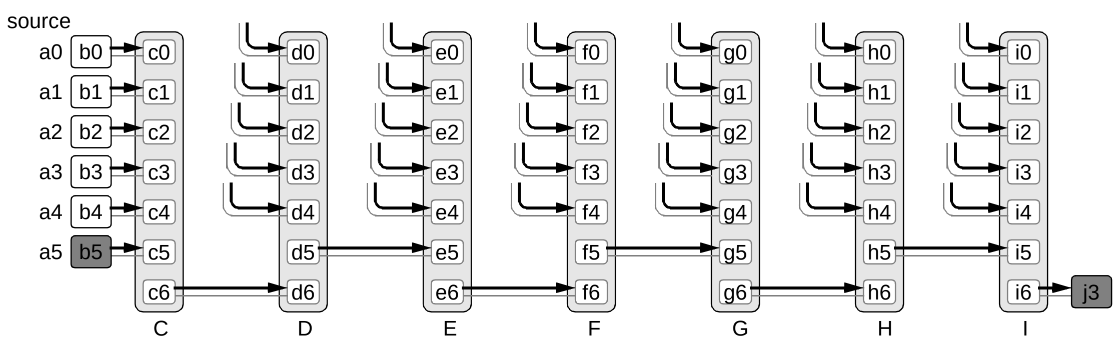


Figure F.3—Six-source topology

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.

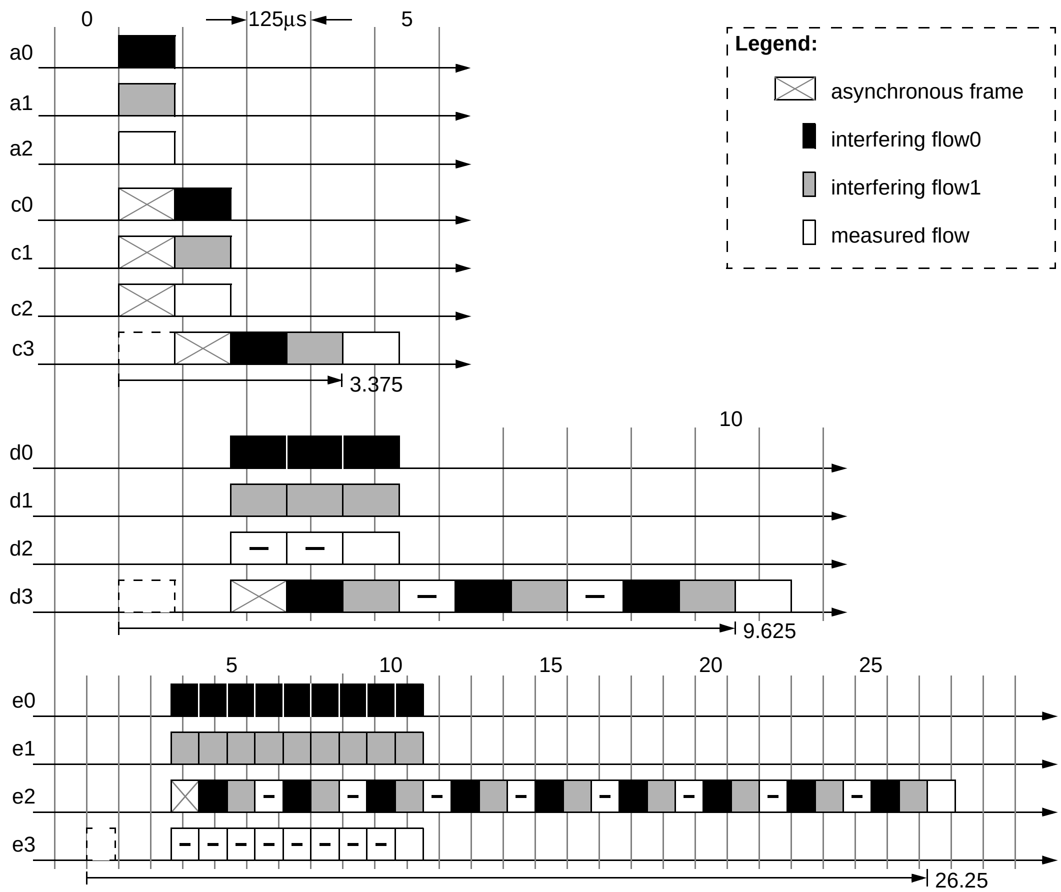


Figure F.4—Three-source bunching timing; input-queue bridges

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.

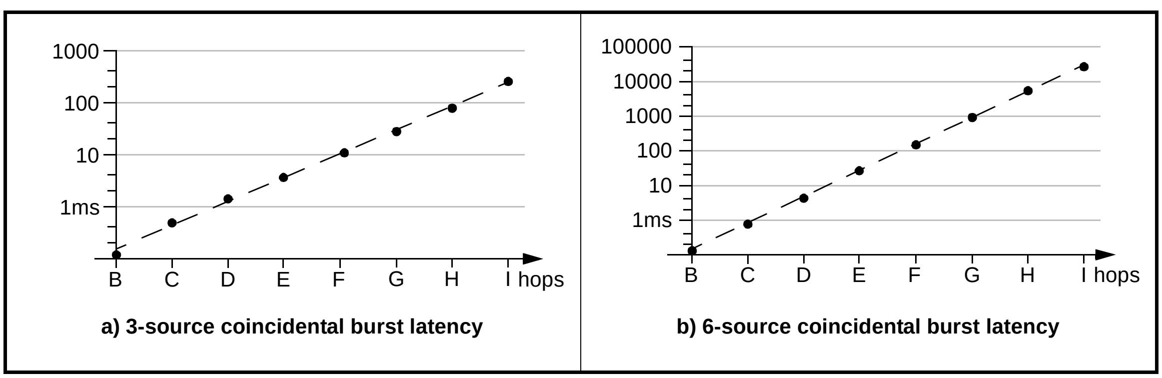
Table F.1—Cumulative bursting latencies

Topology	Units	Measurement point							
		B	**C**	**D**	**E**	**F**	**G**	**H**	**I**
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.

$$delay[n] = mtu \times (n + p^n) \tag{F.1}$$

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.



a) 3-source coincidental burst latency

b) 6-source coincidental burst latency

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 (120µs), 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 {c0,c1,c2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through c3. 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,c2},…,{f0,f1,f3}, only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

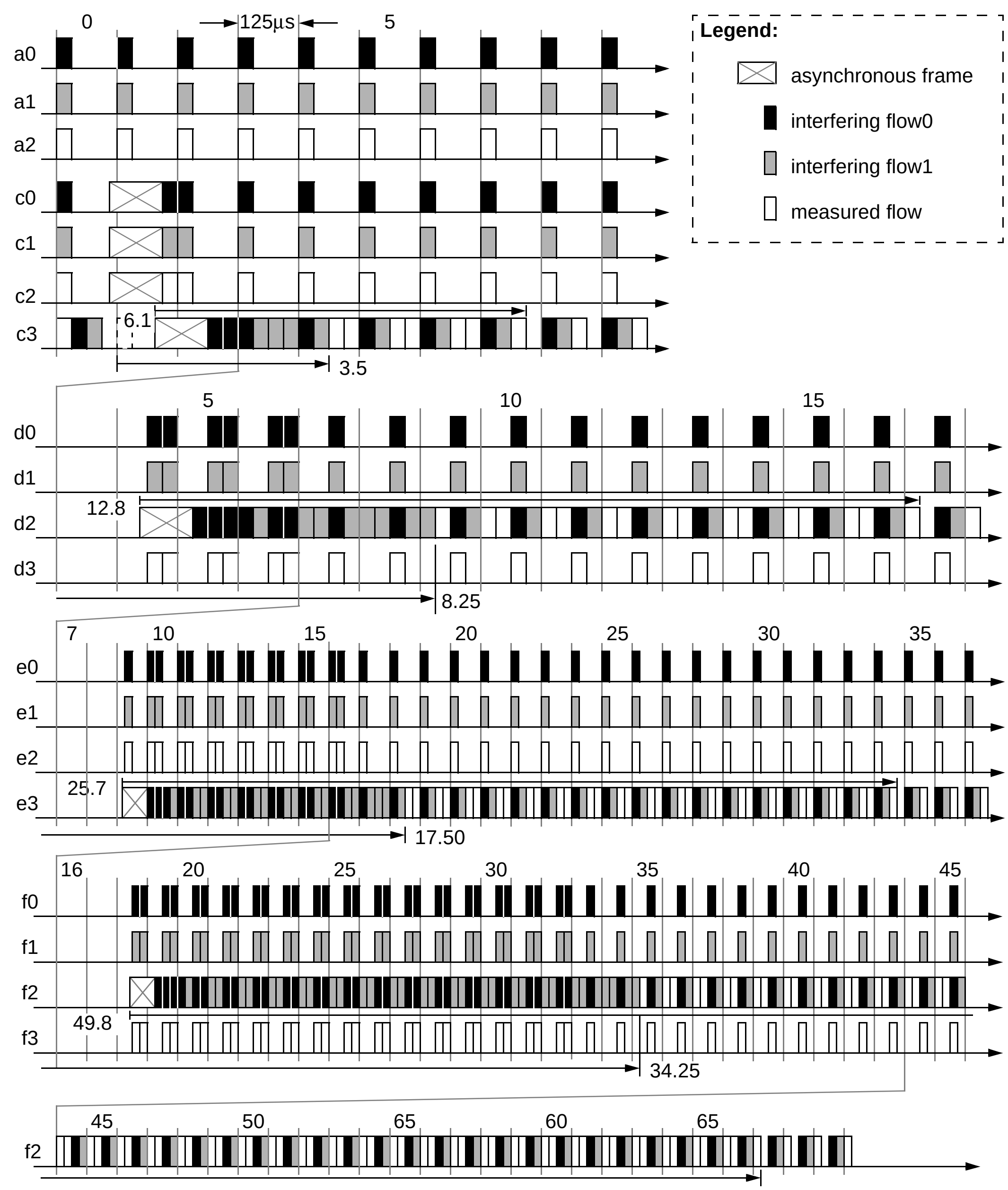


Figure F.6—Three-source bunching; input-queue bridges

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 {c0,c1,c2,c3,c4,c5} concentrates traffic from three talkers; one sixth of the cumulative traffic is forwarded through c6. 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}, …, {d0,d1,d2,d3,d4,d6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

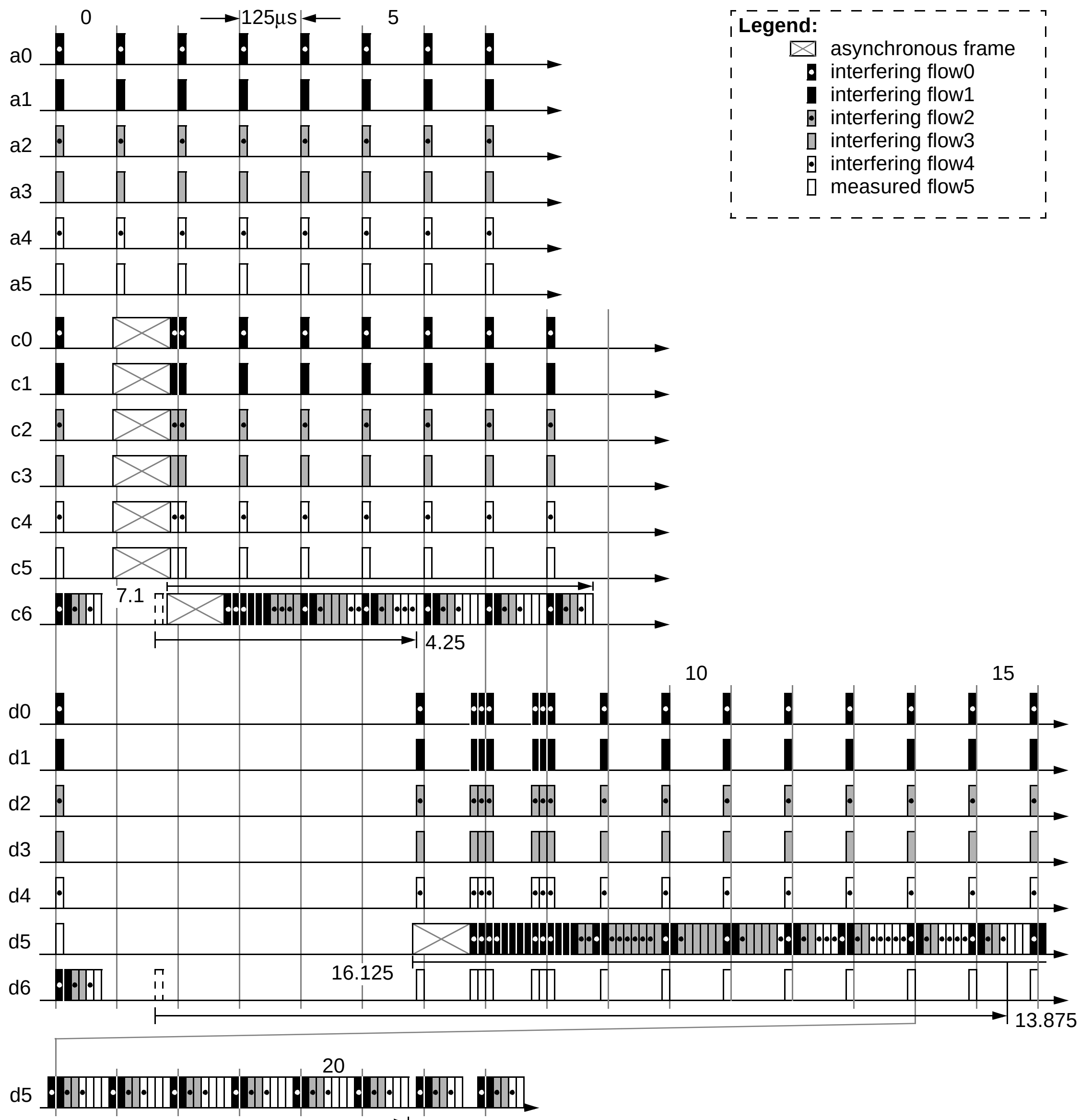


Figure F.7—Six source bunching timing; input-queue bridges

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							
		B	C	D	E	F	G	H	I
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	4.25	13.87	(39.33)	(107.2)	(288.2)	(771)	2058
	ms	0.01	0.56	1.73	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.

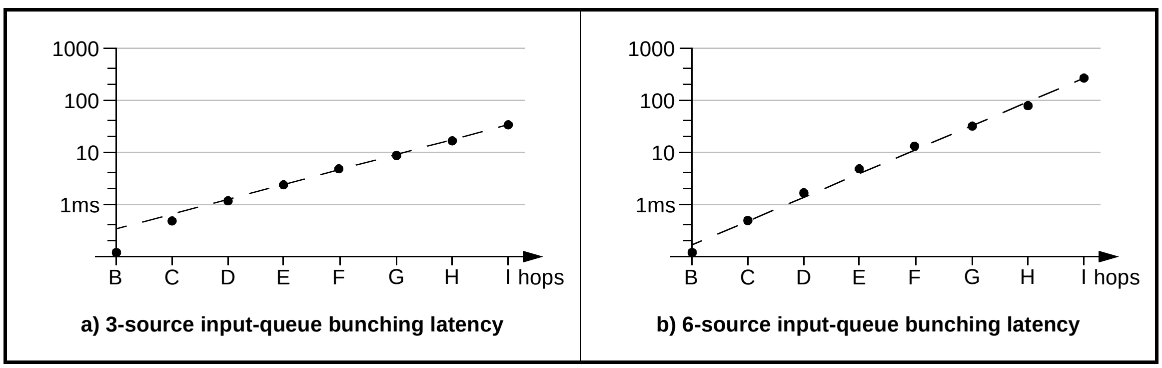


Figure F.8—Cumulative bunching latencies; input-queue bridge

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 {c0,c1,c2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through c3. 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,c2},…,{f0,f1,f3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

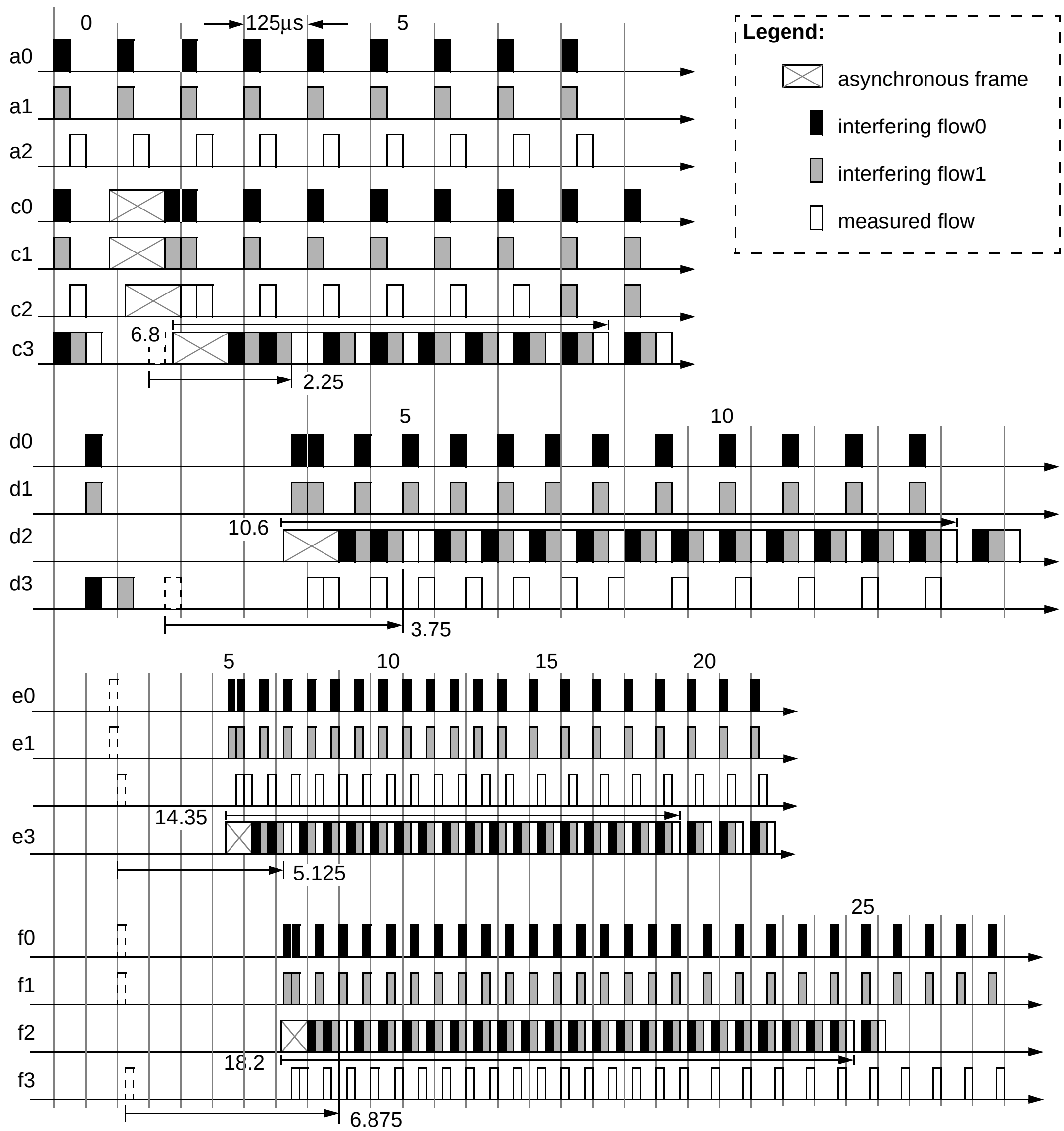


Figure F.9—Three-source bunching; output-queue bridges

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 {c0,c1,c2,c3,c4,c5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port c6. 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},…,{e0,e1,e2,e3,e4,e5} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

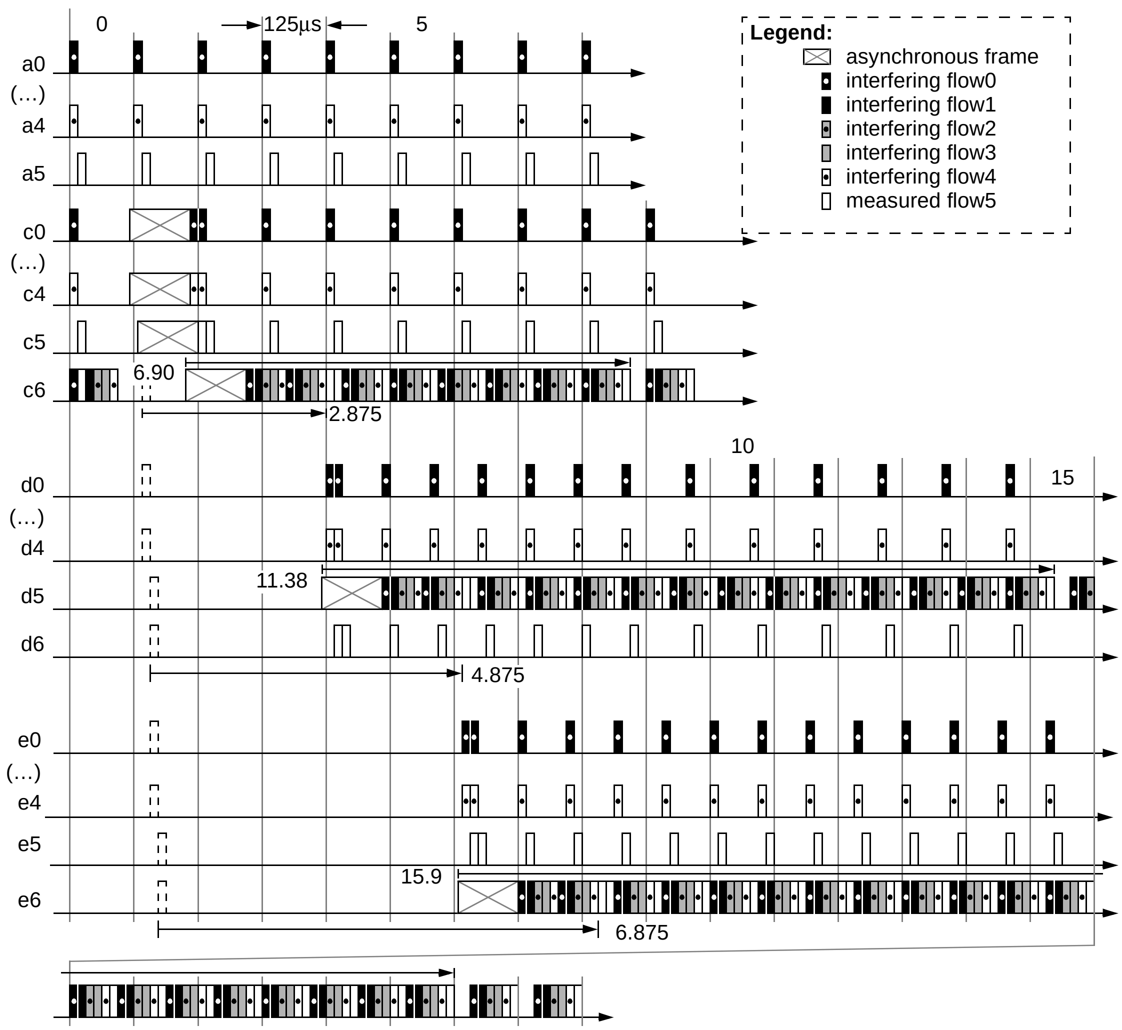


Figure F.10—Six source bunching; output-queue bridges

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	2.25	3.75	5.125	6.875	–	–	–
	ms	0.10	0.27	0.45	0.62	0.83	–	–	–
6-source (see F.2.2.2)	cycles	.875	2.875	4.875	6.875	–	–	–	–
	ms	0.10	0.35	0.59	0.83	–	–	–	–

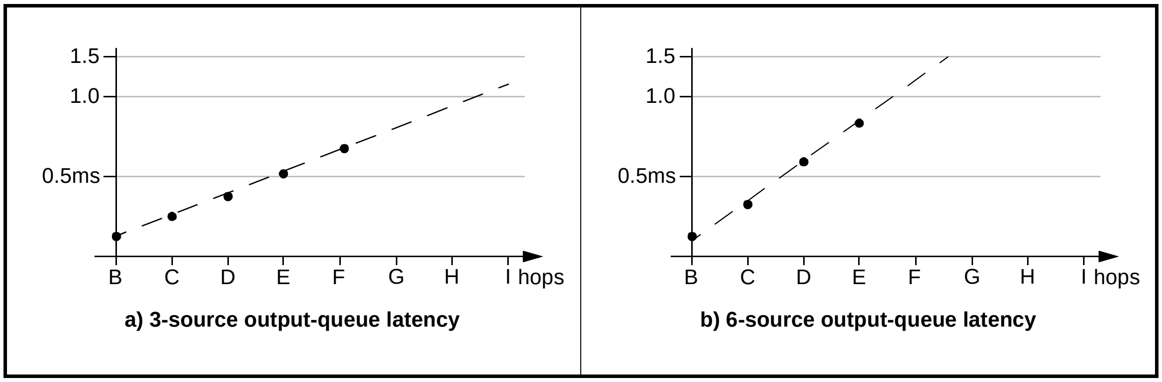


Figure F.11—Cumulative bunching latencies; output-queue bridge

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 {c0,c1,c2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port c3. 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,c2},…,{f0,f1,f3} only illustrate the passing-through listener traffic; the remainder of the traffic is assumed to be routed elsewhere.

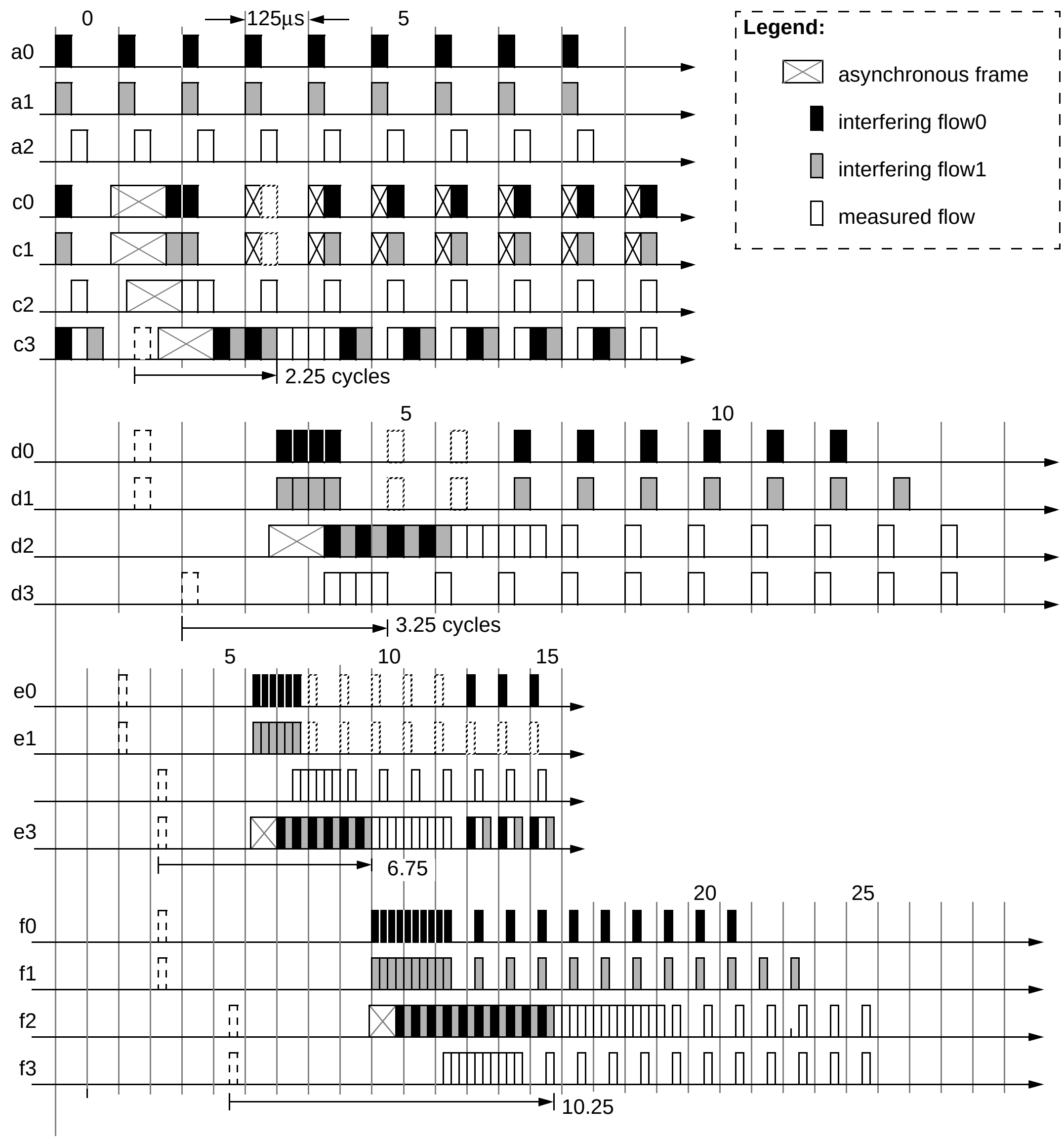


Figure F.12—Three-source bunching; variable-rate output-queue bridges

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 {c0,c1,c2,c3,c4,c5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port c6. 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,d6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

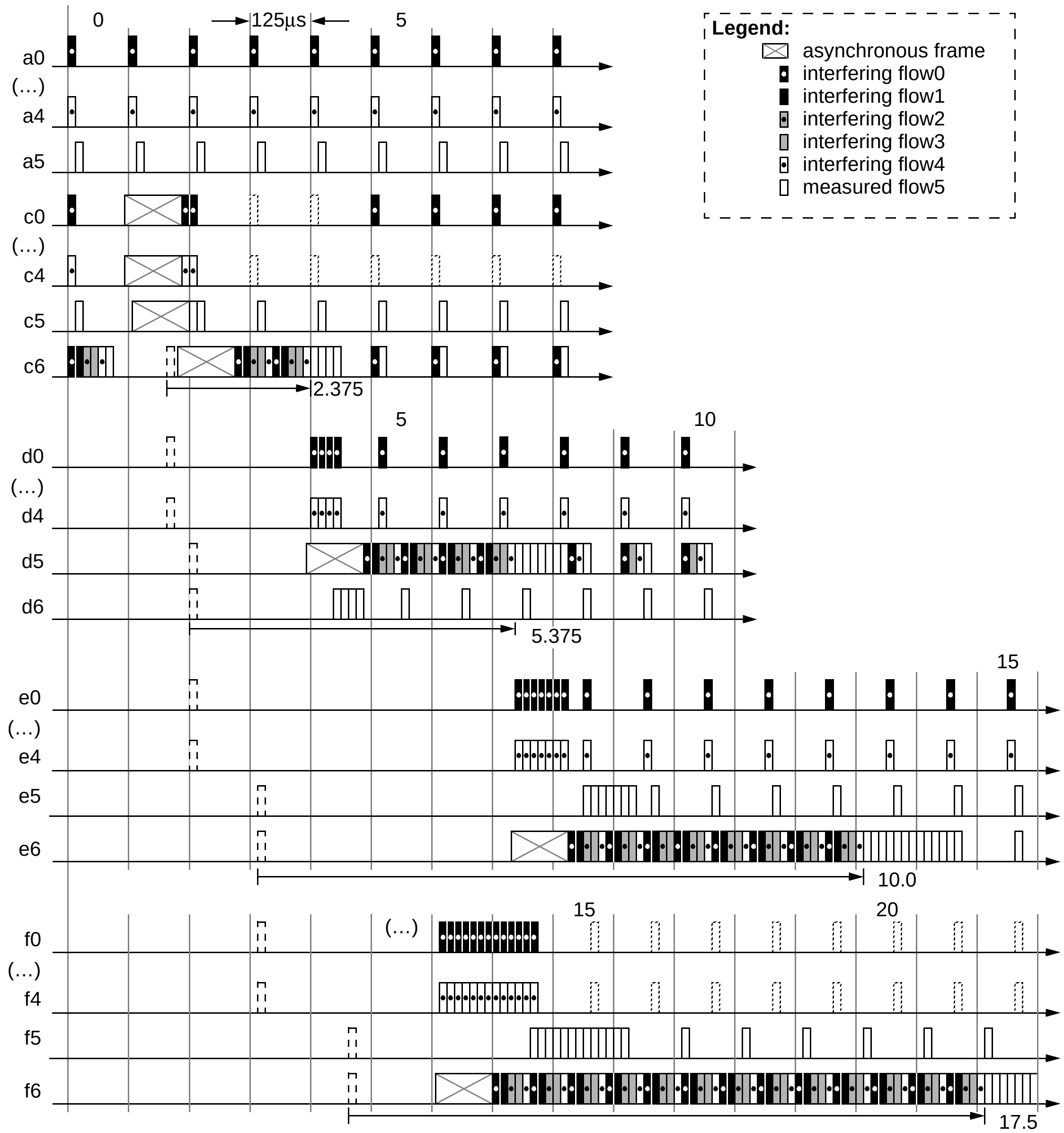


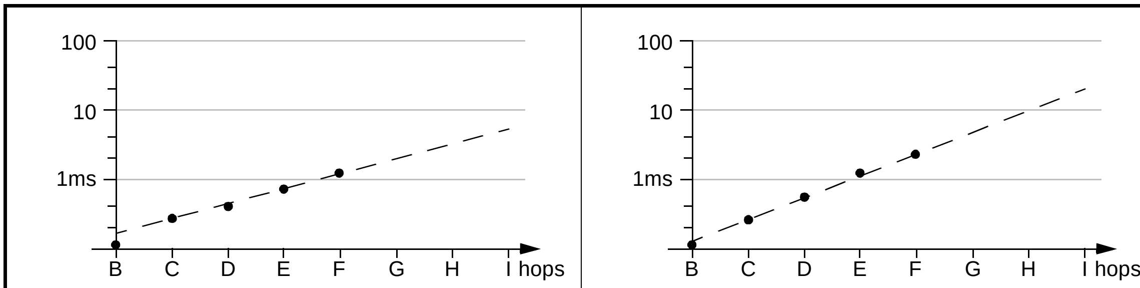
Figure F.13—Six source bunching; variable-rate output-queue bridges

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							
		B	**C**	**D**	**E**	**F**	**G**	**H**	**I**
3-source (see F.2.2.1)	cycles	0.75	2.25	3.35	6.75	10.25	–	–	–
	ms	0.10	0.27	0.40	0.81	1.23	–	–	–
6-source (see F.2.2.2)	cycles	0.75	2.375	5.375	10.0	17.5	–	–	–
	ms	0.10	0.28	0.65	1.20	2.1	–	–	–



a) 3-source coincidental burst latency

b) 6-source coincidental burst latency

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 {c0,c1,c2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port c3. 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.

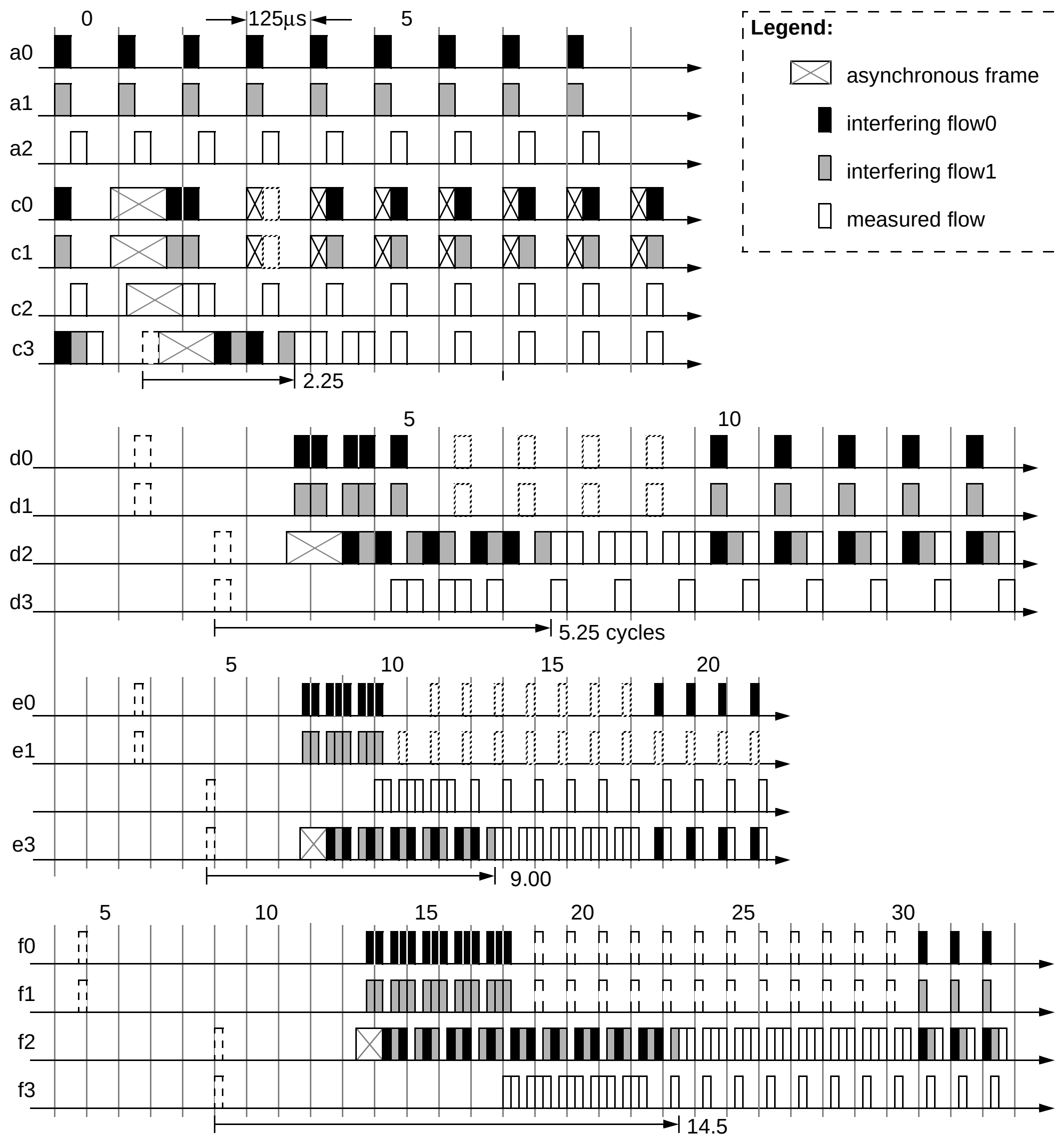


Figure F.15—Three-source bunching; throttled-rate output-queue bridges

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 {c0,c1,c2,c3,c4,c5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port c6. 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},…,{f0,f1,f2,f3,f4,f6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

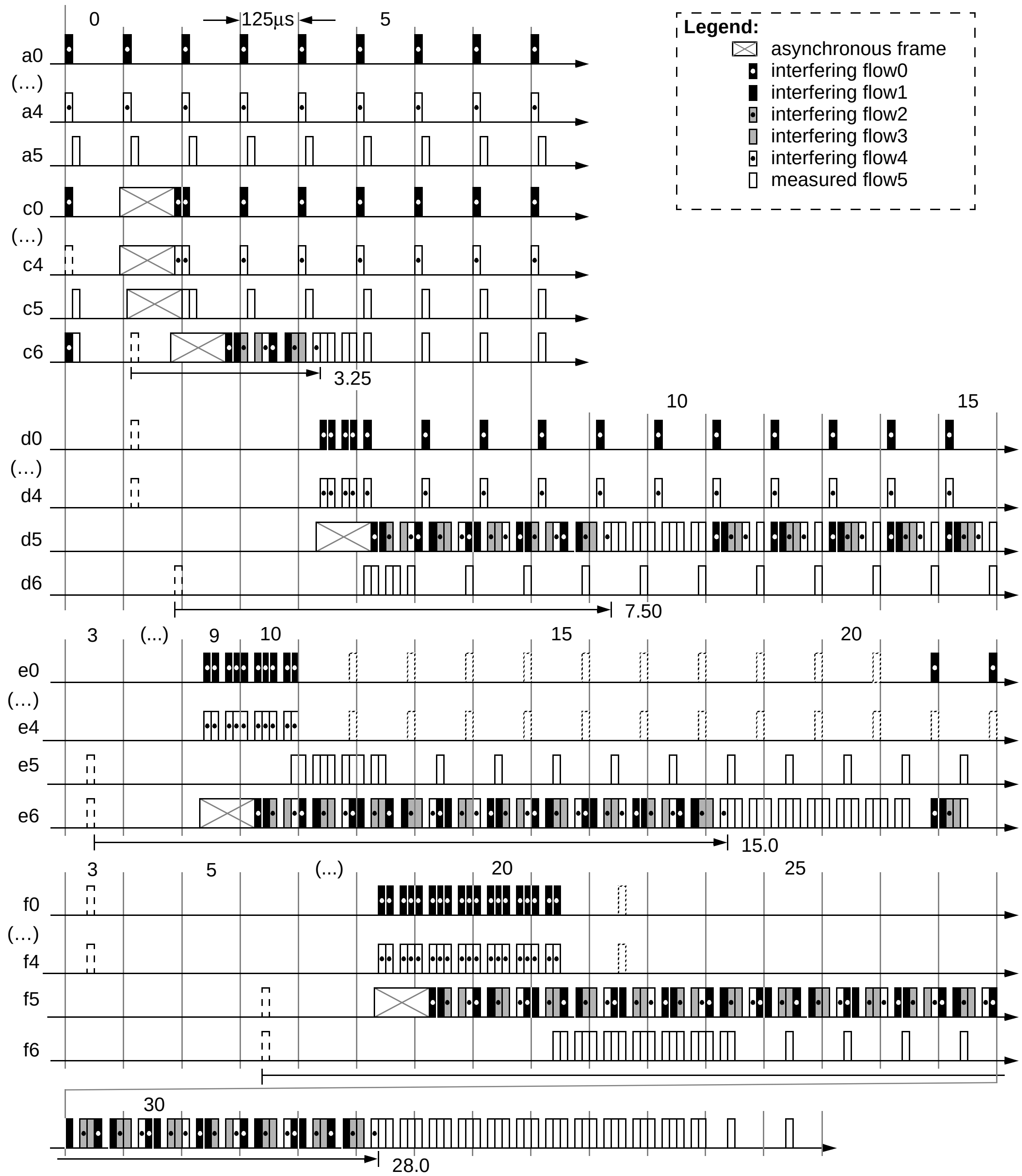


Figure F.16—Six source bunching; throttled-rate output-queue bridges

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							
		B	C	D	E	F	G	H	I
3-source (see F.2.2.1)	cycles	0.75	2.25	5.25	9.00	14.5	–	–	–
	ms	0.09	0.28	0.66	1.13	1.8	–	–	–
6-source (see F.2.2.2)	cycles	0.75	3.25	7.5	15.0	28	–	–	–
	ms	0.09	0.30	0.94	1.88	3.5	–	–	–

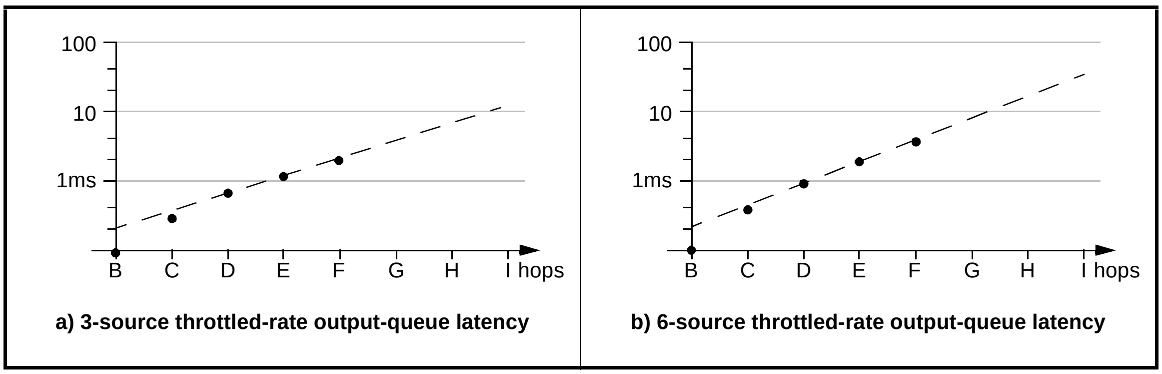


Figure F.17—Cumulative bunching latencies; throttled-rate output-queue bridge

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

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 F.19. Bridge ports {c0,c1,c2} concentrates traffic from three talkers; one third of the cumulative traffic is forwarded through port c3. 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.

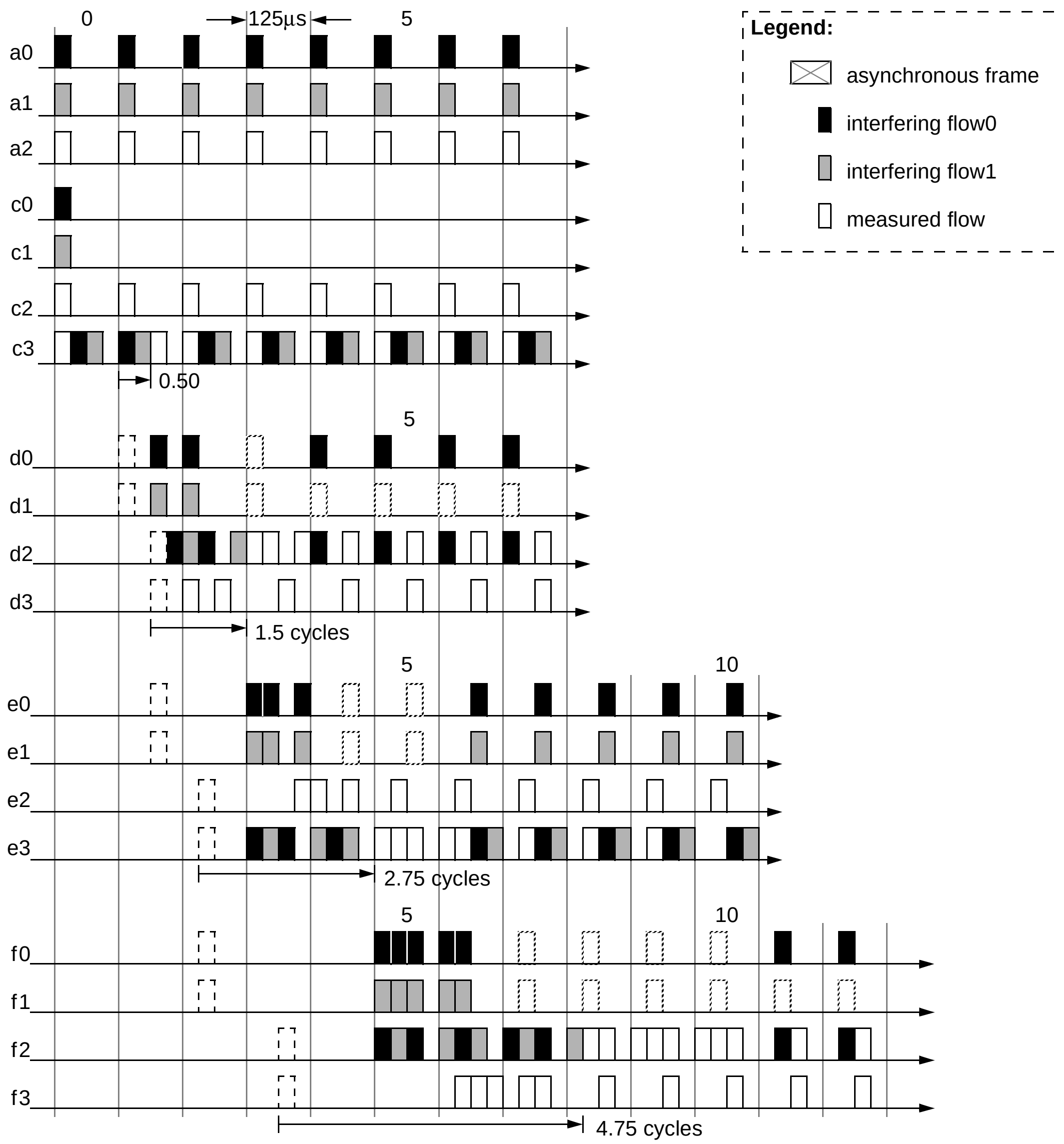


Figure F.18—Three-source bunching; throttled-rate output-queue bridges

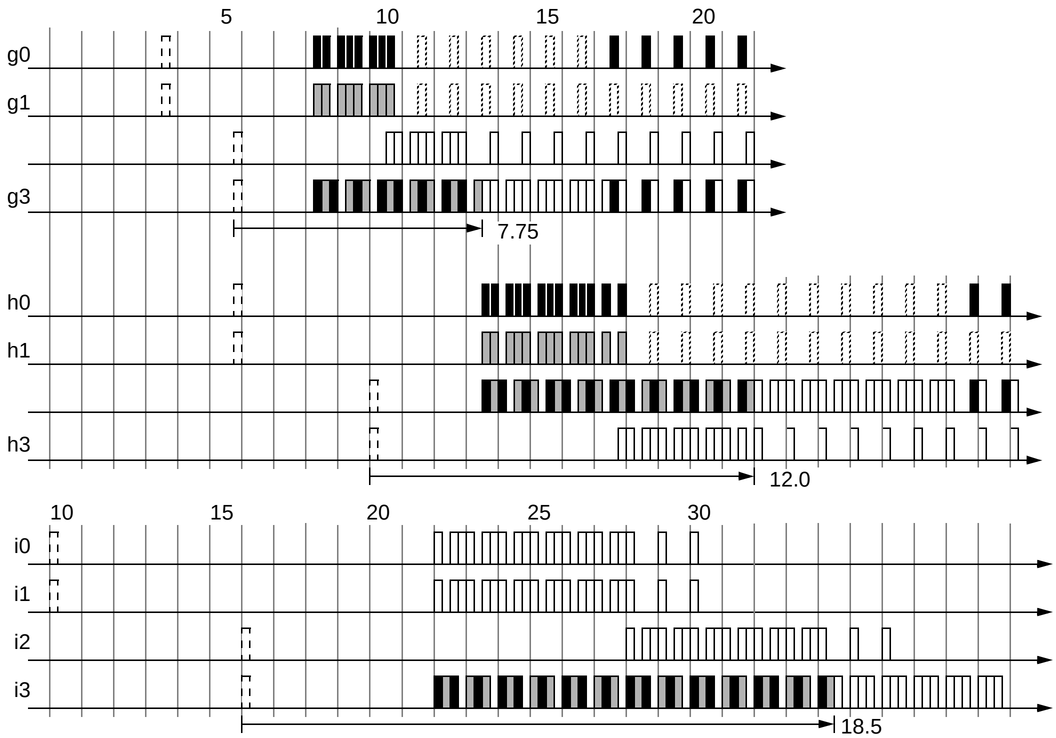


Figure F.19—Three-source bunching; throttled-rate output-queue bridges

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.16. Bridge ports {c0,c1,c2,c3,c4,c5} concentrates traffic from six talkers; one sixth of the cumulative traffic is forwarded through port c6. 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},…,{f0,f1,f2,f3,f4,f6} only illustrate passing-through traffic; the remainder of the traffic is routed elsewhere.

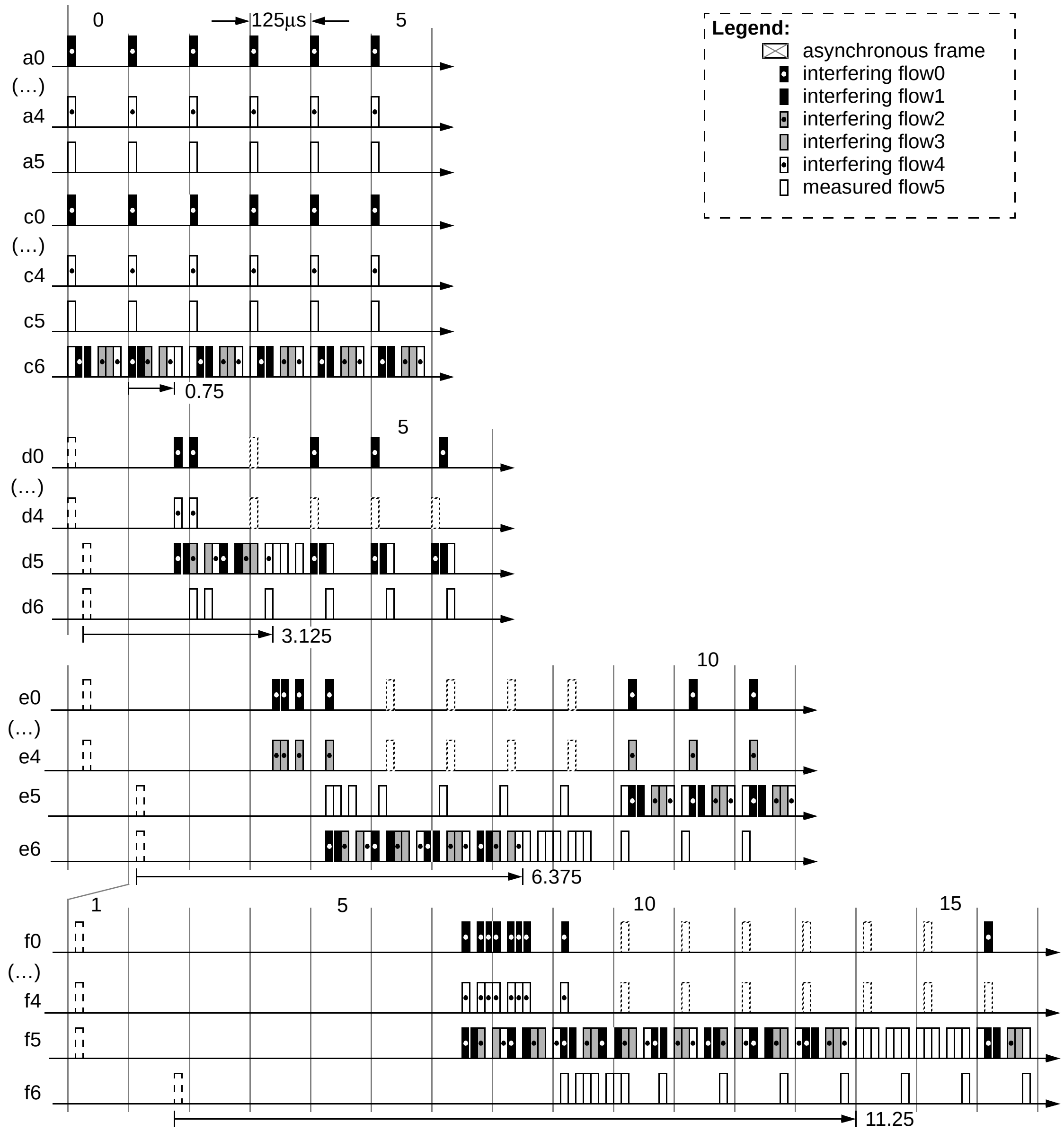


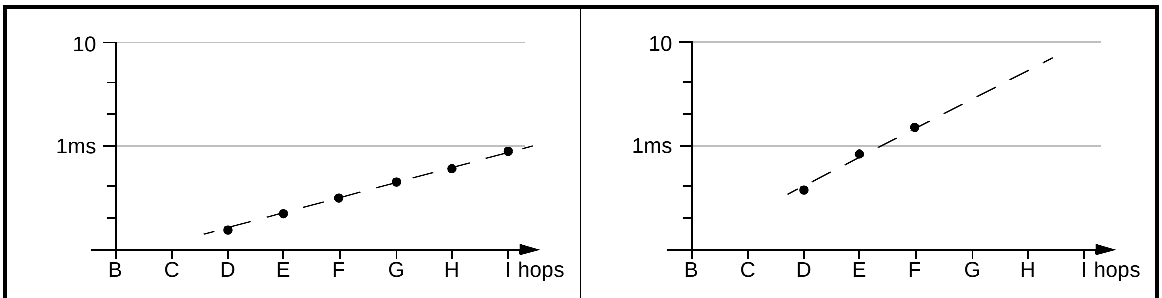
Figure F.20—Six source bunching; classA throttled-rate output-queue bridges

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							
		B	C	D	E	F	G	H	I
3-source (see F.2.2.1)	cycles	–	0.50	1.5	2.75	4.75	7.75	12.0	18.5
	ms	–	0.06	0.19	0.34	0.59	0.97	1.5	2.31
6-source (see F.2.2.2)	cycles	–	0.75	3.125	6.375	11.5	–	–	–
	ms	–	0.09	0.39	0.80	1.44	–	–	–



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: TBD.

F.2.7 Bunching concerns

This subannex evaluates several bridge forwarding scenarios, with the intent of providing guidance for RE capable bridge designs. Observations based on analysis of these scenarios leads to the following concerns towards throttled-rate output-queue bridges:

a) Idling. Bunching allows active links to appear inactive for multiple cycles.
This could affect the stream-present timeout delays associated with subscription protocols.

b) Storage. Additional storage to ensure lossless classA transmissions.
(These properties has been deferred to future revisions of this working paper).

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 Source-routed frame formats

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 G.2. 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.

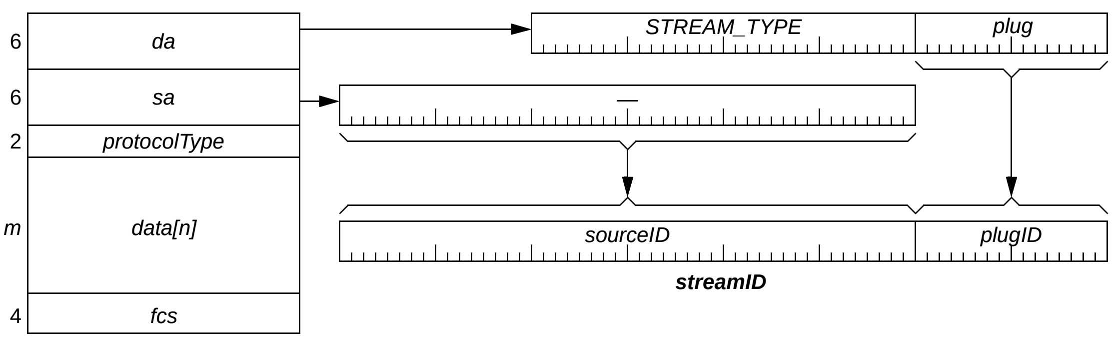


Figure G.1—classA frame formats

This advantages of this approach (which relies on the multicast nature of classA streams) include:

a) Localized. The administration of multicast addresses is managed independently by each talker, eliminating the need to provide, configure, and manage multicast address servers.

b) Efficient. The inclusion of a *protocolType* field to identify a frame's classA nature is unnecessary. Efficiency reduces the need for bridge-aware multi-block frame formats (see 5.3.3).

c) Structured. The stacking order of *protocolType* values is unaffected by its classA nature.

The primary disadvantage of this approach relates to its forwarding through bridges:

a) Different. Within existing bridges, multicast routing decisions are nominally based on the multicast *da* address; the *sa* address is normally ignored.

G.1.2 VLAN routed frame formats

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.2.

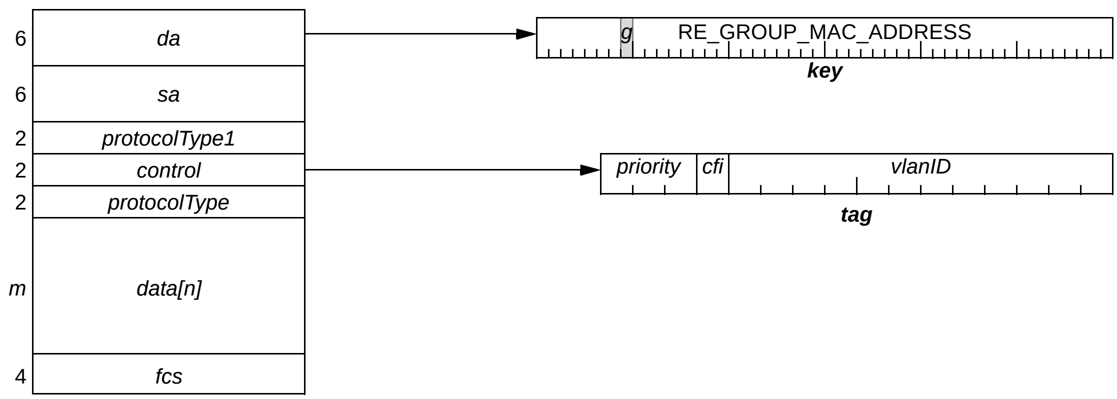


Figure G.2—classA frame formats

A single multicast address (labeled as RE_GROUP_MAC_ADDRESS) identifies the multicast time-sensitive nature of the frame. The 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.

This design approach (which relies on the multicast nature of classA streams) has desirable properties:

a) Similar. The *vlanID* is currently used to selectively route unicast as well as multicast frames.

The primary disadvantage 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.2 Subscription

G.2.1 Simple Reservation Protocol (SRP) overview

Subscription involves explicit negotiation for bandwidth resources, performed in a distributed fashion, flowing over the paths of intended communication. The RE subscription protocols are called Simple Reservation Protocols (SRP), due to their simplicity as compared to the Resource Reservation Protocol (RSVP). SRP shares many of the baseline RSVP features, including the following:

a) SRP is simplex, i.e. reservations apply to unidirectional data flows.
b) SRP is receiver-oriented, i.e., the receiver of a classA stream initiates and maintains the resource reservation used for that stream.
c) SRP maintains “soft” state in bridges, providing graceful support for dynamic membership changes and automatic adaptations to changes in network topology.
d) 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.

a) SRP is symmetric, i.e. the listener-to-talker path is the inverse of the talker-to-listener path.
b) SRP does no not provide for transcoding; any stream is fully characterized by its streamID and bandwidth.

G.2.2 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 RequestRefresh messages; this state is deleted if no matching RequestRefresh messages arrive before the expiration of a “cleanup timeout” interval. Listener’s may also force state deletions by generating an explicit RequestLeave message.

RequestRefresh messages are idempotent. When a route changes, the next RequestRefresh message will initialize the path state to the new route, and future RequestRefresh 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 RequestRefresh 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 classA traffic; this state is deleted if no matching classA traffic arrives before the expiration of a “cleanup timeout” interval. Thus, talker stations or agents may force reservation-state deletions by stopping their transmissions of classA traffic.

SRP sends it messages as layer-2 datagrams with no reliability enhancement. Periodic transmissions by listener 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.

G.2.3 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) Admissions controls (described in previous subclauses) reject any RequestRefresh message that (when combined with previously accepted request) would consume more than 75% of 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.

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

G.2.4 Bridge-resident agents

Subscription facilities establish multicast 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 G.3-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 G.3-b.

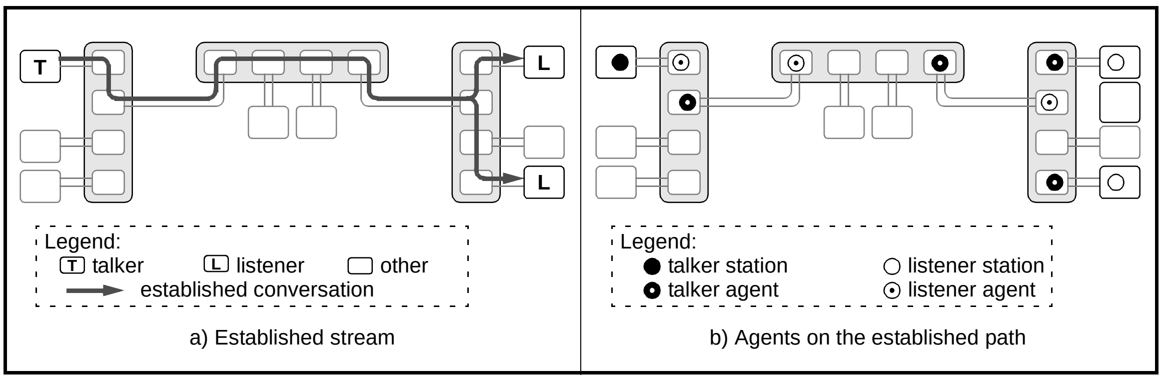


Figure G.3—Agents on an established path

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 rejected if the cumulative bandwidth would exceed 75% of the link capacity. The talker agents are also responsible for sustaining streams of classA data; their absence can result in disconnections of the attached listener agent.

The listener agents are responsible for periodically refreshing their adjacent talker agents, to confirm their continued presence. A persistent absence of refreshes causes the adjacent talker agent to disconnect its stream transmissions and (if appropriate) to inform other station-local agents.

For each established stream within a bridge, the listener agent remains active while all but the last downstream flows are disconnected. The upstream station receives its disconnect notice only after the last of the downstream flows has disconnected.

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

G.2.5 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 G.4.

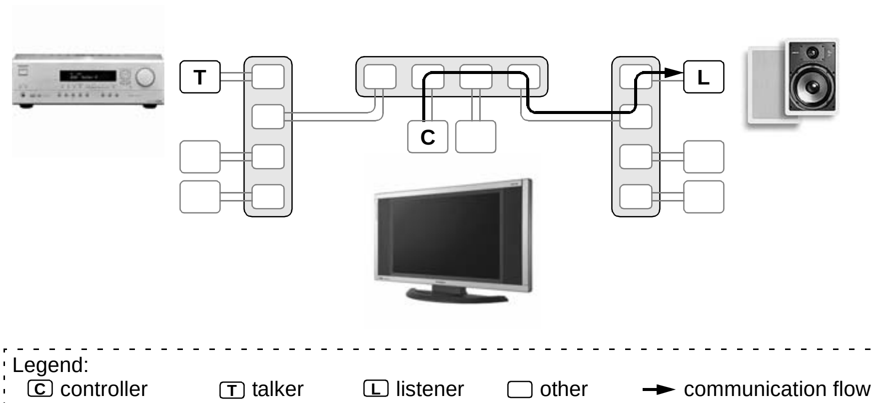


Figure G.4—Controller activation

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.

G.2.6 Pinging the talker

After being activated by a talker, listeners are expected to ping the talkers before initiating subscription operations, as illustrated in Figure G.5. The purpose of the ping is to ensure that bridges have learned listener and talker addresses, allowing frames to be sequentially passed from the listener-to-talker.

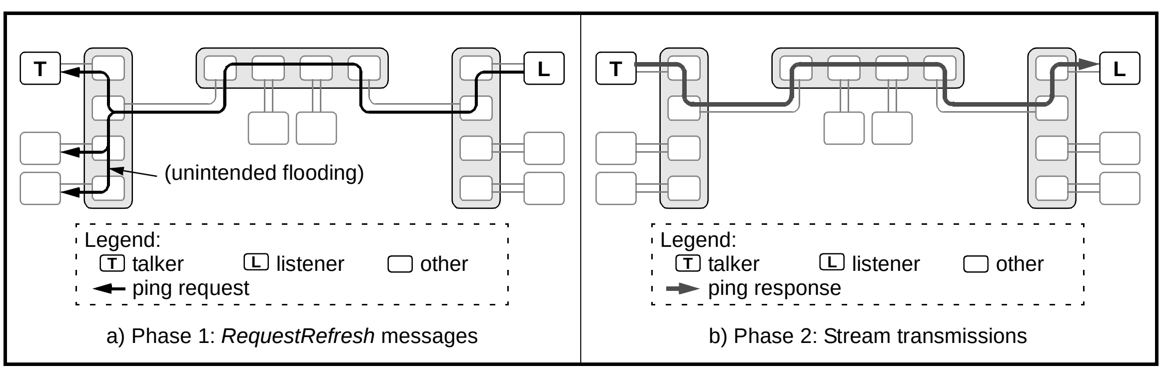


Figure G.5—Pinging the talker

G.2.7 Path creation

Establishing a conversation between a listener and a talker involves sending a RequestRefresh message from the listener towards the talker, illustrated by the dark arrow paths in Figure G.6-a. If available bandwidths are sufficient, the talker starts its stream transmissions, as illustrated by the gray arrow paths in Figure G.6-b.

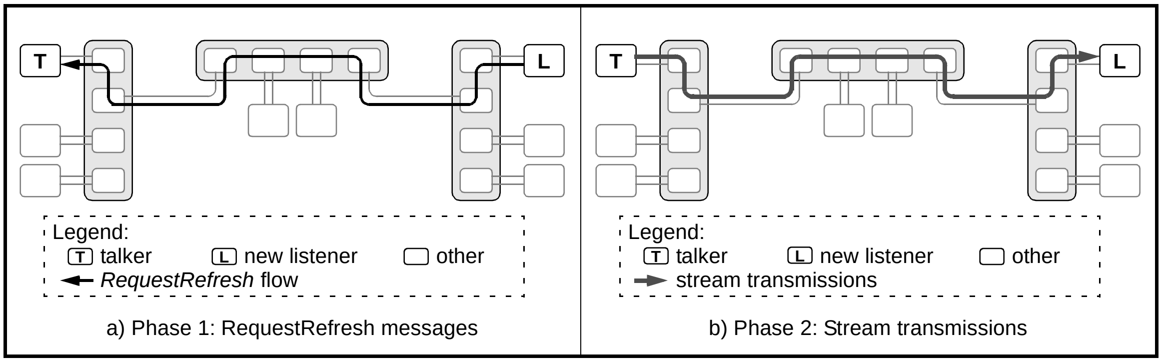


Figure G.6—Path creation

In rare circumstances, some talker addresses may not have been learned and the RequestRefresh message will terminate with a returned ResponseError message. The listener has the option of repeating the RequestRefresh after performing a ping (see G.2.6), which validates the talker presence and activates bridge learning.

Another timeouts is associated with the absence of periodic RequestRefresh messages. In the continued absence of these expected messages, the listener is assumed to be absent or deactivated. Based on this assumption, the associated talker (station or agent) resources are released.

G.2.8 Side-path extensions

A second listener joins an established conversation by sending a RequestRefresh message towards the talker, as illustrated by the dark-arrow path in Figure G.7-a. When an established connection is discovered, the switch (not the talker) returns stream transmissions, as illustrated by the dark-gray path in Figure G.7-b.

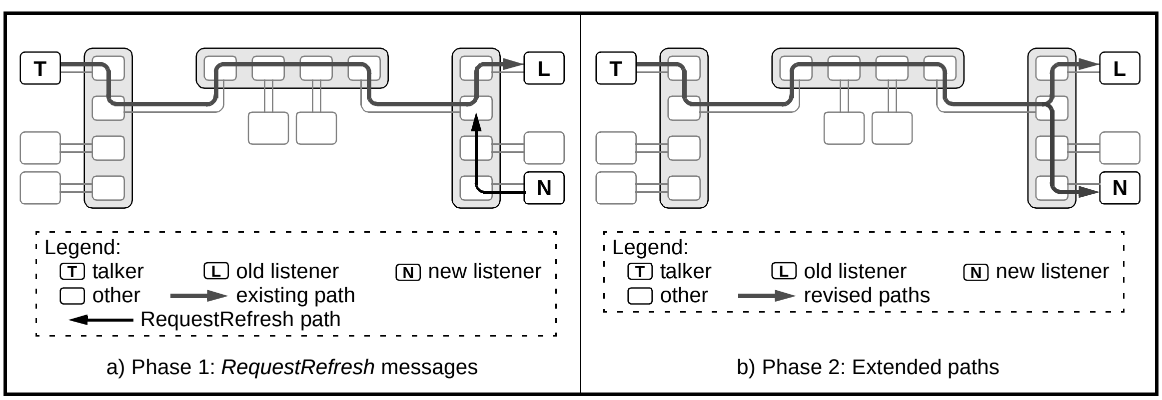
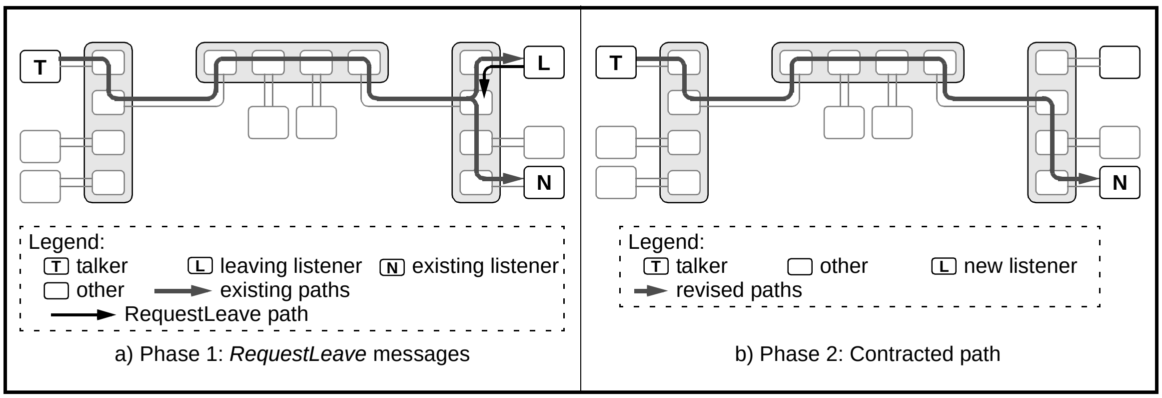


Figure G.7—Side-path extensions

Each talker agent maintains separate state, so that classA traffic can be multicast to the applicable stations, rather than flooded downstream. The distinct markers also allow the switch to detect when the last listener disconnects, so that its previously shared upstream span can be released appropriately.

G.2.9 Side-path release

A retiring listener normally leaves an established conversation, by sending a RequestLeave message towards the talker. That message propagates to the nearest merging bridge connection, as illustrated by the dark-arrow path in Figure G.8-a. When an established/merged connection is discovered, the switch (not the talker) stops the stream transmissions, as illustrated by the disappearance of a side path in Figure G.8-b.

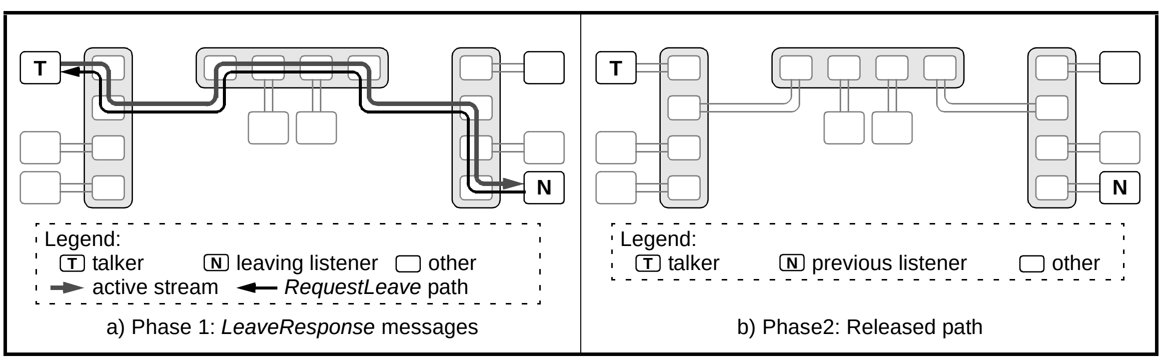


a) Phase 1: *RequestLeave* messages b) Phase 2: Contracted path

Figure G.8—Side-path demolition

G.2.10 Released path

The final listener bandwidth release involves sending a RequestLeave message towards the talker. In this case, that message propagates to the talker, as illustrated by the dark-arrow path in Figure G.9-a. The stream transmissions then stop, as illustrated in Figure G.9-b.



a) Phase 1: *LeaveResponse* messages b) Phase2: Released path

Figure G.9—Released path

G.2.11 Errors and timeouts

G.2.11.1 Subscription failures

A RequestRefresh message can encounter an error while flowing from the listener towards the talker, illustrated by the dark arrow paths in Figure G.10-a. When such errors occur, a ResponseError message is normally returned to the listener, as illustrated by the gray arrow paths in Figure G.10-b.

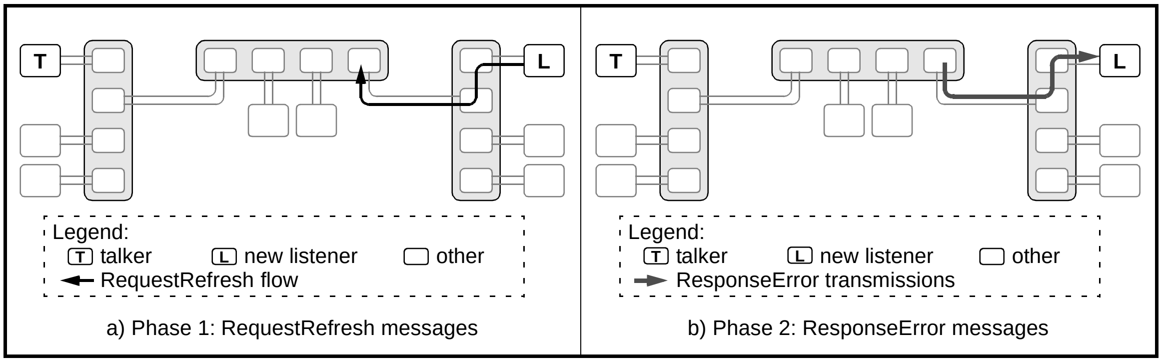


Figure G.10—Error responses

Errors may be associated with a variety of errors including (but not limited to) the following:

a) Insufficient resources. Necessary resources are available within the bridge:
   1) Insufficient bandwidth is available on the link from the talker agent to its adjacent listener.
   2) Insufficient path-related resources are available in the bridge's talker agent.
   3) Insufficient path-related resources are available in the bridge's upstream listener agent.
   4) Insufficient link or memory bandwidth is available with the bridge.
b) Unlearned address. The route from the bridge to the talker is unknown.
   (To avoid complexities and inefficiencies, RequestRefresh messages are never flooded.)

G.2.11.2 Listener-presence timeouts

Listener agents and stations are responsible for refreshing their local talkers, to demonstrate their continued presence. In the absence of these refresh messages, the talkers assume the listener is absent and teardown the inactive path (or inactive branch from the path).

Thus, sustaining the active paths of Figure G.11-a requires periodic refresh messages on each hop, as illustrated in Figure G.11-b. The refresh messages and associated timeouts are performed independently on each span. The messages that establish the path (see G.2.7 and G.2.8) are the same as these listener-initiated messages that sustain the established path.

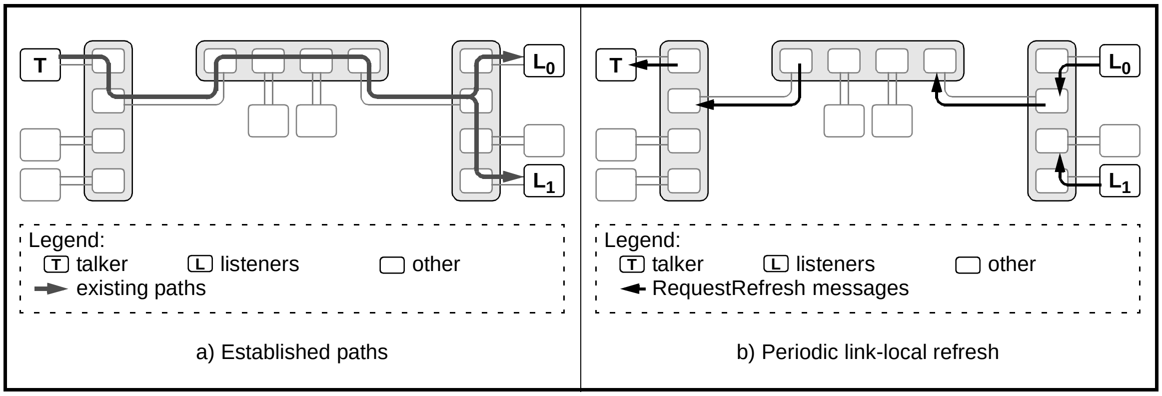


Figure G.11—Side-path demolition

G.2.11.3 Talker-presence timeouts

Talker agents and stations are responsible for updating their local listeners, to demonstrate their continued presence. In the absence of these updates, the listeners assume the talker is absent and teardown the inactive path (or inactive branch from the path).

Thus, sustaining the active paths of Figure G.11-a requires periodic transmissions of classA traffic on each hop (not illustrated). The associated timeouts are performed independently on each span. The frames that transfer classA data are the same as these talker-initiated frames that sustain the established path.

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.

H.2 Formulated responses

TBD

Annex I

(informative)

Comment responses

NOTE—This clause should be skipped on the first reading (reading starts at Clause 1).
This clause is provided for communicating detailed responses to reviewer comments.

I.1 Recent review-comment resolutions

I.1.1 Kevin Gross comments

Alexi has suggested 15ms for instrument to ear latency (my experience says you're good all the way up to 50 ms). I have suggested <0.5 ms as a first choice for voice to ear when headphones are involved and 5 - 50 ms as a second-best choice. I'm not sure where the 10ms figure you're using in equation 5.9 comes from. I've revised some of the section 5.1.4 text to show you what I had in mind...

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. Due to multiple hops and the latency contributions (see Equation 5.9), the constraints on the value of T (see Equation 5.10 and Equation 5.11) yield a T value constraint that is physically impossible for today's digital audio technology to achieve.

$t0 + t1 + t2 + t3 + t4 + t5 + t6 < 0.5$ ms (5.9)
1ms+ T + T +5ms+ T + T +1ms+< 0.5 ms (5.10)
4ÞT + 7ms < 0.5 ms (5.11)
T < -1.6 ms (5.12)

Some 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. The system in figure 5.4, when 0.5 ms network delays are assumed, produces an overall latency that fits comfortably within these relaxed constraints.

4*0.5ms + 7ms = 9 ms (5.13)

-----Original Message-----
From: Gross, Kevin
Sent: Thursday, April 28, 2005 9:16 AM
To: 'David V James'
Subject: RE: [RE] Latencies through RE cables (better URL)

Sure, I'd be happy to review it.

If you include this scenario and accept a <0.5ms delay requirement for it,
something's gonna have to give further down the line.

My suggestion: <0.5ms is not achievable with digital audio systems because you blow your latency budget in A/D and D/A alone. 0-0.5ms is the conventionally desirable operating range for this scenario. 0.5-5ms is nasty due to comb filtering. Although it defies the conventional latency wisdom that less is more, 5-50ms is actually a comfortable place to operate in this scenario; we should shoot for that. Note that your existing 15ms requirement falls in the 5-50ms range.

I.1.2 Michael Johas Teener comments

From: Michael Johas Teener [mailto:Mikejt@broadcom.com]
Sent: Monday, June 06, 2005 3:19 PM
To: David James
Subject: Re: Short prereview scan

a) Your hypertext TOC entries are all wrong… I think your PDF options on Framemaker are wrong…
Response: Fixed.

b) No update to version history
Response: Huh? Version history was updated, but version number was in error.

c) F.1.2 and F.1.3 it isn't clear where the "b" stations are … I think they are the outputs of "a", but it isn't obvious …
Response: A separate column now identifies the source and stations/ports are uniformly labeled.

d) Horiz scale of figures not obvious … are they 8kHz cycles?
Response: Yes, they are 8kHz cycles, now labeled as 125 µs cycles.

e) F.2.5 … it isn't certain what the throttle algorithm is being used (75% for "stream" traffic over a measurement interval of 1 cycle?)
Response: Yes, that is the algorithm. Not yet sure how to clarify or if others should be documented. Good topic for discussion.

I.1.3 Felix Feng comments

From: Feifei Feng [mailto:feng.fei@samsung.com]
Sent: Monday, June 06, 2005 4:55 PM
To: 'David V James'
Subject: RE: Short prereview scan

I'm comfortable with the basic message flows, namely, listener announcing + talker responding (with resources locking and notifying). It reflects our consensus during the ad-hoc conference call.

Comments and questions include:

a) You may explicitly indicate that the listener announcement can reuse the GARP mechanism with few changes. Therefore the simplicity and feasibility of SRP can be emphasized. RequestJoin and RequestLeave will have corresponding primitives in GARP.

b) I'm not sure what the "resources" in page 43 line 5 are referring to? Do you mean the processing power, registration table etc. for GARP?

c) Page39 line53 "Although speculative registration resources are allocated within bridges, these resources are released after timeouts have verified the absence of the talker station". I think there are two scenarios to remove the speculative registration. The first one is to actively detect the timeout from the talker side (no response from upstream in a specified period). The second one is to detect the timeout from the listener side (once the talker's address has been learnt by an intermediate bridge, this bridge will stop sending Join to other upstream bridges. Those bridges will timeout since no Join from downstream). The final solution may choose either of them, or both. It should be further studied. Your description falls into only the first case.

d) Page37 line32 “The state on the now-unused segment of the route will be deleted after a timeout interval”. Similar to Comment 3, clarification might be needed for whether the timeout depends on the upstream refresh or downstream refresh.

I understand that detail specification should be refined only in task force. So it’s ok to just leave Comment c&d under discussion.

Annex J

(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.

```
/// *************************************************************************************************************************************
//                                                                                                         1         1         1         1
//       1         2         3         4         5         6         7         8         9         0         1         2         3
//345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123

#include <assert.h>
#include <stdio.h>

// unsigned char      uint8_t;                                // 1-byte unsigned integer
// unsigned short     uint16_t;                               // 2-byte unsigned integer
// unsigned int       uint32_t;                               // 4-byte unsigned integer
// unsigned long long uint64_t;                               // 8-byte unsigned integer

// signed char        int8_t;                                 // 1-byte signed integer
// signed short       int16_t;                                // 2-byte signed integer
// signed int         int32_t;                                // 4-byte signed integer
// signed long long   int64_t;                                // 8-byte signed integer

#define BASE_TIMER 0                                          // Make this 1 if baseTimer hardware is provided
#define DIFF_SCALE ((double)4096 * ((uint64_t)1 << 31))       // Changes <200PPM to a 32-bit signed integer
#define EXTRACT_CORE(a, b) (((a) << 32) | ((b) >> 32))        // Extract seconds and fraction component
#define FULL_SCALE (0x7FFFFFFF)                               // Biggest 32-bit positive integer
#define LIMIT(a, b, c) MAX(MIN((a), (b)), (c))                // Force base/bounds constraints
#define MAX(a, b) ((a) < (b) ? (b) : (a))                     // Maximum value definition
#define MIN(a, b) ((a) > (b) ? (b) : (a))                     // Minimum value definition
#define MINIMUM_WIDE(a, b) (CompareWide((a), (b)) < 0 ? (a) : (b))
#define ONES64 ~((uint64_t)0)                                 // 64-bit all-ones value
#define SCALE64 ((double)16 * (1 << 30) * (1 << 30))          // Floating-point equivalent of (1<<64)
#define TICKS 10                                              // Number of 10ms intervals within each 100ms interval

typedef struct
{
    uint64_t hi;                                              // more-significant portion
    uint64_t lo;                                              // less-significant portion
} DoubleWide;

typedef struct _PortInfo
{
    struct _PortInfo *portPtr;                                // Points to the next linked port
    unsigned changing:1;                                      // A precedence-changing indication
    unsigned portLevel:4;                                     // Relative priority number of ports
    unsigned portNumber:12;                                   // Port number
    DoubleWide portPrecedence;                                // Incoming frame parameters

    uint8_t  skipCount;                                       // Number of 10ms intervals
    uint32_t cableDelay;                                      // The cable delay, from local master
    uint32_t linkOffset;                                      // The cable difference, from local master
    uint64_t deltaTime;                                       // For inclusion in transmitted frames

                                                              // Best if captured accurately by the PHY
    uint64_t latchRxFlexTime;                                 // Snapshot of flexTimer, on clockSync arrival,
                                                              // available for this clockSync reception.
    uint64_t latchTxFlexTime;                                 // Snapshot of flexTimer, on clockSync departure,
                                                              // available for next clockSync transmission
```

```
    uint64_t savedRxFlexTime;                       // Previous latchRxFlexTime value
    uint64_t savedRxFlexData;                       // Previous clockSync.lastFlexTime value

                                                    // If BASE_TIMER is 1, baseTimer snapshots:
    uint32_t latchRxBaseTime;                       // Captured clockSync base-time arrival
    uint32_t latchTxBaseTime;                       // Captured clockSync base-time departure (delayed)
} PortInfo;

typedef struct
{                                                   // Customized per-station components
    PortInfo *portPtr;                              // Points to a linked-list of ports
    double clockFrequency;                          // Nominal clock frequency, in Hz
    int8_t deviation;                               // Deviation in parts-per-million
    uint64_t eui64;                                 // 64-bit extended unique identifier
    unsigned systemLevel:4;                         // Relative priority number of ports
    unsigned systemNumber:12;                       // Port number

    uint8_t  skipCount;                             // Number of 10ms intervals
    uint32_t myDiffRate;                            // The rate difference, from upstream neighbor
    uint32_t diffRate;                              // The rate difference, from grand-master
    uint32_t linkOffset;                            // The cable difference, from local master
    uint64_t deltaTime;                             // For inclusion in transmitted frames

    DoubleWide thisPrecedence;                      // The precedence of this station
    DoubleWide bestPrecedence;                      // The best observed precedence
    int16_t bestPort;                               // Selected clock-slave port

    uint64_t savedRxFlexTime;                       // Previous latchRxFlexTime value
    uint32_t savedRxBaseTime;                       // Previous latchRxBaseTime value

    uint64_t timeOfDay;                             // Offset and rate-compensated timer value
    uint64_t flexTimerHi;                           // Offset and rate adjustable 64-bit timer
    uint64_t flexTimerLo;                           // Offset and rate adjustable 64-bit timer
    uint64_t flexOffset;                            // Adjustable offset value for flexTimer
    uint64_t flexRate;                              // 40-bit adjustable rate for flexTimer

    uint64_t baseTimer;                             // Fixed-rate fixed-offset 64-bit timer
    uint64_t baseRate;                              // SCALE64/clockFrequency, pre-initialized

    uint32_t savedRxBaseTickTime;                   // Saved values of savedRxBaseTime
    uint32_t savedRxBaseTickData;                   // Saved values of clockSync.lastBaseTime;
} StationInfo;

typedef struct                                      // The clockSync frame, reserved-padded to
{                                                   // the minimum 64-byte frame size.
    uint32_t  da_hi;                                // Ethernet's 48-bit destination address
    uint16_t  da_lo;                                //  "
    uint16_t  sa_hi;                                // Ethernet's 48-bit source address
    uint32_t  sa_lo;                                //  "
    uint16_t  protocolType;                         // Specifies format/meaning of following
    uint8_t   subType;                              // Refined format/meaning specification
    uint8_t   syncCount;                            // Sequence numbers for consistency checks
    uint8_t   hopsCount;                            // Hop counts from the grand master
    uint8_t   reserved;                             // A few reserved bytes, for 64-byte minimum
```

```
    uint16_t  systemTag;                              // Precedence for grand-master election
    uint64_t  uniqueID;                               // Identifier for grand-master election
    uint64_t  lastFlexTime;                           // flexTimer on last clockSync transmission
    uint64_t  deltaTime;                              // Time difference on opposing link
    uint64_t  offsetTime;                             // Cumulative grand-master offset differences
    uint32_t  diffRate;                               // Cumulate grand-master rate differences
    uint32_t  lastBaseTime;                           // baseTimer on last clockSync transmission
    uint32_t  fcs;                                    // Frame check sequence
} ClockSyncFrame;

uint32_t   BaseTimerChange(uint64_t, uint64_t, double);
void       CheckOther(StationInfo *, PortInfo *);
void       ClockSyncArrived(StationInfo *, PortInfo *);
void       ClockSyncDeparted(StationInfo *, PortInfo *);
void       ClockSyncReceive(StationInfo *, PortInfo *, ClockSyncFrame *, uint8_t);
void       ClockSyncTransmit(StationInfo *, PortInfo *, ClockSyncFrame *);
int        CompareWide(DoubleWide, DoubleWide);
int        PrecedenceCheck(StationInfo *, PortInfo *, ClockSyncFrame *);
DoubleWide PrecedenceMerge(uint16_t, uint64_t, uint8_t, uint16_t);
void       TimerTick(StationInfo *);


// Called with:
//   stationInfoPtr  -- the station information context
void
StationSetup(StationInfo *stationInfoPtr)
{
    PortInfo *portPtr;
    StationInfo *siPtr = stationInfoPtr;
    uint16_t systemTag;
    uint16_t i;

    assert(siPtr != NULL);
    siPtr->baseRate = (SCALE64/siPtr->clockFrequency) * ( 1 + siPtr->deviation/(double)1000000);
    systemTag = ((uint16_t)(siPtr->systemLevel) << 12) | siPtr->systemNumber;
    siPtr->thisPrecedence =
     PrecedenceMerge(systemTag, siPtr->eui64, 0, 0);
    for (i = 0, portPtr = siPtr->portPtr; portPtr != NULL; portPtr = portPtr->portPtr, i += 1) {
        portPtr->portNumber = i;
    }
}

// Called with:
//   stationInfoPtr  -- the station information context
//   portInfoPtr     -- the port information context
//   clockSyncPtr -- the contents of a clockSync frame
void
ClockSyncReceive(StationInfo *stationInfoPtr, PortInfo *portInfoPtr, ClockSyncFrame *clockSyncPtr, uint8_t rateAdjust)
{
    PortInfo *piPtr = portInfoPtr;
    StationInfo *siPtr = stationInfoPtr;
    ClockSyncFrame *csPtr = clockSyncPtr;
    uint32_t measuredDelta, receivedDelta, diffRate;
    uint64_t rxDelta, txDelta, clockDelta, cableDelay;
    double tempRate;
```

```
    int8_t grandMaster, slave;

    assert(siPtr != NULL && piPtr != NULL);
    if (PrecedenceCheck(siPtr, piPtr, csPtr))
        return;

    rxDelta = piPtr->savedRxFlexTime - csPtr->lastFlexTime;                          // Measured receive-link delay
    txDelta = csPtr->deltaTime;                                                      // Reported transmit-link delay
    clockDelta = (txDelta - rxDelta)/2;                                              // Local timer differences
    cableDelay = (txDelta + rxDelta)/2;                                              // Cable transmission delay

    grandMaster = ((siPtr->bestPrecedence.lo >> 16) == 0);                           // Grand master has hopsCount==0
    slave = ((siPtr->bestPrecedence.lo & 0X0FFF) == piPtr->portNumber);              // A slave port is so identified
    if (!grandMaster && slave) {                                                     // The slave tracks the master
        diffRate = LIMIT(siPtr->myDiffRate + csPtr->diffRate, FULL_SCALE, -FULL_SCALE);  // Rate-range limitation
        siPtr->flexOffset = csPtr->offsetTime + clockDelta + siPtr->linkOffset;      // Offset compensation
        siPtr->flexRate = siPtr->baseRate + siPtr->baseRate * (diffRate / DIFF_SCALE);   // Rate compensation
        if (rateAdjust)
        {
            // Computed at triggered intervals, measured over the last interval
            measuredDelta = (siPtr->savedRxBaseTime - siPtr->savedRxBaseTickTime);   // Clock-slave difference
            receivedDelta = (csPtr->lastBaseTime - siPtr->savedRxBaseTickData);      // Clock-master difference
            siPtr->savedRxBaseTickTime = siPtr->savedRxBaseTime;                     // Previous saved value
            siPtr->savedRxBaseTickData = csPtr->lastBaseTime;                        // Previous saved value
            tempRate = DIFF_SCALE * ((double)(receivedDelta - measuredDelta)/receivedDelta);  // Local rate difference
            siPtr->myDiffRate = LIMIT(tempRate, FULL_SCALE, -FULL_SCALE);            // Rate difference limits
        }
        siPtr->diffRate = diffRate;                                                  // Passthrough difference
        if (BASE_TIMER)
            siPtr->savedRxBaseTime = piPtr->latchRxBaseTime;
        else
            siPtr->savedRxBaseTime +=                                                // Receiver’s baseTimer snapshot
             BaseTimerChange(siPtr->savedRxFlexTime, piPtr->latchRxFlexTime, diffRate);
    }
    piPtr->cableDelay = cableDelay;                                                  // Local cable-delay knowledge
    piPtr->savedRxFlexTime = piPtr->latchRxFlexTime;                                 // Saved reference time
    piPtr->deltaTime = rxDelta;                                                      // Saved for retransmission
}


// Called with:
//   stationInfoPtr  -- the station information context
//   portInfoPtr     -- the port information context
//   clockSyncPtr    -- the contents of a clockSync frame
void
ClockSyncTransmit(StationInfo *stationInfoPtr, PortInfo *portInfoPtr, ClockSyncFrame *clockSyncPtr)
{
    ClockSyncFrame *csPtr = clockSyncPtr;
    PortInfo       *piPtr = portInfoPtr;
    StationInfo    *siPtr = stationInfoPtr;

    assert(siPtr != NULL && piPtr != NULL && csPtr != NULL);

    // An absent baseTimer is emulated by properly scaling time differences,
    // measured from the last recorded received-clockSync event.
```

```
    //   - baseTime value was computed
    //   - a different normDiffRate value has taken effect
    if (!BASE_TIMER)
        piPtr->latchTxBaseTime = siPtr->savedRxBaseTime +                            // Derived from latchTxFlexRate
          BaseTimerChange(siPtr->savedRxFlexTime, piPtr->latchTxFlexTime, siPtr->diffRate);

    csPtr->hopsCount  = (siPtr->bestPrecedence.lo >> 16) + 1;                          // Increment hop-count value
    csPtr->systemTag = (siPtr->bestPrecedence.hi >> 32);                               // Best preference value
    csPtr->uniqueID  = (siPtr->bestPrecedence.hi << 32) | (siPtr->bestPrecedence.lo >> 32) ;       // Unique differentiation

    csPtr->lastFlexTime = piPtr->latchTxFlexTime;                                      // Send last timer value
    csPtr->deltaTime    = piPtr->deltaTime;                                            // Send received-link delay
    csPtr->lastBaseTime = piPtr->latchTxBaseTime;                                      // Send last baseTimer value
    csPtr->offsetTime   = siPtr->flexOffset;                                           // This station’s cumulative offset
    csPtr->diffRate     = siPtr->diffRate;                                             // Send current diffRate value
}

// Called when a clockSync frame is received, to latch timer values.
// Latches timers are available when ClockSyncReceive() is called.
//
// Called with:
//   stationInfoPtr -- the station information context
//   portInfoPtr     -- the port information context
void
ClockSyncArrived(StationInfo *stationInfoPtr, PortInfo *portInfoPtr)
{
    PortInfo    *piPtr = portInfoPtr;
    StationInfo *siPtr = stationInfoPtr;

    assert(siPtr != NULL);
    piPtr->latchRxFlexTime = EXTRACT_CORE(siPtr->flexTimerHi, siPtr->flexTimerLo);      // Latch seconds:fraction fields
    if (BASE_TIMER)                                                                   // If a BASE_TIMER is present,
        piPtr->latchRxBaseTime = siPtr->baseTimer >> 32;                              // latch its fraction field
}

// Called when a clockSync frame is transmitted, to latch timer values.
// Latches timers are available for the next ClockSyncTransmit() call.
//
// Called with:
//   stationInfoPtr -- the station information context
//   portInfoPtr     -- the port information context
void
ClockSyncDeparted(StationInfo *stationInfoPtr, PortInfo *portInfoPtr)
{
    PortInfo    *piPtr = portInfoPtr;
    StationInfo *siPtr = stationInfoPtr;

    assert(siPtr != NULL);
    piPtr->latchTxFlexTime = EXTRACT_CORE(siPtr->flexTimerHi, siPtr->flexTimerLo);      // Latch seconds:fraction fields
    if (BASE_TIMER)                                                                   // If a BASE_TIMER is present,
        piPtr->latchTxBaseTime = siPtr->baseTimer >> 32;                              // latch its fraction field
}

// Called when a clockSync frame is received, or after a clockSync-silence timeout.
// Performs the grand-master precedence check, with field concatendated as follows:
```

```
//
//   +-------------------------------------------------------------------------------+
//   |   zero    systemTag                   eui64                      00   hops   portTag |
//   +----------'----16----'-------------------64----------------------'--8--'--8--'----16---+
//
// If hops == ONES, this value is considered VOID and has the worse precedence
// Otherwise, the best precedence corresponds to the smallest of two tested values.
//
// Called with:
//   stationInfoPtr -- the station information context
//   portInfoPtr    -- the port information context
//   clockSyncPtr   -- the contents of a clockSync frame
int
PrecedenceCheck(StationInfo *stationInfoPtr, PortInfo *portInfoPtr, ClockSyncFrame *csPtr)
{
    PortInfo    *piPtr = portInfoPtr, *portPtr;
    StationInfo *siPtr = stationInfoPtr;
    uint16_t bestPort, portTag;
    int8_t test;

    assert(siPtr != NULL && piPtr != NULL);
    if (csPtr != NULL && csPtr->hopsCount != 0XFF) {                                         // In the absence of timeouts,
        portTag = ((uint16_t)(piPtr->portLevel) << 12) | piPtr->portNumber;                  // compute the precedence value.
        piPtr->portPrecedence =
         PrecedenceMerge(csPtr->systemTag, csPtr->uniqueID, csPtr->hopsCount, portTag);
    } else {                                                                                 // Otherwise, set precedence
        piPtr->portPrecedence.hi = piPtr->portPrecedence.lo = ~((uint64_t)0);                // to an ignored largest value
    }
    test = CompareWide(piPtr->portPrecedence, siPtr->bestPrecedence);                        // Compare precedence values
    bestPort = (siPtr->bestPrecedence.lo & 0X0FFF);                                          // and find the clock-master.
    if (test < 0 || (test > 0 && bestPort == piPtr->portNumber)) {                           // If the bestPort has changed,
        siPtr->bestPrecedence = MINIMUM_WIDE(piPtr->portPrecedence, siPtr->thisPrecedence);  // reset its value and recompute.
        for (portPtr = siPtr->portPtr; portPtr != NULL; portPtr = portPtr->portPtr) {
            if (portPtr != piPtr)
                CheckOther(siPtr, portPtr);
            portPtr->changing = 1;
        }
    }
    return(csPtr == NULL);                                                                   // Indicates processing impossible
}

// Called at a high clock rate (less than 20 ns) to update flexTimer and baseTimer (if present).
// This routine is intended to illustrate the computations involved in updating hardware timers;
// this code is _not_ expected to be incorporated into firmware.
//
//                                            +-----------------------------------------+
//              0000 0000 0000 0000(hex)      |       fraction          subfraction    | flexRate
//                       |                    +---------32---------.---------32---------+
//                       |                                          |
//   ____________________v____________________.____________________v____________________
//  ,----------------------------------------,----------------------------------------,
//  (                                  flexAdd:128                                     )
//   ---------------------------------------------------------------------------------
//                                        ^   |
//                                        |   v
```

```
//  +-------------------------------------------------------------------------------------+
//  |    superSeconds            seconds              fraction           subfraction     | flexTimer
//  +---------32---------'---------32---------'---------32---------'---------32---------+
//                        \________________________________________/
//                                            |
//                       +--------------------+
//                       |
//                       v
//   -----------------------------------------    +-------------------------------------------+
//  (               offsetAdd:64              )<--|       seconds              fraction      | flexOffset
//   -----------------------------------------    +----------32--------'---------32---------+
//                       |
//                       v
//                   timeOfDay
//                                            +-------------------------------------------+
//                                            |       fraction           subfraction     | baseRate
//                                            +---------32---------.---------32---------+
//                                                                 |
//                                                                 v
//                                             -----------------------------------------
//                                            (                baseAdd:64                )
//                                             -----------------------------------------
//                                                               ^   |
//                                                               |   v
//                                            +-------------------------------------------+
//                                            |       fraction           subfraction     | baseTimer
//                                            +---------32---------'---------32---------+
//
//
// Called with:
//   stationInfoPtr -- the station information context
void
TimerTick(StationInfo *stationInfoPtr)
{
    StationInfo *siPtr = stationInfoPtr;
    int64_t pastTimerLo;

    assert(siPtr != NULL);
    pastTimerLo = siPtr->flexTimerLo;                                                          // Saved to detect overflows
    siPtr->flexTimerLo += siPtr->flexRate;                                                     // Addition of the subfractions
    siPtr->flexTimerHi += (pastTimerLo > siPtr->flexTimerLo) ? 1 : 0;                          // Propagate carry into seconds
    siPtr->timeOfDay = EXTRACT_CORE(siPtr->flexTimerHi, siPtr->flexTimerLo) + siPtr->flexOffset;   // Compensate for offset drifts
    if (BASE_TIMER)
        siPtr->baseTimer += siPtr->baseRate;
}

// Called with:
//   stationInfoPtr -- the station information context
//   currentTime    -- the current flexTimer value
//   diffRate       -- the scaled rate difference
uint32_t
BaseTimerChange(uint64_t lastTime, uint64_t nextTime, double diffRate)
{
    uint64_t delta;
```

```
    delta = nextTime - lastTime;                        // Compute lapsed time
    delta -= delta * (diffRate / DIFF_SCALE);           // Compensate by rate difference
    return(delta);                                      // Return incremenal change
}

// FormPrecedence
// Called with:
//   systemTag      -- the 16-bit most-significant precedence subfield
//   uniqueID       -- the 64-bit unique identifier (EUI-64)
//   hopsCount       -- the hop-count distance from the grand master
//   portTag        -- the tag associated with the port
DoubleWide
PrecedenceMerge(uint16_t systemTag, uint64_t uniqueID, uint8_t hopsCount, uint16_t portTag)
{
    DoubleWide result;

    result.hi = ((uint64_t)systemTag << 32) | (uniqueID >> 32);
    result.lo = (uniqueID << 32) | ((uint32_t)hopsCount << 16) | portTag;
    return(result);
}

// Performs a comparison of 128-bit preceision unsigned values
// Called with:
//   a -- the first of two 128-bit values
//   b -- the final of two 128-bit values
int
CompareWide(DoubleWide a, DoubleWide b)
{
    if (a.hi != b.hi)
        return(a.hi > b.hi ? 1 : -1);
    if (a.lo != b.lo)
        return(b.lo > b.lo ? 1 : -1);
    return(0);
}

// Called with:
//   stationInfoPtr  -- the station information context
//   portInfoPtr     -- the port information context
void
CheckOther(StationInfo *stationInfoPtr, PortInfo *portInfoPtr)
{
    PortInfo       *piPtr = portInfoPtr;
    StationInfo    *siPtr = stationInfoPtr;

    assert(siPtr != NULL && piPtr != NULL);
    if (CompareWide(piPtr->portPrecedence, siPtr->bestPrecedence) < 0)
        siPtr->bestPrecedence = piPtr->portPrecedence;
}
```

Index

T

U