INTERNATIONAL STANDARD

ISO/IEC 13961 IEEE Std 1596

First edition
2000-07

Information technology –

Scalable Coherent Interface (SCI)

Sponsor

Microprocessor and Microcomputer Standards Subcommittee of the IEEE Computer Society

PRICE CODE XC

For price, see current catalogue

CONTENTS

Clause Page

Page

Page

Page

Page

Page

INFORMATION TECHNOLOGY –

SCALABLE COHERENT INTERFACE (SCI)

FOREWORD

1) ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

2) In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

3) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 13961 was prepared by subcommittee 26: Microprocessor systems, of ISO/IEC joint technical committee 1: Information technology.

Annexes A and B are for information only.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute.

The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Foreword to IEEE Std 1596, 1998 Edition

[This foreword is not a part of ISO/IEC 13961:2000, Information technology – Scalable Coherent Interface (SCI).]

The demand for more processing power continues to increase, and apparently has no limit. One can usefully saturate the resources of any computer so easily by merely specifying a finer mesh or higher resolution for the solution of some physical problem (hydrodynamics, for example), that engineers and scientists are desperate for enormously larger computers.

To get this kind of computing power, it seems necessary to use a large number of processors cooperatively. Because of the propagation delays introduced when signals cross chip boundaries, the fastest uniprocessor may be on one chip before long. Pipelining and similar large-mainframe tricks are already used extensively on single-chip processors. Vector processors help, but are hard to use efficiently in many applications. Multiprocessors communicating by message passing work well for some applications, but not for all. The shared-memory multiprocessor looks like the best strategy for the future, but a great deal of work will be needed to develop software to use it efficiently.

It is important to support both the shared-memory and the message-passing models efficiently (and at the same time) in order to support optimal software for a wide range of problems, especially for a system that dynamically allocates processors and perhaps changes its configuration depending on the nature of its load.

SCI started from an attempt to increase the bandwidth of a backplane bus past the limits set by backplane physics in order to meet the needs of new generations of processor chips, some of which can single-handedly saturate the fastest buses. We soon learned that we had to abandon the bus structure to achieve our goals.

Backplane performance is limited by physics (distributed capacitances and the speed of light) and by a bus's one-at-a-time nature, an inherent bottleneck. To gain performance far beyond what buses and backplanes can do, one needs better signaling techniques and the concurrent use of many signaling paths.

Rather than using bused backplane wires, SCI is based on point-to-point interconnect technology. This design approach eliminates many of the physics problems and results in much higher speeds. SCI in effect simulates a bus, providing the bus services one expects (and more) without using buses.

Committee Membership

The specification has been developed with the combined efforts of many volunteers. The following is a list of those who were members of the Working Group while the draft and final specification were compiled:

David B. Gustavson, Chair

David V. James, Vice Chair

Nagi Aboulenein
Knut Alnes
Robert H. Appleby
Kurt Baty
Amir Behroozi
David L. Black
Andre Bogaerts
Paul Borrill
Patrick Boyle
David Brearley, Jr.
Charles Brill
Haakon Bugge
Jan Buytaert
Jay Cantrell
Mike Carlton
Fred L. Chong
Graham Connolly
James R.(Bob) Davis
W. Kenneth Dawson
Stephen R. Deiss
Gary Demos
Roberto Divia
Gregg Donley
Wayne Downer
Guy Fedorkow
Peter Fenner
David Ford
Stein Gjessing
Torstein Gleditsch
James Goodman
Robert J. Greiner
Charles Grimsdale
Emil N. Hahn
Horst Halling
Craig Hansen
Marit Jenssen
Rajeev Jog
Svein Erik Johansen
Sverre Johansen
Ross Johnson
Anatol Kaganovich
Alain Kagi
Hans Karlsson*
Tom Knight
Michael J. Koster
Ernst Kristiansen
Stein Krogdahl
Ralph Lachenmaier
Branko Leskovar
Dieter Linnhofer
Robert McLaren
Mark Mellinger
Svein Moholt
Viggy Mokkarala
John Moussouris
Hans Muller
Klaus D. Muller
Ellen Munthe-Kaas
Russell Nakano
Tom Nash
Steve Nelson
Julian Olyansky
Chris Parkman
Dan Picker
Phil Ponting
Steve Quinton
Jean F. Renardy
Randy Rettberg
Morten Schanke
Gene Schramm
James L. Schroeder
Tim Scott
Donald Senzig
Gurindar Sohi
Robert K. Southard
Joanne Spiller
Paul Sweazey
Lorne Temes
Manu Thapar
John Theus
Mike van Brunt
Phil Vukovic
Anthony Waitz
Richard Walker
Steve Ward
Carl Warren*
Steinar Wenaas
Mike Wenzel
Richard J. Westmore
Wilson Whitehead
Hans Wiggers
Mark Williams
Philip Woest
S. Y. Wong
Ken Wratten
Chu-Sun Yen

*deceased

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

M. R. Aaron
Scott Akers
Ray S. Alderman
John Allen
Knut Alnes
Richard P. Ames
Bjorn Bakka
David M. Barnum
Kurt Baty
Harrison A. Beasley
Amir Behroozi
Janos Biri
David Black
William P. Blase
Andre Bogaerts
W. C. Brantley
David Brearley, Jr.
Haakon Bugge
Kim Clohessy
Graham Connolly
Jonathan C. Crowell
W. Kenneth Dawson
Stephen Deiss
Dante Del Corso
Stephen L. Diamond
Jean-Jacques Dumont
William P. Evertz
Guy Federkow
Timothy R. Feldman
Peter Fenner
Gordon Force
Stein Gjessing
Andy Glew
Patrick Gonia
James Goodman
Charles Grimsdale
David Hawley
Phil Huelson
Zoltan R. Hunor
Edgar Jacques
David V. James
Kenneth Jansen
Rajeev Jog
Sverre Johansen
Ross Johnson
Jack R. Johnson
Anatol Kaganovich
Christopher Koehle
Michael J. Koster
Ernst H. Kristiansen
Ralph Lachenmaier
Glen Langdon, Jr.
Gerry Laws
Minsuk Lee
Branko Leskovar
Anthony G. Lubowe
Svein Moholt
James M. Moidel
James D. Mooney
Klaus-Dieter Mueller
Ellen Munthe-Kaas
Cuong Nguyen
J. D. Nicoud
Dan O Connor
Mira Pauker
Donald Pavlovich
Thomas Pittman
Steve Quinton
Richard Rawson
Steven Ray
Randy Rettburg
Hans Roosli
Paul Rosenberg
Carl Schmiedekamp
James L. Schroeder
Don Senzig
Philip Shutt
Michael R. Sitzer
Gurindar Sohi
Robert K. Southard
Joanne Spiller
David Stevenson
Robert Stewart
Paul Sweazey
Daniel Tabak
Daniel Tarrant
Lorne Temes
Manu Thapar
Michael G. Thompson
Chris Thomson
Joseph P. Trainor
Robert Tripi
Robert J. Voigt
Phil Vukovic
Yoshiaki Wakimura
Richard Walker
Eike Waltz
Carl Warren*
Richard J. Westmore
Hans A. Wiggers
Mark Williams
Andrew Wilson
Joel Witt
Ken Wratten
David L. Wright
Chu Yen
Oren Yuen
Janusz Zalewski

*deceased

When the IEEE Standards Board approved this standard on 19 March 1992, it had the following membership:

Marco W. Migliaro, Chair **Donald C. Loughry**, Vice Chair

Andrew G. Salem, Secretary

Dennis Bodson	Donald N. Heirman	T. Don Michael*
Paul L. Borrill	Ben C. Johnson	John L. Rankine
Clyde Camp	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster*	Joseph Koepfinger*	Gary S. Robinson
David F. Franklin	Irving Kolodny	Martin V. Schneider
Ramiro Garcia	D. N. Jim Logothetis	Terrance R. Whittemore
Thomas L. Hannan	Lawrence V. McCall	Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Fernando Aldana

Satish K. Aggarwal

James Beall

Richard B. Engelman

Stanley Warshaw

This standard was was approved by the American National Standards Institute on 23 October 1992. It was reaffirmed by IEEE in 1998.

– Blank page –

INFORMATION TECHNOLOGY –

SCALABLE COHERENT INTERFACE (SCI)

1 Introduction

1.1 Document structure

This International Standard describes a communication protocol that provides the services required of a modern computer bus, but at far higher performance levels than any bus could attain. Packet protocols on unidirectional point-to-point transmission links emulate a sophisticated bus without incurring the inherent bus physics or bus contention problems.

This International Standard is partitioned into clauses that serve several distinct purposes:

Clause 1: Introduction provides background for understanding the Scalable Coherent Interface (SCI) protocols, and may be skipped by those already familiar with these concepts. The descriptions in this clause are somewhat simplified, and should not be considered part of the SCI specification.

Clause 2: References, glossary and notation defines the terminology used within this standard and lists references that are required for implementing the standard.

Clause 3: Logical protocols and formats defines the packets and protocols that implement transactions (like reads and writes) between SCI nodes. This clause uses text and figures as introductory material, to establish a frame of reference for the formal specification.

Clause 4: Cache-coherence protocols provides background information for understanding the protocols used by two or more SCI nodes to maintain coherence between cached copies of shared data. The coherence protocols contain many options. This clause describes the minimal subset of these protocols, a typical set of options that are likely to be implemented, and also the full set of protocols.

Clause 5: C-code structure explains the structure of the C code that defines the logical (packet symbol processing) and cache-coherence protocols. The precise specifications of the logical-level packet protocols and the cache-coherence protocols, which involve a large number of state-transition details, are expressed in C code because it is difficult to state them unambiguously in English, and so that they can be tested thoroughly under simulation.

Clause 6: Physical layers defines a mechanical package and several physical links that may be used to implement the logical protocols. This clause uses text and figures to specify the mechanical and electrical characteristics of several physical links.

Annexes A and B: These annexes describe other system-related concepts that have influenced the design of this standard. These may be useful for understanding the rationale behind some of the SCI design decisions.

Bibliography provides a variety of references that may be useful for understanding the terminology, notation, or concepts discussed within this standard.

C code: The C code is published as a text file on an IBM-format diskette. This was done for the convenience both of the casual reader of this standard, who will not delve into the details of the C code, and also of the serious user, who will wish to understand the C code thoroughly, executing it on a computer. Though the C code takes precedence over this International Standard in case of inconsistency, this International Standard provides considerable explanation and illustration to help develop an intuitive understanding that will make the C code more comprehensible.

1.2 SCI overview

1.2.1 Scope and directions

Purpose: To define an interface standard for very high-performance multiprocessor systems that supports a coherent shared-memory model scalable to systems with up to 64 K nodes. This standard is to facilitate assembly of processor, memory, I/O, and bus adaptor cards from multiple vendors into massively parallel systems with throughputs ranging up to more than 10^{12} operations per second.

Scope: This standard will encompass two levels of interface, defining operation over distances less than 10 m. The *physical* layer will specify electrical, mechanical, and thermal characteristics of connectors and cards. The *logical* level will describe the address space, data transfer protocols, cache coherence mechanisms, synchronization primitives, *control* and status registers, and initialization and error recovery facilities.

The preceding statements were those submitted to and approved by the IEEE Standards Board as the definition of the SCI project. These goals have been met and exceeded: support for message-passing was added, and the operating distance is not limited to 10 m. (The intent of that limitation was to make clear that this is not yet-another Local Area Network.)

The real distinction between SCI and a network has more to do with the memory-access-based model SCI uses and the distributed cache-coherence model.

The practical operating distance depends more on the throughput and performance needed than on any absolute limit built into the specification. Very long links would yield unacceptable performance for many users (but perhaps not all).

In particular, the fibre-optic physical layer can extend the SCI paradigm over distances long enough to link a computer to its I/O devices, or to link several nearby processors. No arbitrary length limit would be appropriate, but practical considerations including the throughput requirements and the cost of transmitters and receivers will set the lengths that people consider useful.

A very-high-priority goal was that SCI be cost-effective for small systems as well as for the massively parallel ones mentioned in the purpose statement above. SCI's low pin count and simple ring implementation make medium-performance, few-processor systems easier to build with SCI than with bused backplane systems; a two-layer backplane should be sufficient, and three layers should be enough to support the optional geographical addressing mechanism. The SCI interface, complete with transceivers, fits into a single IC package that includes much of the logic needed to support the cache-coherence protocols. This economy for small systems leads to the expectation that SCI processor boards will be built in high volume, making them inexpensive enough to be assembled in large numbers for building supercomputers at low cost.

SCI also simplifies the construction of reliable systems. SCI Type 1 modules are well protected against electrostatic discharge and electromagnetic interference, and can be safely inserted while the remainder of the system remains powered. SCI supports live insertion and withdrawal by using a single supply voltage (with on-board conversion as needed) and staggered pin lengths in the connector to guarantee safe sequencing. Note, however, that system software plays an important role in live insertion or removal of a module because the resources provided by that module have to be allocated and deallocated appropriately.

In systems where several modules share a ringlet, the removal of one module interrupts all communication via that ringlet, so the resources on those modules also have to be deallocated. A similar situation arises in any system that may have multiple processors resident on one field-replaceable board: all have to be deallocated when any one is replaced. The system software for handling the deallocation and reallocation of these resources is outside SCI's scope.

Although SCI does not provide fault tolerance directly in its low-level protocols, it does provide the support needed for implementing fault-tolerant operation in software. With this recovery software, the SCI coherence protocols are robust and can recover from an arbitrary number of detected transmission failures (packets that are lost or corrupted).

The SCI paradigm removes the limits that bus structures place on throughput, but its latency is of course limited by the speed of signal propagation (less than the speed of light). Ever-increasing throughput can be expected as technology improves, but the organization of hardware and software will have to take into account the relatively constant latency (delay between request and response), which is proportional to the physical size of the system.

The last generation of buses approached the ultimate limits of performance, leading to the concept of an ultimate standard. However, the initially defined SCI physical layers are likely just the first of a series of implementations having higher or lower performance levels. The 1 Gbyte/s link speed specified for the initial ECL/copper-backplane implementation was chosen based on a combination of marketing and engineering considerations. From a marketing point of view, it was necessary to define a territory that did not disturb the markets for present 32-bit standards or present networks, and from an engineering point of view this link speed was near the edge of what available signalling technology and integrated circuit technology could support.

New technologies, such as better cables, connectors, transceivers; IC packages with more pins or higher power-dissipation capabilities; or faster ICs, could make it practical or desirable to implement SCI on new physical-layer standards. Such standards, with different link widths or bit rates, will be developed from time to time. However, packet formats and higher level coherence protocols will be the same across all these physical implementations. That should make the problem of interfacing one SCI system to another relatively simple – SCI already includes the necessary mechanisms to cope easily with speed differences.

1.2.2 The SCI approach

The objective of SCI was to define an interconnect system that *scales* well as the number of attached processors increases, that provides a *coherent* memory system, and that defines a simple *interface* between modules.

SCI developers initially hoped to make a better backplane bus to meet these goals, but soon realized no bus could do the job. Bus speeds are limited by the distance a signal must travel and the propagation delay across a backplane. In asynchronous buses, the limit is the time needed for a handshake signal to propagate from the source to the target and for a response to return to the source. In synchronous buses, it is the time difference between clock and data signals that originate in different places.

Transmission lines in a backplane bus are affected by reflections caused by multiple connectors, as well as by variations in loading as the number of inserted modules changes. This makes a backplane bus an imperfect transmission line at best.

Furthermore, a backplane bus can only handle one data transmission at a time and therefore becomes a bottleneck in multiprocessor systems. Although bridges can be used to extend the bus concept to a multiple-bus topology, these bridges are expected to be more costly and less efficient than SCI switches. Support for an efficient switch greatly influenced the design of the SCI protocols.

SCI solves these problems by defining a radically different interconnect system. SCI defines an interface standard that enables a system integrator to connect boards using many different interconnect configurations. These configurations may range from simple rings to complex multistage switching networks. SCI modules still may plug into a backplane – it holds the connectors in place; it is just not wired as a bus.

SCI uses point-to-point unidirectional communication between neighbouring nodes, greatly reducing the nonideal-transmission-line problems. The bandwidth of the point-to-point link depends on the transmission medium. A Type 18 DE 500 link is 2 bytes wide and data are transferred at 1 Gbyte/s, using differential ECL signalling and both edges of a 250 MHz clock.

The clock rate can be much higher for point-to-point links than for buses. For a given data rate this makes it possible to use faster clocking to reduce the link width. This reduces the pin count for bus interface logic, so that the entire bus interface can be integrated on a single chip. Thus, timing skews can be tightly specified, since components are inherently well matched in a single-chip design. A large number of requests can be outstanding at the same time, making SCI well suited for high-performance multiprocessor systems. SCI allows up to 64 K nodes to be connected in a single system. Since each node could itself be a multiprocessor, the SCI addressing mechanism should be sufficient to support the next generation of massively parallel computer systems.

Cache coherence is an important part of the proposed standard. Switching networks cannot easily provide reliable broadcast or eavesdrop capabilities. Hence the SCI coherence protocols are based on single-responder directed bus transactions and distributed directories, where processors sharing cache lines are linked together by pointers. Broadcasts are generally software, not hardware, operations, though the protocols do support some (noncoherent) broadcast transactions that may be useful in certain applications.

1.2.3 System configurations

An SCI node relies on feedback arriving on its input link to *control* its behaviour on its output link. Thus there must always be a ring-like connection, with the output of one node providing the input to another. Implementations of this structure range from a small ring connecting two nodes (one of which might be the port to a fast switch) to a large ring consisting of many nodes. The term ringlet is often used to imply a ring that has a relatively small number of nodes, up to perhaps half a dozen. Few applications will perform well with large rings because each node sees traffic generated by all the other nodes on the ring; for some I/O applications, however, large rings may be appropriate.

One node on each ring (called the scrubber) is assigned certain housekeeping tasks, such as initializing the ring to the point that each node is addressable, maintaining certain timers, and discarding damaged packets so they don't circulate indefinitely.

For performance, fault tolerance or other reasons many systems will require more than one ringlet. Agents, which consist of two or more SCI node interfaces to different ringlets, with appropriate routing mechanisms, are used to allow nodes on different ringlets to communicate with one another in a transparent way.

One can build useful switch fabrics consisting of many ringlets with a few processor nodes and agents on each. Or one can use more traditional switch mechanisms that have SCI interfaces at their extremities but transparently use whatever internal data transfer and switching techniques they prefer.

1.2.4 Initial physical models

The logical portion of the SCI specification defines the format and function of fields in packets that are sent from one SCI node to another over any one of several different physical link layers.

SCI links continually transmit symbols that contain 16 data bits plus packet-delimiter and clock information. The clock provides a precise timing reference that the receiver uses for extracting data from the incoming signals. A symbol is either part of a packet (a contiguous sequence of symbols marked by the packet delimiter) or an idle symbol (transmitted during the interval between packets to maintain synchronism between the link and the receiver). On a backplane, where signal wires are relatively inexpensive, an entire symbol may be sent each clock period. On longer-distance interconnects, where signal wires are relatively expensive, the symbols may be sent one bit at a time.

The notation used by SCI for names of link types is:

Type <number of signals> <kind of signals> <bit rate per signal in Mb/s>.

Type 18 DE 500 signals support high-performance boards plugged into a system backplane or cable links connecting proprietary physical packages. Symbols are sent bit-parallel, using differential drivers and receivers. High transmission rates can be achieved by having all signal drivers and receivers in the same integrated circuit package, which also contains high-speed queues, as illustrated in figure 1.

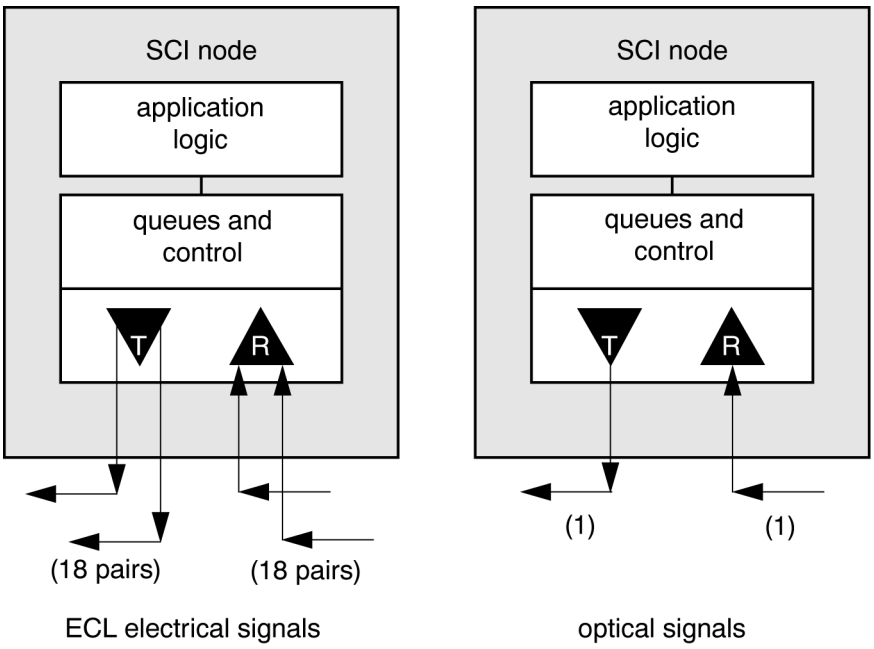


Figure 1 – Physical-layer alternatives

The initial interface chips were VLSI chips that included the transmitters, receivers, high-speed queues, and most of the cache-coherence protocols. Subsequent implementations generally removed the coherence logic, leaving that to the province of the system's memory controller. Several implementations initially ran at reduced speed for compatibility with standard CMOS processes. Some included the SCI interface as just a small part of a system chip that included processor and other application-specific logic. The complexity of the protocol is very low, with some implementors reporting that they used less than 25 k gates,

much less than some common low-end interface products. The inherent power dissipation of the SCI interface is less than that of interfaces to bused backplanes, since the differential signal levels are smaller (less than 1 V), there are fewer signals, and transmission impedances are significantly higher.

The Fiber-Optic Physical Layer Type 1-FO-1250 is intended to support longer-distance local communications (tens to thousands of meters). The fiber versions of SCI could be used to connect back-end peripherals to the central system, or could provide high-bandwidth communication between workstations and servers in a local computing environment. Packets are sent in a bit-serial fashion, as illustrated in figure 1.

Low-cost LEDs can support communication bandwidths of less than 1 Gb/s over short fiber hops. Higher-cost single-mode lasers and fibers are required for higher bandwidth communications over longer distances. Many applications will find it attractive to use coaxial cable instead of fiber for short hops, avoiding the optical/electrical conversion costs.

Fiber-optic interfaces are expected to consist of high-speed bipolar front-ends that convert between a high-bandwidth serial bit-stream and a lower-bandwidth symbol-stream. Lower-speed back-end circuits could be implemented in less expensive CMOS technologies.

New link standards will be defined from time to time to take advantage of advances in technology or to accommodate the needs of particular markets.

1.2.5 SCI node model

An SCI node needs to be able to transmit packets while concurrently accepting other packets addressed to itself and passing packets addressed to other nodes. Because an input packet might arrive while the node is transmitting an internally generated packet, FIFO storage is provided to hold the symbols received while the packet is being sent. Since a node transmits only when its bypass FIFO is empty, the minimum bypass FIFO size is determined by the longest packet that the node originates. Idle symbols received between packets provide an opportunity to empty the bypass FIFO in preparation for the next transmission.

Input and output FIFOs are needed in order to match node processing rates to the higher link-transfer rate. Since there is no facility for delaying the transmissions of symbols within a packet, each node ensures that all symbols within one packet are available for transmission at full link speed. Similarly the node is able to receive a packet at full speed. Since node application logic is not expected to match the SCI link speeds, FIFO storage is needed for both transmit and receive functions, as illustrated in figure 2.

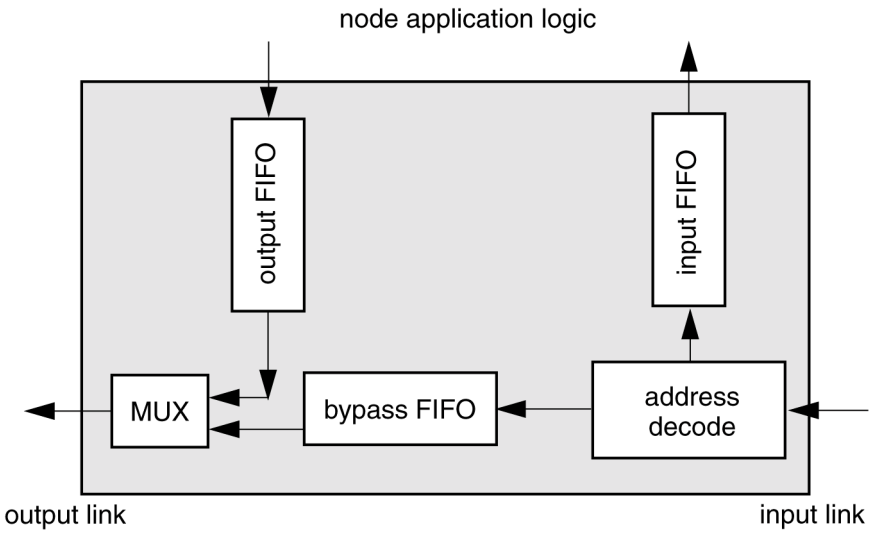


Figure 2 – SCI node model

1.2.6 Architectural parameters

The SCI system is generally considered to have a 64-bit architecture, because of its address size (16 bits for node selection plus 48 bits for use within each node). The data width is less constrained, however. SCI usually sends data in multiples of 16 bytes, and the most significant size assumption is the 64-byte coherence-line size.

SCI is described in terms of a distributed shared-memory model with cache coherence, because that is the most complex service SCI provides. However, SCI also provides message-passing mechanisms and noncoherent transactions for those who need or prefer them. All of these transactions can be dynamically mixed in one system as desired.

1.2.7 A common CSR architecture

Control and status registers (CSRs) are an important part of the proposed standard. The CSR definitions are essential for all initialization and exception handling. A few of the CSRs are SCI-specific, but the majority of the necessary definitions are provided by the CSR Architecture standard (IEEE Std 1212-1991)[1].

SCI uses the 64-bit-fixed addressing model defined by the CSR Architecture. The 64-bit address space is divided into subspaces, one for each of 64 K equal-sized nodes, as illustrated in figure 3. When compared to other address-extension schemes, the fixed address-field partitioning dramatically simplifies packet routing; however, it complicates software's memory-mapping model, since the memory addresses provided by different memory nodes can no longer be contiguous.

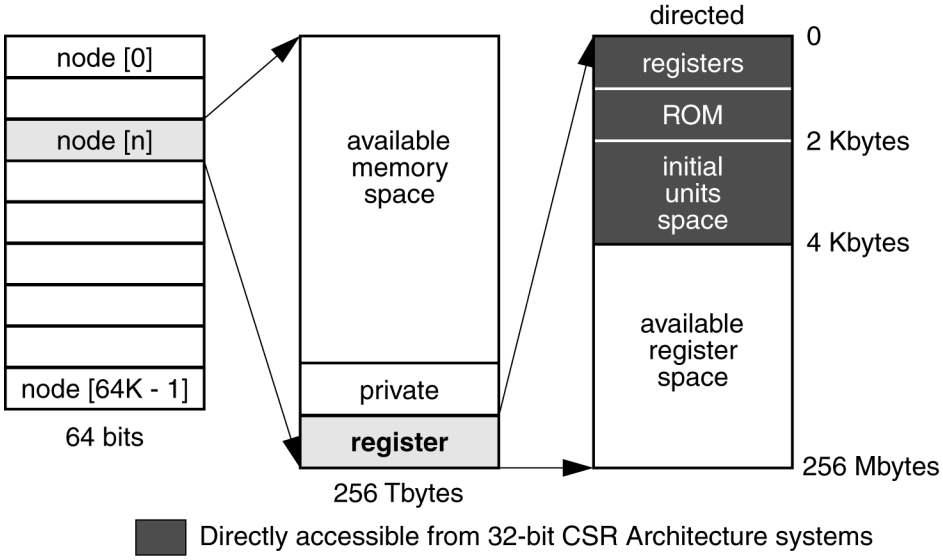


Figure 3 – 64-bit-fixed addressing

The upper 16 bits of the address specify the responder *nodeId* value; the remaining 48 bits specify the address-offset in the addressed node. The highest 256 Mbytes of each node's 256 Tbytes contain the CSR registers as defined in the CSR Architecture. Since SCI's broadcast transactions are block moves with no responses, only the directed (i.e., not broadcast) CSR registers are supported.

[1] Information on references can be found in 2.1.

Only a portion of the 64-bit address space is accessible from 32-bit systems bridged to SCI. The initial 4 Kbytes of each node's directed CSR address space as defined by the CSR Architecture could be directly mapped into 32-bit addresses, using the 10 bus-address and 6 module-address bits to form an SCI node address. In addition, a small portion (3.5 Gbytes) of the memory address space in node[0] could be directly mapped from the 32-bit memory address space. However, the address-map conventions used by bridges to other buses are beyond the scope of the SCI standard.

1.2.8 Structure of the specification

This specification covers a great deal of new territory, and has required some new approaches for presenting the material in a way that is precise and not easily misunderstood. Much of this International Standard is tutorial and explanatory in nature, to develop the way of thinking and the level of understanding needed to properly interpret and use the precise specification. The most important part of this standard is the packet protocol. Packet transmission is in turn implemented on some physical signalling layer, and that in turn may be incorporated into a standard mechanical package.

Except for the packet formats and physical implementation specifications, such as module, connector, power and signal levels, this specification is expressed in the C computer language. English text should be considered explanatory, and C listings, the definitive specification. Though C is known to have some ambiguities (such as the order of evaluation of parts of certain expressions), they are easily avoided in this application. In addition to making this specification unambiguous, another significant advantage of the C specification is that it is executable so that it can be incorporated into other software to test the operation of the specification under simulation or to test a real implementation of the specification.

1.3 Interconnect topologies

1.3.1 Bridged systems

To ensure the early availability of the wide range of I/O interface boards that any system needs in order to become accepted and useful, the SCI standard was heavily influenced by the need to bridge to other system buses. Conversions between SCI and other bus standards are performed by bus bridges, as illustrated in figure 4.

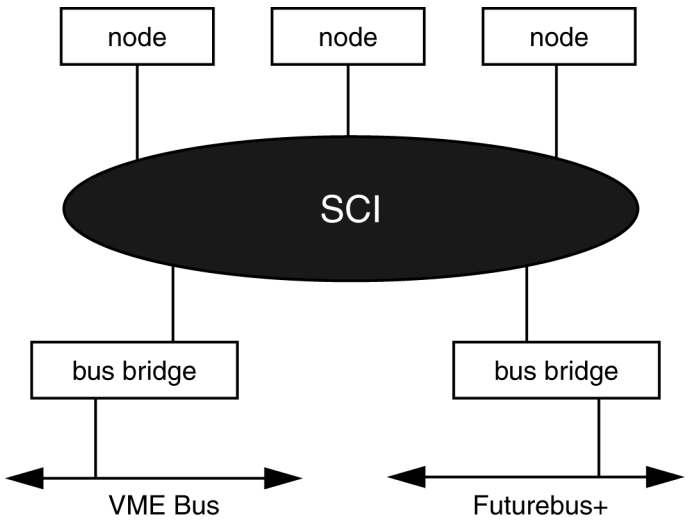


Figure 4 – Bridged systems

Indivisible uncached lock transactions (such as swap, compare&swap, fetch&add) are supported, but not implemented as indivisible read-modify-write transaction sequences. Since indivisible sequences are hard to implement in large switches, indivisible lock operations are performed at the responder upon request. A standard set of lock transaction subcommands is defined in order to communicate the intent of the requester to the hardware at the responder that will carry out the operation. Bus bridges may translate these lock transactions into indivisible sequences where appropriate.

Most remote DMA adapters generate uncached bus transactions; bus bridges can convert these into coherent transaction sequences. If the remote bus supports coherent transfers, the bus bridge can also convert between coherence protocols. Futurebus+R (see [B11][2], [B3], and [B4]) and SCI have the same coherence line size, which simplifies that conversion process.

1.3.2 Scalable systems

SCI protocols are scalable, which means that they are efficient and cost-effective for uses ranging from low-end desktop computers to high-end massively parallel processing (MPP) systems. One future vision of a massively parallel processor consists of large numbers of single-board computers connected through a high-performance switch.

To make this vision a reality, SCI is designed to be used in simple passive backplane configurations, or as the basis for constructing switches, or as the interface between multiprocessor boards and vendor-dependent proprietary high-performance interconnects. Such configurations are introduced in the following clauses.

1.3.3 Interconnected systems

SCI is based on packets sent from one node to another over unidirectional links. This specification defines a way to send these packets 16 bits at a time over short distances (on the order of meters), and one bit at a time over longer distances (on the order of a kilometer).

The bit-serial version of SCI makes use of fiber-optic links or short coaxial cables. It might be used as a high-performance peripheral bus connecting storage servers to back-end processors, or as a local-area bus connecting distributed workstations and file servers.

1.3.4 Backplane rings

The simplest SCI interconnect is a single ring. Larger configurations could consist of multiple rings connected through bridges. The highest performance configurations would probably be based on switching interconnects, like the butterfly switch. From a node interface perspective, the interface to a simple ring and to a complex switch is the same (one input link and one output link). The lowest-cost SCI configuration makes use of a passive backplane; the nodes are electrically connected as a ring. The ring connection could join adjacent slots (which results in one long link to connect the ends) or alternate slots (to shorten the maximum link length). On a sequential ring, a node's physical and electrical neighbours are the same, as illustrated in figure 5.

---

[2] The numbers in brackets preceded by the letter B correspond to those of the bibliography.

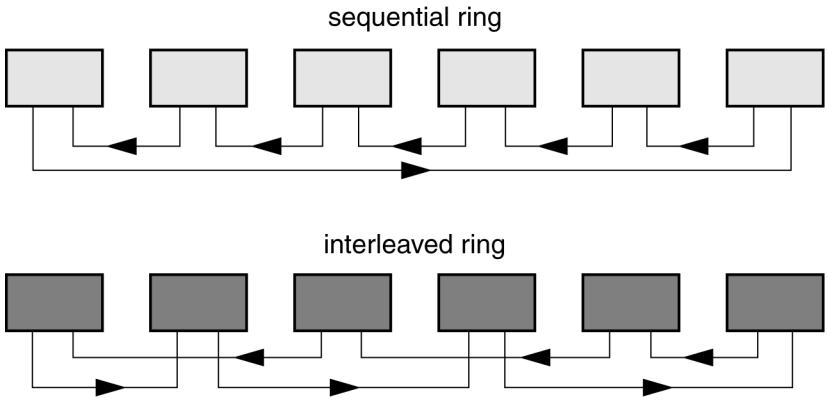


Figure 5 – Backplane rings

On an interleaved ring a node's physical and electrical neighbours differ. Even-numbered boards attach to one ring direction; odd-numbered boards to the other, thus minimizing the maximum distance between nodes. To support partially populated topologies, implementations are expected to use pass-through cards in empty slots, to provide jumper-card pairs for bypassing empty slots, or to use self-bridging connectors (that short inputs to outputs when the slot is empty).

There is also provision for doubled (or even trebled) SCI connections to a module, making bridges and redundant fault-tolerant systems possible. With multiple rings arranged so that at least one ring skips any given slot, one can maintain partial system operation even when one module is removed – the rings connected to that slot are broken, but the other rings can connect the remaining modules via bridges.

For some applications it may be desirable to use SCI signals on cable links to connect devices that do not fit conveniently into the standard SCI modules.

1.3.5 Interconnected rings

Since the SCI protocols have been designed to minimize the transit time for packets that pass through a switch from one ringlet to another, they can be readily applied to multiple-ring topologies. For example, a grid of processors can be easily and efficiently interconnected by horizontal and vertical ringlets, as illustrated in figure 6. In this illustration, each processor has two SCI interfaces; one interface attaches to the horizontal ringlet and the other attaches to the vertical ringlet.

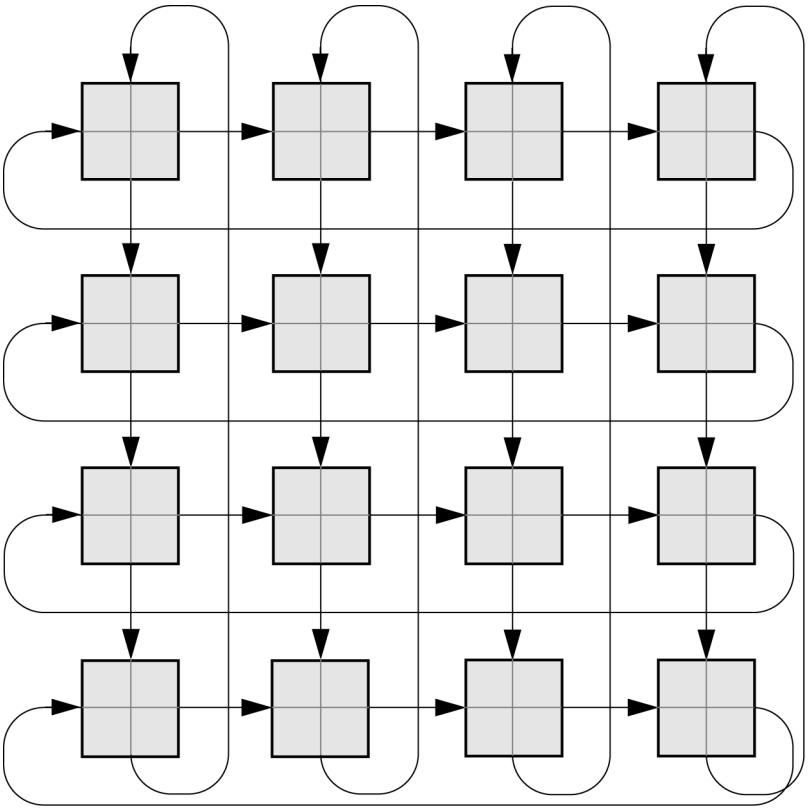


Figure 6 – Interconnected rings

Additional dimensions (for example, a 3-D cube) can be supported by increasing the number of ports on each processor (one for each dimension). Such structures are known as *k*-ary n-cubes, where *k* is the number of nodes on each ringlet and *n* is the number of dimensions. For a fixed number of processors, the number *k* can be increased to reduce the cost of the switch elements or may be decreased to reduce the contention on each ringlet.

1.3.6 Rectangular grid interconnects

SCI can also be used as an interconnect to form grids of processors. Nodes with four SCI interfaces can form a bidirectional interconnect, where different ringlets connect each node to its adjacent neighbours. Nodes with two SCI interfaces can form a unidirectional interconnect, where the ringlets form squares of nodes, as illustrated in figure 7.

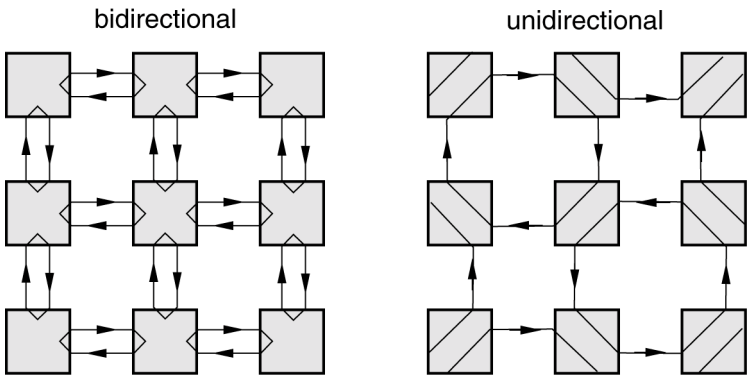


Figure 7 – 2-D processor grids

1.3.7 Butterfly switches

SCI can also be used to implement butterfly-like interconnects. Before SCI, these NlogN switches were generally implemented with a unidirectional data transfer and a reverse flow-*control* signal. The switch is wrapped around, so one processor node appears to connect to both sides of the switch.

SCI ringlets can be used to implement such switches by partitioning the transmission paths into separate ringlets, horizontal and diagonal, as shown in figure 8.

The dotted-line ringlet-completion path in this figure is an implied node-internal data path that connects one access port to another.

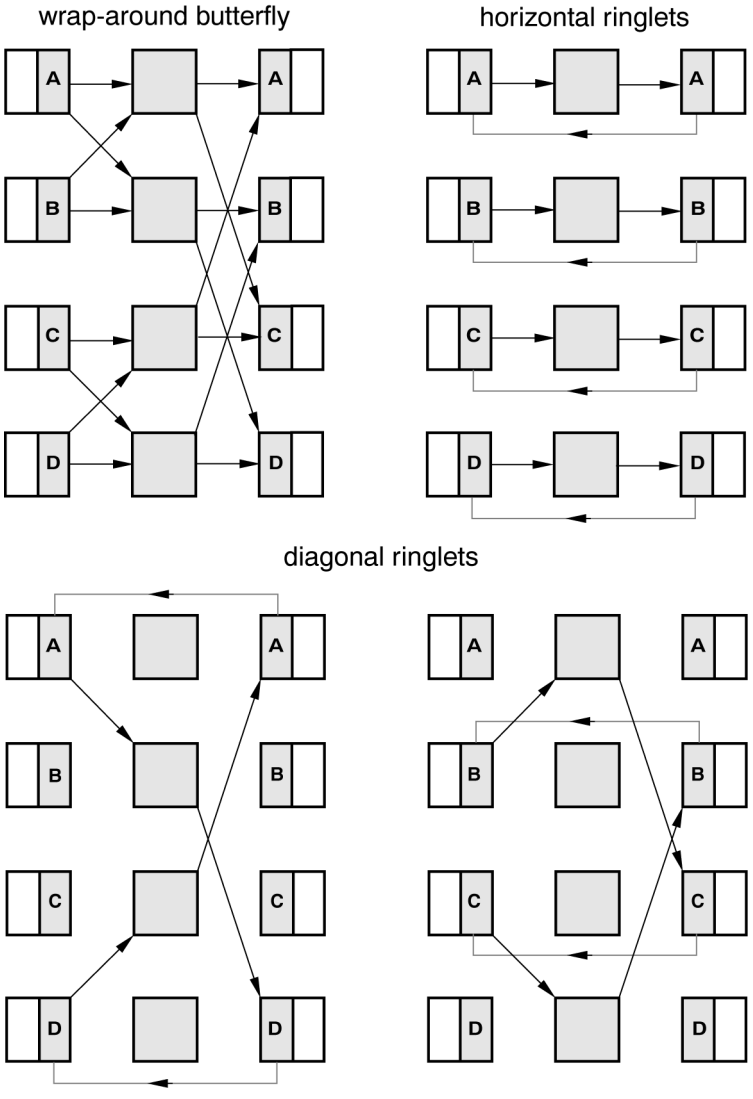


Figure 8 – Butterfly ringlets

1.3.8 Vendor-dependent switches

A switch may internally implement specialized vendor-dependent protocols to route SCI packets. Each node is attached to the switch by an SCI ringlet obeying normal SCI protocols, as shown in figure 9. SCI provides the interface between the nodes and the queues in the switch interfaces. To avoid deadlock, two queues are provided in each direction, one for requests and one for responses. This prevents requests from using up all the queue space and thus blocking completion of their responses. This strategy is followed throughout SCI.

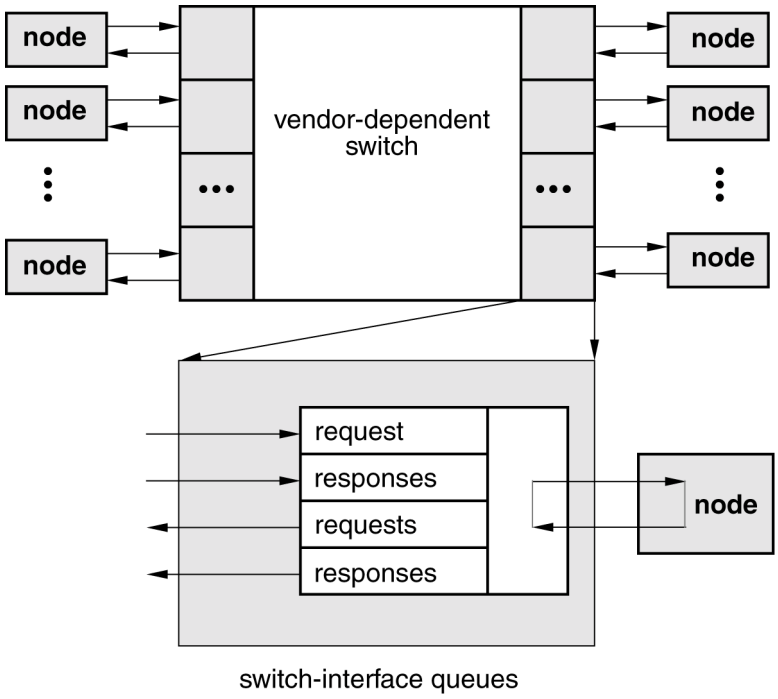


Figure 9 – Switch interface

1.4 Transactions

Transactions are performed by sending packets from a queue in one node to a queue in another. A packet consists of an unbroken sequence of 16-bit s*ymbols*. It contains address, command, and status information in a header, optional data in one of several allowed lengths, and a check symbol. When a packet arrives at a node to which it is not addressed, it is passed on to the next node with no change except possibly to the flow control information in the header. When a packet arrives at its destination address it is stored by that node for processing, and is not passed on to the next node.

An SCI packet originates at a source and is addressed to a single target. In going from source to target the packet may possibly pass through intermediate nodes or agents (explained later). Such single-requestor/single-responder protocols are highly scalable.

Transactions are initiated by a *requester* and completed by a *responder*. Transactions consist of two subactions. During the *request subaction* address and command are transferred from requester to responder. The *response subaction* returns completion status from responder to requester. Depending on the transaction command, data are transferred in the *request subaction* (writes), the *response subaction* (reads), or both subactions (locks).

A *subaction* consists of two packet transmissions, one sent on the output link and the other received on the input link. A subaction is initiated by a source, which generates a *send* packet. The subaction is completed by the destination, which returns an *echo* packet. Hence a typical transaction involves the transfer of four packets, as illustrated in figure 10.

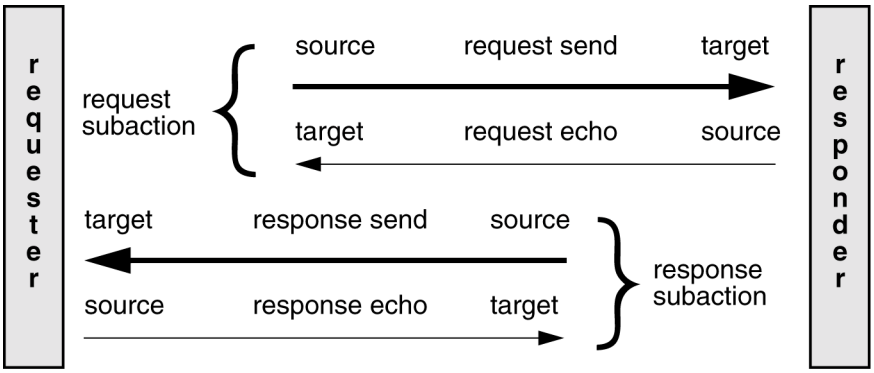


Figure 10 – Subactions

1.4.1 Packet formats

The first symbol of the header, *targetId*, contains the final target's nodeId, and is sufficient for a node to quickly recognize packets addressed to it. During the passage of a packet through an SCI system, intermediate agents look at the *targetId* symbol (and possibly other symbols) to route the packet, and intermediate nodes look at it to determine whether they should accept the packet. This and other packet symbols are shown, in simplified form, in figure 11.

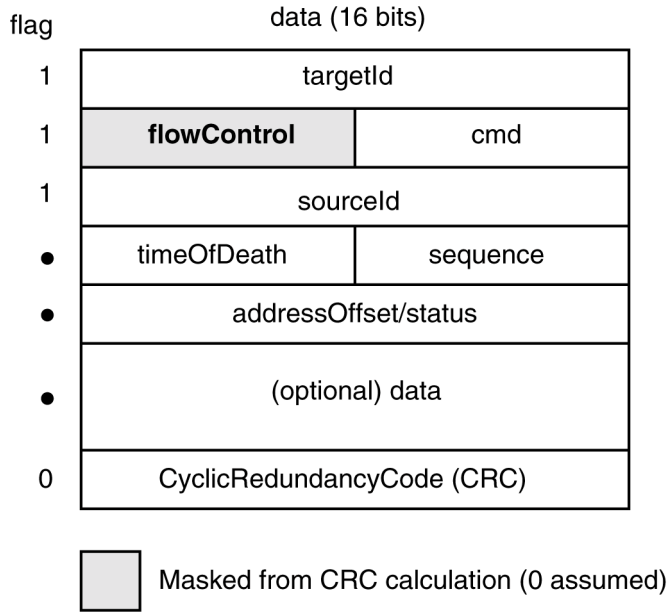


Figure 11 – Send-packet format, simplified

The second symbol, *command*, provides flow-control information and the transaction command field. The flow-control field, which contains localized flow-control information, may be changed many times before a packet reaches its destination. This information is excluded from the CRC calculation, so the CRC remains unchanged (and error coverage is not compromised) as the packet is routed toward its final destination.

The command field specifies the type of packet (read00, readsb, writesb, etc.). In a request-send packet, the command specifies the action to be performed by the responder. In a response-send packet, the command specifies the amount of data returned. In an echo packet, the command field indicates whether the corresponding send packet was accepted.

The third symbol contains the *sourceId*, allowing the target to identify the originator of the packet. All packets include a 6-bit sequence number (which distinguishes between multiple currently pending transactions from one requester). The location of this field differs for send and echo packets.

Appended to each packet is a 16-bit cyclic redundancy code (CRC), that is generated when the packet is created by the source, is optionally checked by agents, and is checked before the packet is processed by the target. The CRC is generated based on a parallelized version of the 16-bit ITU-T CRC.

Note that a flag bit is associated with each symbol. A zero-to-one transition of the flag bit is used to identify the first symbol of a packet. The one-to-zero transition of the flag bit occurs near the end of the packet (1 or 4 symbols before the packet's end, for echo and send packets respectively). A loss of link synchronization will generally cause improper flag patterns and CRCs.

Other information that is included in some packet types includes the following:

1) *Time of death*. The *timeOfDeath* is a time-stamp field in send packets, that specifies the time at which the packet should be discarded. This simplifies error recovery protocols by bounding the lifetime of all outstanding packets.
2) *Address offset*. The 48-bit *addressOffset* field in request-send packets transfers an address offset to the responder. Although this is often used to select specific memory or register locations, the interpretation of (most of) this field is responder-architecture dependent.
3) *Status.* The 48-bit status field in response-send packets returns the transaction status from the responder to the requester.
4) *Extended header.* A packet may include an additional 16 bytes of header. The presence of the extended header is signalled by a bit in the command field. A small portion (four bytes) of the extended header is defined for certain cache-coherence transactions. The remainder of the extended header is reserved for definition by future extensions to the SCI standard.
5) *Data bytes*. The data section contains a data block of 0, 16, or 64 bytes. SCI systems may optionally support 256-byte transfers for higher efficiency.

1.4.2 Input and output queues

Queues are used to hold SCI packets that cannot be immediately forwarded or processed at their intermediate or final destinations. The simplest responder node has two queues. The input queue holds request packets that have been stripped from the input link but have not yet been processed. The output queue holds response packets to be sent on the output link when bandwidth is available. These queues are illustrated in figure 12.

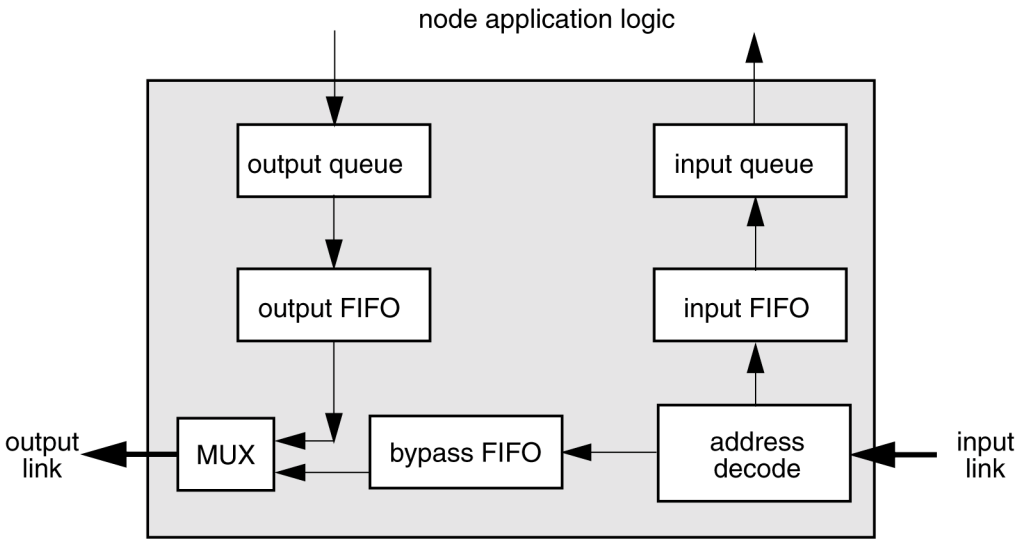


Figure 12 – Responder queues

Packets in the output queue are sent when the bypass FIFO is empty and the node's flow-control mechanism (see 3.6) permits it. Another packet (or packets) may arrive on the input link while an output packet is being sent. If they are not addressed to this node, the bypass FIFO holds these incoming packets for delayed transmission after the output queue packet has been sent. Thus, the bypass FIFO needs to be as large as the longest packet sent through the output queue.

While the bypass FIFO is nonempty, symbols arriving between packets (called idle symbols) are merged and their contents are saved for delayed retransmission. Thus, most idle symbols provide an opportunity to decrease by one the number of saved symbols in the bypass FIFO. When the bypass FIFO is empty, and the flow-control mechanism is re-enabled, another packet may be sent from the output queue.

When a send packet is emitted, the packet is saved in the output queue until a confirming echo packet is received. The addressed target node strips the send packet from the interconnect and creates an echo packet, which is returned to the source. There are two types of echo packet. If the target node can save the send packet, a done echo is returned. If the target node lacks queue space, it discards the send packet and returns a retry echo.

When a done echo is returned to the source the corresponding send packet is discarded (i.e., its queue space is freed for reuse). When a retry echo is returned to the source the corresponding send packet is resent. Resending after a retry echo packet is often called busy-retry, and the discarded send packet is said to have been busied by the destination node.

Note that send packets can be discarded by targets that have no space to save them, but returned echo packets are always accepted. Sources need to allocate space for echo packets before transmitting send packets.

1.4.3 Request and response queues

Many SCI nodes have requester as well as responder capabilities. To avoid system deadlocks on these full-duplex nodes, request and response subactions are processed through separate queues. Thus, each node logically has a pair of request and response subaction queues, as shown in figure 13.

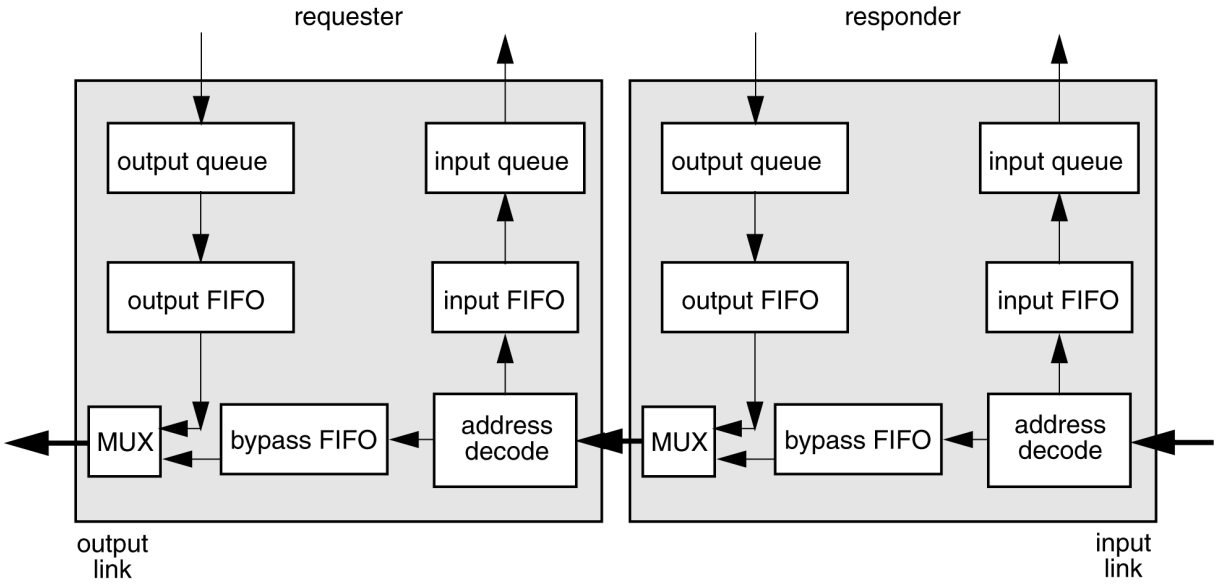


Figure 13 – Logical requester/responder queues

For performance and cost reasons, a single bypass FIFO is desirable. With suitable allocation protocols, the two bypass FIFOs can be merged into one, as illustrated in figure 14.

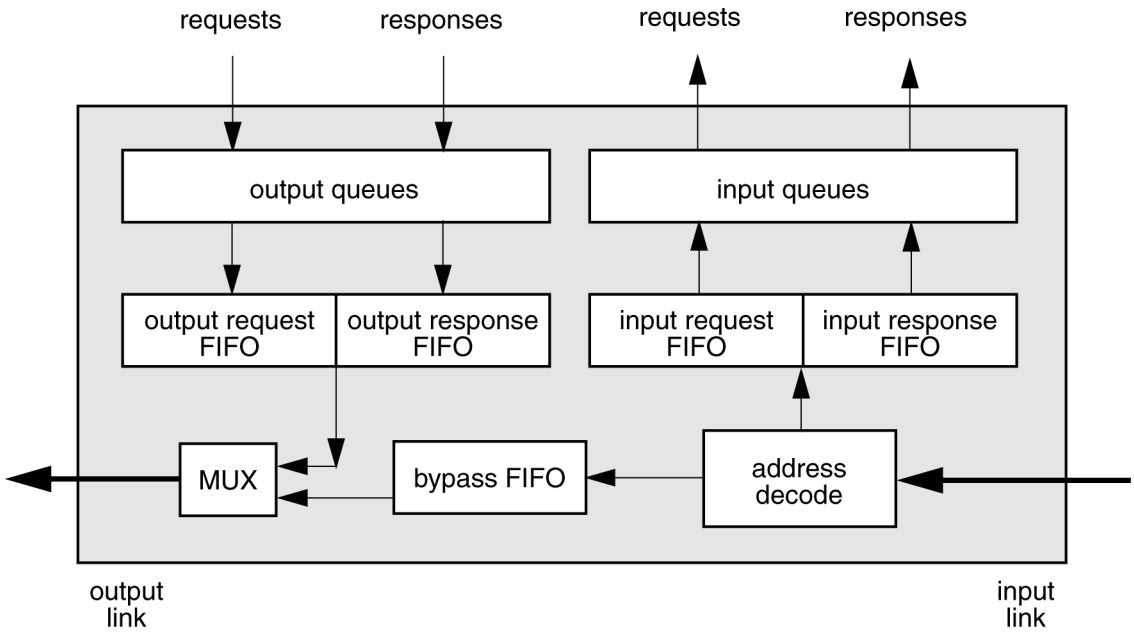


Figure 14 – Paired request and response queues

Pairs of input and output FIFOs are still required, to ensure that requests and responses can be processed independently. The input and output queues can be dynamically or statically allocated for holding requests and responses, if these queues can be bypassed when a FIFO entry is available. Forward progress is ensured because at least one entry is always available for holding input-request, input-response, output-request, and output-response packets respectively.

1.4.4 Switch queues

The concept of independent queue pairs can be extended to switches. For example, the queues in a simple bridge (suitable for use in hierarchical topologies) between two SCI ringlets are illustrated in figure 15.

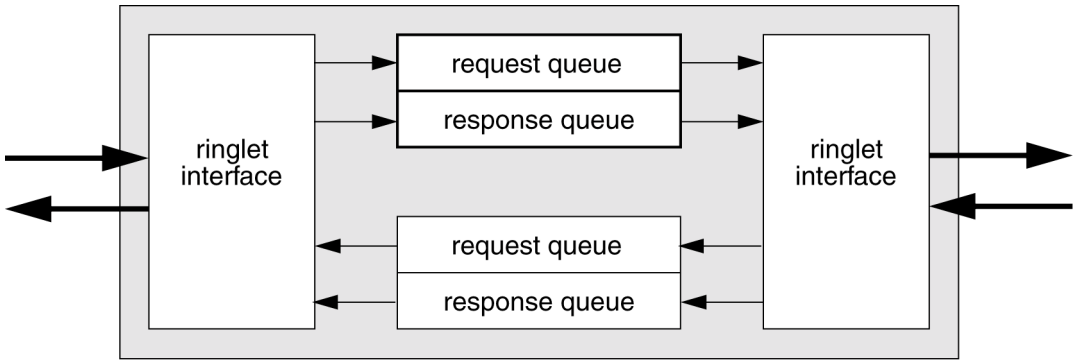


Figure 15 – Basic SCI bridge, paired request and response queues

More-complex topologies could have loops in the physical configuration (e.g., a toroidal topology formed by connecting the top and bottom edges and the right and left edges of a 2-D mesh). Additional queues may be needed to avoid hardware deadlocks due to possible circular dependencies in such systems.

1.4.5 Subactions

When requester and responder are on the same lightly loaded ringlet (i.e., local), a transaction involves four packet transmissions, as illustrated in figures 16 and 17. (Shading is used to indicate the queue that holds the relevant packet. The queue state in figure 16 is shown as it would be just before receipt of the illustrated packet.) The request subaction involves the transfer of a request packet from the requester to the responder (steps 1 and 2). The responder's processing involves the consumption of the request packet and the generation of a response packet. The response subaction involves the return of a response packet from the responder to the requester (steps 3 and 4).

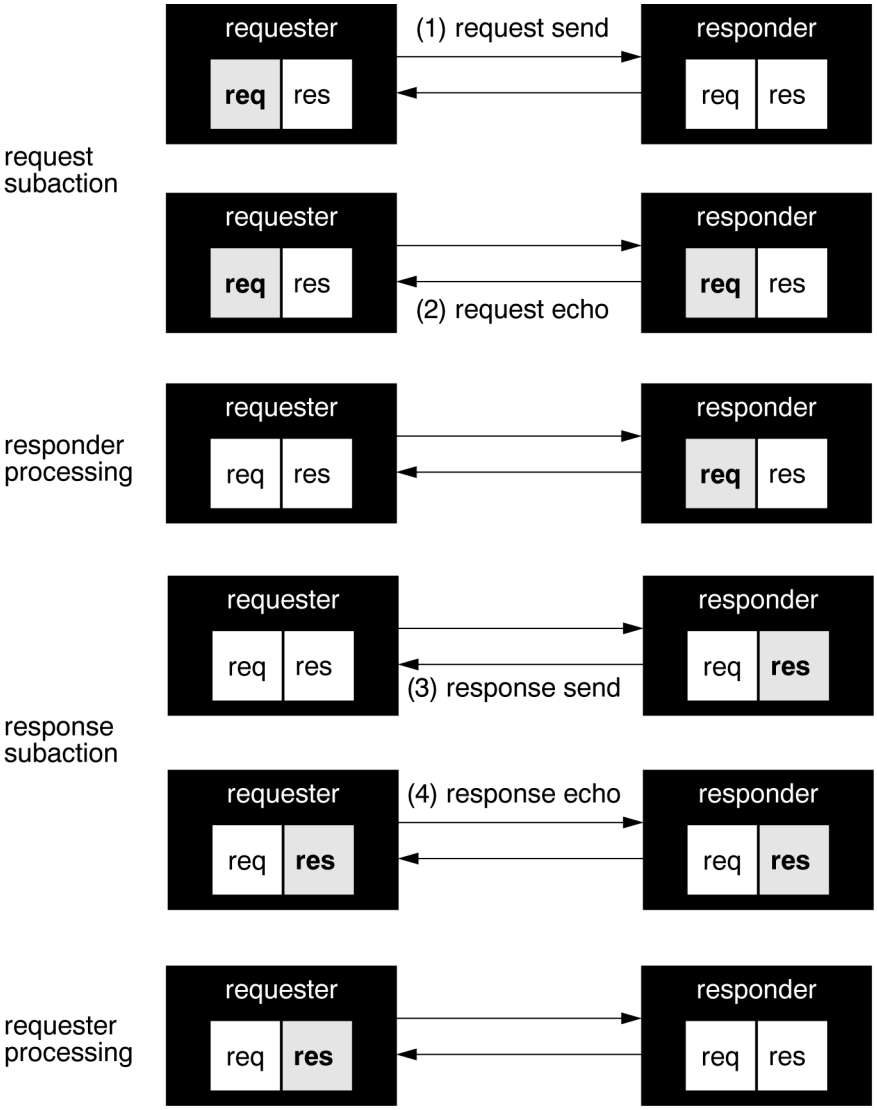


Figure 16 – Local transaction components

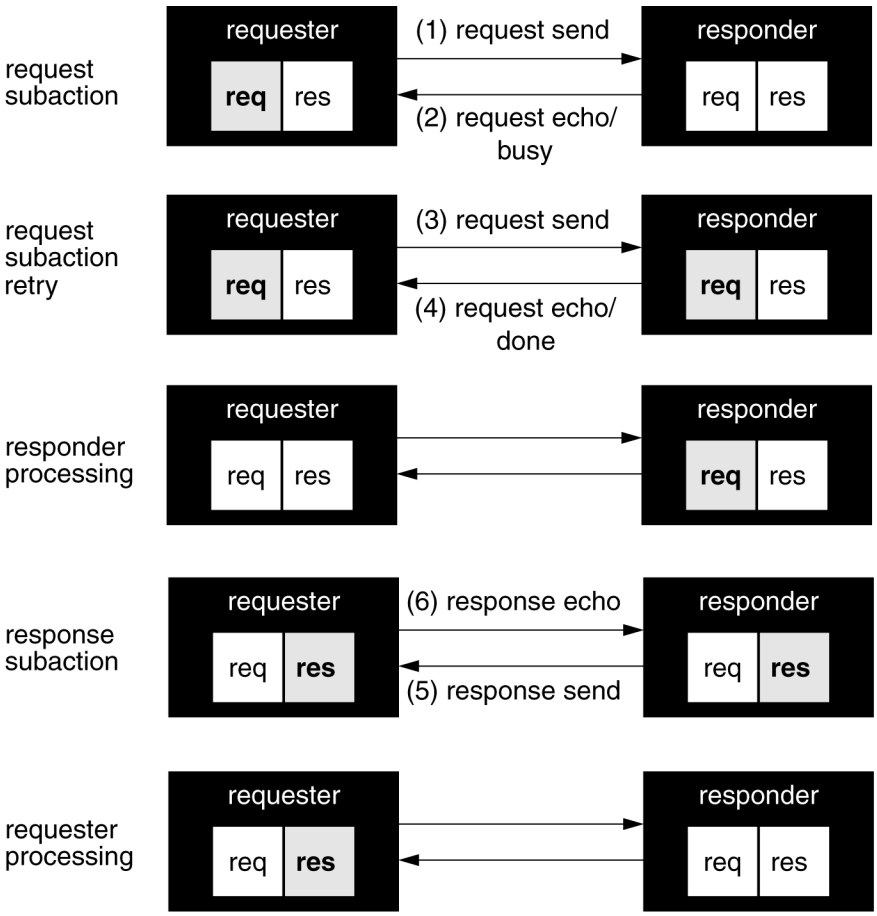


Figure 17 – Local transaction components (busied by responder)

Each subaction consists of a *send packet* (steps 1 and 3 in figure 16) that transfers information between a producer and a consumer and an *echo packet* (steps 2 and 4) acknowledging the receipt of the information. Each packet is sent between a *source* and a *target*. The producer is a source for request-send and response-echo packets and a target for request-echo and response-send packets.

The producer saves a copy of the request-send (or response-send) packet until a returned request-echo (or response-echo) packet confirms its acceptance at the consuming node. The echo packet may sometimes indicate that the consumer queues were busy (full) and that the send packet was discarded. These busied packets are retransmitted until they are accepted by the consumer. *Bandwidth allocation protocols* are used to guarantee that all producers will eventually transmit their send packets; *queue allocation protocols* guarantee that consumers will eventually accept these send packets (or a busied retransmission of them, see 3.7).

For example, consider a heavily loaded system, where there is contention for the shared responder subaction queues. If the responder's request queue is full, the first request-send packet may be busied and retransmitted as illustrated in figure 17. The queue state in this figure is shown as it would be just before completion of each illustrated subaction.

The first request-send packet (1) is busied by the responder, which initially has a full request queue. The request-send packet is discarded and the busy status (2) is returned in the first request-echo packet. Later another request-send packet (3) is sent from the requester to the responder and (in this example) is accepted; receipt of the request-send packet is confirmed by the status returned in the request-echo packet (4).

Although not illustrated in figure 17, either the request-send or the response-send packet may be busied many times, but will eventually be accepted. Simple ageing protocols guarantee that the oldest busied transactions are eventually accepted.

1.4.6 Remote transactions (through agents)

A packet starts at an original-producer (source) node, addressed to a final-consumer (target) node. For a remote transaction the source and target nodes are on different rings. The packet will then be accepted by consumer queues in intermediate agents (e.g., bridges or switches) for forwarding to the target. Each intermediate agent behaves like a producer when forwarding the packet to its final-consumer node. A given packet has only one final consumer, but may be processed by a number of consumer/producer pairs as it moves from agent to agent.

A remote transaction is initiated by the requester as though it were local. The packets forming the transaction are queued and forwarded by intermediate agents. To the requester, the agent behaves like a responder; to the responder, the agent behaves like a requester. An agent typically acts on behalf of many nodes, and thus accepts packets with any of a set of addresses (a different set on each side). The steps involved in the completion of a remote SCI transaction are illustrated in figure 18, for a lightly loaded system (no subaction queues are full) with a single intermediate agent. In this figure, the queue state is shown as it was before the start of each illustrated subaction.

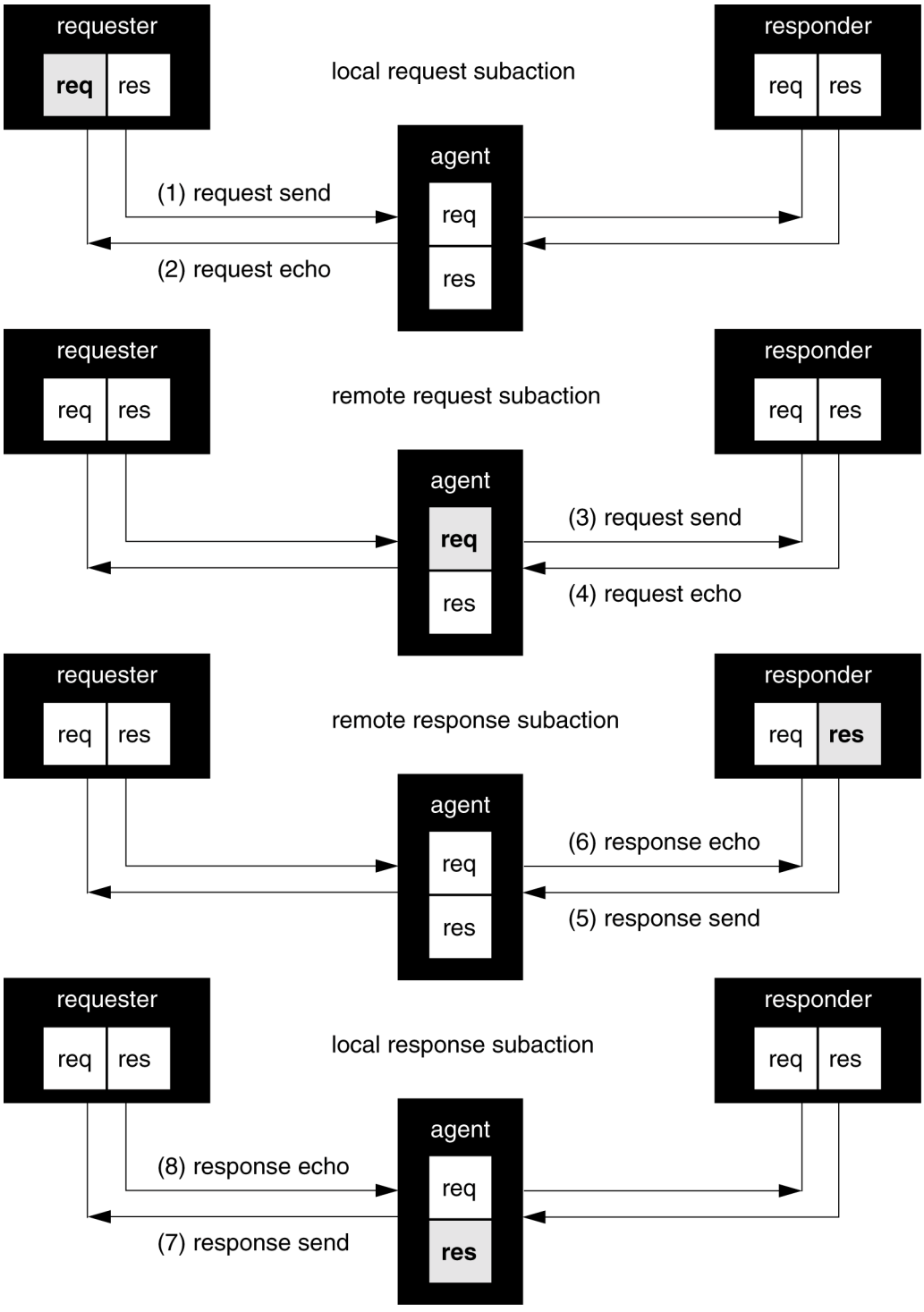


Figure 18 – Remote transaction components

The initial request subaction (1 and 2) transmits the request packet from the local requester to the intermediate agent. The remote request subaction (3 and 4) forwards the request packet from the agent to the remote responder. After confirmation that the request has been accepted by the responder, the intermediate agent discards subaction information (residual history); its send buffers can immediately be reused for other purposes.

Note that subactions do not care whether they are local or remote; only agents need know that the subaction is not local. Note also that echoes merely confirm delivery to the next agent, not necessarily to the final consumer, and that queues in agents take responsibility for further transmission.

After the request has been processed by the responder, the remote response subaction (5 and 6) transmits the response packet from the remote responder to the intermediate agent. The local response subaction (7 and 8) forwards the response packet from the agent to the original requester. After confirmation of the response being accepted by the requester, the responder and the intermediate agent have no queued send packets; their send buffers can be immediately reused for other purposes.

An active agent can be pipelined; forwarding of the request-send packet (3) can begin before the request-echo packet is returned (2) to the requester. The same is true for the response; the response-send packet (7) can begin before the response echo (6) is returned to the responder. Note that an agent must also keep a copy in its queue until echo confirmation has occurred.

This mechanism applies in general for any number of intermediate agents. The routing of packets in a system is determined by the set of agents, each with its own set of addresses to accept.

1.4.7 Move transactions

A *move* transaction is like a write transaction, with the exception that no response subaction is returned. A move transaction is expected to be used when large amounts of data are transferred and timeliness is more important than confirmed delivery, such as for repetitive data transfers to a video frame buffer. Although more efficient than a write transaction, the lack of a response (which provides the responder's completion status) limits move-transaction uses to specialized applications or constrained configuration topologies.

A move transaction is a specialized noncoherent write transaction that has a request subaction but (for improved efficiency) no response subaction. Flow control, performed at the subaction level, ensures that request-send packets are not discarded when attempting to enter congested queues. However, transmission errors (which are normally reported in response subactions) will not be detected by the standard lower-level protocols (but could be by application-specific higher-level ones). The steps involved in the completion of a remote SCI move transaction are illustrated in figure 19, for a lightly loaded system.

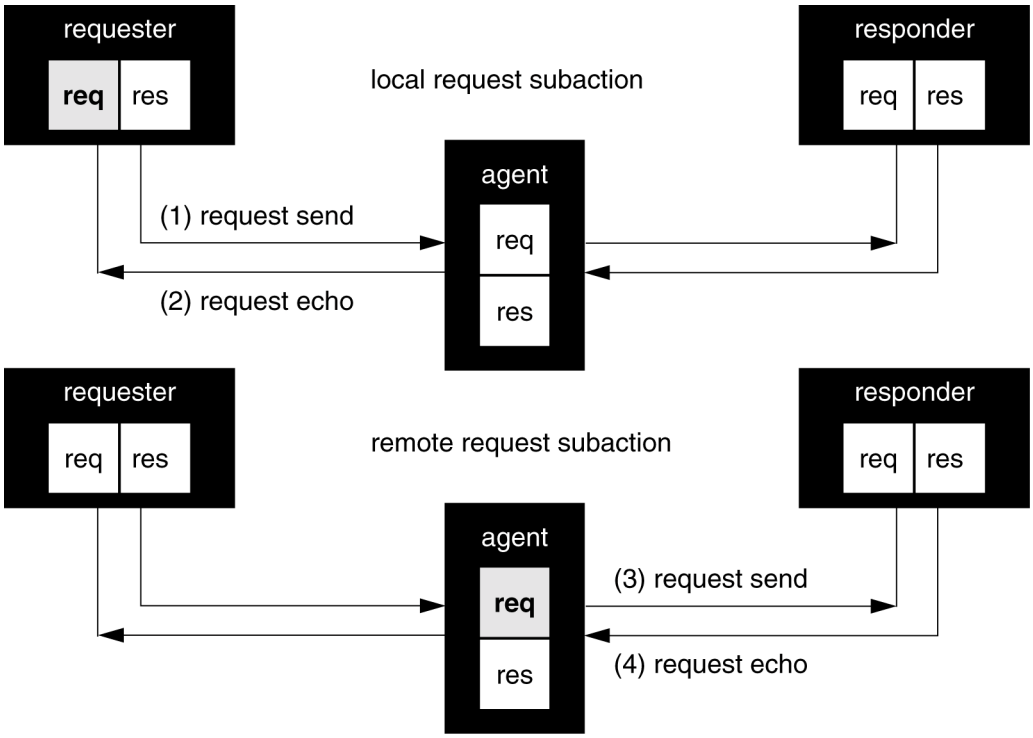


Figure 19 – Remote move-transaction components

The local request subaction (1 and 2) transmits the move-request packet from the requester to the intermediate agent. The remote request subaction (3 and 4) forwards the move-request packet from the agent to the responder. The final agent is informed when the request is queued in the responder, but the requester receives no such confirmation. Since transactions may be reordered while passing through an interconnect, there is no standard way for either the requester or the agent to confirm when or if the move transaction has completed.

Since move transactions have no response, there is no standard way to return agent or responder error status to the requester. Intermediate agents and responders are expected to provide mechanisms for logging these errors, but these error logging mechanisms are beyond the scope of the SCI standard.

Since move transactions have no confirming response, there is no reliable way to use their *transactionId* values to differentiate between distinct move transactions. Thus, producers with two or more active move transactions could become confused, when two or more active move transactions generated the same request-echo packet. To avoid these confusions, producers are expected to temporarily inhibit transmission of new move requests when their echoes could be confused with those that are already expected from other active requests. (An active request is one that has been sent but whose echo has not been returned).

1.4.8 Broadcast moves

Some applications can benefit from the optional capability of efficiently broadcasting a packet to multiple destinations using a single transaction. Application examples include some kinds of image processing such as HDTV (high-definition television) signal processing, systolic processing arrangements, and massively parallel architectures such as neural networks. Special protocols are used to ensure forward progress, since a move transaction might sometimes be accepted by some of the nodes but not all (when some of the consumer queues are temporarily full).

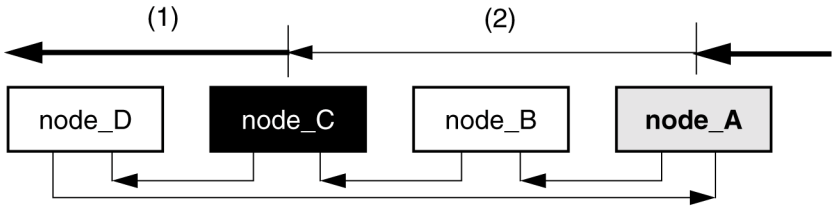
In the worst case, a broadcast consumes the same bandwidth as sending the packet repeatedly to all its *N* destinations. In the best case, it reduces the consumed bandwidth by a factor of *N*, when there are *N* broadcast-capable nodes on the ring. Note that broadcast transactions are ignored by nodes that do not support this optional capability.

Several subaction command codes are allocated for broadcast functions. Half of these codes are for starting broadcast messages; the other half are for the resumption of a previously initiated broadcast. Except for having multiple effective target addresses, broadcast (*start* and resume) transactions are functionally equivalent to directed moves (they do not have a response subaction and they do not participate in cache coherence).

On a local ringlet, a start-broadcast packet is sent from the broadcaster to itself, with a special start-move command code (*smove*) that enables the eavesdrop capability on other ringlet-local nodes. The command code for the broadcast is decoded by all those nodes that have broadcast capability; the *smove* is ignored by nodes that do not support broadcast, based on its target address.

If all acceptance queues are free, the *smove* packet returns to its source (*node_C*) and is stripped. The originating broadcaster *node_C* recognizes that no echo is needed, but updates its send queues as though one were received. The strategy of not echoing one's own send packet*s* is efficient, simplifies the allocation-priority sampling protocols, and applies to directed send packet*s* as well.

If an eavesdropper's acceptance queues are full, it strips (1) the packet and returns (2) an echo, as illustrated in figure 20.



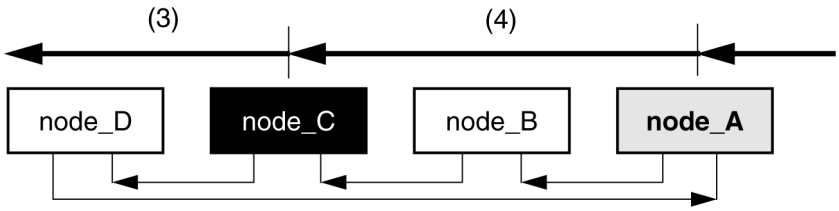
1) smove send packet, target = source = R

2) echo packet, target = R, source = A

Figure 20 – Broadcast starts

In this example, the broadcast transaction has been originated by a remote node (*node_R*) and is being forwarded to this ringlet through *node_C*. Just as for other SCI transactions, the send packet's sourceId field is provided by the original source, not the local agent.

When the busied transaction is re-sent (3) by *node_C*, the retried packet contains the resume-broadcast command (*rmove*) and is directed to the node that returned the echo (*node_A*). The resume-broadcast packet is directed in the sense that it is ignored by other ringlet-local nodes. While the acceptance queues are full, the *rmove* packets are stripped and echoed by *node_A* and re-sent by *node_C*. When the acceptance queues become free, *node_A* converts the packet into its original *smove* form (4) for distribution to the downstream nodes, as illustrated in figure 21.



3) rmove send packet, target = A, source = R

4) smove send packet, target = source = R

Figure 21 – Broadcast resumes

When the *rmove* transaction is accepted by *node_A*, its target address is restored to the value provided by the source and its command value is restored to the original *smove* value. When this queued packet is passed to an adjacent ringlet it looks like the original broadcast. Restoring the resume-broadcast to its start-broadcast form also requires regeneration of the CRC value, since the target and command fields change. Note that waiting for the sourceId before converting the packet to its original form requires two extra levels of pipelining in the node's packet processing (one more than needed by a ring scrubber).

The *smove* transaction completes when it is stripped by the originating *node_C*. This broadcast is never busied, even if *node_C*'s acceptance queues are full. This is because the broadcast actions were already performed on *node_C*, before the send packet was originally transmitted.

1.4.9 Broadcast passing by agents

The routing algorithms for an agent's directed and broadcast transactions may differ, to prevent broadcasts from travelling from one ringlet through a switch or a bridge to another ringlet and back again, thereby circulating in the system indefinitely.

For example, consider two ringlets connected to each other via two distinct symmetric bridges. A broadcast could start on one ringlet, propagate to the first bridge, pass to the other ringlet, circulate around to the second bridge, and then propagate back onto the original ringlet. There would be an infinite loop and an increasing number of packets the original packet would go past the bridge while the bridge creates a new one.

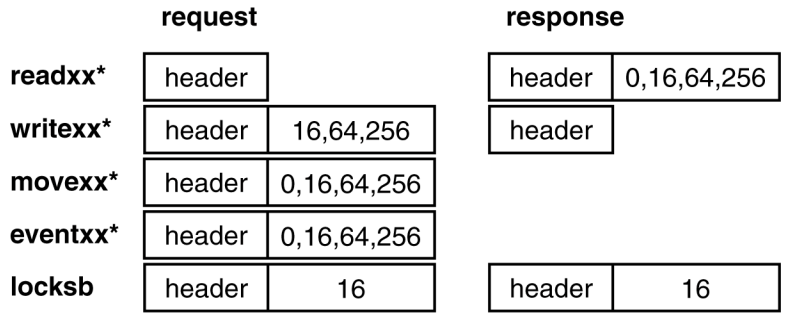
Normally an agent needs only to look at the *targetId* and its own internal routing tables to decide whether or not to pass a packet to its remote side. That is, routing decisions depend entirely on packet destinations. However, agents that support broadcast transactions look at the *sourceId* field in broadcast packets, and broadcasts have a special routing table. The table indicates which broadcasts are to be passed, based on *sourceId* comparisons. When properly initialized, such tables prevent the return of broadcasts that previously left this ringlet.

Such broadcast routing tables need to be set up at initialization time. Proper setup of these routing tables involves treating each node in the system as the potential root of a tree whose branches are formed by the other ringlets and agents in the system. System initialization procedures are expected to put these broadcast tree routes into the broadcast tables with the specific purpose of creating efficient paths that have no loops. These procedures may optionally take into account traffic patterns in the system in order to optimize path assignments where path choices exist.

Note that the implementation of the broadcast routing table in an agent, like the normal routing table, need not be a table lookup. In some configurations, the routing can be done algorithmically with *sourceId* range-checking logic. However, the specification of the routing tables or range-checking logic is beyond the scope of this standard.

1.4.10 Transaction types

Several types of transactions are supported, including reads, writes, and locks. The primary difference between these transactions is the amount of data transferred, and in which subaction is as illustrated in figure 22.



Note: **xx** represent one of the allowed data block lengths
(number of data bytes, on the right after the header)

Figure 22 – Transaction formats

Readxx transactions copy data from the responder to the requester; writexx transactions copy data from the requester to the responder. Readxx and writexx transactions both have responses, which are used to return the completion status from the responder.

Movexx transactions copy data from the requester to the responder. Movexx transactions are more efficient than their nearly equivalent writexx transactions, but there is no provision for returning the completion status from the responder.

Eventxx transactions copy data from the requester to the responder. Eventxx transactions have no flow control, and are never rejected. Therefore, the protocols in this standard can provide no guarantee of data delivery. Event00 is used for synchronizing time-of-day clocks. If the other Eventxx transactions are used for moving data, the system designer must provide sufficient storage for that data outside the normal request-queue storage that is managed by SCI's flow control mechanisms.

Locksb transactions copy data from the requester to the responder. The responder indivisibly updates the affected address, based on the command value and the request-subaction's data. The response subaction returns the previous (unmodified) data and status. These non-coherent transactions support fetch&add as well as compare&swap update operations.

Shorter transactions, such as a 1-byte write transaction, are formatted as 16-byte transactions, but only a portion of the data is used. These selected-byte read and write transactions are useful when accessing *control* registers (which are less than 16 bytes in size, and whose side-effects are sometimes dependent on the transaction size).

1.4.11 Message passing

SCI supports message passing, as defined by the CSR Architecture. A standard noncoherent write64 transaction is used to send short unsolicited messages to a specified CSR register within the target node. Two techniques for sending longer messages can be used:

1) *Concatenated packets.* Two or more 64-byte write transactions are concatenated to form a longer message.
2) *Indirect pointer.* A long message transfer (from A to B) is initiated by a short unsolicited message from A to B. This message includes a pointer to the longer message, which remains stored in memory at A. After processing the message pointer, the processor on node B reads the long message from node A.

To simplify flow-control protocols (and buffer allocation), the indirect-pointer approach is recommended.

1.4.12 Global clocks

The SCI standard supports global time synchronization, as defined by the CSR Architecture. SCI nodes can maintain local clocks (formatted as 64-bit integer-seconds/fraction-seconds counters). Hardware provides mechanisms for detecting drifts between clocks, and software is responsible for correcting the drifts as they are detected. Several expected uses of the clocks are as follows:

1) *System debugging.* If the optional trace feature is implemented, the route of a packet with its trace-bit set can be reconstructed by logging (with an accurate time stamp) packet arrivals at switching points in the interconnect.
2) *Time of death.* If the optional *timeOfDeath* value is provided in the packet header, stale send packets can be safely discarded before they might be misinterpreted.
3) *Real-time data.* A global clock can be used to synchronize the activities of multiple data-acquisition nodes (such as A/D and D/A converters).

On a traditional backplane, a *clockStrobe* signal can be broadcast to synchronize clocks on observing nodes. Clock synchronization on SCI is more complex, since signal paths are daisy-chained or switched rather than bused (see 3.12.4.1).

1.4.13 Allocation protocols

Depending on system configurations and dynamic loading conditions, the cumulative bandwidth requirements of multiple requesters can exceed the capacity of a shared interconnect or the bandwidth of a shared responder. When the cumulative bandwidth exceeds the available bandwidth, allocation protocols apportion the oversubscribed resources to the multiple requesters.

Most of the bandwidth is (optionally) apportioned unfairly to the highest-priority transactions. However, a small portion of the bandwidth is always apportioned fairly, as illustrated in figure 23. There are four priority levels: 0 through 3 are the lowest through highest priority respectively. The allocation protocols allocate most of the bandwidth (approximately 90 %) to those transactions with the highest priority that is currently being used; the remaining bandwidth is allocated fairly to those transactions having priorities less than the current highest priority.

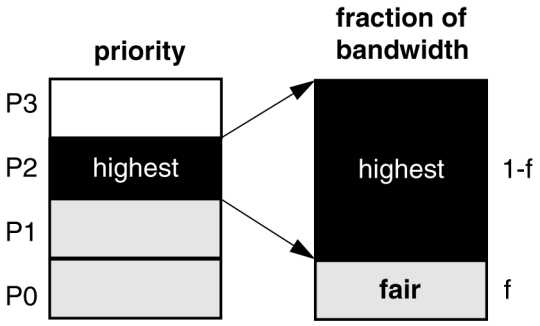


Figure 23 – Bandwidth partitioning

For the lower-priority nodes, the relative node priority has no effect on the allocation of this bandwidth. However, under dynamic loading conditions, the higher-priority nodes are likely to become the highest-priority nodes more often, which then increases their apportioned bandwidth.

Although this partial fairness scheme complicates allocation protocols, having even a little guaranteed bandwidth fairly allocated simplifies SCI in other ways, which include the following:

1) *Forward progress.* The impact of transient hardware or software priority inversions is minimized. A high-priority process can be temporarily blocked by a low-priority process without deadlocking the system.
2) *Deterministic timeouts.* For any system configuration, deterministic worst-case transaction timeout values can be calculated. These values are necessary for initializing the timeout hardware.
3) *Queue-allocation protocols.* Partial fairness bounds the time limit for retrying busied transactions. This simplifies queue-allocation protocols, which wait for retries of previously busied transactions.

Bandwidth allocation protocols apportion bandwidth on a local ringlet. When many *requesters* and many responders are on the same ringlet, allocation protocols apportion the shared ringlet bandwidth. SCI bandwidth-allocation protocols are similar in effect to bus arbitration protocols.

Queue allocation protocols allocate queue entries in a responder or switch component. When many requesters access the same responder, the responder's allocation protocols allocate the limited responder-queue bandwidth. SCI queue-allocation protocols and bus-bridge busy-retry protocols are similar in function.

Bandwidth allocation protocols apportion bandwidth when the interconnect is the bottleneck; queue allocation protocols apportion bandwidth when a shared responder (or intermediate agent) is the bottleneck. These bottlenecks are illustrated in figure 24. Shading indicates congestion.

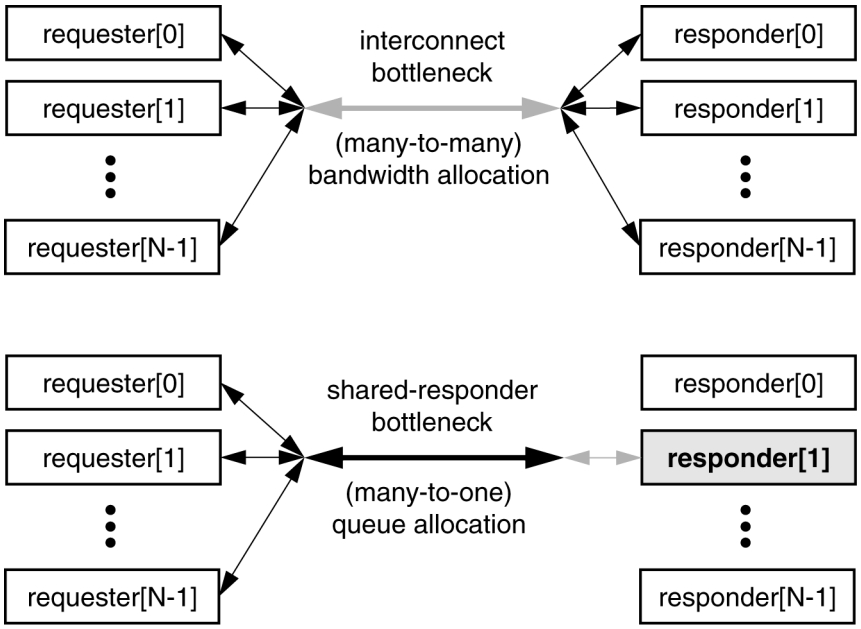


Figure 24 – Resource bottlenecks

Requester nodes assign a two-bit transaction priority to their transactions. This transaction priority affects the bandwidth and queue allocation protocols, which assign most of the available bandwidth to the highest-priority nodes. A send packet's effective priority is usually equal to its transaction priority, but may be temporarily increased because of higher-priority packets that are blocked behind it. This priority-modification process is called priority inheritance. Priority inheritance is supported by SCI, whose send packets contain the transaction priority as well as the effective priority.

Allocation of prioritized bandwidth has a delayed effect. Transmission of future packets is inhibited based on the state of other nodes in the recent past. On large systems, these protocols can effectively apportion bandwidth but have little effect on reducing the latency for random accesses.

Traditional backplane bus arbitration takes longer, but simultaneously senses the priority of all nodes, so priority information is more current and more directly affects latency. Note that this bus virtue comes at the price of severely limiting the bandwidth and the maximum number of nodes.

1.4.14 Queue allocation

Most bus designers are familiar with arbitration protocols, which are similar in function to SCI's bandwidth allocation protocols. When bus transactions are unified (not split into separate request and response subactions) and never busied, fair arbitration protocols are sufficient to ensure that all transactions eventually complete. However, when bus transactions are split into request and response subactions, many requesters may access a shared responder node, and its available queues may be filled. When queues are filled, request subactions are terminated with a busy status, which forces them to be retried until the queue eventually has space.

In the absence of queue-reservation protocols, some retried *request subaction*s could never be sent successfully. Although queues may be emptied quickly, they could consistently be refilled by one or several other requesters, while the one *requester* is continually busied, as illustrated in figure 25.

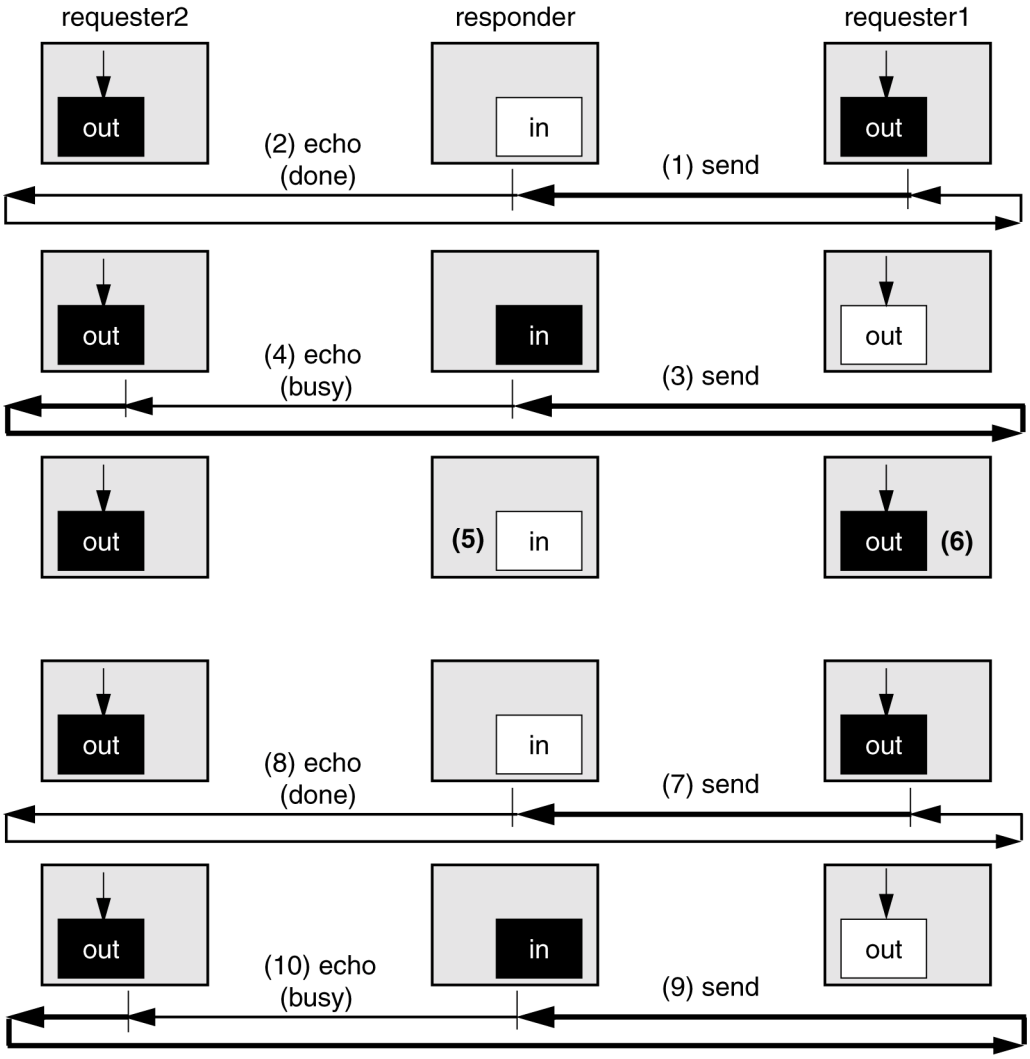


Figure 25 – Queue allocation avoids starvation

In this illustration, requester1 initially sends (1) a request-send packet to the responder; since the responder's queue is empty, the packet is accepted. The returned request-echo packet indicates (2) the request send was accepted without error. However, this request-send packet temporarily fills the responder's input-request queue.

Before the responder has processed its input-request queue, another request-send packet is sent (3) from requester2; since the responder's queue is full, the packet is rejected. The returned request-echo packet indicates (4) the subaction was busied and should be quickly retried.

Soon thereafter, the responder's input-request queue is emptied (5) and another request-send packet is generated (6) within requester1. The new request subaction is sent (7) from requester1; since the responder's queue is empty, the packet is accepted. The returned request-echo packet indicates (8) the request send was accepted without error.

Then requester2 resends (9) its previously busied request-send packet, but since the responder's queue is once again full the packet is rejected. The returned request-echo packet indicates (10) the subaction was busied and should be quickly retried.

If this cycle repeats, the less-fortunate requester2 could be forever starved by the activity of requester1. The SCI allocation protocols avoid such starvation conditions by reserving space for the older send packets that are busied. See 3.7 for details.

1.5 Cache coherence

1.5.1 Interconnect constraints

High-performance processors use local caches to reduce effective memory-access times. In a multiprocessor environment this leads to potential conflicts; several processors could be simultaneously observing and modifying local copies of shared data.

Cache-coherence protocols define mechanisms that guarantee consistent data even when data are locally cached and modified by multiple processors. The SCI cache-coherence protocol can be hardware based, thus reducing both the operating system complexity and the software effort to ensure consistency. Many cache-coherence protocols rely on the broadcasting of all transactions. This broadcasting allows use of eavesdropping and intervention techniques to achieve data consistency. Broadcast transactions are inherent in a bus-based system, but are not feasible for large high-speed distributed systems. Therefore, broadcast and eavesdropping mechanisms are not used by the SCI cache-coherence mechanism.

1.5.2 Distributed directories

SCI uses a distributed directory-based cache-coherence protocol. Each shared line of memory is associated with a distributed list of processors sharing that line. All nodes with cached copies participate in the update of this list.

Every memory line that supports coherent caching has an associated directory entry that includes a pointer to the processor at the head of the list. Each processor cache-line tag includes pointers to the next and previous nodes in the sharing list for that cache line. Thus, all nodes with cached copies of the same memory line are linked together by these pointers. The resulting doubly linked list structure is shown in figure 26.

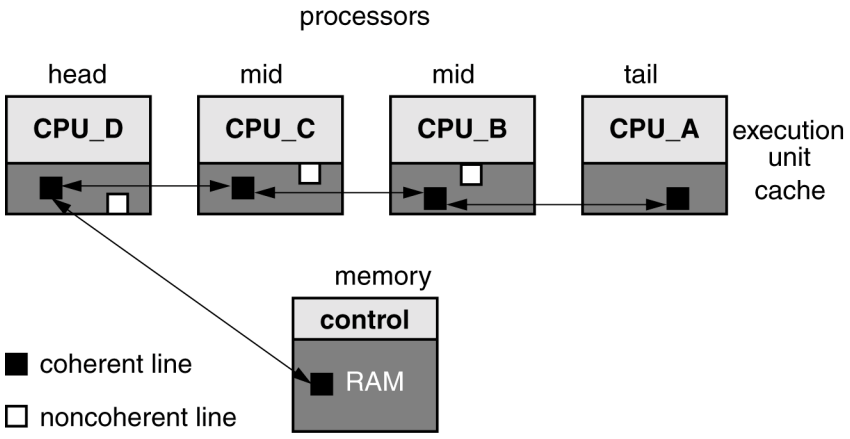


Figure 26 – Distributed cache tags

Note that this illustrates the logical organization of the directory's sharing-list structure for one line, which may be different for each line that is cached. The processors are always shown on the top and the shared memory location is shown on the bottom. These logical illustrations should not be confused with the physical topology of a system; SCI expects that processors and memory will often be found on the same node.

Coherence protocols can be selectively enabled, based on bits in the processor's virtual-address-translation tables. Depending on processor architecture and application requirements, pages could be coherently cached, noncoherently cached, or not cached at all.

This distributed-list concept scales well. Even when the number of nodes in a list grows dramatically, the memory-directory and processor-cache-tag sizes remain unchanged. However, memory-directory storage and processor-cache-tag storage represent extra fixed-percentage overheads for cache-coherence protocols.

The list pointer values are the node addresses of the processors (caches). When a node accesses memory to get a copy of coherently shared data, memory saves the requesting node's address. If there are currently no cached copies, the requesting node becomes the head of a new list. (The memory directory is updated with the new node address.) If other nodes have cached copies of the data, the pointer to the head of the sharing list is returned from memory. The requesting node inserts itself at the head of the list and gets its data from the previous head.

With the exception of the pairwise sharing option, write access is restricted to the node at the head of the list. To get write access, a requesting node creates an exclusive copy by inserting itself at the head of the list and purging the remainder of the list entries. SCI supports both weak and strong sequential consistency, as determined by the processor architecture. A weakly ordered write instruction can be executed before the sharing-list purge completes, while a strongly ordered write must wait for purge completion.

1.5.3 Standard optimizations

Standard optimizations are defined that improve the performance of common kinds of coherence updates, as follows:

1) *Fresh copies.* The *fresh* memory state indicates that all shared copies are read-only; the data can be returned from memory when a new processor is attaching to the head of the previous sharing list.
2) *DMA transfers.* DMA data can be read directly from the sharing-list head without changing the directory state. DMA writes (of full 64-byte lines) can be performed directly to memory, although a list of old copies (purge list) will be returned to the writer if the data were being shared.
3) *Pairwise sharing.* When data are shared by a producer (the writer) and a consumer (the reader), data are directly transferred from one cache to the other. The directory pointers need not be changed, and memory is not involved in the cache-to-cache transfer.

1.5.4 Future extensions

As well as supporting a wide range of interoperable options, the SCI standard intends to support several compatible future extensions. This allows implementations to quickly use the existing specification, while providing opportunities to expand the SCI capabilities when more experience is available. Although the future extensions are beyond the scope of the SCI standard, a short overview is intended to provide the reader with insights on how this standard may evolve in the future.

1.5.4.1 Out-of-band QOLB

The SCI standard supports the concept of delaying distribution of shared data, by queuing additional requesters until a cache line has been released by its current owner (queued on lock bit, called QOLB). The coherence protocols define the QOLB option to avoid transferring shared cache lines until the data can be used. Although QOLB controls the flow of cache lines between caches, an additional lock bit is needed to validate ownership of the cache-line data; within the SCI standard, this lock bit is expected to be contained within the 64 bytes of cache-line data.

A future extension to the SCI coherence protocols could implement a more-transparent lock bit, by providing an out-of-band lock bit for every 64-byte cache line. The advantage of using out-of-band lock bits is that compiler support of QOLB is made much easier. As an example, consider an array of objects, each of which needs a lock bit. The QOLB protocols assumed that the lock-bit and its affected data are contained within the same cache line. Although the compiler can make each object slightly larger, this would change the size of each array object.

If lock bits are implemented as a one-bit cache-line tag, which is located in an out-of-band data address, then the size of array elements is unaffected by the lock bits. To implement these lock bits, each cache line would be assumed to have a 513th bit associated with it. A reserved bit in the header could be used to efficiently transfer this bit in response-send packets; a bit in the extended header could be used to transfer this bit in request-send packets. Processors would be expected to provide special loadQolb and swapQolb instructions to read and modify this out-of-band lock bit, based on the cache-line address being accessed. Special operating system software would be expected to save and restore these extra bits when the data is swapped to secondary storage.

The encoding of this out-of-band lock bit has been deferred, so that it can be reconsidered when the coding requirements of the logarithmic extensions (discussed in the following subclause) are known.

1.5.4.2 Logarithmic extensions

On a large heavily loaded system, hot spots may occur at or near a heavily shared memory controller. To eliminate such hot spots, coherence protocols should support the possibility of combining list-prepend requests in the interconnect. Such hot spots not only degrade the performance of the requesting processor, they degrade the performance of other transactions that share portions of the congested connection path. Although coherent combining is not defined in this specification, it is planned as part of P1596.2, a compatible extension to the SCI standard.

A possible way to support coherent combining is as follows. While queued in a switch buffer, two requests to the same physical memory address (read A and read B) can be combined. The combining generates one response (status A), that is immediately returned to one of the requesters, and one modified request (read A-B), that is routed toward memory. Additional requests (read C) can also be combined with the modified request, as illustrated in figure 27.

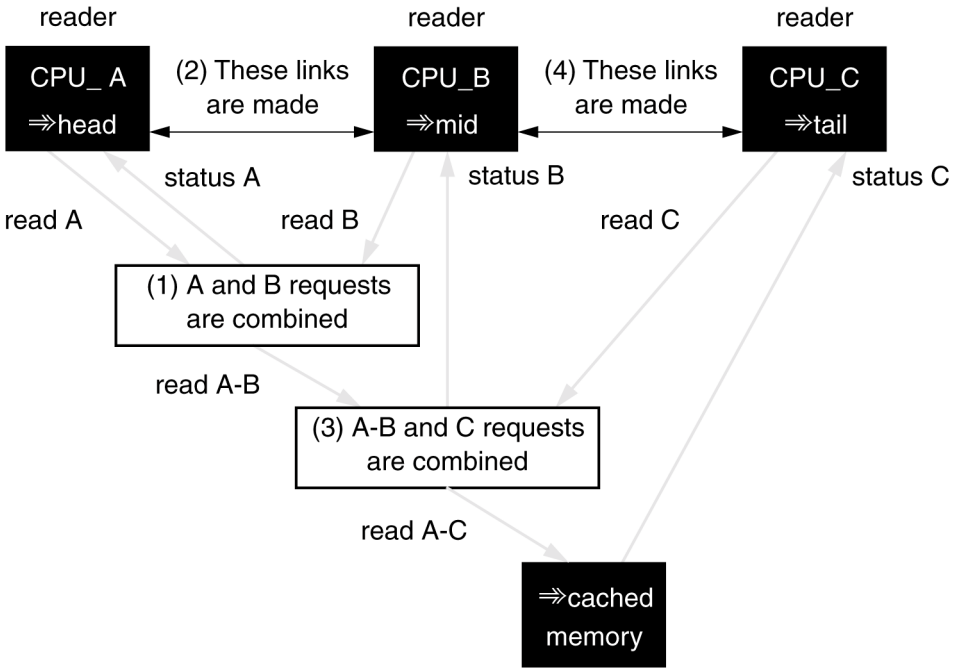


Figure 27 – Request combining

These read transactions can be combined in the interconnect or at the front-end of the memory controller.

When request combining reduces the hot spot latencies, the distribution of data to the other sharing-list entries may become the performance bottleneck. Extensions to the coherence protocols are being developed to reduce the linear latencies normally associated with data distribution and invalidations. Linear latencies can be reduced to logarithmic latencies by adding a third sharing-list pointer to SCI's forward and backward pointers to form a tree structure, as illustrated in figure 28.

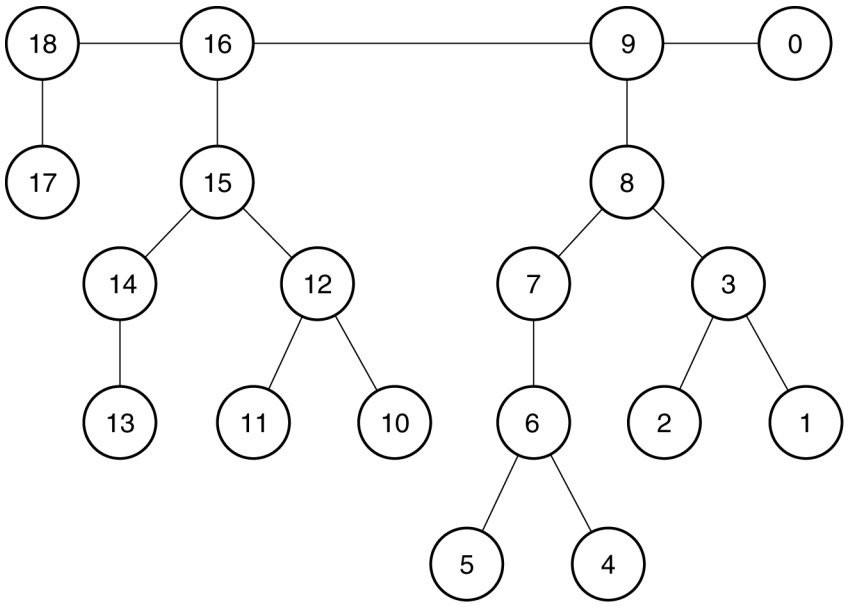


Figure 28 – Binary tree

The three pointers per cache line define a binary tree. Shared data can be routed through the tree to quickly distribute new copies of read-shared data. A writer can also route purges through the tree to quickly invalidate other read-only copies. Deadlock avoidance for forwarding of data and purges can be handled correctly.

The support for binary trees is planned as a compatible extension to SCI (P1596.2). It is an authorized standards project that has not been completed at the time of this International Standard's publication. For current information contact the chairman of that working group.

1.5.5 TLB purges

Most SCI systems will have processors that use virtual addressing. Such processors cache their most recent virtual-to-physical address translations in special translation lookaside buffers (TLBs). When page-table entries are changed, remotely cached TLB entries need to be purged.

TLB replacements are usually handled by software that purges the corresponding remote entries when page-table entries are changed. Three remote TLB purge mechanisms are supported by SCI:

1) *Indirect purging.* The TLB purge address is left (1) in a memory-resident message. Remote processors are interrupted (2), read their messages, purge their local TLB entries, and return their completion status to memory (3).
2) *Direct purging.* The TLB purge address is written to a *control* register on each remote processor. The response from the *control* register write is delayed until the TLB purge has completed.
3) *Coupled purging.* Physically addressed TLB entries can be implemented as cached versions of page-table entries. When the page table is modified the cache-coherence protocols are used to invalidate the TLB entries in the other processors.

The first two of these TLB-purge options are illustrated in figure 29, for processor P-1 purging a TLB entry in processor P-2. The third option has some dependency interlocks that must be clearly understood to ensure correctness while avoiding deadlock.

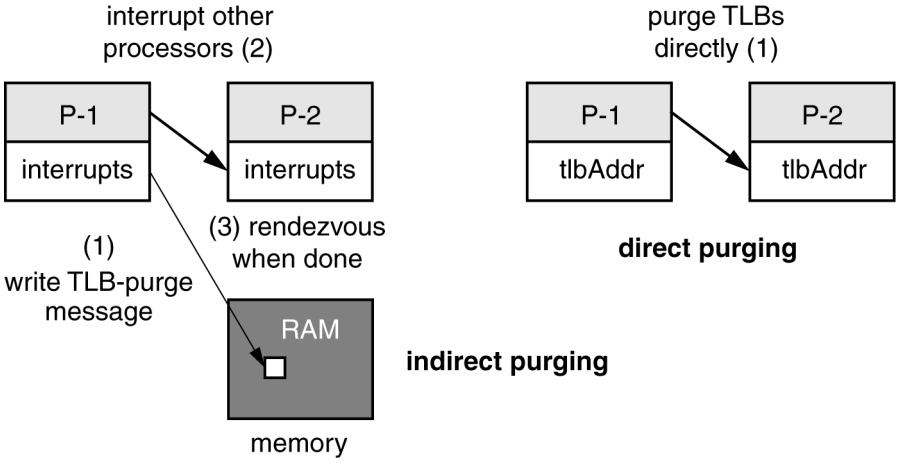


Figure 29 – TLB purging

1.6 Reliability, availability, and support (RAS)

1.6.1 RAS overview

Maintainability has been a primary concern in the design of SCI. To simplify maintenance, the SCI protocols have been defined with the following precepts in mind:

1) *Conceptual simplicity*. Although high-performance circuits may be complex when implemented, the functions provided by the SCI interconnect should be conceptually simple.
2) *Minimum options*. It is better to standardize on one nonoptimal option than to support a wide variety of options in the field.

Rather than describing a formal RAS strategy, this clause describes the major decisions (in the logical protocols) that were influenced by the RAS objectives and strategies.

1.6.2 Autoconfiguration

Each ringlet has a scrubber node that is responsible for monitoring ringlet activity and discarding stale or corrupted packets and idle symbols. To minimize human errors in the configuration process, the scrubber is automatically selected when the ringlet is initialized. This avoids the use of human-settable switches, which could accidentally be set to conflicting values.

The scrubber-selection process is based on an 80-bit unique identifier. The 16 most-significant bits of this identifier can be set manually, so that a pre-specified scrubber can be selected whenever the ringlet is initialized. The least-significant 64 bits of the number are used to break ties, when two or more nodes have the same value for the 16 most-significant bits. These 64 bits are assigned at node manufacturing time, or may be generated randomly (based on a real, not pseudo-, random number generator).

The initial addresses on each ringlet are automatically assigned by the scrubber, based on the distance of the node from the scrubber. In larger systems with multiple ringlets, each of the scrubbers initially assigns the same sequence of nodeId values to the nodes on its ringlet. Initialization software eventually overrides these initial values and assigns unique *nodeId* values to all nodes on all ringlets in the system.

1.6.3 Control and status registers

In the design of the control and status registers (as defined by the CSR Architecture), the following issues were considered:

1) *Autoconfiguration*. When new nodes are inserted, the old boot code should still work on the new system. The new configuration can be automatically detected and dynamically initialized. Autoconfiguration support includes the following features:
   a) *Standard ID-ROM*. Each node has ROM. A standard portion of the ROM identifies the node's name and initialization characteristics.
   b) *Standard selftests*. With standardized selftests, a node can be partially initialized before its I/O driver software is available.
2) *Distributed error logs.* The CSR Architecture provides the framework for implementing distributed error logs, one on each node in the system. These error logs supplement the standardized error status codes when attempting to isolate the source of an error.

See the CSR Architecture for details. Note that most of the definitions therein are shared by related buses (Futurebus+ and Serial Bus) as well.

1.6.4 Transmission-error detection and isolation

In a large system, a significant number of errors may occur during packet transmissions. SCI protocols are designed to detect these errors readily and isolate them. Although a small portion of each packet has no error detection coverage, these fields are only used for arbitration purposes; an error in them would affect only the packet's ringlet-local effective priority, not the packet's correct interpretation.

To reliably detect transmission errors, all packets are protected by a 16-bit ITU-T CRC code. The packet's flow-control information (which dynamically changes during packet routing) is excluded from the CRC calculation. Thus, the CRC is unchanged by intermediate (switch) hops between the original source and the final target. This simplifies implementation of switches and improves reliability of error checking (coverage is not compromised while a new CRC is being appended to unprotected data).

Timeouts are also used to detect transmission errors. Whenever possible, these timeouts are designed to be self-calibrating (so they cannot be incorrectly set). An exception is the response timeout, which has to be set by software (based on knowledge of system configuration and design parameters). Allocation protocols ensure a minimal amount of fairly apportioned bandwidth, so proper timeout values that detect hardware transmission errors are independent of the system's real-time software loading.

Addressing errors are a form of transmission error; although the data are not corrupted during transmission, there is no target to properly acknowledge the packet. These addressing errors are quickly detected and reported by ringlet scrubbers, so that these (software-related) errors will not be confused with other (hardware-related) types of transmission errors.

When possible, error status is returned to the *requester* in the response-send packet, using a 4-bit status code. The status code distinguishes among error categories. This helps isolate the cause of the problem (for an address-ID error), or the location of additional information (for a responder-data error).

1.6.5 Error containment

To simplify recovery from transmission errors, errors are contained (whenever possible). For example, the conversion of a send packet into an echo packet is delayed so that the integrity of the send packet can be reflected in its echo.

Often transmission of a packet or echo has begun before it is discovered to be invalid. This is commonly done to reduce latency. In such a case the correct CRC is computed for the data as transmitted, and then certain bits are complemented to produce a recognizable bad CRC value. This process is called stomping the CRC, and makes it possible to discriminate between packets newly discovered to be bad and those that have already been detected but are still propagating. Thus error logging can record the bad packet at only the first checking location after the failure, making discovery of the failure point easier. The stomped CRC is a bad CRC, and has the normal effect that the packet will eventually be discarded.

Error containment also influenced the *time-of-death* fields (which are optionally included in all send packets). When a response timeout is generated, the *time-of-death* value can be used to guarantee that residual send packets have been deleted. This simplifies error recovery, since stale packets (which could be confused with newly generated transactions) are never delivered.

1.6.6 Hardware fault retry (ringlet-local, physical layer option)

Ringlet-local hardware fault-retry may be supported (as a physical layer option) on individual ringlets. However, hardware fault-retry is not supported for end-to-end transmissions, since the failures introduced by the (much more complex) end-to-end retry hardware would most likely offset most of the benefits it could provide. For example, hardware fault retry could be used to improve the reliability of transmission over a less reliable intermediate medium, as illustrated in figure 30.

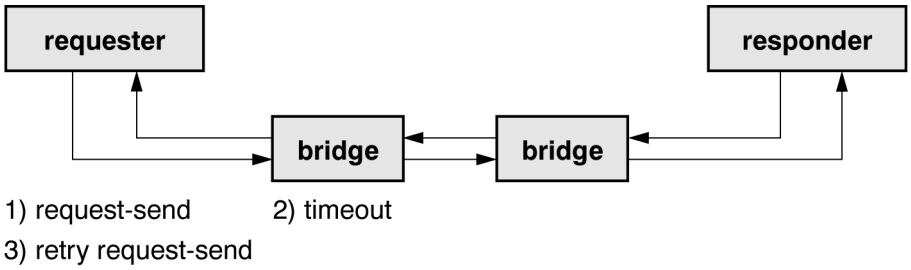


Figure 30 – Hardware fault-retry sequence

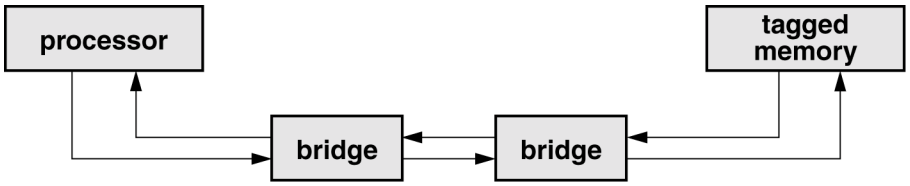
Hardware fault retry has significant costs; special accounting hardware is needed to log sequence numbers needed for duplicate suppression, and each packet is lengthened by prepending these sequence numbers. The SCI standard does not define a hardware fault-retry mechanism.

1.6.7 Software fault recovery (end-to-end)

Several forms of software fault recovery are well supported. When accessing noncoherent CSRs, many transactions can be safely retried by software. The retry is not as simple as it first sounds; after the failure, the success of the first transaction is unknown (it may have succeeded or failed). Reads (to SCI-defined registers) have no side effects, so reads of these registers can be safely retried (one and two reads are equivalent, they both have no side effects).

Many writes have side effects, but can safely be retried (the side effects of one and two identical writes are the same). Retrying writes to CSRs where one and two writes have different side-effects is harder. For these registers, the CSR Architecture recommends using sequence-number bits in the data; these bits can be used by software (to verify the success or failure of the initial transaction attempt). Designers should carefully consider these problems and avoid creating needless difficulties for error recovery.

Software can perform end-to-end fault retry on *coherent memory transactions*. Since coherent memory has a tag identifying the last owner, the previously dirty entries can be identified after the fault is detected. Transaction fault recovery involves flushing the old dirty copy to memory and destroying the (possibly now corrupted) sharing-list structure, as illustrated in figure 31. After the data have been flushed, the sharing list is rebuilt automatically using the standard coherence protocols.



1) request-send
2) timeout (SPLIT_TIMEOUT)
3) check memory tags / flush dirty data
4) retry request-send

Figure 31 – Software fault-retry on coherent data

Although the error recovery is relatively inefficient, its infrequent use should have a minimal impact on system performance.

1.6.8 System debugging

A *trace* bit is provided to selectively enable *packet logging* as packets are routed through the system. Since a globally synchronized time-of-day clock is provided (see 3.4.6), packets can be accurately time-stamped as they are logged. The use of time stamps allows the route of the packet (at logging locations) to be reconstructed based on the log contents. The detailed implementation and use of the *trace* bit is beyond the scope of the SCI standard.

1.6.9 Alternate routing

On a single ringlet, the SCI protocols are intolerant to faults since one failure brings down the entire ringlet. However, redundant-ringlet systems are feasible. Switches or bridges between ringlets can isolate each ringlet from the failure of others.

Even though a ringlet has failed, its nodes could still be interrogated and diagnosed using a redundant low-cost diagnostic bus (Serial Bus). Although Serial Bus is not intended to be a redundant operational bus, it can assist in identifying the failed field-replaceable unit.

1.6.10 Online replacement

The SCI standard supports online replacement of modules, in that the full system need not be idled while a module is being replaced. Software is expected to isolate the module before it is replaced, taking account of any resources that that module was providing to the rest of the system. For example, coherently cached data has to be flushed to memory before a processor caching it can be replaced.

The physical specification section of the SCI standard defines mechanical and electrical interfaces that support online replacement. These specifications allow a module to be replaced without disrupting the electrical power supplied to other nodes in the system. The CSR Architecture defines the behaviour of modules during the on-line replacement process.

Replacing a module temporarily breaks the ringlet. A switch could isolate this ringlet from the remainder of the system while the module is being replaced. Alternatively, fault-recovery software could retry transactions that were lost while the module was being replaced. These ringlet-isolation and fault-recovery protocols are beyond the scope of the SCI standard.

2 References, glossary, and notation

2.1 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of IEC and ISO maintain registers of currently valid International Standards.

EIA IS-64 (1991), *2 mm Two-Part Connectors for Use with Printed Boards and Backplanes* [3)]

IEC 60793-1, *Optical fibres – Part 1: Generic specification* [4)]

IEC 60793-2, *Optical fibres – Part 2: Product specifications*

IEEE Std 1301-1991, *IEEE Standard for a Metric Equipment Practice for Microcomputers – Coordination Document) (ANSI)* [5)]

IEEE Std 1301.1-1991, *IEEE Standard for a Metric Equipment Practice for Microcomputers – Convection-Cooled with 2 mm Connectors (ANSI)*

ISO/IEC 13213:1994 [ANSI/IEEE Std 1212, 1994 Edition], *Information technology – Microprocessor systems – Control and Status Registers (CSA) Architecture for microcomputer buses* [6)]

ISO/IEC 9899:1990, *Programming languages – C*

[3)] EIA publications are available from Global Engineering, 1990 M Street NW, Suite 400, Washington, DC, 20036, USA.

[4)] IEC publications are available from IEC Customer Service Centre, Case postale 131, 3 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

[5)] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[6)] ISO publications are available from the ISO Central Secretariat, Case postale 56, 1 rue de Varembé, CH-1211 Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the American National Standards Institute.