ASX An Object Oriented Framework for Developing Distributed Applications

Douglas C. Schmidt
University of California, Irvine, CA

6th USENIX C++ Conference, Cambridge, MA

Presented by Venkat

OOAD – CSCI 5448
Prof. Kenneth Anderson

Agenda

- Distributed Computing and role of ASX
- Structure and functionality of the ASX framework
- ASX based implementations (TCP/IP protocol)
- Performance experiment results
- Concluding remarks

Distributed Computing

- Collaboration through connectivity and interworking
- Performance through parallel processing
- Reliability and availability through replication
- Scalability and portability through modularity
- Extensibility through dynamic (re)configuration
- Cost effectiveness through resource sharing

- Limitations with conventional tools and techniques used to develop distributed application software (more on next slide)

More on Limitations

- Conventional application development environments (such as UNIX, Windows NT, and OS/2) lack type-safe, portable, re-entrant, and extensible system call interfaces and component libraries.

- Techniques based upon algorithmic decomposition, which limit the extensibility, reusability, and portability of distributed applications.

What is ASX?

- Adaptive Service eXecutive (ASX) ASX is an object-oriented framework containing automated tools and reusable components that collaborate to simplify the development, configuration, and reconfiguration of distributed applications on shared memory multi-processor platforms.

- Facilitates application extensibility, component reuse, and performance enhancement

What does ASX do?

- ASX decouples application independent components from application specific components making it more extensible and portable (For example, it is possible to dynamic reconfigure one or more services in an ASX-based application without requiring the modification, recompilation, re-linking, or restarting of a running system).

- “Service” - primary unit of configuration in the ASX framework
  - Services may be simple (such as returning the current time-of-day) or highly complex (such as a real-time distributed PBX event traffic monitor).

What is a framework?

- A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications

    - The components in a framework typically include classes, class hierarchies, class categories and object

    - Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration process associated with complex application domains such as graphical user interfaces, databases, operating system kernels, and communication subsystems.

Object Oriented (OO) architecture of ASX

- ASX provides the following class categories
    - Stream class category
    - Reactor class category
    - Service Configurator class category
    - Concurrency class category
    - IPC SAP class category

- A complete distributed application may be formed by combining components in each of the following class categories via C++ language features such as inheritance, aggregation, and template instantiation

OO architecture of ASX

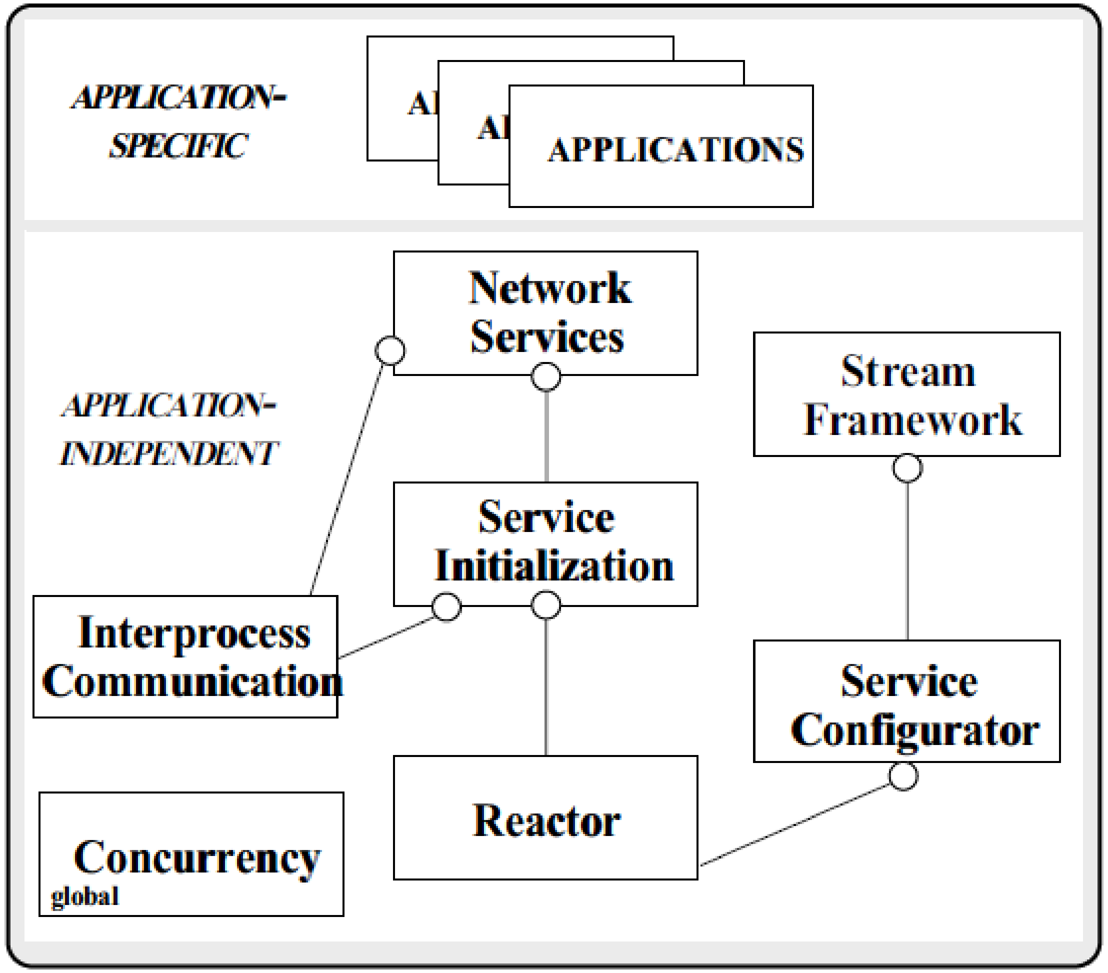


Figure 1: Class Categories in the ASX Framework

Relationships between components (refer previous slide for example)

- Solid rectangles indicate class categories
- Solid clouds indicate objects
- Nesting indicates composition relationships between objects
- Undirected edges indicate some type of link exists between two objects
- Dashed clouds indicate classes
- Directed edges indicate inheritance relationships between classes
- Undirected edge with a small circle at one end indicates either a composition or uses relation between two classes

Stream Class Category

- Responsible for coordinating one or more Streams (Stream is an object used to configure and execute application-specific services into the ASX framework)

- Primary components of the ASX Stream class category
  - Stream Class
  - Module Class
  - Task Class
  - Multiplexer Class

ASX Stream Class Category

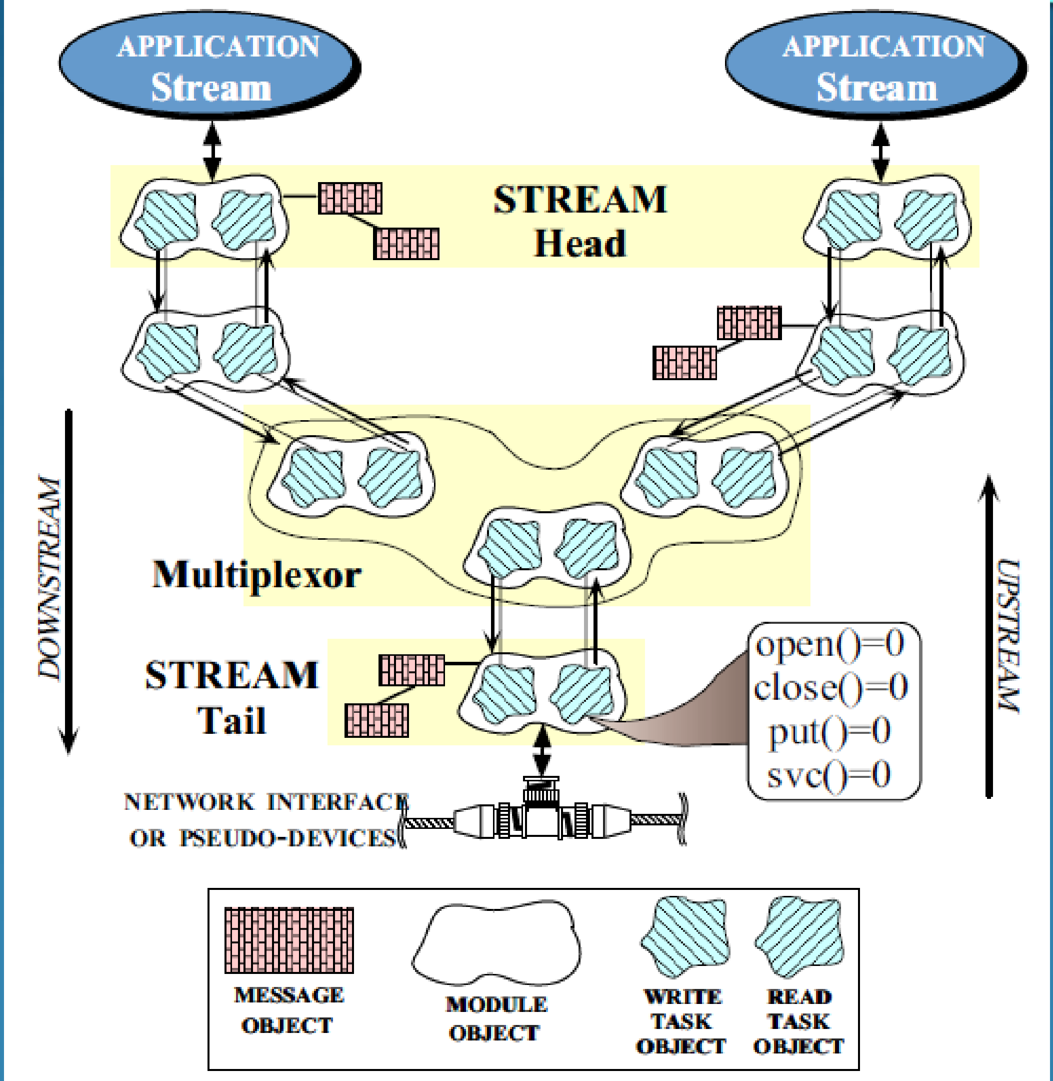


Figure 2: Components in the `Stream` Class Category

Stream Class Category

- Stream class object:
  - Provides a bi-directional get/put-style interface
  - Allows applications to access a stack of one or more hierarchically-related service Modules.
  - Applications send and receive data and control messages through the inter-connected Modules that comprise a particular Stream object.

- Module class:
  - A Stream is formed by inter-connecting a series of Module objects.
  - Module objects in a Stream are loosely coupled, and collaborate with adjacent Module objects by passing typed messages.
  - Each Module object contains a pair of pointers to objects that are service-specific subclasses of the Task class

Stream Class Category

- Task class:
  - One Task subclass handles read-side processing for messages sent upstream to its Module layer
  - Other Task subclass handles write-side processing messages send downstream to its Module layer.
  - Defines four pure virtual methods (open, close, put, and svc).
    - open() and close() for performing service-specific Task initialization and termination activities.
    - put() is invoked when a Task at one layer in a Stream passes a message to an adjacent Task in another layer.
    - svc() method is used to perform service-specific processing asynchronously with respect to other Tasks in its Stream.

- Multiplexer class:
  - Multiplexors are used to route Message Blocks between inter-related streams (such as those used to implement complex protocol families in the Internet and the ISO OSI reference models).

Reactor Class Category

- Responsible for de-multiplexing
  (1) temporal events generated by a timer driven callout queue
  (2) I/O events received on communication ports, and
  (3) signal events

  and dispatching the appropriate pre-registered handler(s) to process these events.

- Reactor encapsulates the functionality of the select and poll I/O de-multiplexing mechanisms within a portable and extensible C++ wrapper

Reactor Class Category

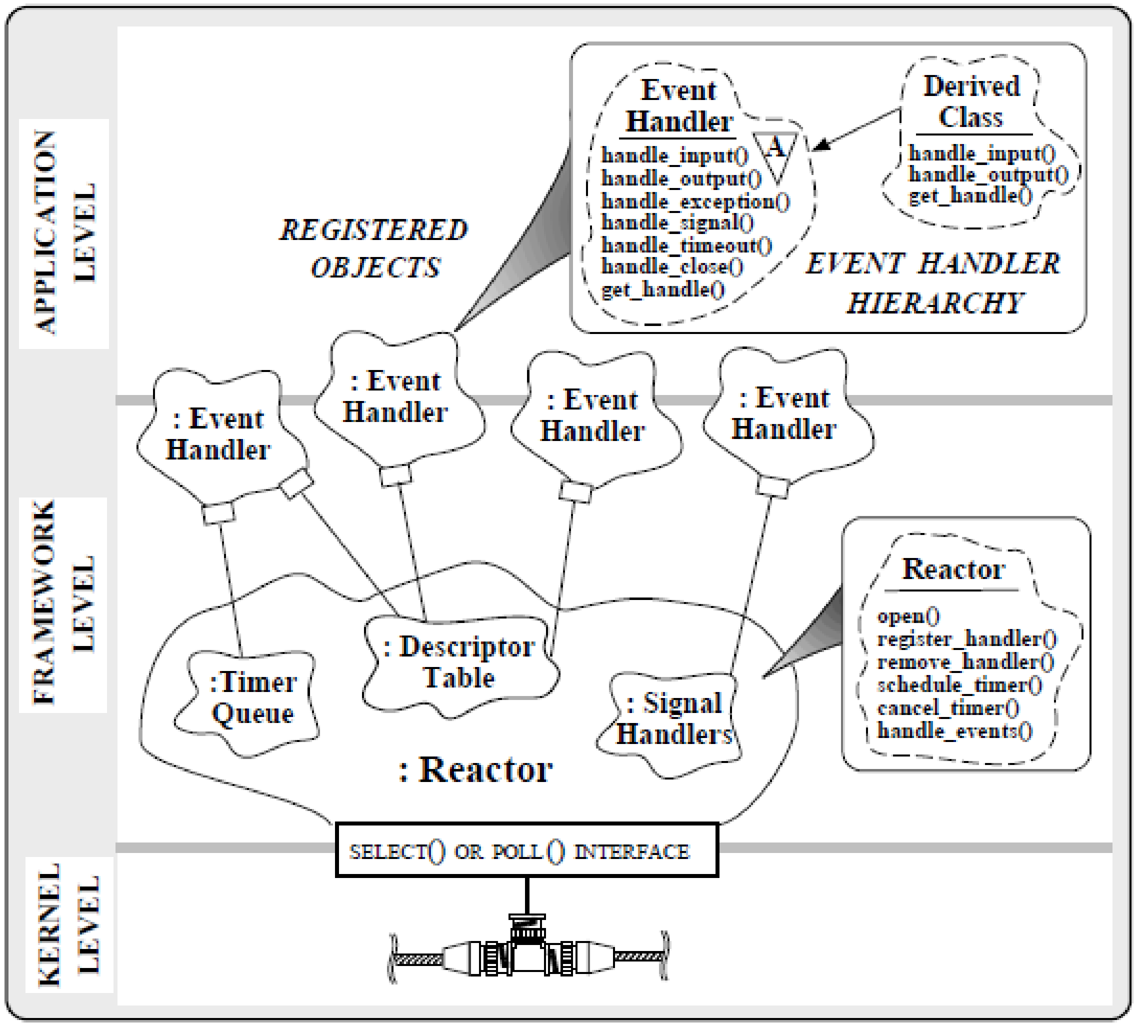


Figure 4: Components in the Reactor Class Category

Reactor Class Category

- The Reactor contains a set of methods(refer previous slide):
  - Certain methods register, dispatch, and remove I/O descriptor-based and signal-based handler objects from the Reactor.
  - Other methods schedule, cancel, and dispatch timer-based handler objects.
  - These handler objects all derive from the Event Handler abstract base class(which specifies an interface for event registration and service handler dispatching).

- When an application instantiates and registers a composite I/O descriptor-based service handler object, the Reactor extracts the underlying I/O descriptor from the object. This descriptor is stored in a table along with I/O descriptors from other registered objects.

Service Configurator class category

- Responsible for explicitly linking or unlinking services dynamically into or out of the address space of an application at run-time.

- The Service Configurator components include …

  - Service Object Inheritance hierarchy (flexibility in inserting and removing services from an application at runtime)

  - Service Repository class (collectively control and coordinate the Service Objects that comprise an application's currently active services)

  - Service Config class (uses a configuration file to guide its configuration and reconfiguration activities)

Service Configurator class category

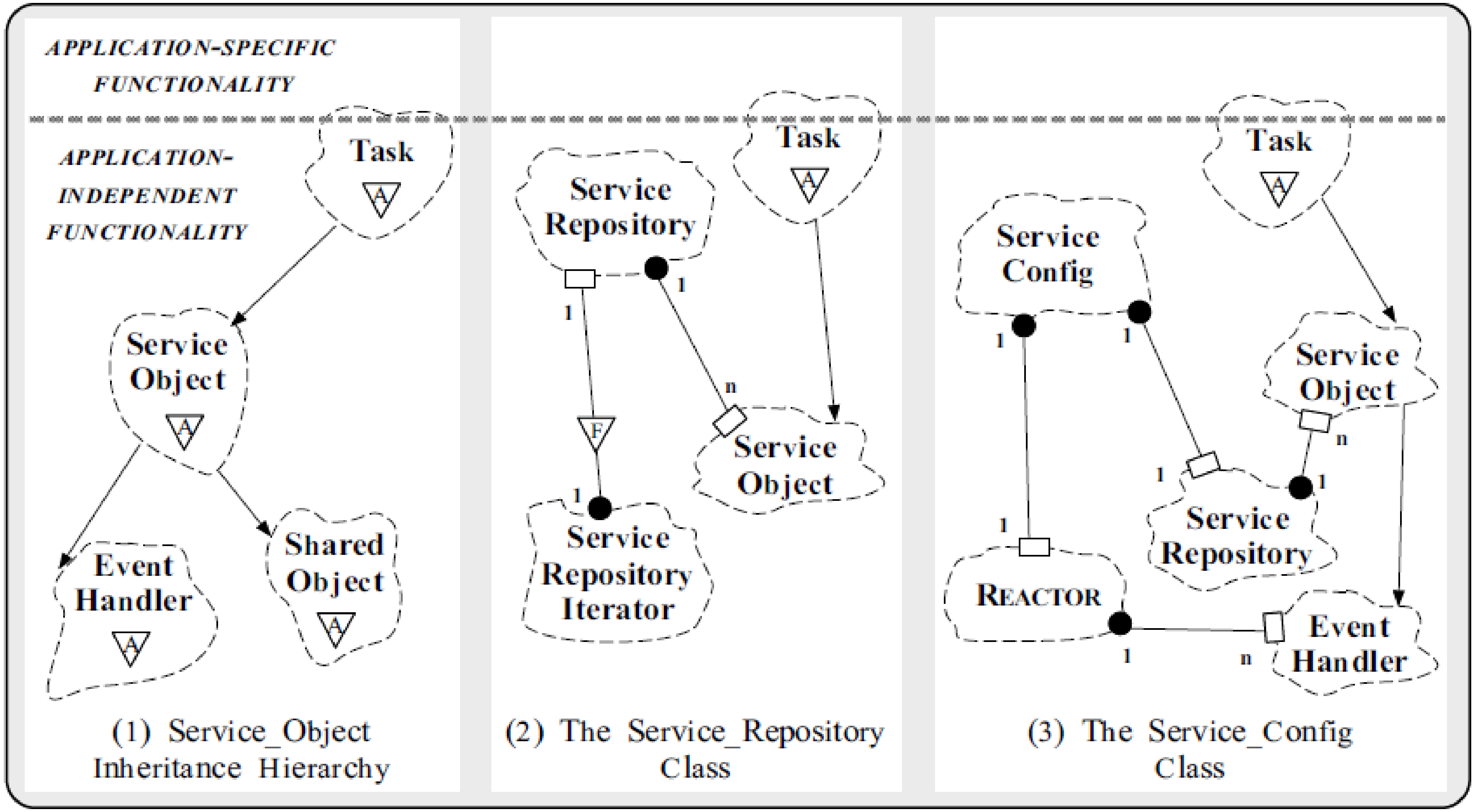

APPLICATION-SPECIFIC FUNCTIONALITY
APPLICATION-INDEPENDENT FUNCTIONALITY
Task
A
Service Object
A
Event Handler
A
Shared Object
A
(1) Service_Object Inheritance Hierarchy
Service Repository
1
1
Task
A
F
n
1
Service Object
Service Repository Iterator
(2) The Service_Repository Class
Task
A
Service Config
1
1
Service Object
n
1
1
1
Service Repository
REACTOR
1
n
Event Handler
(3) The Service_Config Class

Figure 5: Components in the `Service Configurator` Class Category

Concurrency Class Category

- Responsible for spawning, executing, synchronizing, and gracefully terminating services at run-time via one or more threads of control within one or more processes

- The concurrency class category includes ..
  - The Synch class (Mutex, Condition, etc.)
  - The thread manager class (suspend_all, resume_all, etc.)

IPC SAP Class Category

- Encapsulates standard OS local and remote IPC mechanisms (such as sockets and TLI) within a type-safe and portable object-oriented interface.
- IPC SAP stands for “Inter-Process Communication Service Access Point.”
- Forest of class categories are rooted at the IPC SAP base class (next slide)
- Class categories includes SOCK SAP (encapsulates the socket API), TLI SAP (encapsulates the TLI API), SPIPE SAP (encapsulates the UNIX SVR4 STREAM pipe API), and FIFO SAP (encapsulates the UNIX named pipe API).

IPC SAP Class Category

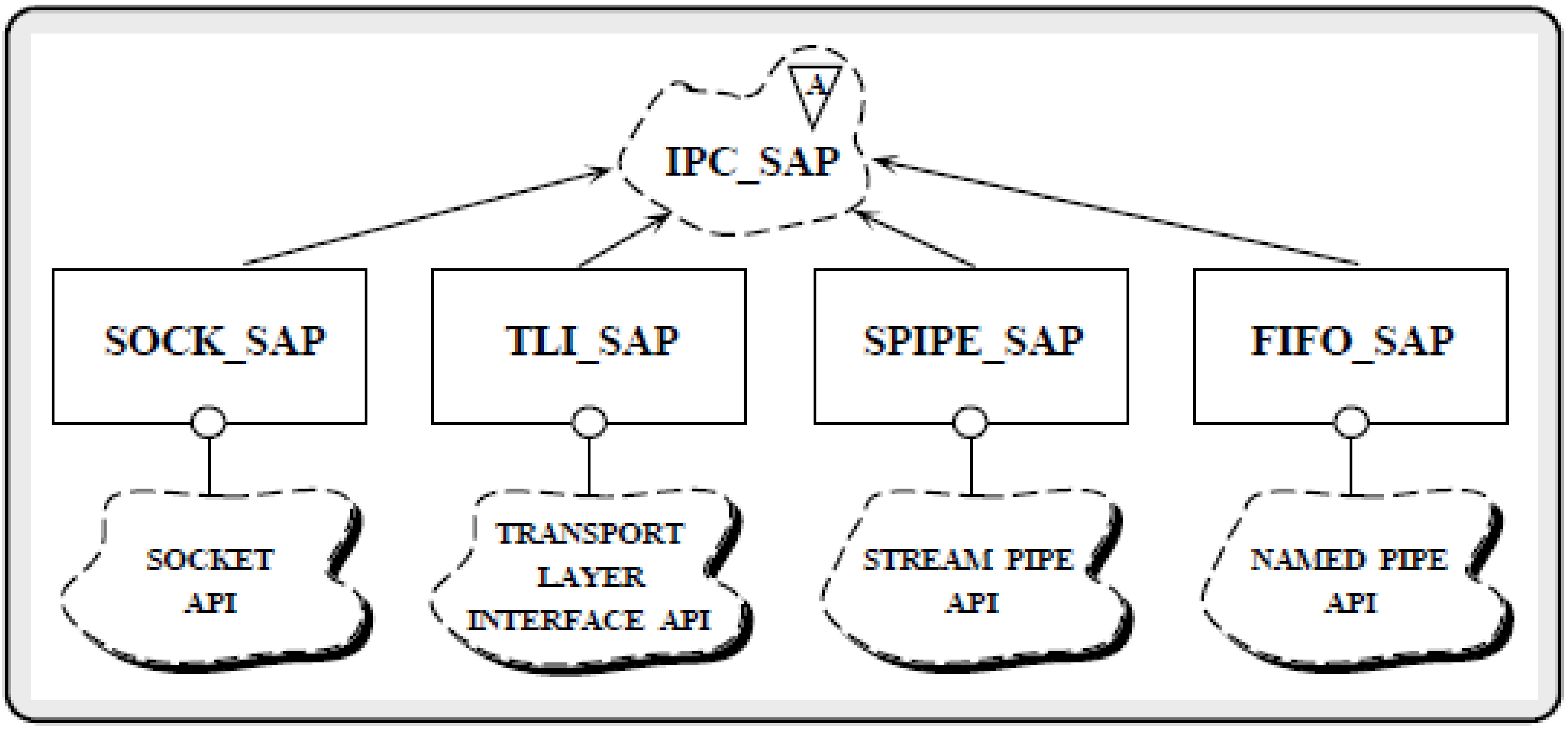


Figure 6: Components in the IPC SAP Class Category

Performance Experiments on the Communication Subsystem

- Communication subsystem comprises of …
  - Protocol functions (e.g. routing)
  - Operating system mechanics (e.g. memory management)

- Three basic elements for a process architecture
  - Control & data messages between applications
  - Protocol processing task units
  - Processing elements executing above task units

- Two basic process architectures (next slide)
  - Task-based
  - Message-based

Process Architecture

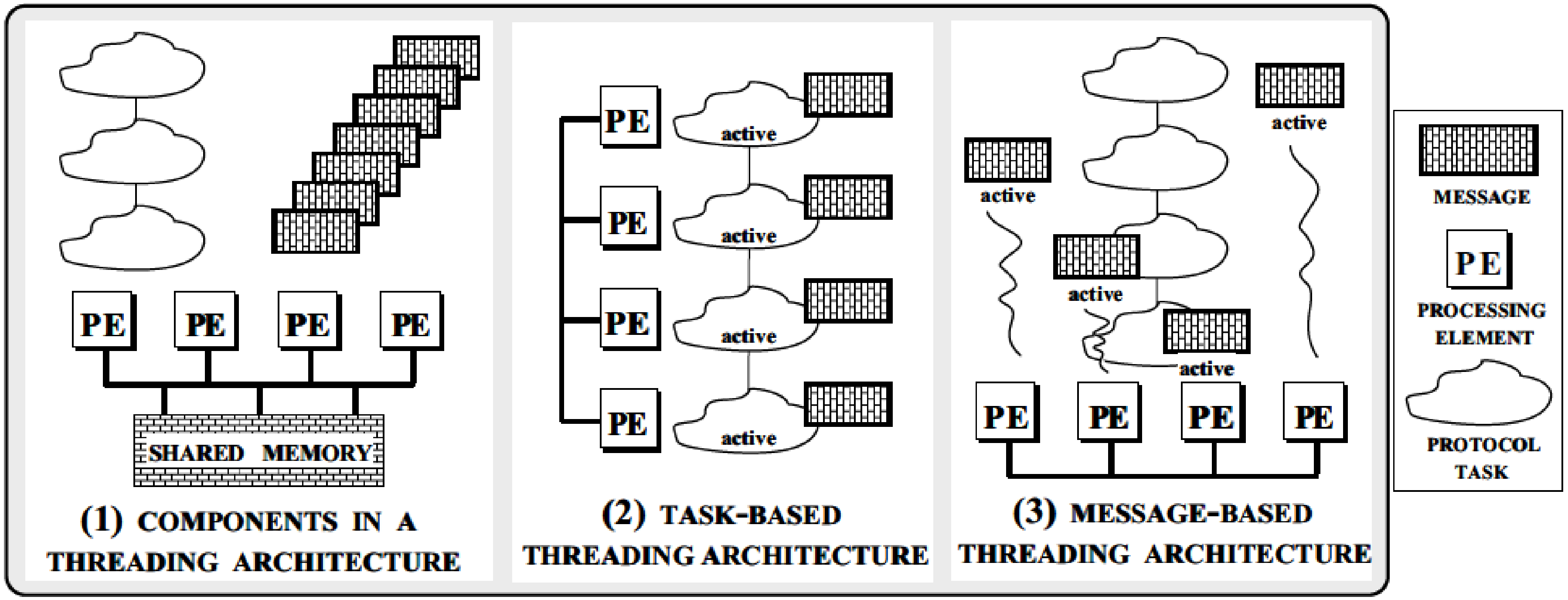


Figure 7: Process Architecture Components and Interrelationships

Performance Experiments on the Communication Subsystem

- Multiprocessor platform used for testing:
  - Sun 690MP SPARCserver, which contains 4 SPARC 40 MHz processing elements (PEs), each capable of performing at 28MIPs.
  - The operating system used for the experiments is release 5.3 of SunOS, which provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel

Performance Experiments on the Communication Subsystem

- Communication protocols
  - Two types of protocol stacks were used
    1) connectionless UDP transport protocol
    2) connection oriented TCP protocol

- The protocol stacks developed contain the data-link, transport, and presentation layers

- Protocol stacks were developed by specializing existing components in the ASX framework via techniques involving inheritance and parameterized types

Performance Experiments on the Communication Subsystem

- Task-based Process Architecture (Fig. 8 next slide)
    - Protocol-specific processing for the data-link and transport layer are performed in two Modules clustered together into one thread.
    - Presentation layer and application interface processing is performed in two Modules clustered into a separate thread.
    - These threads cooperate in a producer/consumer manner, operating in parallel on the header and data fields of multiple incoming and outgoing messages

Task-based Process Architecture

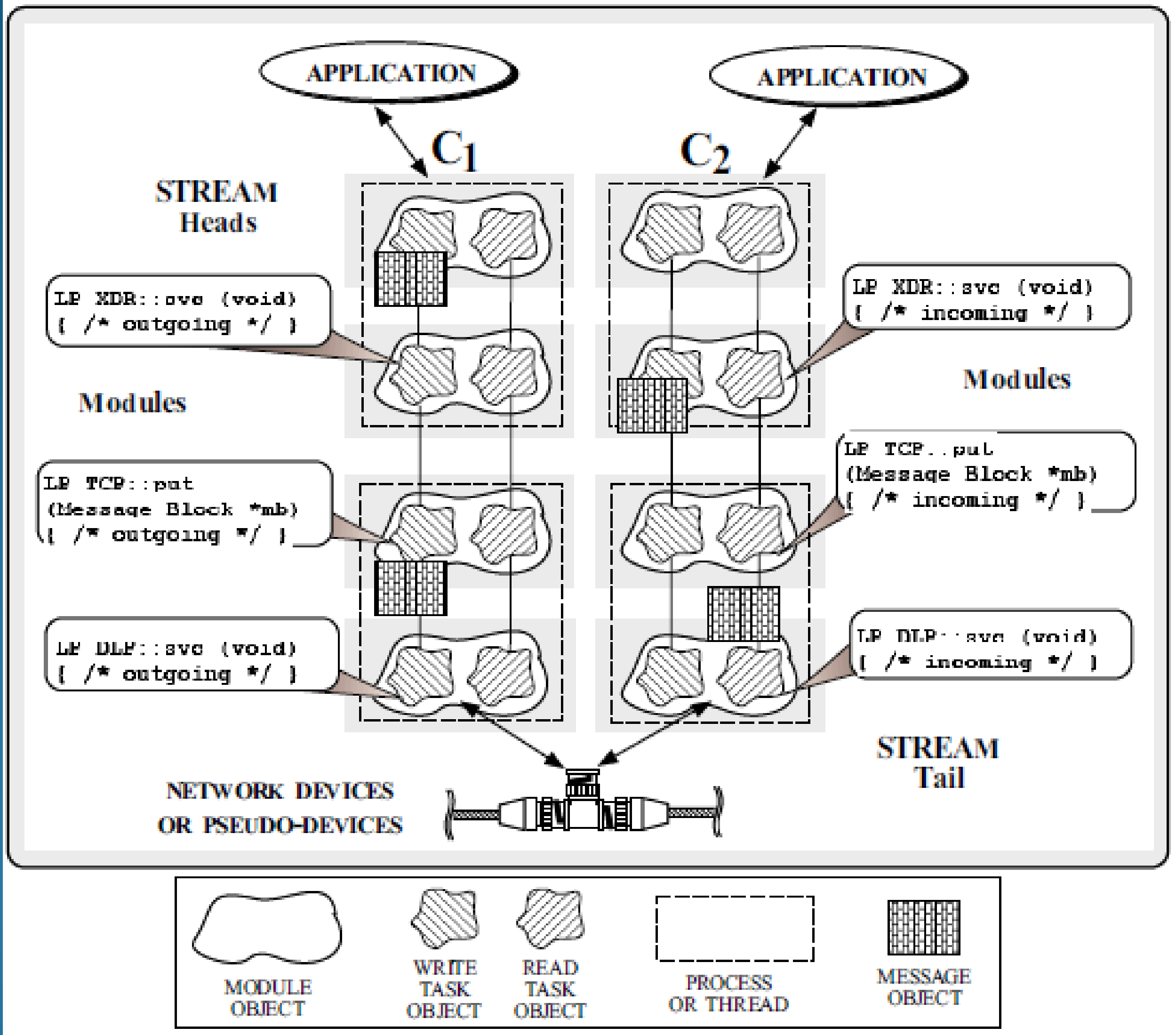

APPLICATION
APPLICATION
C1
C2
STREAM Heads
LP XDR::svc (void) { /* outgoing */ }
LP XDR::svc (void) { /* incoming */ }
Modules
Modules
LP TCP::put (Message Block *mb) { /* outgoing */ }
LP TCP::put (Message Block *mb) { /* incoming */ }
LP DLP::svc (void) { /* outgoing */ }
LP DLP::svc (void) { /* incoming */ }
STREAM Tail
NETWORK DEVICES OR PSEUDO-DEVICES
MODULE OBJECT
WRITE TASK OBJECT
READ TASK OBJECT
PROCESS OR THREAD
MESSAGE OBJECT

Figure 8: A Task-based Process Architecture

Performance Experiments on the Communication Subsystem

- Message-based Process Architecture (Fig. 9)
  - An incoming message is handled by the MP DLP::svc method, which manages a pool of pre-spawned threads.
  - Each message is associated with a separate thread that escorts the message synchronously through a series of interconnected Tasks in a Stream.
  - Each layer of the protocol stack performs its protocol functions and then makes an up-call to the next adjacent layer in the protocol stack by invoking the Task::put method in that layer.
  - The put method executes the protocol tasks associated with its layer.

Message-based Process Architecture

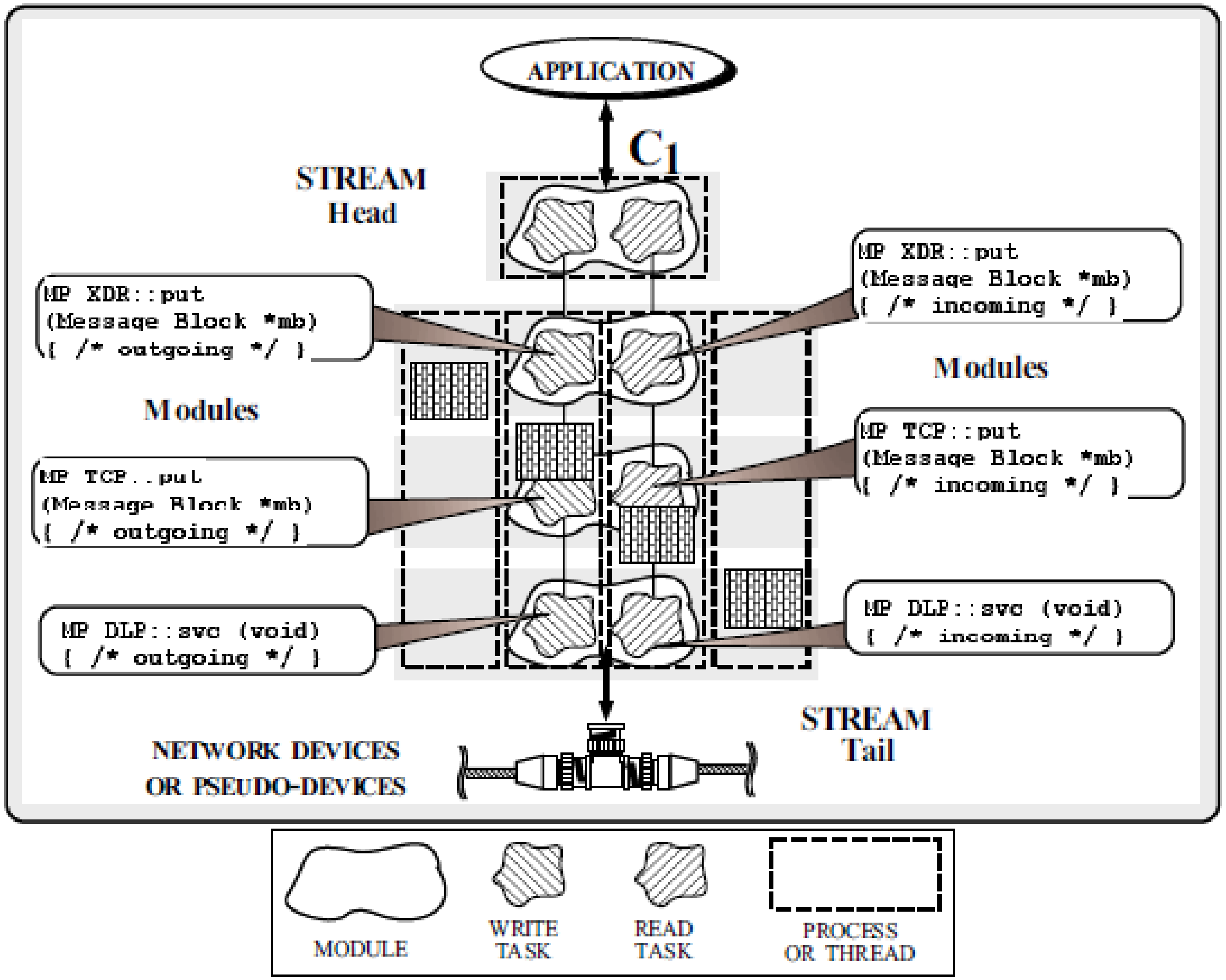

APPLICATION
C1
STREAM Head
MP XDR::put (Message Block *mb) { /* outgoing */ }
Modules
MP TCP::put (Message Block *mb) { /* outgoing */ }
MP DLP::svc (void) { /* outgoing */ }
MP XDR::put (Message Block *mb) { /* incoming */ }
Modules
MP TCP::put (Message Block *mb) { /* incoming */ }
MP DLP::svc (void) { /* incoming */ }
NETWORK DEVICES OR PSEUDO-DEVICES
STREAM Tail
MODULE
WRITE TASK
READ TASK
PROCESS OR THREAD

Figure 9: A Message-based Process Architecture

Process Architecture Experiment Results

- Three types of measurements were obtained for each combination of process architecture and protocol stack
  - Total throughput
  - Context switching overhead, and
  - Synchronization overhead

- The charts in the next few slides depict these results (they are self explanatory).
  - CO – connection oriented
  - CL - Connectionless

Process Architecture Throughput

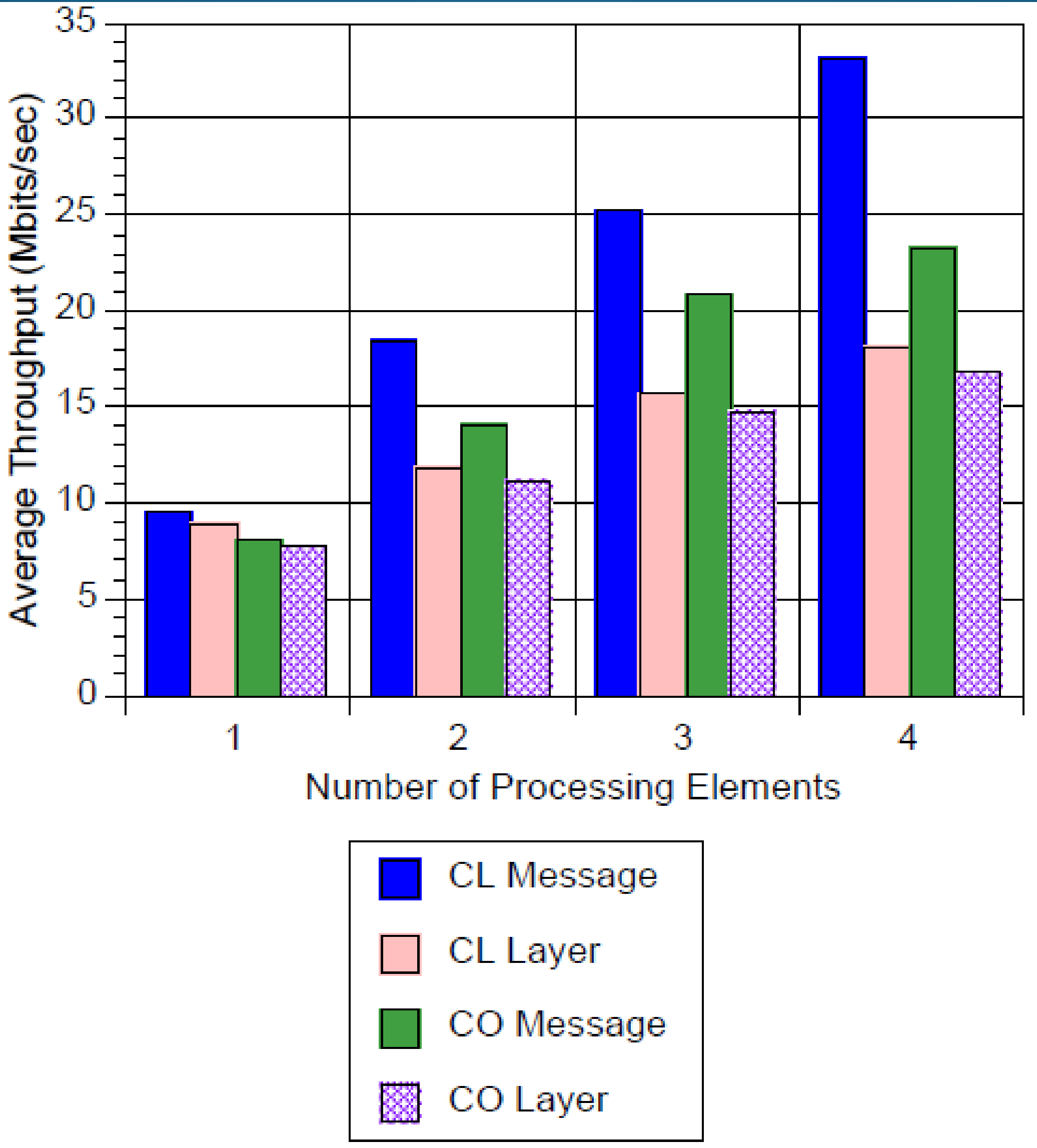
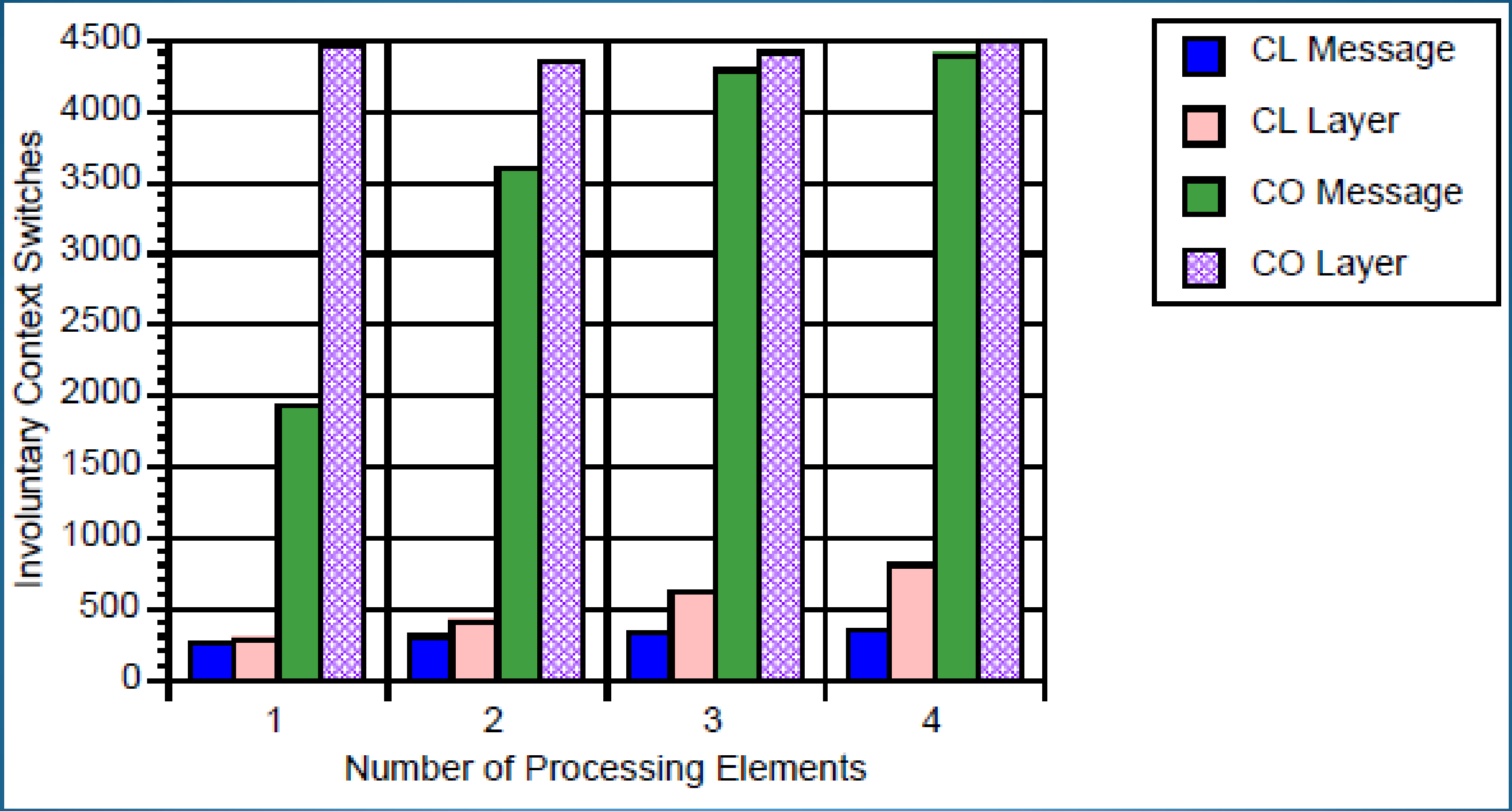
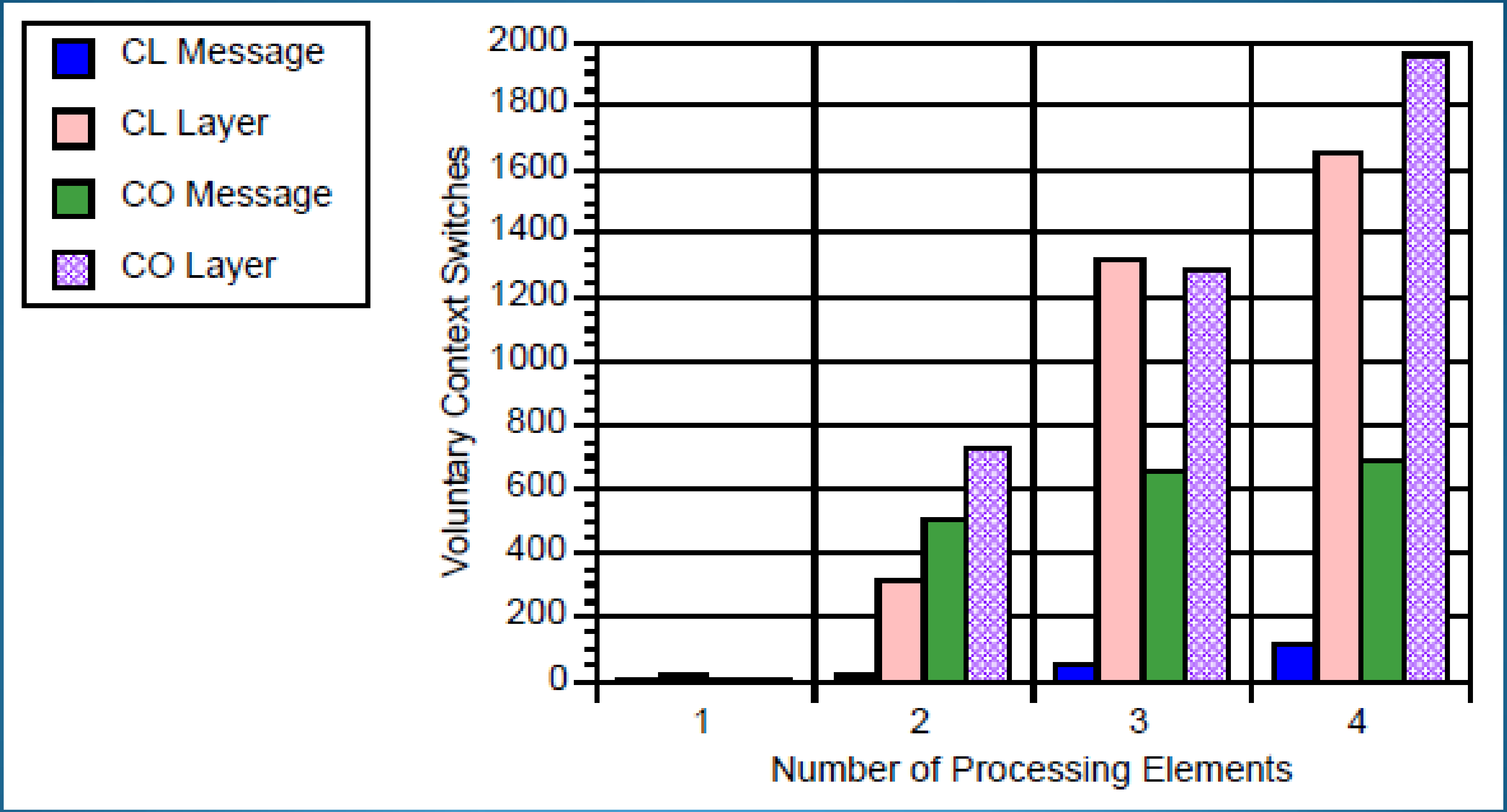


Figure 10: Process Architecture Throughput

Process Architecture Involuntary Context Switching



Process Architecture Voluntary Context Switching



Process Architecture Locking Overhead

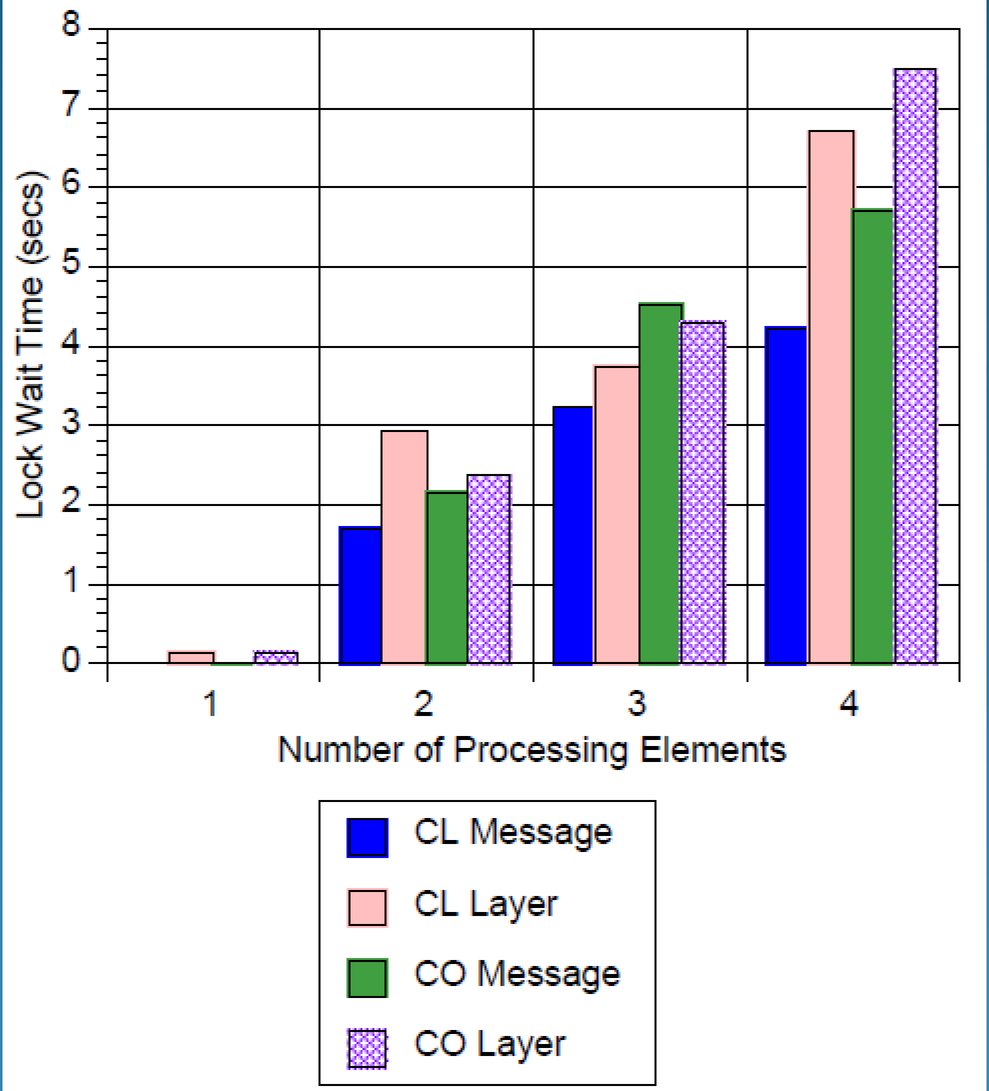


Figure 12: Process Architecture Locking Overhead

Concluding remarks

- Developing distributed applications that effectively utilize parallel processing remains a complex and challenging task

- ASX provides an extensible object-oriented framework that simplifies the development of distributed applications on shared memory multi-processor platforms

- A key aspect of concurrent distributed application performance involves the type of process architecture selected

Concluding remarks

• Task-based process architectures incur relatively high levels of context switching and synchronization overhead when compared to message-based architectures

• Components in the ASX framework have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products

• More on this topic? http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.4884

Thanks

Questions.