Chapter 1: Introduction

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press
Definition

- Autonomous processors communicating over a communication network
- Some characteristics
  - No common physical clock
  - No shared memory
  - Geographical separation
  - Autonomy and heterogeneity
Distributed System Model

Figure 1.1: A distributed system connects processors by a communication network.
Relation between Software Components

Figure 1.2: Interaction of the software components at each process.
Motivation for Distributed System

- Inherently distributed computation
- Resource sharing
- Access to remote resources
- Increased performance/cost ratio
- Reliability
  - availability, integrity, fault-tolerance
- Scalability
- Modularity and incremental expandability
Parallel Systems

- **Multiprocessor systems** (direct access to shared memory, UMA model)
  - Interconnection network - bus, multi-stage switch
  - E.g., Omega, Butterfly, Clos, Shuffle-exchange networks
  - Interconnection generation function, routing function

- **Multicomputer parallel systems** (no direct access to shared memory, NUMA model)
  - bus, ring, mesh (w w/o wraparound), hypercube topologies
  - E.g., NYU Ultracomputer, CM* Connection Machine, IBM Blue gene

- **Array processors** (colocated, tightly coupled, common system clock)
  - Niche market, e.g., DSP applications
**UMA vs. NUMA Models**

Figure 1.3: Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.
Figure 1.4: Interconnection networks for shared memory multiprocessor systems. (a) Omega network (b) Butterfly network.
Omega Network

- $n$ processors, $n$ memory banks
- $\log n$ stages: with $n/2$ switches of size 2x2 in each stage
- Interconnection function: Output $i$ of a stage connected to input $j$ of next stage:
  \[ j = \begin{cases} 
  2i & \text{for } 0 \leq i \leq n/2 - 1 \\
  2i + 1 - n & \text{for } n/2 \leq i \leq n - 1
  \end{cases} \]
- Routing function: in any stage $s$ at any switch:
  to route to dest. $j$,
  if $s + 1$th MSB of $j = 0$ then route on upper wire
  else [s + 1th MSB of $j = 1$] then route on lower wire
Interconnection Topologies for Multiprocessors

Figure 1.5: (a) 2-D Mesh with wraparound (a.k.a. torus) (b) 3-D hypercube
Flynn’s Taxonomy

Figure 1.6: SIMD, MISD, and MIMD modes.

- **SISD**: Single Instruction Stream Single Data Stream (traditional)
- **SIMD**: Single Instruction Stream Multiple Data Stream
  - scientific applications, applications on large arrays
  - vector processors, systolic arrays, Pentium/SSE, DSP chips
- **MISD**: Multiple Instruction Stream Single Data Stream
  - E.g., visualization
- **MIMD**: Multiple Instruction Stream Multiple Data Stream
  - distributed systems, vast majority of parallel systems
Terminology

- **Coupling**
  - Interdependency/binding among modules, whether hardware or software (e.g., OS, middleware)

- **Parallelism:** $T(1)/T(n)$.
  - Function of program and system

- **Concurrency of a program**
  - Measures productive CPU time vs. waiting for synchronization operations

- **Granularity of a program**
  - Amt. of computation vs. amt. of communication
  - Fine-grained program suited for tightly-coupled system
Message-passing vs. Shared Memory

- **Emulating MP over SM:**
  - Partition shared address space
  - Send/Receive emulated by writing/reading from special mailbox per pair of processes

- **Emulating SM over MP:**
  - Model each shared object as a process
  - Write to shared object emulated by sending message to owner process for the object
  - Read from shared object emulated by sending query to owner of shared object
Classification of Primitives (1)

- Synchronous (send/receive)
  - Handshake between sender and receiver
  - Send completes when Receive completes
  - Receive completes when data copied into buffer

- Asynchronous (send)
  - Control returns to process when data copied out of user-specified buffer
Classification of Primitives (2)

- **Blocking (send/receive)**
  - Control returns to invoking process after processing of primitive (whether sync or async) completes

- **Nonblocking (send/receive)**
  - Control returns to process immediately after invocation
  - Send: even before data copied out of user buffer
  - Receive: even before data may have arrived from sender
Non-blocking Primitive

\[
\text{Send}(X, \text{destination}, \text{handle}_k) \quad // \text{handle}_k \text{ is a return parameter}
\]

\[
\quad ...
\]

\[
\quad ...
\]

\[
\text{Wait}(\text{handle}_1, \text{handle}_2, \ldots, \text{handle}_k, \ldots, \text{handle}_m) \quad // \text{Wait always blocks}
\]

Figure 1.7: A nonblocking \textit{send} primitive. When the \textit{Wait} call returns, at least one of its parameters is posted.

- Return parameter returns a system-generated handle
  - Use later to check for status of completion of call
  - Keep checking (loop or periodically) if handle has been posted
  - Issue \text{Wait}(\text{handle}_1, \text{handle}_2, \ldots) call with list of handles
  - Wait call blocks until one of the stipulated handles is posted
Blocking/nonblocking; Synchronous/asynchronous; send/receive primitives

Figure 1.8: Illustration of 4 send and 2 receive primitives
Asynchronous Executions; Message-passing System

Figure 1.9: Asynchronous execution in a message-passing system
Synchronous Executions: Message-passing System

Figure 1.10: Synchronous execution in a message-passing system

In any round/step/phase: \((send \mid internal)^* (receive \mid internal)^*\)

(1) \(Sync\_Execution(int \ k, \ n) \) // \(k\) rounds, \(n\) processes.
(2) \(\text{for } r = 1 \text{ to } k \text{ do}\)
(3) proc \(i\) sends msg to \((i + 1) \mod n\) and \((i - 1) \mod n\);
(4) each proc \(i\) receives msg from \((i + 1) \mod n\) and \((i - 1) \mod n\);
(5) compute app-specific function on received values.
Synchronous vs. Asynchronous Executions (1)

- **Sync vs async processors; Sync vs async primitives**
- **Sync vs async executions**
- **Async execution**
  - No processor synchrony, no bound on drift rate of clocks
  - Message delays finite but unbounded
  - No bound on time for a step at a process
- **Sync execution**
  - Processors are synchronized; clock drift rate bounded
  - Message delivery occurs in one logical step/round
  - Known upper bound on time to execute a step at a process
Synchronous vs. Asynchronous Executions (2)

- Difficult to build a truly synchronous system; can simulate this abstraction
- Virtual synchrony:
  - async execution, processes synchronize as per application requirement;
  - execute in rounds/steps
- Emulations:
  - Async program on sync system: trivial (A is special case of S)
  - Sync program on async system: tool called synchronizer
System Emulations

Figure 1.11: Sync ↔ async, and shared memory ↔ msg-passing emulations

- Assumption: failure-free system
- System A emulated by system B:
  - If not solvable in B, not solvable in A
  - If solvable in A, solvable in B
Challenges: System Perspective (1)

- Communication mechanisms: E.g., Remote Procedure Call (RPC), remote object invocation (ROI), message-oriented vs. stream-oriented communication
- Processes: Code migration, process/thread management at clients and servers, design of software and mobile agents
- Naming: Easy to use identifiers needed to locate resources and processes transparently and scalably
- Synchronization
- Data storage and access
  - Schemes for data storage, search, and lookup should be fast and scalable across network
  - Revisit file system design
- Consistency and replication
  - Replication for fast access, scalability, avoid bottlenecks
  - Require consistency management among replicas
Challenges: System Perspective (2)

- Fault-tolerance: correct and efficient operation despite link, node, process failures
- Distributed systems security
  - Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
- Scalability and modularity of algorithms, data, services
- Some experimental systems: Globe, Globus, Grid
Challenges: System Perspective (3)

- API for communications, services: ease of use
- Transparency: hiding implementation policies from user
  - Access: hide differences in data rep across systems, provide uniform operations to access resources
  - Location: locations of resources are transparent
  - Migration: relocate resources without renaming
  - Relocation: relocate resources as they are being accessed
  - Replication: hide replication from the users
  - Concurrency: mask the use of shared resources
  - Failure: reliable and fault-tolerant operation
Challenges: Algorithm/Design (1)

- Useful execution models and frameworks: to reason with and design correct distributed programs
  - Interleaving model
  - Partial order model
  - Input/Output automata
  - Temporal Logic of Actions

- Dynamic distributed graph algorithms and routing algorithms
  - System topology: distributed graph, with only local neighborhood knowledge
  - Graph algorithms: building blocks for group communication, data dissemination, object location
  - Algorithms need to deal with dynamically changing graphs
  - Algorithm efficiency: also impacts resource consumption, latency, traffic, congestion
Challenges: Algorithm/Design (2)

- Time and global state
  - 3D space, 1D time
  - Physical time (clock) accuracy
  - Logical time captures inter-process dependencies and tracks relative time progression
  - Global state observation: inherent distributed nature of system
  - Concurrency measures: concurrency depends on program logic, execution speeds within logical threads, communication speeds
Challenges: Algorithm/Design (3)

- **Synchronization/coordination mechanisms**
  - Physical clock synchronization: hardware drift needs correction
  - Leader election: select a distinguished process, due to inherent symmetry
  - Mutual exclusion: coordinate access to critical resources
  - Distributed deadlock detection and resolution: need to observe global state; avoid duplicate detection, unnecessary aborts
  - Termination detection: global state of quiescence; no CPU processing and no in-transit messages
  - Garbage collection: Reclaim objects no longer pointed to by any process
Challenges: Algorithm/Design (4)

- Group communication, multicast, and ordered message delivery
  - Group: processes sharing a context, collaborating
  - Multiple joins, leaves, fails
  - Concurrent sends: semantics of delivery order

- Monitoring distributed events and predicates
  - Predicate: condition on global system state
  - Debugging, environmental sensing, industrial process control, analyzing event streams

- Distributed program design and verification tools

- Debugging distributed programs
Challenges: Algorithm/Design (5)

- Data replication, consistency models, and caching
  - Fast, scalable access;
  - coordinate replica updates;
  - optimize replica placement
- World Wide Web design: caching, searching, scheduling
  - Global scale distributed system; end-users
  - Read-intensive; prefetching over caching
  - Object search and navigation are resource-intensive
  - User-perceived latency
Challenges: Algorithm/Design (6)

- Distributed shared memory abstraction
  - Wait-free algorithm design: process completes execution, irrespective of actions of other processes, i.e., $n-1$ fault-resilience
  - Mutual exclusion
    - Bakery algorithm, semaphores, based on atomic hardware primitives, fast algorithms when contention-free access
  - Register constructions
    - Revisit assumptions about memory access
    - What behavior under concurrent unrestricted access to memory?
      - Foundation for future architectures, decoupled with technology (semiconductor, biocomputing, quantum . . .)
  - Consistency models:
    - coherence versus access cost trade-off
    - Weaker models than strict consistency of uniprocessors
Challenges: Algorithm/Design (7)

- Reliable and fault-tolerant distributed systems
  - Consensus algorithms: processes reach agreement in spite of faults (under various fault models)
  - Replication and replica management
  - Voting and quorum systems
  - Distributed databases, commit: ACID properties
  - Self-stabilizing systems: "illegal" system state changes to "legal" state; requires built-in redundancy
  - Checkpointing and recovery algorithms: roll back and restart from earlier "saved" state
  - Failure detectors:
    - Difficult to distinguish a "slow" process/message from a failed process/never sent message
    - Algorithms that "suspect" a process as having failed and converge on a determination of its up/down status
Challenges: Algorithm/Design (8)

- Load balancing: to reduce latency, increase throughput, dynamically. E.g., server farms
  - Computation migration: relocate processes to redistribute workload
  - Data migration: move data, based on access patterns
  - Distributed scheduling: across processors
- Real-time scheduling: difficult without global view, network delays make task harder
- Performance modeling and analysis: Network latency to access resources must be reduced
  - Metrics: theoretical measures for algorithms, practical measures for systems
  - Measurement methodologies and tools
Applications and Emerging Challenges (1)

- **Mobile systems**
  - Wireless communication: unit disk model; broadcast medium (MAC), power management etc.
  - CS perspective: routing, location management, channel allocation, localization and position estimation, mobility management
  - Base station model (cellular model)
  - Ad-hoc network model (rich in distributed graph theory problems)

- **Sensor networks**: Processor with electro-mechanical interface

- **Ubiquitous or pervasive computing**
  - Processors embedded in and seamlessly pervading environment
  - Wireless sensor and actuator mechanisms; self-organizing; network-centric, resource-constrained
  - E.g., intelligent home, smart workplace
Applications and Emerging Challenges (2)

- Peer-to-peer computing
  - No hierarchy; symmetric role; self-organizing; efficient object storage and lookup; scalable; dynamic reconfig

- Publish/subscribe, content distribution
  - Filtering information to extract that of interest

- Distributed agents
  - Processes that move and cooperate to perform specific tasks; coordination, controlling mobility, software design and interfaces

- Distributed data mining
  - Extract patterns/trends of interest
  - Data not available in a single repository
Applications and Emerging Challenges (3)

- Grid computing
  - Grid of shared computing resources; use idle CPU cycles
  - Issues: scheduling, QOS guarantees, security of machines and jobs

- Security
  - Confidentiality, authentication, availability in a distributed setting
  - Manage wireless, peer-to-peer, grid environments
    - Issues: e.g., Lack of trust, broadcast media, resource-constrained, lack of structure