Contents

- Parallel computing.
- Concurrency.
- Parallelism levels; parallel computer architecture.
- Distributed systems.
- Processes, threads, events, and communication channels.
- Global state of a process group.
- Logical clocks.
- Runs and cuts.
- The snapshot protocol.
- Atomic actions.
- Consensus algorithms.
- Modeling concurrency with Petri Nets.
- Client-server paradigm.
The path to cloud computing

- Cloud computing is based on ideas and the experience accumulated in many years of research in parallel and distributed systems.
  - Cloud applications are based on the client-server paradigm with a relatively simple software, a thin-client, running on the user's machine, while the computations are carried out on the cloud.
  - Concurrency is important; many cloud applications are data-intensive and use a number of instances which run concurrently.
  - Checkpoint-restart procedures are used as many cloud computations run for extended periods of time on multiple servers. Checkpoints are taken periodically in anticipation of the need to restart a process when one or more systems fail.
  - Communication is at the heart of cloud computing. Communication protocols which support coordination of distributed processes travel through noisy and unreliable communication channels which may lose messages or deliver duplicate, distorted, or out of order messages.
Parallel computing

- Parallel hardware and software systems allow us to:
  - Solve problems demanding resources not available on a single system.
  - Reduce the time required to obtain a solution.
- The *speed-up* $S$ measures the effectiveness of parallelization:
  \[
  S(N) = \frac{T(1)}{T(N)}
  \]
  $T(1) \rightarrow$ the execution time of the sequential computation.
  $T(N) \rightarrow$ the execution time when $N$ parallel computations are executed.
- **Amdahl's Law**: if $\alpha$ is the fraction of running time a sequential program spends on non-parallelizable segments of the computation then
  \[
  S = \frac{1}{\alpha}
  \]
- **Gustafson's Law**: the *scaled speed-up* with $N$ parallel processes
  \[
  S(N) = N - \alpha(N-1)
  \]
Concurrent execution can be challenging.

- It could lead to **race conditions**, an undesirable effect when the results of concurrent execution depend on the sequence of events.
- Shared resources must be protected by **locks/semmaphores/monitors** to ensure serial access.
- Deadlocks and livelocks are possible.

The four Coffman conditions for a deadlock:

- **Mutual exclusion** - at least one resource must be non-sharable, only one process/thread may use the resource at any given time.
- **Hold and wait** - at least one processes/thread must hold one or more resources and wait for others.
- **No-preemption** - the scheduler or a monitor should not be able to force a process/thread holding a resource to relinquish it.
- **Circular wait** - given the set of n processes/threads \( \{P_1, P_2, P_3, \ldots, P_n\} \). Process \( P_1 \) waits for a resource held by \( P_2 \), \( P_2 \) waits for a resource held by \( P_3 \), and so on, \( P_n \) waits for a resource held by \( P_1 \).
A monitor provides special procedures to access the data in a critical section.
More challenges

- *Livelock* condition: two or more processes/threads continually change their state in response to changes in the other processes; then none of the processes can complete its execution.

- Very often processes/threads running concurrently are assigned priorities and scheduled based on these priorities. *Priority inversion*, a higher priority process/task is indirectly preempted by a lower priority one.

- Discovering parallelism is often challenging and the development of parallel algorithms requires a considerable effort. For example, many numerical analysis problems, such as solving large systems of linear equations or solving systems of PDEs (Partial Differential Equations), require algorithms based on domain decomposition methods.
Parallelism

- Fine-grain parallelism → relatively small blocks of the code can be executed in parallel without the need to communicate or synchronize with other threads or processes.
- Coarse-grain parallelism → large blocks of code can be executed in parallel.
- The speed-up of applications displaying fine-grain parallelism is considerably lower than those of coarse-grained applications; the processor speed is orders of magnitude larger than the communication speed even on systems with a fast interconnect.
- Data parallelism → the data is partitioned into several blocks and the blocks are processed in parallel.
- Same Program Multiple Data (SPMD) → data parallelism when multiple copies of the same program run concurrently, each one on a different data block.
Parallelism levels

- **Bit level parallelism.** The number of bits processed per clock cycle, often called a word size, has increased gradually from 4-bit, to 8-bit, 16-bit, 32-bit, and to 64-bit. This has reduced the number of instructions required to process larger size operands and allowed a significant performance improvement. During this evolutionary process the number of address bits have also increased allowing instructions to reference a larger address space.

- **Instruction-level parallelism.** Today’s computers use multi-stage processing pipelines to speed up execution.

- **Data parallelism or loop parallelism.** The program loops can be processed in parallel.

- **Task parallelism.** The problem can be decomposed into tasks that can be carried out concurrently. For example, SPMD. Note that data dependencies cause different flows of control in individual tasks.
Parallel computer architecture

- Michael Flynn’s classification of computer architectures is based on the number of concurrent control/instruction and data streams:
  - SISD (Single Instruction Single Data) – scalar architecture with one processor/core.
  - SIMD (Single Instruction, Multiple Data) - supports vector processing. When a SIMD instruction is issued, the operations on individual vector components are carried out concurrently.
  - MIMD (Multiple Instructions, Multiple Data) - a system with several processors and/or cores that function asynchronously and independently; at any time, different processors/cores may be executing different instructions on different data. We distinguish several types of systems:
    - Uniform Memory Access (UMA).
    - Cache Only Memory Access (COMA).
    - Non-Uniform Memory Access (NUMA).
Distributed systems

- Collection of autonomous computers, connected through a network and distribution software called middleware which enables computers to coordinate their activities and to share system resources.

- Characteristics:
  - The users perceive the system as a single, integrated computing facility.
  - The components are autonomous.
  - Scheduling and other resource management and security policies are implemented by each system.
  - There are multiple points of control and multiple points of failure.
  - The resources may not be accessible at all times.
  - Can be scaled by adding additional resources.
  - Can be designed to maintain availability even at low levels of hardware/software/network reliability.
Desirable properties of a distributed system

- Access transparency - local and remote information objects are accessed using identical operations.
- Location transparency - information objects are accessed without knowledge of their location.
- Concurrency transparency - several processes run concurrently using shared information objects without interference among them.
- Replication transparency - multiple instances of information objects increase reliability without the knowledge of users or applications.
- Failure transparency - the concealment of faults.
- Migration transparency - the information objects in the system are moved without affecting the operation performed on them.
- Performance transparency - the system can be reconfigured based on the load and quality of service requirements.
- Scaling transparency - the system and the applications can scale without a change in the system structure and without affecting the applications.
Processes, threads, events

- Dispatchable units of work:
  - Process → a program in execution.
  - Thread → a light-weight process.
- State of a process/thread → the ensemble of information we need to restart a process/thread after it was suspended.
- Event → is a change of state of a process.
  - Local events.
  - Communication events.
- Process group → a collection of cooperating processes; the processes work in concert and communicate with one another in order to reach a common goal.
- The global state of a distributed system consisting of several processes and communication channels is the union of the states of the individual processes and channels.
Messages and communication channels

- **Message** → a structured unit of information.

- **Communication channel** → provides the means for processes or threads to communicate with one another and coordinate their actions by exchanging messages. Communication is done only by means of `send(m)` and `receive(m)` communication events, where \( m \) is a message.

- The state of a communication channel: given two processes \( p_i \) and \( p_j \), the state of the channel \( \xi_{i,j} \) from \( p_i \) to \( p_j \) consists of messages sent by \( p_i \) but not yet received by \( p_j \).

- **Protocol** → a finite set of messages exchanged among processes to help them coordinate their actions.
Space-time diagrams display local and communication events during a process lifetime. Local events are small black circles. Communication events in different processes are connected by lines from the send event and to the receive event.

(a) All events in process $p_1$ are local; the process is in state $\sigma_1$ immediately after the occurrence of event $e_1^1$ and remains in that state until the occurrence of event $e_1^12$.

(b) Two processes $p_1$ and $p_2$; event $e_1^2$ is a communication event, $p_1$ sends a message to $p_2$; $e_2^3$ is a communication event, process $p_2$ receives the message sent by $p_1$.

(c) Three processes interact by means of communication events.
Global state of a process group

- The global states of a distributed computation with \( n \) processes form an \( n \)-dimensional lattice.
- How many paths to reach a global state exist? The more paths, the harder is to identify the events leading to a given state. A large number of paths increases the difficulties to debug a system.
- In case of two threads the number of paths from the initial state \( \Sigma^{(0,0)} \) to the state \( \Sigma^{(m,n)} \) is:
  \[
  N^{(m,n)} = (m + n) / (m!n!)
  \]
- In the two dimensional case the global state \( \Sigma^{(m,n)} \) can only be reached from two states, \( \Sigma^{(m-1,n)} \) and \( \Sigma^{(m,n-1)} \)
(a) The lattice of the global states of two processes with the space-time showing only the first two events per process.

(b) The six possible sequences of events leading to the state $\Sigma^{2,2}$
Communication protocols - coordination

- Communication in the presence of channel failures, a major concern. It is impossible to guarantee that two processes will reach an agreement in case of channel failures (see next slide).

- In practice, error detection and error correction codes allow processes to communicate reliably through noisy digital channels. The redundancy of a message is increased by more bits and packaging a message as a codeword.

- Communication protocols implement:
  - Error control mechanisms – using error detection and error correction codes.
  - Flow control - provides feedback from the receiver, it forces the sender to transmit only the amount of data the receiver can handle.
  - Congestion control - ensures that the offered load of the network does not exceed the network capacity.
Process coordination in the presence of errors; each message may be lost with probability $\epsilon$. If a protocol consisting of $n$ messages exists, then the protocol should be able to function properly with $n-1$ messages reaching their destination, one of them being lost.
Time and time intervals

- Process coordination requires:
  - A global concept of time shared by cooperating entities.
  - The measurement of time intervals, the time elapsed between two events.
- Two events in the global history may be unrelated, neither one is the cause of the other; such events are said to be concurrent events.
- Local timers provide relative time measurements. An isolated system can be characterized by its history expressed as a sequence of events, each event corresponding to a change of the state of the system.
- Global agreement on time is necessary to trigger actions that should occur concurrently.
- Timestamps are often used for event ordering using a global time base constructed on local virtual clocks.
Logical clocks

- Logical clock (LC) → an abstraction necessary to ensure the clock condition in the absence of a global clock.
- A process maps events to positive integers. LC(e) the local variable associated with event e.
- Each process time-stamps each message $m$ it sends with the value of the logical clock at the time of sending:
  $$\text{TS}(m) = \text{LC}(\text{send}(m)).$$
- The rules to update the logical clock:
  $$\text{LC}(e) = \text{LC} + 1 \quad \text{if } e \text{ is a local event}$$
  $$\text{LC}(e) = \max (\text{LC}, (\text{TS}(m)+1)) \quad \text{if } e \text{ is a receive event.}$$
Three processes and their logical clocks.
Message delivery rules; causal delivery

- The communication channel abstraction makes no assumptions about the order of messages; a real-life network might reorder messages.
- First-In-First-Out (FIFO) delivery → messages are delivered in the same order they are sent.
- Causal delivery → an extension of the FIFO delivery to the case when a process receives messages from different sources.
- Even if the communication channel does not guarantee FIFO delivery, FIFO delivery can be enforced by attaching a sequence number to each message sent. The sequence numbers are also used to reassemble messages out of individual packets.
Message receiving and message delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.
Violation of causal delivery when more than two processes are involved; message $m_1$ is delivered to process $p_2$ after message $m_1$, though message $m_1$ was sent before $m_3$. Indeed, message $m_3$ was sent by process $p_1$ after receiving $m_1$, which in turn was sent by process $p_3$ after sending message $m_1$.
Runs and cuts

- **Run** → a total ordering of all the events in the global history of a distributed computation consistent with the local history of each participant process; a run implies a sequence of events as well as a sequence of global states.

- **Cut** → a subset of the local history of all processes.

- **Frontier of the cut** in the global history of n processes → an n-tuple consisting of the last event of every process included in the cut.

- Cuts provide the intuition to generate global states based on an exchange of messages between a monitor and a group of processes. The cut represents the instance when requests to report individual state are received by the members of the group.
Consistent and inconsistent cuts and runs

- **Consistent cut** → a cut closed under the causal precedence relationship.
- A consistent cut establishes an instance of a distributed computation; given a consistent cut we can determine if an event $e$ occurred before the cut.
- **Consistent run** → the total ordering of events imposed by the run is consistent with the partial order imposed by the causal relation.
- The causal history of event $e$ is the smallest consistent cut of the process group including event $e$. 
Inconsistent and consistent cuts.

- The cut $C_1=(e_1^4, e_2^5, e_3^2)$ is inconsistent because it includes $e_2^4$ - the event triggered by the arrival of the message $m_3$ at process $p_2$ but does not include $e_3^3$, the event triggered by process $p_2$ sending $m_3$ thus, the cut $C_1$ violates causality.

- The cut $C_2=(e_1^5, e_2^6, e_3^3)$ is consistent; there is no causal inconsistency, it includes event $e_2^6$ - the sending of message $m_4$ without the effect of it, the event $e_3^4$ - the receiving of the message by process $p_3$. 

Dan C. Marinescu

Chapter 2

28
The causal history of event $e_2^5$ is the smallest consistent cut including $e_2^5$. 

Chapter 2
The snapshot protocol of Chandy and Lamport

- A protocol to construct consistent global states.
- The protocol has three steps:
  - Process $p_0$ sends to itself a *take snapshot* message.
  - Let $p_i$ be the process from which $p_i$ receives the *take snapshot* message for the first time. Upon receiving the message, the process $p_i$ records its local state and relays the *take snapshot* along all its outgoing channels without executing any events on behalf of its underlying computation; channel state is set to empty and process $p_i$ starts recording messages received over each of its incoming channels.
  - Let $p_s$ be the process from which $p_i$ receives the *take snapshot* message beyond the first time; process $p_i$ stops recording messages along the incoming channel from $p_s$ and declares channel state between processes $p_i$ and $p_s$ as those messages that have been recorded.
Six processes executing the snapshot protocol of Chandy and Lamport.
Concurrency

- Required by system and application software:
  - Reactive systems respond to external events; e.g., the kernel of an operating system, embedded systems.
  - Improve performance - parallel applications partition the workload and distribute it to multiple threads running concurrently.
  - Support a variable load and shorten the response time - distributed applications, including transaction management systems and applications based on the client-server.
Context switching when a page fault occurs during the instruction fetch phase.

Virtual Memory Manager attempts to translate the virtual address of a next instruction of thread 1 and encounters a page fault. Thread 1 is suspended waiting for the event signaling that the page was paged-in from the disk.

The Scheduler dispatches thread 2.

The Exception Handler invokes the Multi-Level Memory Manager.

Chapter 2

Dan C. Marinescu
Atomic actions

- Parallel and distributed applications must take special precautions for handling shared resources.
- Atomic operation → a multi-step operation should be allowed to proceed to completion without any interruptions and should not expose the state of the system until the action is completed.
- Hiding the internal state of an atomic action reduces the number of states a system can be in thus, it simplifies the design and maintenance of the system.
- Atomicity requires hardware support:
  - Test-and-Set → instruction which writes to a memory location and returns the old content of that memory cell as non-interruptible.
  - Compare-and-Swap → instruction which compares the contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.
All-or-nothing atomicity

- Either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted; a transaction is either carried out successfully, or the record targeted by the transaction is returned to its original state.

- Two phases:
  - Pre-commit → during this phase it should be possible to back up from it without leaving any trace. Commit point - the transition from the first to the second phase. During the pre-commit phase all steps necessary to prepare the post-commit phase, e.g., check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space must be carried out; during this phase no results should be exposed and no actions that are irreversible should be carried out.
  - Post-commit phase → should be able to run to completion. Shared resources allocated during the pre-commit cannot be released until after the commit point.
The states of an all-or-nothing action.
Storage models. Cell storage does not support all-or-nothing actions. When we maintain the version histories it is possible to restore the original content but we need to encapsulate the data access and provide mechanisms to implement the two phases of an atomic all-or-nothing action. The journal storage does precisely that.
Atomicity

- Before-or-after atomicity → the effect of multiple actions is as if these actions have occurred one after another, in some order.
- A systematic approach to atomicity must address several delicate questions:
  - How to guarantee that only one atomic action has access to a shared resource at any given time.
  - How to return to the original state of the system when an atomic action fails to complete.
  - How to ensure that the order of several atomic actions leads to consistent results.
Consensus protocols

- Consensus → process of agreeing to one of several alternates proposed by a number of agents.
- Consensus service → set of $n$ processes; clients send requests, propose a value and wait for a response; the goal is to get the set of processes to reach consensus on a single proposed value.

Assumptions:

- Processes run on processors and communicate through a network; processors and network may experience failures, but not Byzantine failures.
- Processors: (i) operate at arbitrary speeds; (ii) have stable storage and may rejoin the protocol after a failure; (iii) send messages to one another.
- The network: (i) may lose, reorder, or duplicate messages; (ii) messages are sent asynchronously; it may take arbitrary long time to reach the destination.
Paxos

- Paxos → a family of protocols to reach consensus based on a finite state machine approach.
- Basic Paxos considers several types of entities:
  - Client → agent that issues a request and waits for a response.
  - Proposer → agent with the mission to advocate a request from a client, convince the acceptors to agree on the value proposed by a client, and to act as a coordinator to move the protocol forward in case of conflicts.
  - Acceptor → agent acting as the fault-tolerant memory of the protocol.
  - Learner → agent acting as the replication factor of the protocol and taking action once a request has been agreed upon.
  - Leader → a distinguished proposer.
- Quorum → subset of all acceptors.
- A proposal has a proposal number $pn$ and contains a value $v$.
- Several types of requests flow through the system, prepare, accept.
The Paxos algorithm has two phases

- **Phase I.**
  - **Proposal preparation:** A proposer (the leader) sends a proposal \((pn=k,v)\). The proposer chooses a proposal number \(pn=k\) and sends a prepare message to a majority of acceptors requesting:
    - that a proposal with \(pn<k\) should not be accepted.
    - The proposal with the highest number \(pn<k\) already accepted by each acceptor.
  - **Proposal promise:** An acceptor must remember the proposal number of the highest proposal number it has ever accepted and the highest proposal number it has ever responded to. It can accept a proposal with \(pn=k\) if and only if it has not responded to a prepare request with \(pn>k\); if it has already replied to a prepare request for a proposal with \(pn>k\) it should not reply.

- **Phase II.**
  - **Accept request:** If the majority of acceptors respond, then the proposer chooses the value \(v\) of the proposal as follows:
    - the value \(v\) of the highest proposal number selected from all the responses.
    - an arbitrary value if no proposal was issued by any of the proposers

  Proposer sends an accept request message to a quorum of acceptors with \((pn=k,v)\).
  - **Accept:** If an acceptor receives an accept message for a proposal with the proposal number \(pn=k\) it must accept it if and only if it has not already promised to consider proposals with a \(pn>k\).
The flow of messages for the Paxos consensus algorithm. Individual clients propose different values to the leader who initiates the algorithm. Acceptor A accepts the value in message with proposal number \( pn=k \); acceptor B does not respond with a promise while acceptor C responds with a promise, but ultimately does not accept the proposal.

1. Prepare – the leader chooses a proposal with proposal number \( pn=k \).
2. Promise – an acceptor promises to accept the proposal only if it has not responded to a proposal with \( pn>k \). (B does not respond)
3. Accept request – the leader chooses the value \( v \) of the highest proposal number from all acceptors who have sent a promise and sends it to all of them.
4. Accept – an acceptor accepts a proposal with \( pn=k \) only if it has not promised to accept a proposal with \( pn>k \). Stores the accepted value in persistent memory.
5. The leader accepts a proposal if the quorum of acceptors send an accept message for the proposal with \( pn=k \). (C does not accept)
The basic Paxos with three actors: proposer (P), three acceptors (A1, A2, A3) and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are (a) Successful first round when there are no failures. (b) Successful first round of Paxos when an acceptor fails.
Petri Nets (PNs)

- PNs $\rightarrow$ bipartite graphs; tokens that flow through the graph.
- Used to model the dynamic rather than static behavior of systems, e.g., detect synchronization anomalies.
- Bipartite graph $\rightarrow$ graphs with two classes of nodes; arcs always connect a node in one class with one or more nodes in the other class.
- PNs are also called Place-Transition (P/T) Nets. The two classes of nodes are: places and transitions. Arcs connect one place with one or more transitions or a transition with one or more places.
- Firing of a transition removes tokens from its input places and adds them to its output places.
Modeling concurrency with Petri Nets

- Petri Nets can model different activities in a distributed system.
  - A transition may model the occurrence of an event, the execution of a computational task, the transmission of a packet, a logic statement.
  - The input places of a transition model the pre-conditions of an event, the input data for the computational task, the presence of data in an input buffer, the pre-conditions of a logic statement.
  - The output places of a transition model the post-conditions associated with an event, the results of the computational task, the presence of data in an output buffer, or the conclusions of a logic statement.
Petri Nets firing rules. (a) An unmarked net with one transition $t_1$ with two input places, $p_1$ and $p_2$ and one output place, $p_3$. (b) The marked net, the net with places populated by tokens; the net before firing the enabled transition $t_1$. (c) The marked net after firing transition $t_1$ two tokens from place $p_1$ and one from place $p_2$ are removed and transported to place $p_3$. 
Petri Nets modeling.

- (a) Choice; only one of the transitions, $t_1$ or $t_2$, may fire.
- (b) Symmetric confusion; transitions $t_1$ and $t_3$ are concurrent and, at the same time, they are in conflict with $t_2$. If $t_2$ fires, then $t_1$ and/or $t_3$ are disabled.
- (c) Asymmetric confusion; transition $t_1$ is concurrent with $t_3$ and it is in conflict with $t_2$ if $t_3$ fires before $t_1$. 
(a) (b) (c) (d)
Classes of Petri Nets

- Classified based on the number of input and output flow relations from/to a transition or a place and by the manner in which transitions share input places:
  - State Machines - are used to model finite state machines and cannot model concurrency and synchronization.
  - Marked Graphs - cannot model choice and conflict.
  - Free-choice Nets - cannot model confusion.
  - Extended Free-choice Nets - cannot model confusion but allow inhibitor arcs.
  - Asymmetric Choice Nets - can model asymmetric confusion but not symmetric one.
Classes of Petri Nets

Asymmetric Choice  Free Choice  State Machines  Marked Graphs

Petri Nets
The client-server paradigm

- Based on enforced modularity → the modules are forced to interact only by sending and receiving messages.

- This paradigm leads to:
  - A more robust design, the clients and the servers are independent modules and may fail separately.
  - The servers are stateless, they do not have to maintain state information; the server may fail and then come up without the clients being affected or even noticing the failure of the server.
  - An attack is less likely because it is difficult for an intruder to guess the format of the messages or the sequence numbers of the segments, when messages are transported by TCP.
(a) Email service; the sender and the receiver communicate asynchronously using inboxes and outboxes. Mail demons run at each site.

(b) An event service supports coordination in a distributed system environment. The service is based on the publish-subscribe paradigm; an event producer publishes events and an event consumer subscribes to events. The server maintains queues for each event and delivers notifications to clients when an event occurs.
World Wide Web.
The three-way handshake involves the first three messages exchanged between the client and the server. Once the TCP connection is established the HTTP server takes its time to construct the page to respond the first request; to satisfy the second request, the HTTP server must retrieve an image from the disk.
The response time includes the RTT, the server residence time, and the data transmission time.

Chapter 2
A Web client can: (a) communicate directly with the server; (b) communicate through a proxy; (c) use tunneling to cross the network.