Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

C4

Features of Distributed Algorithms

Ioan Salomie

Fall 2017

Distributed Application and Control Algorithm executions

- **Distributed Application execution**
  - Consists of the execution of the statements (including the communication statements), within the distributed application program (business logic)
- **Control Algorithms**
  - Executed in order to monitor the application execution or to perform various auxiliary functions
  - Performs functions such as:
    - » creating a spanning tree,
    - » creating a connected dominating set,
    - » achieving consensus among the nodes,
    - » distributed transaction commit,
    - » distributed deadlock detection,
    - » global predicate detection,
    - » termination detection,
    - » global state recording, check-pointing,
    - » memory consistency enforcement in distributed shared memory systems

Centralized and Distributed Algorithms (1)

- Centralized (Distributed) Algorithms (CA)
  - One processor (possible few) – performs predominant amount of work
  - Other processors – play relatively smaller roles in accomplishing the joint task such as requesting / supplying information (periodically or when queried)
  - A typical system configuration suited for centralized algorithms is the **client–server** configuration
    - » Much of today's commercial software is written using this configuration
  - Drawbacks (from a theoretical perspective):
    - » **Potential bottleneck** - The single server is a potential bottleneck for both processing and bandwidth access on the links
    - » **Single point of failure** - this problem is alleviated in practice by using replicated servers distributed across the system => the overall configuration is not as centralized any more

Centralized and Distributed Algorithms (2)

- Distributed Algorithm (DA)
  - Each processor plays an equal role in
    - » sharing the message overhead,
    - » time overhead, and
    - » space overhead.
  - Difficult to design efficient, purely distributed algorithms
- Consider the problem of recording a global state of all the nodes
  - The well-known Chandy–Lamport algorithm (studied before) is distributed – yet one node (typically the initiator), is responsible for assembling the local states of the other nodes, and hence plays a slightly different role

Centralized and Distributed Algorithms (3)

- Centralized / Distributed and Superimposed Topologies
  - Algorithms designed to run on a **logical-ring** superimposed topology = > fully distributed algorithms
  - Algorithms designed to run on the **logical tree** and other asymmetric topologies with a pre-designated root node => have some asymmetry that mirrors the asymmetric topology
- Fully distributed algorithms are ideal but only partly distributed algorithms are sometimes more practical to implement in real systems
- **Note**. The advances in peer-to-peer networks, ubiquitous and ad-hoc networks, blockchain technology (distributed consensus algorithms) and mobile systems will require distributed solutions

Symmetric and Asymmetric Algorithms

- Symmetric algorithms
  - All the processors execute the same logical functions
- Asymmetric algorithms
  - Different processors execute logically different (but perhaps partly overlapping) functions
  - A centralized algorithm is always asymmetric
  - An algorithm that is not fully distributed is also asymmetric
  - In the client–server configuration
    - » The clients and the server execute asymmetric algorithms
  - In a tree configuration
    - » The **root** and the **leaves** usually perform some functions that are different from each other, and that are different from the functions of the **internal nodes** of the tree
  - Applications where there are asymmetries in the roles of the cooperating processors => associated asymmetric algorithms
    - » A typical example: when one processor initiates the computation of some global function (e.g., min, sum)

Anonymous Algorithms

- Anonymous system
  - Process / processor identifiers are not used to make any execution decisions in the distributed algorithm
- Anonymous algorithm
  - An algorithm that run on anonymous system
  - ⇒ The algorithm is not using process identifiers or processor identifiers in the code
- In famous algorithms (such as the Bakery algorithm for mutual exclusion on platforms providing read/write atomicity only, or the "wait-wound" or "wound-die" algorithms used for deadlock detection/avoidance or for transaction serializability in databases)
  - Process identifier is used in resolving ties or contentions that are otherwise unresolved despite the symmetric and non-centralized nature of the algorithms

Uniform Algorithms

- An uniform algorithm is not using **n**, (the number of processes in the system) in its code
- Algorithms that run on a logical ring where the nodes communicate only with their neighbors are uniform
  - In the following courses will study a uniform algorithm for leader election (on logical ring superimposed topologies)
- Advantages
  - Allows scalability transparency
    - » Processes can join or leave the distributed execution without intruding on the other processes (except its immediate neighbors that need to be aware of any changes in their immediate topology)

Adaptive Algorithms

- Consider:
  - X, a problem to be solved on a system with **n** nodes
  - C(X), the context of X
  - k <= n the number of nodes "participating" in C(X) when the algorithm to solve X is executed
- Adaptive algorithm
  - The complexity of the algorithm can be expressed in terms of **k** rather than in terms of **n**
- Example
  - If the complexity of a mutual exclusion algorithm can be expressed in terms of the actual number of nodes contending for the critical section when the algorithm is executed, then the algorithm is adaptive

Deterministic and Non-deterministic Executions (1)

- Each execution defines a partial order on the events in the execution.
- In a deterministic execution (even in asynchronous systems)
  - Repeated re-execution will reproduce the same partial order on the events.
  - Useful property for applications such as debugging, detection of unstable predicates, and for reasoning about global states.
- In a non-deterministic execution
  - Any re-execution of that program may result in a very different outcome, and
  - Any assertion about a non-deterministic execution can be made only for that particular execution (=> non-deterministic executions are difficult to reason with)
  - Different re-executions may result in different partial orders because of variable factors such as
    - » lack of an upper bound on message delivery times and unpredictable congestion and
    - » local scheduling delays on the CPUs due to timesharing

Deterministic and Non-deterministic Executions (2)

- Deterministic receive primitive
  - Specifies the source from which it wants to receive a message
- Non-deterministic receive primitive
  - Can receive a message from any source
    - » The message delivered to the process is:
      - the first message that is queued in the local incoming buffer, or
      - the first message that comes in subsequently if no message is queued in the local incoming buffer
- Distributed programs with deterministic execution
  - Contains no non-deterministic receives
- Distributed programs with non-deterministic execution
  - Contains at least one non-deterministic receive primitive

Synchronous and Asynchronous systems (1)

- **Synchronous system** satisfies the following three properties:
  - **(P1)** There is a known upper bound on the message communication delay
  - **(P2)** There is a known bounded drift rate for the local clock of each processor with respect to real-time
    - » The drift rate between two clocks is defined as the rate at which their values diverge
  - **(P3)** There is a known upper bound on the time taken by a process to execute a logical step in the execution

Synchronous and Asynchronous systems (2)

- In an **asynchronous system**
  - None of the three properties P1, P2, P3 of synchronous systems are satisfied
  - Systems can be designed that satisfy some combination but not all of the criteria that define a synchronous system
  - The algorithms to solve any particular problem may be very different based on the model assumptions

  => it is important to clearly identify the system model before
- Distributed systems are inherently asynchronous
  - **Synchronizers** - provide the abstraction of a synchronous execution for the asynchronous systems

Online / Offline Algorithms

- On-line algorithms
  - Execute as the data is being generated
- Off-line algorithms
  - Require all the data to be available before algorithm execution begins
- On-line algorithm advantages
  - Debugging
    - » On-line debugging can detect errors when they occur, as opposed to collecting the entire trace of the execution and then examining it for errors
  - Scheduling
    - » On-line scheduling allows for dynamic changes to the schedule to account for newly arrived requests with closer deadlines
- Comparison
  - On-line algorithms are more desirable

Wait-free and Lock-free Algorithms (1)

- Can be viewed as a special class of fault-tolerant algorithms
- Wait-free / lock-free algorithms offer a very high degree of robustness
- Wait-free
  - Wait free - The ability of a process (thread) to complete its execution irrespective of the actions of other processes (threads)
  - Allow multiple threads to read/write shared resources in a concurrent way without corrupting the resource
- Lock-free – The ability of processes to concurrently access shared resources without locking
  - No synchronization primitives (such as mutex or semaphores) are involved
  - Contrast to algorithms that protect shared resources with locks
- All wait-free algorithms are lock-free (not inverse)
- Designing a wait-free / lock-free algorithm is expensive and not always possible

Wait-free Algorithms (2)

- Applications of wait-free algorithms
  - Mutual exclusion synchronization for the distributed shared memory abstraction
    - » The objective was to enable a process to access its critical section, even if the process in the critical section fails or misbehaves by not exiting from the critical section

Failure Models

- Failure model - specifies the manner in which the component(s) of the system may fail
  - Many failure models have been studied
  - The assumed failure model has impact on the implementation of the Distributed Algorithms
- **k-fault tolerant** system
  - **k** - Maximum number of system components that may fail without affecting system's specified behavior
  - System component – processes, links or a combination

Failure Models

- In DS both processes and communication channels may fail
- Failures taxonomy (Hadzilacos and Toueg [1994])
  - Omission failures,
  - Arbitrary failures
  - Timing failures

Failure Models
Omission Failures

- Omission failure - a process or communication channel fails to perform actions that it is supposed to do

Processes

- Crash process (process halts and does not execute any processing steps)
- Design of fault tolerant services is simplified if it can be assumed that the services on which they depend crash cleanly – that is, their processes either function correctly or else stop
- In **synchronous systems**: timeouts are used to detect crash processes;
- In **asynchronous systems** timeouts can only detect that the process is not responding (crash or slow process)
- Failure is difficult to be detected in asynchronous DS => reaching agreement is difficult in asynchronous DS in the presence of failures
- **Fail-stop** process is a crashed process detected using timeouts by other processes in Distributed synchronous systems

Failure Models
Omission Failures

Communication channels

- Send-omission failure
  - Loss of messages between the sending process and the outgoing message buffer
- Receive-omission failure
  - Loss of messages between the incoming message buffer and the receiving process
- Channel-omission failure
  - Loss of message in between buffers
- All these are known as **dropping message failures**
- Dropping message main causes
  - Lack of buffer space at the receiver or at an intervening gateway,
  - Network transmission error, detected by a checksum carried with the message data

Failure Models
Arbitrary Failures

- Arbitrary failure (Byzantine failure) – any type of error may occur
- Worst failure semantics
- Process level
  - Arbitrarily omits intended processing steps or takes unintended processing steps
  - Cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply
- Communication channel
  - Message contents may be corrupted,
  - Nonexistent messages may be delivered or
  - Real messages may be delivered more than once

Failure models
Timing failures

- All previous failure models apply to both synchronous and asynchronous systems
- Timing failures apply to synchronous systems, and manifest themselves as some or all of the following at each process:
  - **(i)** general omission failures
  - **(ii)** process clocks violating their pre-specified drift rate
  - **(iii)** process violating the bounds on the time taken for a step of execution
- In term of severity
  - Timing failures are more severe than general omission failures
  - Timing failures are less severe than Byzantine failures with message authentication

Failure models
Fault Tolerance

- DS - collection of components offering services
- Objective: construct reliable services from components that may exhibit failures
  - Ex. multiple servers that hold **replicas of data** can continue to provide a service when one of them crashes
- Steps
  - Detecting failure
  - Making failure transparent to the user
    - » Hide the failure (examples: message retransmission when fails to arrive, redundant resources)
    - » Attenuate the failure
    - » Tolerate the failure (web browser cannot contact server, informs the user and let them to decide)
    - » **Use replicated and redundant resources**
  - Recovery from failures
    - » Automatic recovery from partial faults
  - New research area: autonomic computing - see IBM's page for more details:
    - » https://www-01.ibm.com/software/info/topic/autonomic.html

Failure models
Fault Tolerance

Means for achieving fault tolerance

- Redundancy and Replication of DS resources and
- Recovery algorithms

Fault tolerance is close related to the concepts of **availability** and **consistency of DS**

Failure models
Fault Tolerance

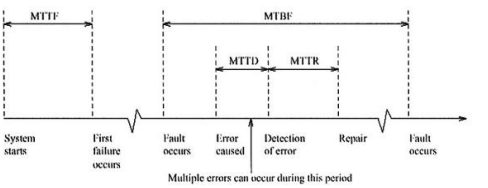
Availability

- How to measure:
  - Percentage calculated as the operational time per total time considered
- To increase availability: the failure must be **detected** and **recovery** procedure initiated (this process is called **failover**)
  - Detection: periodically monitoring the status of each server (heart beating algorithms) by the master node (more difficult in P2P systems)
  - Recovery: achieved by replication and redundancy

Failure models
Fault Tolerance

Availability (cont.)

- Metrics related to availability
  - Mean Time to Failure (MTTF),
  - Mean Time to Repair (MTTR),
  - Mean Time Between Failures (MTBF)
  - Mean Time to Detect Failures (MTTD)



Failure models
Fault Tolerance

Availability (cont.)

- MTTF - the expected time that a DS system will operate before the first failure occurs
  - N - the number of identical systems measured for the considered time period T
  - ti - the time that system i operates befc encountering the first failure, consideri the start time t=0
- MTTR - the is the average time required to repair a DS system
  - ti – time to repair the i-th of N faults
- Mean time between failure (MTBF)
  - T is the time period considered
  - n_{avg} is the average number of failures
- Average number of failures (n_{avg})
  - N the number of systems, each being operated for time T
  - n_i the number of failures encountered by the i-th system

$$MTTF = \frac{\sum_{i=1}^{N} t_i}{N}$$

$$MTTR = \frac{\sum_{i=1}^{N} t_i}{N}$$

$$MTBF = \frac{T}{n_{avg}}$$

$$n_{avg} = \sum_{i-1}^{N} \frac{n_i}{N}$$

Failure models
Fault Tolerance

Availability (cont.)

- Examples of availability values for T = 1

Availability	Downtime	System
90%	> 1 month	Basic PC
99%	Approx. 4 days	Server
99.9%	Approx. 9 hours	Cluster
99.99%	Approx. 1 hour	XMP (Extreme Memory Profile)
99.999%	Approx. 2 min	Net Switches of good provider
99.9999% (four nines)	Approx. 2 sec	Critical controller

Failure models
Fault Tolerance

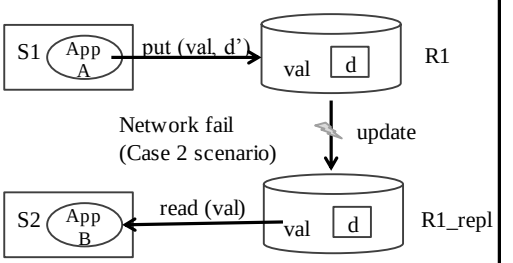
Consistency

Consistency and data replication

- Data replication is common in DS for achieving:
  - Fault Tolerance and
  - performance (avoid overload of a single bottleneck resource, communication delays),
  - fault tolerance or
- Examples:
  - replicated databases
  - mirrored heavy used WWW sites
- Data replication compromises data consistency
- **Tradeoff** in DS between **performance** (favored by weak consistency) and **correctness** (favoring strong consistency)

Failure models
Fault Tolerance

Consistency



Failure models
Fault Tolerance

Consistency

Engineering weak consistency

- Simple approach - all updates are made to a **primary copy**
- Primary copy propagates updates to a number of backup copies
  - The system must converge to a consistent state as updates propagate
  - Issue: the updates do not (in general) reach all replicas in the same total order => conflicting updates => timestamp using logical clocks to discriminate order
- Queries can be made to any copy but the primary copy can be queried for the most-up-to-date value
  - Example: DNS domains have a distinguishing name server per domain plus a few replicas

Failure models
Fault Tolerance

Consistency

Engineering weak consistency

- Performance issues:
  - for large scale DS primary copy becomes bottleneck and
  - updates the rest of the copies from primary is slow
- Reliability issues
  - Primary copy can fail
  - Have a **hot standby** to which updates are made synchronously with the primary copy
- Scalable approach
  - Distribute a number of first class replicas
  - Issues: concurrent updates and queries (needs to be further solved)

Failure models
Fault Tolerance

Consistency

Engineering strong consistency

- Use a transactional approach

```
startStrongConsistency
    execute the same update to all replicas
endStrongConsistency
```

- Commit – all changes are made, persistent and visible
- Abort – No changes are made to any object
- Implementation
  - Update : lock all objects, execute update, unlock all objects
  - Query: read from any replica => consistent values

Failure models
Fault Tolerance

Consistency

Engineering strong consistency

- Update issues
  - Some replicas may reside at the end of slow communication lines
  - Some replicas may fail, be slow or overloaded
  - => low availability (the reason for replication) due to delays in responding to queries
  - The situation cannot be accepted to abort the transaction-oriented approach due to one replica that have problems
  - Solution: Majority voting scheme: **Quorum Assembly** algorithm

CAP Theorem

Main question:

- Can we build a large scale DS that:
  - Scales to an high (unbound) number of transactions using a high (unlimited) data repositories **and**
  - Always **available** with high efficiency in serving users requests **and**
  - Provide **strong consistency** guarantees

- Eric Brewer at the *Symposium on Principles of Distributed Computing* (2000) proposed the following conjecture (known as **CAP Theorem**):

"No distributed system can simultaneously provide all three of the following properties: Consistency, Availability and Partition-resilience"

or in other words,

"Every DS can only provide two of the following three properties: Consistency, Availability and Partition-resilience"

CAP Theorem

- **C = Consistency**
  All operations (R/W) should leave the system in a consistent state or "all DS nodes see the same data at the same time"

- **A = Availability**
  All requests to the active (non-failed) servers must get a response or "a guarantee that every request receives a response about whether it succeeds or not"
- **P = Partition-resilience** (tolerance to network partitions)
  The systems continue to operate despite the arbitrary partitioning due to network/router failures

- Note. As P is necessary for DS, the real choice is between A and C

CAP Theorem

- C + P
  - Always consistent, even in the case of network partition but a reachable replica may deny service without the agreement of the others (e.g. quorum)
  - Examples: Distributed-lock systems (Chubby), Paxos protocol on consensus, BigTable, MongoDB
- A + P
  - A reachable replica provides service even in a network partition but may be inconsistent
  - Cassandra, Amazon Dynamo, CouchDB, Riak
- C + A
  - Available and Consistent unless there is a partition
  - Oracle RAC, IBM DB2 Parallel, GFS (Google File System), HDFS (Hadoop)

Complexity measures and metrics

- Performance of sequential algorithms - is measured using the **time and space complexity** in terms of
  - the lower bounds representing the best case
  - the upper bounds representing the worst case
  - the exact bound
- Performance of distributed algorithms
  - The definitions of space and time complexity need to be refined, and
  - Message complexity also needs to be considered for message-passing systems
- System topology
  - The system topology is usually assumed to be an **undirected un-weighted graph** G = (N, L)

Complexity measures and metrics

- Graph G(N, L) properties
  - n = | N |, l = | L |
  - Graph eccentricity
    - » The **eccentricity** of a **graph** vertex in a connected **graph** is the maximum **graph** distance between that vertex and any other vertex of the graph
    - » Maximum eccentricity is **graph diameter**
  - Degree of vertex (number of edges incident to the vertex)
- Tree embedded in the graph
  - h – tree depth

Complexity measures and metrics

- Main assumption
  - Identical code runs at each processor
  - If this assumption is not valid, then different complexities need to be stated for the different codes
- The complexity measures
  - Space complexity per node
  - System wide space complexity
  - Time complexity per node
  - System wide time complexity
  - Message complexity

Complexity measures and metrics

Space complexity per node

- This is the memory requirement at a node
- The best case, average case, and worst case memory requirement at a node can be specified

Complexity measures and metrics

System wide space complexity

- The system space complexity (best case, average case, or worst case) is not necessarily **n x** (n times) the corresponding space complexity (best case, average case, or worst case) per node
- For example
  - The algorithm may not permit all nodes to achieve the best case at the same time
- If during execution, the worst case occurs at one node, then the worst case will not occur at all the other nodes in that execution

Complexity measures and metrics

Time complexity per node

- Measures the processing time per node
  - Does not explicitly account for the message propagation/transmission times, which are measured as a separate metric

Complexity measures and metrics

System wide time complexity

- Serial execution
  - (by different processes) as in the case of an algorithm in which only the unique token-holder is allowed to execute

  => the overall time complexity is additive
- Concurrent execution at all processors

  => the system time complexity is not n times the time complexity per node

Complexity measures and metrics
Message complexity

Message complexity

- a space component
- a time component

Complexity measures and metrics
Message complexity

Space complexity

- Number of messages
  - Directly contributes to space complexity
  - Indirectly contributes to time component
- Size of messages

Time complexity

- Time complexity depends on the degree of concurrency in the sending of the messages
  - all messages are sequentially sent (with reference to the execution partial order)
  - or all processes can send concurrently
  - or something in between
- For asynchronous executions
  - Time complexity component is measured in terms of sequential message hops, i.e., the length of the longest chain in the partial order $(E, \prec)$ on the events
- For synchronous executions,
  - Time complexity component is measured in terms of rounds (also termed as steps or phases)

Complexity measures and metrics
Conclusion (1)

- It is usually difficult to determine all of the above complexities for most algorithms
- It is important to be aware of the different factors that contribute towards the overhead
- When stating the complexities, it should also be specified whether the algorithm has a synchronous or asynchronous execution
- Depending on the algorithm, further metrics such as the number of send events, or the number of receive events, may be of interest

Complexity measures and metrics
Conclusion (2)

- Case of message multicast
  - It should be stated whether a multicast send event is counted as a single event or not
  - Whether the message multicast is counted as a single message or as multiple messages needs to be clarified
    - » This would depend on whether or not hardware multicasting is used by the lower layers of the network protocol stack
- Case of shared memory systems
  - The message complexity is not an issue if the shared memory is not being provided by the distributed shared memory abstraction over a message-passing system

Complexity measures and metrics

Conclusion (3)

- Additional considerations for complexity measures
  - The size of shared memory, as opposed to the size of local memory (shared memory is expensive, local memory is not expensive)
  - The number of synchronization operations using synchronization variables (useful metric because it affects the time complexity)