FACULTY
OF INFORMATICS
Masaryk University

PA160: Net-Centric Computing II.

Distributed Systems

Luděk Matyska

Spring 2024

FACULTY
OF INFORMATICS
Masaryk University

Lecture overview

Distributed Systems

- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware

- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Service Oriented Architecture (SAO)

Web Services

Issues Examples

- Fault Tolerance in Distributed Systems
- Replication in Distributed Systems
- Transactions in Distributed Systems
- Scheduling/Load-balancing in Distributed Systems

Conclusion

Replication in Distributed Systems

- What is *Replication*
  - Having multiple copies of the same data at several nodes
- For read-only replicas, the implementation is easy – just copy it
- The problem is with writing (as usual)
  - how to ensure *consistency* between the replicas
- *Read-after-write* consistency

FACULTY OF INFORMATICS
Masaryk University

Transactions in Distributed Systems

- A *transaction* is a series of operations that fulfil
  - *Atomicity*
    - The transaction is completed entirely or not at all
  - *Consistency*
    - The system is always in a consistent state
  - *Isolation*
    - Each transaction is run independently on the other transactions
  - *Durability*
    - Once completed, it endures

  ACID

FACULTY
OF INFORMATICS
Masaryk University

Transactions in Distributed Systems

- Operations
  - *Begin*
    - Initiate new transaction
  - *Commit*
    - End the transaction and make all results visible
  - *Abort*
    - End the transaction and all changes made during the transaction are undone
- Roles
  - *Client*
    - The issuer
  - *Coordinator*
    - Controls the whole transaction (handles all operations)
  - *Server*
    - The component whose state is changed by the transaction
    - Registered with the coordinator

FACULTY
OF INFORMATICS
Masaryk University

Flat and Nested Transactions

- Flat transaction
  - Has a single initiating (*Begin*) and a single end point (*Commit* or *Abort*)
  - If locking is used, transactions are serialized
- Nested transactions
  - A transaction includes other transactions (called sub-transactions)
  - Depth is not limited
- The nested transactions use distributed system to its full potential

Atomic Commit

- If transaction updates data on multiple nodes, then
  - Either all nodes must commit, or all must abort
  - If any node crashes, all must abort

  This is called *atomic commitment* problem
- Atomic commit versus consensus

Consensus	Atomic commit
One or more nodes propose a value	Every nodes votes whether to commit or abort
Any one of the proposed values is accepted	Must commit if all nodes vote to commit; must abort if even a single node votes to abort
Crashed nodes can be tolerated if quorum is working	Must abort if even a single node crashes

Two-phase Commit

- The most common algorithm to implement atomic commit
- Basic principles
  - Client starts a regular transaction
  - When a client is ready to commit, it sends a *commit request* to the transaction coordinator
  - Coordinator send a *prepare* message to each server participating in the transaction
  - Each server replies with a message indicating whether it is able to commit the transaction
    - Each server must ensure it can commit but did not commit yet

This is the *first phase*

FACULTY
OF INFORMATICS
Masaryk University

Two-phase Commit

- *Second phase*
  - The coordinator collects the messages
  - If all nodes reply with *ok/prepared*, the coordinator decides to commit
  - If at least one node does not confirm, all does not reply within a timeout, the coordinator decides to abort
  - The coordinator send the decision to all replicas
- What if coordinator crashes?

FACULTY
OF INFORMATICS
Masaryk University

Two-phase Commit – Coordinator Crashes

- Coordinator's actions
  - Coordinator writes its decision to disk
  - After recovery, it sends the decision to nodes (or abort, if no decision written)
  - The nodes are blocked till coordinator recovers
- Coordinator is thus a *single point of failure*
- Avoidance through the consensus algorithm or total order broadcast
- Paxos Commit
  - All nodes send the vote through the total order broadcast
  - Each node counts the votes
    - Only the first vote from a node is counted
    - If any vote is abort, the whole transaction is aborted
  - Total order broadcast ensures the same delivery order on all nodes
  - Total order broadcast ensures the same decision on each node

Scheduling/Load-balancing in Distributed Systems

For concurrent execution of interacting processes:

- **communication** and **synchronization between processes** are the two essential system components

Before the processes can execute, they need to be:

- **scheduled** and
- **allocated** with resources

Why scheduling in distributed systems is of special interest?

- because of the issues that are different from those in traditional multiprocessor systems:
  - *the communication overhead is significant*
  - *the effect of underlying architecture cannot be ignored*
  - *the dynamic behaviour of the system must be addressed*
- local scheduling (on each node) + global scheduling

Scheduling/Load-balancing in Distributed Systems

- let's have a pool of jobs
  - there are some inter-dependencies among them
- and a set of nodes (processors) able to reciprocally communicate

Scheduling

The term **scheduling** means assigning jobs to the machines/processors in a way which *minimizes the time/communication overhead necessary to compute them.*

Load-balancing

The term **load-balancing** means a process that tries to keep all machines in a (distributed) system equally occupied.

Scheduling/Load-balancing in Distributed Systems

- the scheduling/load-balancing task can be represented using graph theory:
  - the pool of N jobs with dependencies can be described as a graph $G(V, U)$, where
    - the nodes represent the jobs (processes)
    - the edges represent the dependencies among the jobs/processes (e.g., an edge from i to j requires that the process i has to complete before j can start executing)
- the graph G has to be split into p parts, so that:
  - $N = N_1 \cup N_2 \cup \cdots \cup N_p$
  - which satisfy the condition, that $|N_i| \approx \frac{|N|}{p}$, where
    - $|N_i|$ is the number of jobs assigned to the processor i, and
    - p is the number of processors, and
    - the number/cost of the edges connecting the parts is minimal
  - the *objectives*:
    - uniform jobs' load-balancing
    - minimizing the communication (the minimal number of edges among the parts)
- the splitting problem is *NP-complete*
  - the heuristic approaches have to be used

FACULTY OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in Distributed Systems

- **Objectives:**
  - to enhance overall system performance metric
  - to minimize process completion time and processor utilization
- Load-balancing deals with equal distribution of load among processors/machines
- Scheduling and load-balancing complements and to some extent overlaps each other
  - the same overall objective
  - can be seen as two phases of the same process
    - depends on the point of view – tasks or processors/machines

Scheduling/Load-balancing in Distributed Systems

- Available techniques:
  - Task Assignment
    - individual related tasks of the process are scheduled to appropriate processors/machines
  - Load Balancing
    - the workload is balanced mong processors/machines in the system
  - Load Sharing
    - to assure no processor/machine is idle if there is a process/task waiting to be processed

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in Distributed Systems

- the “proper” approach to the scheduling/load-balancing problem depends on the following criteria:
  - *jobs’ cost*
  - *dependencies among the jobs*
  - *jobs’ locality*

Scheduling/Load-balancing in Distributed Systems

Jobs' Cost

- the job's cost may be known:
  - before the whole problem set's execution
  - during problem's execution, but before the particular job's execution
  - just after the particular job finishes
- *cost's variability* – all the jobs may have (more or less) the same cost or the costs may differ
- the problem classes based on jobs' cost:
  - all the jobs have the same cost: *easy*
  - the costs are variable, but, known: *more complex*
  - the costs are unknown in advance: *the most complex*

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in Distributed Systems

Dependencies Among the Jobs

- is the order of jobs' execution important?
- the dependencies among the jobs may be known:
  - before the whole problem set's execution
  - during problem's execution, but before the particular job's execution
  - are fully dynamic
- the problem classes based on jobs' dependencies:
  - the jobs are fully independent on each other: *easy*
  - the dependencies are known or predictable: *more complex*
    - flooding
    - in-trees, out-trees (balanced or unbalanced)
    - generic oriented trees (DAG)
  - the dependencies dynamically change: *the most complex*
    - e.g., searching/lookup problems

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in Distributed Systems

Locality

- do all the jobs communicate in the same/similar way?
- is it suitable/necessary to execute some jobs “close” to each other?
- when the job’s communication dependencies are known?
- the problem classes based on jobs’ locality:
  - the jobs do not communicate (at most during initialization): *easy*
  - the communications are known/predictable: *more complex*
    - regular (e.g., a grid) or irregular
  - the communications are unknown in advance: *the most complex*
    - e.g., a discrete events’ simulation

Scheduling/Load-balancing in DSs – Approaches

- in general, the "proper" solving method depends on the time, when the particular information is known
- basic solving algorithms' classes:
  - *static* – offline algorithms
  - *semi-static* – hybrid approaches
  - *dynamic* – online algorithms

- some (but not all) variants:
  - static load-balancing
  - semi-static load-balancing
  - self-scheduling
  - distributed queues
  - DAG planning

Load Balancing Algorithms

GeeksforGeeks

Load Balancing Algorithms

Static

Dynamic

Deterministic

Probabilistic

Centralized

Distributed

Cooperative

Non-Cooperative

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Approaches

Semi-static load-balancing

- suitable for problem sets with slow changes in parameters, and with locality importance
- iterative approach
  - uses static algorithm
  - the result (from the static algorithm) is used for several steps (slight unbalance is accepted)
  - after the steps, the problem set is recalculated with the static algorithm again
- often used for:
  - particle simulation
  - calculations of slowly-changing grids (but in a different sense than in the previous lectures)

Scheduling/Load-balancing in DSs – Approaches

Self-scheduling I.

- a centralized pool of jobs
- idle processors pick the jobs from the pool
- new (sub)jobs are added to the pool
- \+ ease of implementation
- suitable for:
  - a set of independent jobs
  - jobs with unknown costs
  - jobs where locality does not matter
- unsuitable for too small jobs – due to the communication overhead
  - ⇒ coupling jobs into bulks
    - *fixed size*
    - *controlled coupling*
    - *tapering*
    - *weighted distribution*

Scheduling/Load-balancing in DSs – Approaches

Self-scheduling II. – Fixed size & Controlled coupling

Fixed size

- typical offline algorithm
- requires much information (number and cost of each job, ...)
- it is possible to find the optimal solution
- theoretically important, not suitable for practical solutions

Controlled coupling

- uses bigger bulks in the beginning of the execution, smaller bulks in the end of the execution
  - lower overhead in the beginning, finer coupling in the end
- the bulk's size is computed as: $K_i = \lceil \frac{R_i}{p} \rceil$
  where:
  - R_i ...the number of remaining jobs
  - p ...the number of processors

Scheduling/Load-balancing in DSs – Approaches

Self-scheduling II. – Tapering & Weighted distribution

Tapering

- analogical to the Controlled coupling, but the bulks' size is further a function of jobs' variation
- uses historical information
  - low variance $\Rightarrow$ bigger bulks
  - high variance $\Rightarrow$ smaller bulks

Weighted distribution

- considers the nodes' computational power
- suitable for heterogenous systems
- uses historical information as well

Scheduling/Load-balancing in DSs – Approaches

Distributed Queues

- ≈ self-scheduling for distributed memory
- instead of a centralized pool, a queue on each node is used (per-processor queues)
- suitable for:
  - distributed systems, where the locality does not matter
  - for both static and dynamic dependencies
  - for unknown costs
- an example: diffuse approach
  - in every step, the cost of jobs remaining on each processor is computed
  - processors exchange this information and perform the balancing
  - locality must not be important

FACULTY OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Approaches

Centralised Pool vs. Distributed Queues

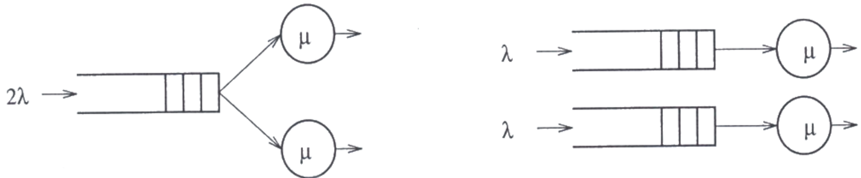


Figure: Centralised Pool (left) vs. Distributed Queues (right).

FACULTY OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

DAG Planning

DAG Planning

- another graph model
  - the nodes represent the jobs (possibly weighted)
  - the edges represent the dependencies and/or the communication (may be also weighted)
- e.g., suitable for digital signal processing
- basic strategy – divide the DAG so that the communication and the processors' occupation (time) is minimized
  - NP-complete problem
  - takes the dependencies among the jobs into account

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Design Issues I.

When the scheduling/load-balancing is necessary?

- for middle-loaded systems
  - lowly-loaded systems – rarely job waiting (there's always an idle processor)
  - highly-loaded systems – little benefit (the load-balancing cannot help)

What is the performance metric?

- mean response time

What is the measure of load?

- must be easy to measure
- must reflect performance improvement
- example: queue lengths at CPU, CPU utilization

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Design Issues I.

Types of policies:

- *static* (decisions hardwired into system), *dynamic* (uses load information), *adaptive* (policy varies according to load)

Policies:

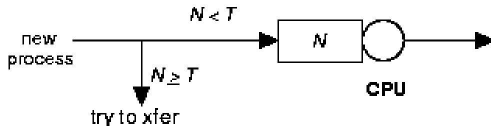
- *Transfer policy:* when to transfer a process?
  - threshold-based policies are common and easy
- *Selection policy:* which process to transfer?
  - prefer new processes
  - transfer cost should be small compared to execution cost
    - $\Rightarrow$ select processes with long execution times
- *Location policy:* where to transfer the process?
  - polling, random, nearest neighbour, etc.
- *Information policy:* when and from where?
  - demand driven (only a sender/receiver may ask for), time-driven (periodic), state-change-driven (send update if load changes)

FACULTY OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Design Issues II.

Sender-initiated Policy

- *Transfer policy*



- *Selection policy:* newly arrived process
- *Location policy:* three variations
  - Random – may generate lots of transfers
    - ⇒ necessary to limit max transfers
  - Threshold – probe n nodes sequentially
    - transfer to the first node below the threshold, if none, keep job
  - Shortest – poll N_p nodes in parallel
    - choose least loaded node below T
    - if none, keep the job

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Design Issues II.

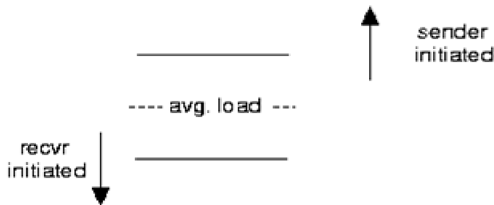
Receiver-initiated Policy

- *Transfer policy:* if departing process causes load $< T$, find a process from elsewhere
- *Selection policy:* newly arrived or partially executed process
- *Location policy:*
  - Threshold – probe up to N_p other nodes sequentially
    - transfer from first one above the threshold; if none, do nothing
  - Shortest – poll n nodes in parallel
    - choose the node with heaviest load above T

Scheduling/Load-balancing in DSs – Design Issues II.

Symmetric Policy

- combines previous two policies without change
  - nodes act as both senders and receivers
- uses average load as the threshold



FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Case study

V-System (Stanford)

- state-change driven information policy
  - significant change in CPU/memory utilization is broadcast to all other nodes
- *M* least loaded nodes are receivers, others are senders
- sender-initiated with new job selection policy
- *Location policy:*
  - probe random receiver
  - if still receiver (below the threshold), transfer the job
  - otherwise try another

FACULTY OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Case study

Sprite (Berkeley) I.

- *Centralized information policy:* coordinator keeps info
  - state-change driven information policy
  - Receiver: workstation with no keyboard/mouse activity for the defined time period (30 seconds) and below the limit (active processes $<$ number of processors)
- *Selection policy:* manually done by user $\Rightarrow$ workstation becomes sender
- *Location policy:* sender queries coordinator
- the workstation with the foreign process becomes sender if user becomes active

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Case study

Sprite (Berkeley) II.

- *Sprite process migration:*
    - facilitated by the Sprite file system
    - state transfer:
        - swap everything out
        - send page tables and file descriptors to the receiver
        - create/establish the process on the receiver and load the necessary pages
        - pass the control
    - the only problem: communication-dependencies
        - solution: redirect the communication from the workstation to the receiver

Scheduling/Load-balancing in DSs

Process Migration

- A transfer of a process from one environment/machine to another
- Two types
  - *Non-preemptive* – process is moved before the execution started
  - *Preemptive* – process is moved during the execution
    - process = code + data + stack

  We also speak about *weak* and *strong* mobility, resp.
- key reasons: *performance* and *flexibility*
- flexibility:
  - dynamic configuration of distributed system
  - clients don't need preinstalled software (download on demand)

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs

Process Migration in Heterogenous Systems

- In general, only weak mobility is supported in common systems (recompile code, no run time information)
- Hiding the heterogeneity
  - Interpreters (scripting languages)
  - Programming language oriented virtual machines (e.g. Java)
  - Full virtual machines

  These approaches can support preemption even in heterogenous environment

FACULTY
OF INFORMATICS
Masaryk University

Lecture overview

Distributed Systems
- Key characteristics
- Challenges and Issues
- Distributed System Architectures
- Inter-process Communication

Middleware
- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

Service Oriented Architecture (SAO)

Web Services

Issues Examples
- Fault Tolerance in Distributed Systems
- Replication in Distributed Systems
- Transactions in Distributed Systems
- Scheduling/Load-balancing in Distributed Systems

Conclusion