FACULTY
OF INFORMATICS
Masaryk University

PA160: Net-Centric Computing II.

Distributed Systems

Luděk Matyska

Slides by: Tomáš Rebok

Spring 2022

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

Conclusion

FACULTY
OF INFORMATICS
Masaryk University

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

FACULTY
OF INFORMATICS
Masaryk University

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

Load-balancing

The term **load-balancing** means assigning the jobs to the processors in the way, which minimizes the time/communication overhead necessary to compute them.

- load-balancing – divides the jobs among the processors
- scheduling – defines execution order of the jobs (on each processor)
  - load-balancing and planning are tightly-coupled (synonyms in DSs)
- **objectives:**
  - enhance overall system performance metric
    - process completion time and processor utilization
  - location and performance transparency

FACULTY
OF INFORMATICS
Masaryk University

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

Scheduling/Load-balancing in Distributed Systems

An illustration

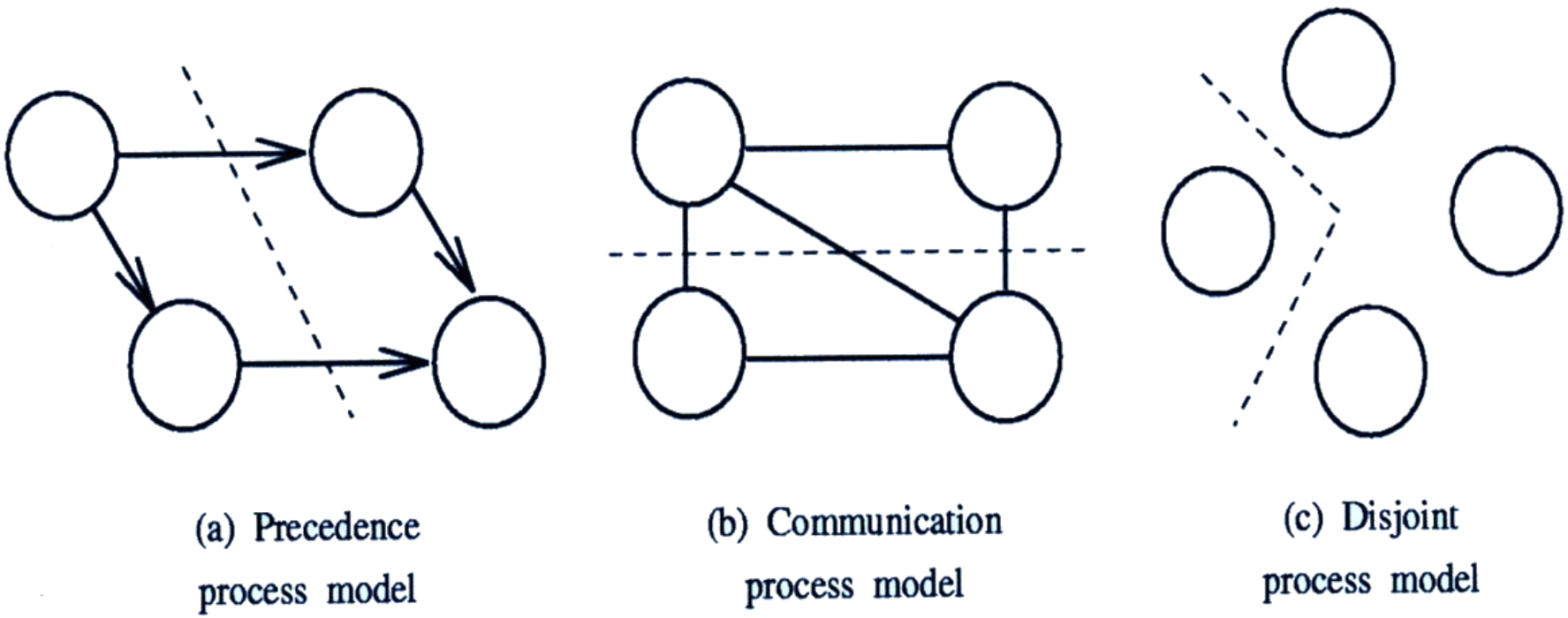


Figure: An illustration of splitting 4 jobs onto 2 processors.

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*

FACULTY
OF INFORMATICS
Masaryk University

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

- communicate all the jobs 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

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

- 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

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

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)

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

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
  - $\Rightarrow$ coupling jobs into bulks
    - fixed size
    - controlled coupling
    - tapering
    - weighted distribution

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

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

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

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

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs – Solving Methods

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

Scheduling/Load-balancing in DSs – Solving Methods

Centralised Pool vs. Distributed Queues

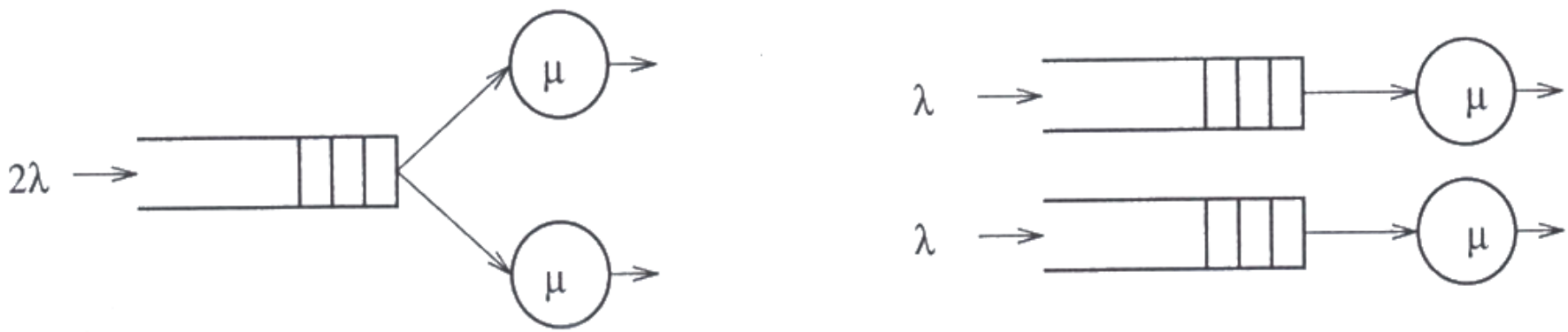


Figure: Centralised Pool (left) vs. Distributed Queues (right).

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:

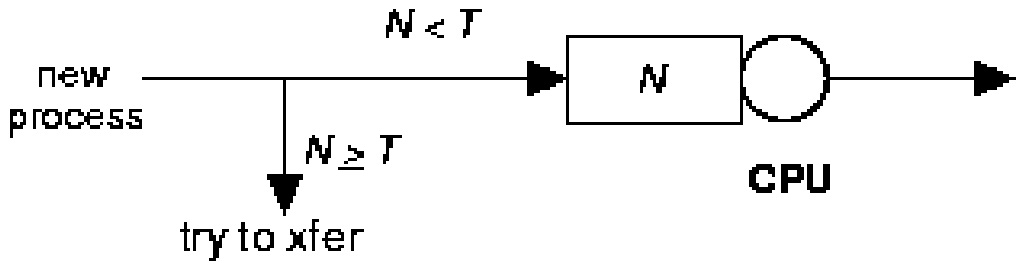
- *Transfer policy:* when to transfer a process?
  - threshold-based policies are common and easy
- *Selection policy:* which process to transfer?
  - prefer new processes
  - transfer cost should be small compared to execution cost
    - $\Rightarrow$ select processes with long execution times
- *Location policy:* where to transfer the process?
  - polling, random, nearest neighbor, 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
    - $\Rightarrow$ 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

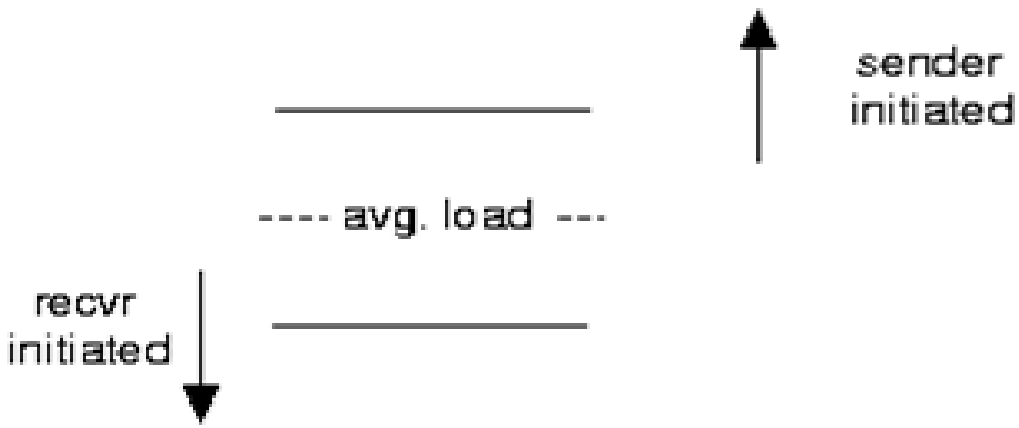
- *Transfer policy:* if departing process causes load $< T$, find a process from elsewhere
- *Selection policy:* newly arrived or partially executed process
- *Location policy:*
  - Threshold – probe up to N_p other nodes sequentially
    - transfer from first one above the threshold; if none, do nothing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

FACULTY
OF INFORMATICS
Masaryk University

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

FACULTY
OF INFORMATICS
Masaryk University

Scheduling/Load-balancing in DSs

Code and Process Migration

- key reasons: *performance* and *flexibility*
- flexibility:
  - dynamic configuration of distributed system
  - clients don't need preinstalled software (download on demand)
- process migration (*strong mobility*)
  - process = code + data + stack
  - examples: Condor, DQS
- code migration (*weak mobility*)
  - transferred program always starts from its initial state
- *migration in heterogeneous systems:*
  - only weak mobility is supported in common systems (recompile code, no run time information)
  - the virtual machines may be used: interpreters (scripts) or intermediate code (Java)

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in Distributed Systems I.

- single machine systems
  - failures are all or nothing
    - OS crash, disk failures, etc.
- distributed systems: multiple independent nodes
  - partial failures are also possible (some nodes fail)
  - probability of failure grows with number of independent components (nodes) in the system
- **fault tolerance:** system should provide services despite faults
  - *transient faults*
  - *intermittent faults*
  - *permanent faults*

Fault Tolerance in Distributed Systems I.

Failure Types

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in Distributed Systems II.

- *handling faulty processes:* through redundancy
- organize several processes into a group
  - all processes perform the same computation
  - all messages are sent to all the members of the particular group
  - majority needs to agree on results of a computation
  - ideally, multiple independent implementations of the application are desirable (to prevent identical bugs)
- use process groups to organize such processes

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in Distributed Systems III.

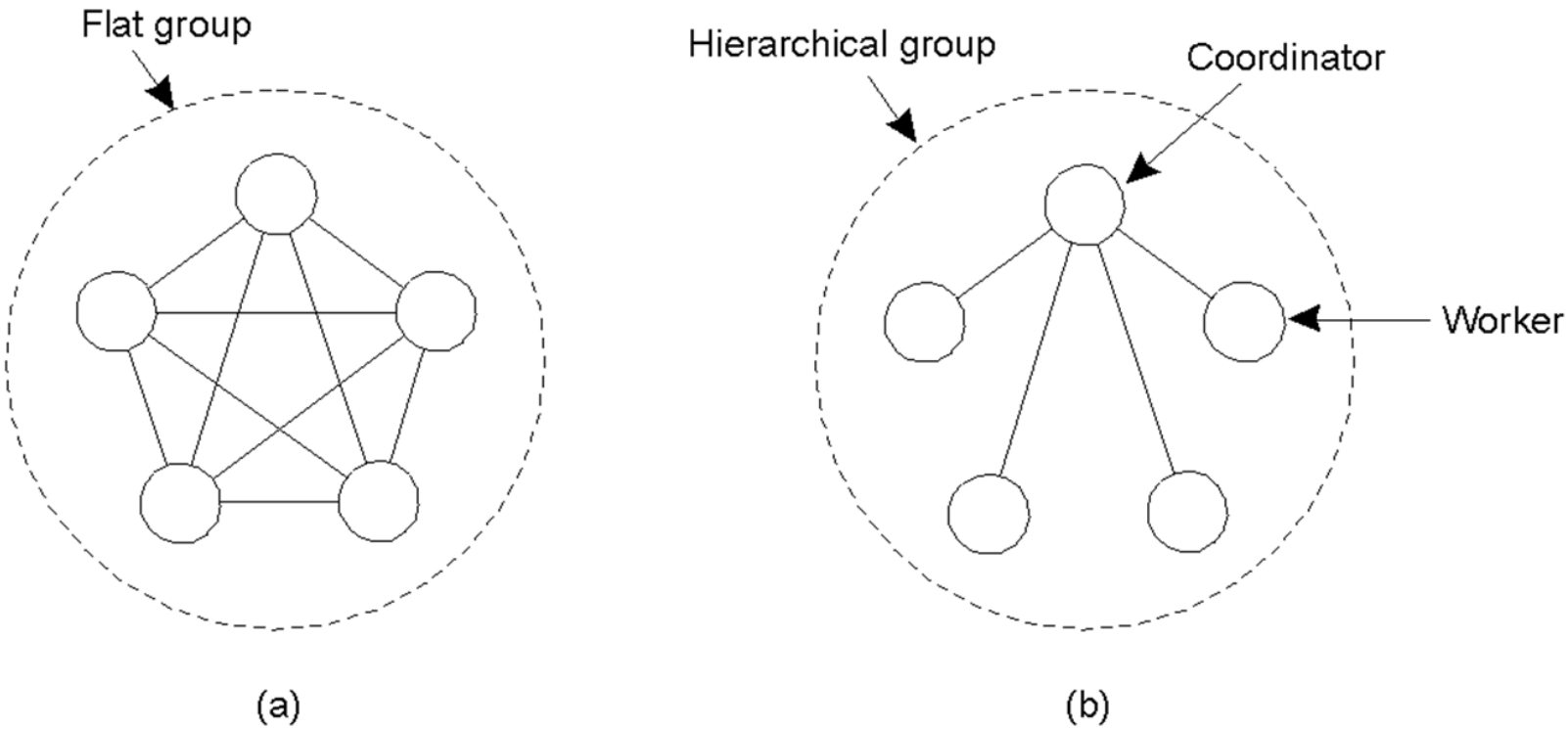


Figure: Flat Groups vs. Hierarchical Groups.

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

- How should processes agree on results of a computation?
- *K-fault tolerant:* system can survive k faults and yet function
  - assume processes fail silently
    - $\Rightarrow$ need $(k+1)$ redundancy to tolerant k faults
- *Byzantine failures:* processes run even if sick
  - produce erroneous, random or malicious replies
  - byzantine failures are most difficult to deal with

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Byzantine Generals Problem:

- four generals lead their divisions of an army
  - the divisions camp on the mountains on the four sides of an enemy-occupied valley
- the divisions can only communicate via messengers
  - messengers are totally reliable, but may need an arbitrary amount of time to cross the valley
    - they may even be captured and never arrive
- if the actions taken by each division is not consistent with that of the others, the army will be defeated
- we need a scheme for the generals to agree on a common plan of action (attack or retreat)
  - even if some of the generals are traitors who will do anything to prevent loyal generals from reaching the agreement
- the problem is nontrivial even if messengers are totally reliable
  - with unreliable messengers, the problem is very complex
  - Fischer, Lynch, Paterson: in asynchronous systems, it is impossible to reach a consensus in a finite amount of time

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Formal definition of the agreement problem in DSs:

- let's have a set of distributed processes with initial states $\in 0, 1$
- *the goal:* all the processes have to agree on the same value
  - additional requirement: it must be possible to agree on both 0 or 1 states
- basic assumptions:
  - *system is asynchronous*
    - no bounds on processes' execution delays exist
    - no bounds on messages' delivery delay exist
    - there are no synchronized clocks
  - *no communication failures* – every process can communicate with its neighbors
  - *processes fail by crashing* – we do not consider byzantine failures

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Formal definition of the agreement problem in DSs: cont'd.

- *implications:*
  - $\Rightarrow$ there is no deterministic algorithm which resolves the consensus problem in an asynchronous system with processes, which may fail
  - because it is impossible to distinguish the cases:
    - a process does not react, because it has failed
    - a process does not react, because it is slow
  - practically overcomed by establishing timeouts and by ignoring/killing too slow processes
    - timeouts used in so-called Failure Detectors (see later)

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast

- *if there was a proper type of fault-tolerant broadcast, the agreement problem would be solvable*
- various types of broadcasts:
  - *reliable broadcast*
  - *FIFO broadcast*
  - *causal broadcast*
  - *atomic broadcast* – the broadcast, which would solve the agreement problem in asynchronous systems

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Reliable Broadcast

- basic features:
  - Validity – if a correct process broadcasts m, then it eventually delivers m
  - Agreement – if a correct process delivers m, then all correct processes eventually deliver m
  - (Uniform) Integrity – m is delivered by a process at most once, and only if it was previously broadcasted
- possible to implement using `send`/`receive` primitives:
  - the process p sending the broadcast message marks the message by its identifier and sequence number
    - and sends it to all its neighbors
  - once a message is received:
    - if the message has not been previously received (based in sender's ID and sequence number), the message is delivered
    - if the particular process is not message's sender, it delivers it to all its neighbors

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – FIFO Broadcast

- the reliable broadcast cannot assure the messages' ordering
  - it is possible to receive a subsequent message (from the sender's view) before the previous one is received
- *FIFO broadcast:* the messages from a single sender have to be delivered in the same order as they were sent
- FIFO broadcast = Reliable broadcast + FIFO ordering
  - if a process p broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m
  - $broadcast_p(m) \rightarrow broadcast_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$
- a simple extension of the reliable broadcast

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Causal Broadcast

- the FIFO broadcast is still not sufficient: it is possible to receive a message from a third party, which is a reaction to a particular message before receiving that particular message
  - $\Rightarrow$ *Causal broadcast*
- Causal broadcast $=$ Reliable broadcast $+$ causal ordering
  - if the broadcast of a message m *happens before* the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m
  - $broadcast_p(m) \rightarrow broadcast_q(m') \Rightarrow deliver_r(m) \rightarrow deliver_r(m')$
- can be implemented as an extension of the FIFO broadcast

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Atomic Broadcast

- even the causal broadcast is still not sufficient: sometimes, it is necessary to guarantee the proper in-order delivery of all the replicas
  - two bank offices: one of them receives the information about adding an interest before adding a particular amount of money to the account, the second one receives these messages contrariwise
    - $\Rightarrow$ inconsistency
  - $\Rightarrow$ Atomic broadcast
- Atomic broadcast = Reliable broadcast + total ordering
  - if correct processes p and q both deliver messages m, m', then p delivers m before m' if and only if q delivers m before m'
  - $deliver_p(m) \rightarrow deliver_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$
- does not exist in asynchronous systems

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Fault-tolerant Broadcast – Timed Reliable Broadcast

- a way to practical solution
- introduces an upper limit (time), before which every message has to be delivered
- Timed Reliable broadcast $=$ Reliable broadcast $+$ timeliness
  - there is a known constant Δ such that if a message is broadcasted at real-time t, then no correct (any) process delivers m after real-time $t + \Delta$
- feasible in asynchronous systems
- A kind of “approximation” of atomic broadcast

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems – Failure Detectors I.

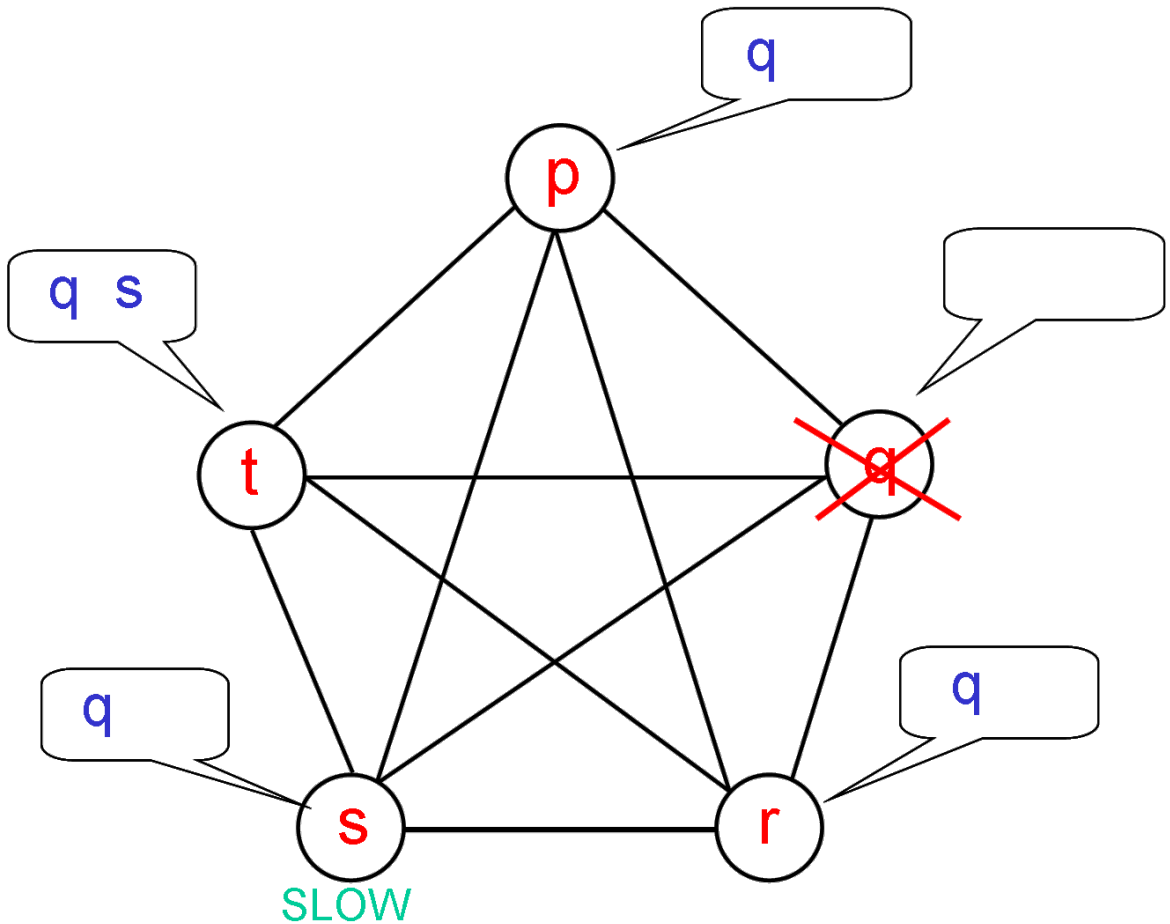
- impossibility of consensus caused by inability to detect slow process and a failed process
  - synchronous systems: let's use timeouts to determine whether a process has crashed
  - $\Rightarrow$ Failure Detectors

Failure Detectors (FDs):

- a distributed oracle that provides hints about the operational status of processes (which processes had failed)
  - FDs communicate via atomic/time reliable broadcast
- every process maintains its own FD
  - and asks just it to determine, whether a process had failed
- however:
  - hints may be incorrect
  - FD may give different hints to different processes
  - FD may change its mind (over & over) about the operational status of a process

FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems – Failure Detectors II.



FACULTY
OF INFORMATICS
Masaryk University

Fault Tolerance in DSs – Agreement in Faulty Systems

Perfect Failure Detector:

- properties:
  - Eventual Strong Completeness – eventually every process that has crashed is permanently suspected by all non-crashed processes
  - Eventual Strong Accuracy – no correct process is ever suspected
- hard to implement
- is perfect failure detection necessary for consensus? **No.**
  - $\Rightarrow$ weaker Failure Detector

weaker **Failure Detector:**

- properties:
  - Strong Completeness – there is a time after which every faulty process is suspected by every correct process
  - Eventual Strong Accuracy – there is a time after which no correct process is suspected
- can be used to solve the consensus
  - this is the weakest FD that can be used to solve the consensus

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

Conclusion

FACULTY
OF INFORMATICS
Masaryk University

Distributed Systems – Further Information

- **FI courses:**
  - PA150: Advanced Operating Sytems Concepts (doc. Staudek)
  - PA053: Distributed Systems and Middleware (doc. Tůma)
  - PA039: Supercomputer Architecture and Intensive Computations (prof. Matyska)
  - PA177: High Performance Computing (LSU, prof. Sterling)
  - IV100: Parallel and distributed computations (doc. Královič)
  - IB109: Design and Implementation of Parallel Systems (dr. Barnat)
  - etc.
- **(Used) Literature:**
  - W. Jia and W. Zhou. Distributed Network Systems: From concepts to implementations. Springer, 2005.
  - A. S. Tanenbaum and M. V. Steen. Distributed Systems: Principles and paradigms. Pearson Prencite Hall, 2007.
  - G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems: Concepts and design. Addison-Wesley publishers, 2001.
  - Z. Tari and O. Bukhres. Fundamentals of Distributed Object Systems: The CORBA perspective. John Wiley & Sons, 2001.
  - etc.