Distributed Systems

8 Migration/Load Balancing

May-25-2009

Gerd Liefländer

System Architecture Group

Today’s Schedule

- Classification of Migration
  - Code Migration
    - Process-/Task Migration Models
    - Implementation
  - Data Migration
- Implementation of Migration
- Load Balancing
- Motivation
- Taxonomy of Load Balancing Schemes
- Needed Information for Load Balancing
- Load Balancing Policies
- Load Sharing

What is Migration?

Two major approaches

- Code migration (traditional)
  - *Weak migration*: only code
    - Java class loading
  - *Strong migration*: code and execution state
    - Process migration
    - Java object migration (via RMI)
- Data migration (newer)
- Examples
  - Juggle: Automatic object and thread distribution in a VM
  - Java Party: A distributed Companion to Java
  - Emerald

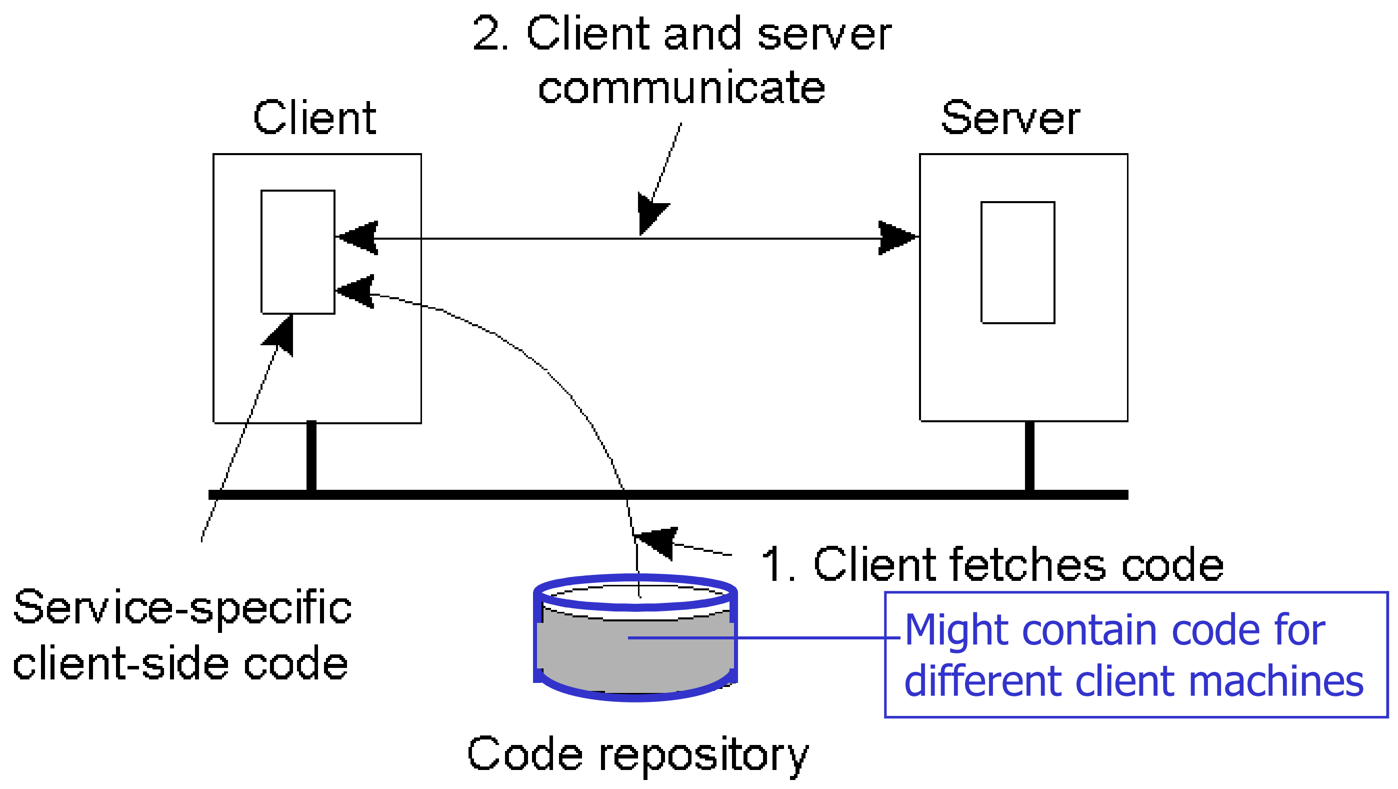
Why Migration?

- Performance
  - Move code on a faster machine
  - Move code to a lightly loaded machine
  - Move code closer to its data (e.g. a data base)
- Availability
  - Move code to a node that will not be shut down in the near future
- Flexibility
  - Allow to dynamically configure a distributed system

Performance Reasons

- Migrating a process to another node in a DS might induce a lot of migration overhead and later follow up costs
- However, migrating from a heavily loaded node to a lightly/loaded one might improve overall system performance
- A search query can be implemented a small program, moving from node to node collecting all search results
- A client processing a very large amount of data from a specific server may be better off executing on the server machine

Code Fetching to install a DS


2. Client and server communicate
Client
Server
Service-specific client-side code
1. Client fetches code
Might contain code for different client machines
Code repository

- Principle of dynamically configuring a client to a server.
  Client first fetches necessary software for future interaction with the sever, then it invokes the server.
- However, you have to trust the downloaded code

Traditional Code Migration

- Moving a “not yet created task” ~ downloading code
- Moving a non-active task to another machine is not that hard (in homogeneous systems)
  - Some resources at load time have to be released at the source node and reserved/allocated at the target node
- Moving an “active” task from one machine to another can involve a lot of overhead
  - ⇒ In any case, migrate **iff** you’re sure to gain either performance or availability
- When migrating an active task you can do
  - complete or partial migration
  - in any case you have to migrate a sufficient amount of its context state from one machine to another

Modeling for Code Migration

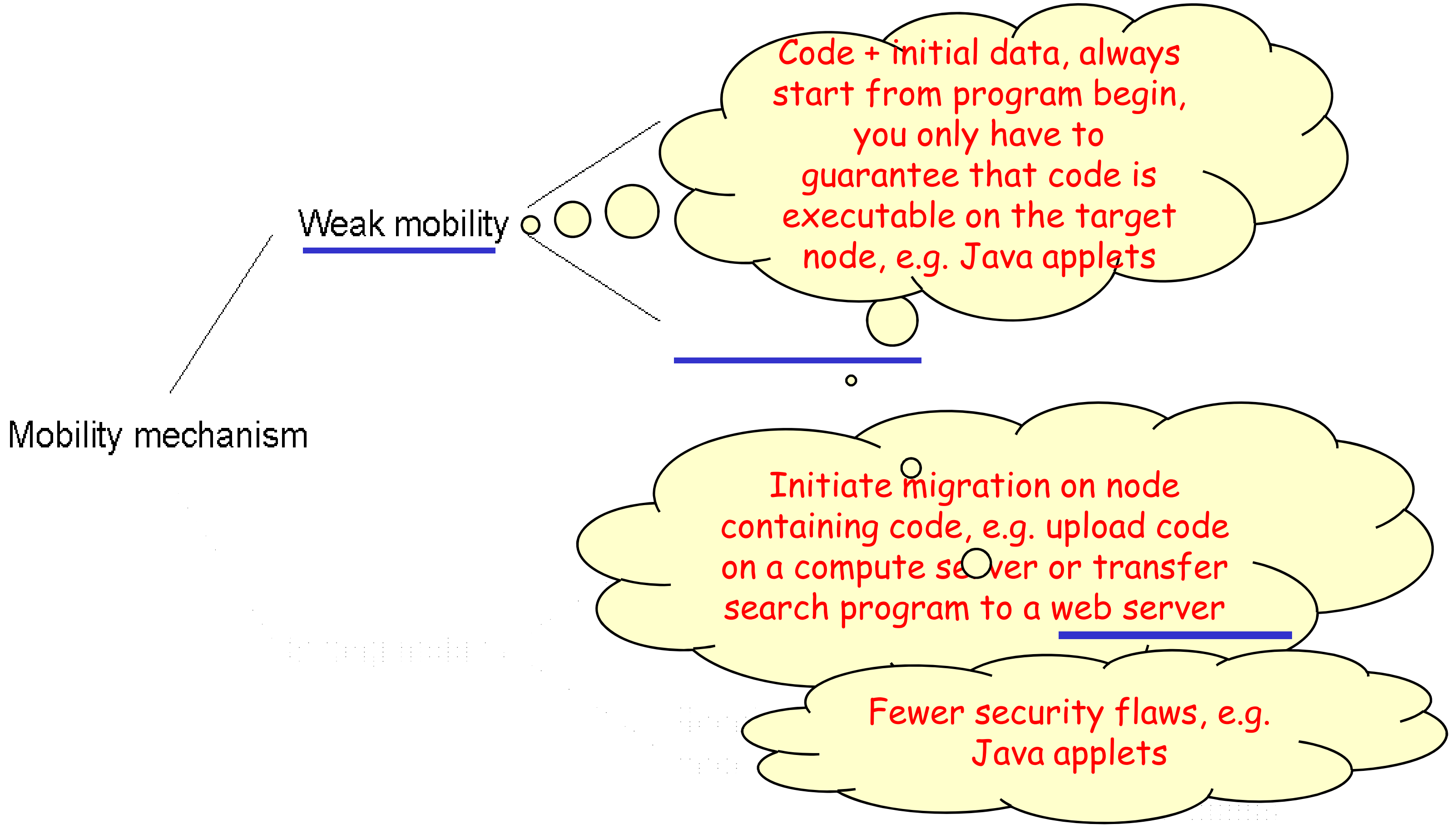
Framework described by Fugetta*:

A process consists of 3 "segments"

- Code segment
- Resource segment
  - Links(handles) to (external) resources, e.g.
    - files or devices
    - other processes
- Execution segment
  - Process context (environment)

* Fuggeta, A. et al: "Understanding Code Mobility", IEEE Trans. Software Engineering, 1999, p. 717

Models for Code Migration


Mobility mechanism
Weak mobility
Code + initial data, always start from program begin, you only have to guarantee that code is executable on the target node, e.g. Java applets
Initiate migration on node containing code, e.g. upload code on a compute server or transfer search program to a web server
Fewer security flaws, e.g. Java applets

- Alternatives for code migration

Weak Migration

- Migrate (download) a software component to a new target node, e.g.
  - Process or task
  - Object
  - …
- Relocation transparency
- Passive components
  - Migrate complete object or AS (i.e. data + code)
  - Must wait until current activity has terminated

Strong Migration

- Active software components
  - Migrate a running task or a process
  - Additionally migrate its
    - state & context
    - instruction pointer (program counter)
    - register set
    - stack …
  - Need support from OS because execution environment of activity must be preserved, e.g.
    - open files with current file pointer and access rights etc., IPC with remote or local partners
    - maintain a stub for forwarding incoming signals and messages to the target machine

Strong Process Migration

1. Stop active process
2. Take *snapshot* of process
3. Transmit snapshot + process to target
4. Recreate process at target node with given snapshot image
5. Bind open descriptors
   - Unattached, fastened, or fixed *resources*
   - Binding by identifier, value, or type
6. Resume process
   - OS and architecture specific
   - Language independent

Data Space Management & Binding

- Binding by *identifier* (strong)
  - Execution environment (EU) requires that at any time it is to this uniquely identified resource, i.e. this resource can not be substituted by another one of the same type
- Binding by *value* (mediocre)
  - At any moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration
- Binding by *type* (weak)
  - EU requires that at any moment the bound resource is compliant with a given type, no matter what its actual value or identity is
  - Typical for resources that are available at any node, like system variables, libraries or devices, e.g. a display

Bindings, Resources, DS Management

Degree of binding

Resource-to-machine binding

Rebind to another printer

	Unattached	Fastened	Fixed
By identifier	**MV or GR**	**GR** (or MV)	**GR**
By value	CP (or MV, GR)	GR (or CP)	GR
By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

Strongest form via ID, e.g. use an absolute URL for a specific web site in case of a shared resource, otherwise migrate resource together with the task

Binding by value is weaker, cause you only need to provide a *resource* with delivering the same value, e.g. using a standard library

By type is weakest binding form, e.g. usage of a local printer, you want to print on whatever printer

GR Establish a global system-wide reference	MV move the resource
CP Copy the value of the resource	RB Rebind task to locally available resource

Binding, Resources

Resource-to-machine binding

	Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV, GR)	GR (or CP)	GR
By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

Degree of binding →

Unattached resources are very easy to migrate,
e.g. a data file associated with a program

Fastened resources might be migrated but at high cost, e.g.
complete web sites or a local data base

Fixed resources ***cannot*** be migrated

Migration in Heterogeneous Systems

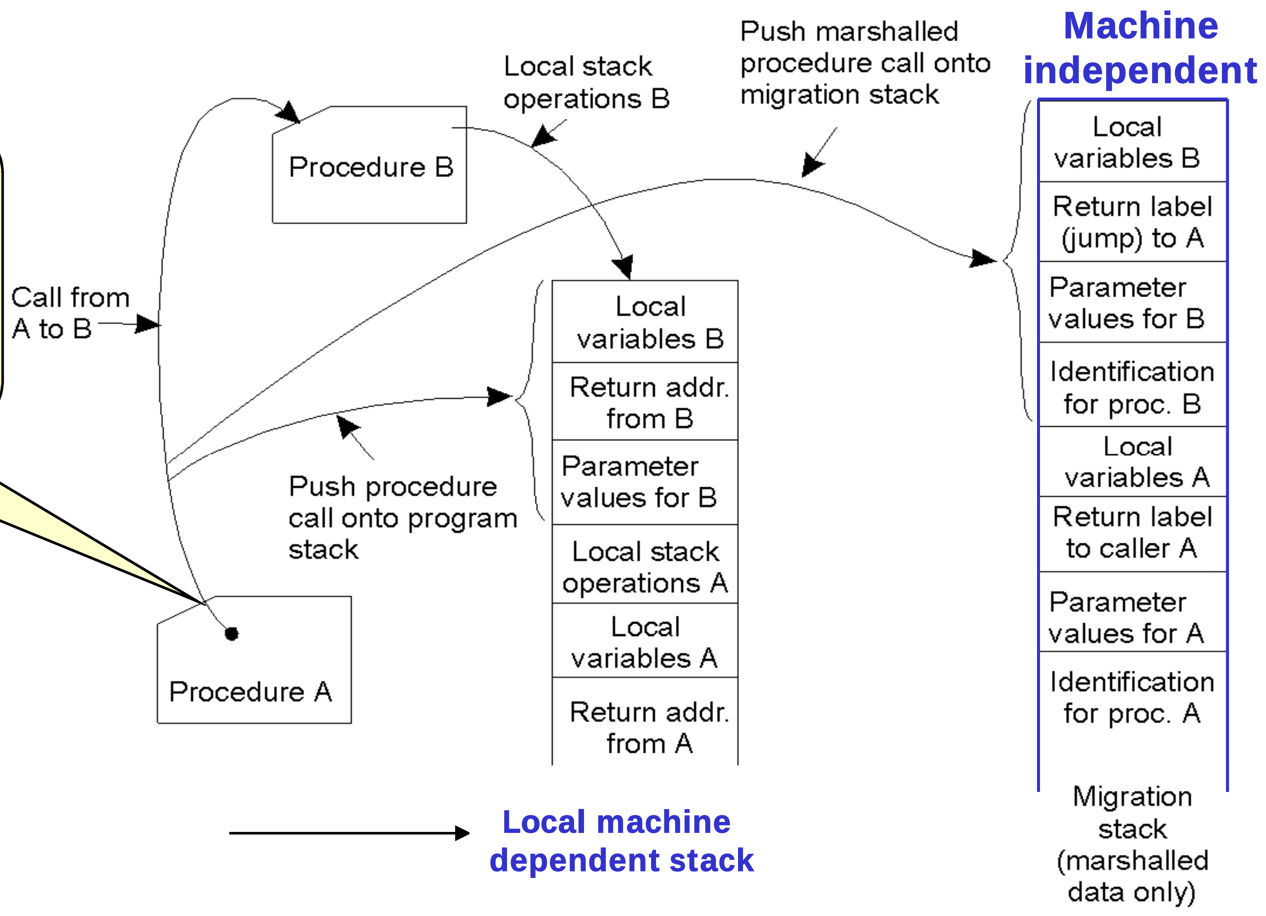
- Up to now we could expect, that a migrated process can be easily resumed on teh target machine
- What if the new machine has a different hardware?
- Make sure, that the program can be executed on each node, where it could be migrated to (either recompilation or support of multiple binary codes)
- Make sure that the execution segment is properly represented and interpreted by each platform
- Weak mobility is easy to achieve: simply recompile the program or maintain multiple binaries

Strong Migration in Het. Systems

- How to transform the execution segment? It is highly platform dependent
- Each execution segments contains the current stack (local values and register values)
- To transfer an execution segment, make sure no platform dependent data is stored
- Restrict code migration to specific points within the code, e.g. migration can take place only when a procedure is called; runtime system maintains a copy of the execution stack in a machine independent format-migration stack
- Migration stack is updated each time a procedure is called, or when a return from the procedure occurs

Migration in Heterogeneous Systems

Reduce migration points at run time, e.g. before next procedure call or use an intermediate code



Principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous distributed system

Implementing Migration

Implementing Task Migration

5 variants of migrating tasks

- eager all (complete)
- pre-copy
- eager dirty
- copy-on-reference
- flushing

Complete Migration

- Eager (all): Transfer entire task, i.e. with all 3 segments
  - Clean approach, no trace of task left behind
  - When the task was waiting for signals or messages, *how to inform the signaler or the sender?*

  ⇒Tasks with waiting signals from a peripheral cannot be migrated without a substitute at the source node that is able to forward all results from a previously initiated I/O

  - If AS = large and if task does not need most of it

    ⇒ this approach is quite expensive

Due to heavy traffic on the net and additional storage requirements on target machine this affects not only the migrating task, but also other non related tasks

Pre-Copy Migration

- Task continues to execute on source node while its AS is copied to the target
  - Pages that have been modified on the source during this pre-copy operation have to be copied a second time
  - Reduces time that a task is temporarily “frozen”

Eager Dirty Migration

- Transfer only mapped and modified pages
  - Transfer other pages *on demand* from background storage of source machine
    - *How to implement, e.g. the disk addresses of unmapped pages may be only valid on the source machine*
    - Two possibilities:
      - While copying the address space tables to the target machine, translate all disk addresses of the source to remote disk addresses
      - Source machine involved throughout the life of the task, i.e. it has to maintain page/segment tables and has to offer remote paging support, i.e. each page fault on the target machine is tunneled to the source machine
      - Good choice if task is only partly migrating to another machine (e.g. only a thread)

Copy-On-Reference Migration

- Migrate pages when referenced
  - variation of eager dirty
  - lowest initial cost of task migration

Flushing Migration

- Pages are cleared from main memory by flushing dirty pages to disk
  - Later use copy-on-reference policy
  - Relieves the source machine of holding mapping information for migrated task in its main memory

Summary

- If a task is a multi-threaded application and the basic migration unit is a thread, then use:
  - eager dirty or
  - copy-on-reference or
  - flushing

Similar considerations apply if a migrated task has open files,
i.e. a thread running on target machine might never access the file,
so why should we migrate open files when migrating a thread?

Distributed Systems

8 Load Balancing

May-25-2009

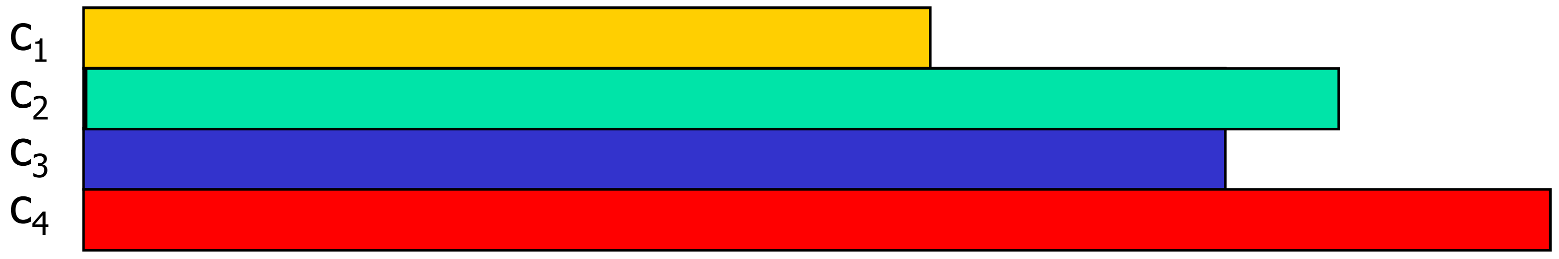
Gerd Liefländer

System Architecture Group

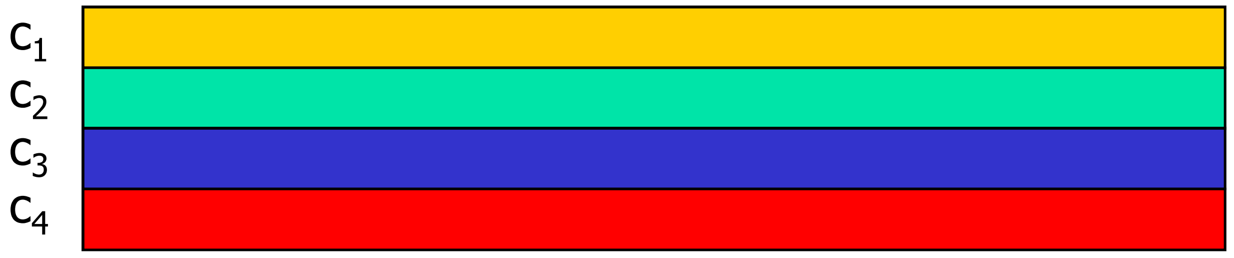
Motivation

Why Load Balancing?

- To achieve a *fair & robust distribution of computations* across nodes to increase performance & availability



Imperfect load balancing



Perfect load balancing

Idea behind Load Balancing

- Try to effectively and efficiently use your resources in your DS
- Try to get system & performance data describing the current and future load of each node in your DS as precisely as possible, but also as cheap as possible
- Try to satisfy
  - your customers by low turnaround times
  - as well as your managers by high resource usage, but low power consumption

Principles of Load Balancing

- Distributed server, e.g.
  - Dispatcher + w worker processes $\Rightarrow$
  - Load can be easily distributed to w nodes
- Load
  - Any instance that consumes resources like
    - CPU
    - Ram usage
    - Network bandwidth …
  - e.g. tasks, processes, KLTs

Principles of Load Balancing

- Distributed Multiprocessing Server, e.g.
  - Team model
    - Worker get requests from a global mailbox, e.g. ray-tracing (pull-model)
  - Pipeline-model
    - Intermediate results are handed from process to process

Load Balancing on n>1 Workstations

- Often a WS is not fully used
  - Users often do other things
  - During night a WS is almost inactive completely
- Start workers on currently not used WS
  - Problem: trust
    - Need trusted WS
  - Problem: user wants to use its WS
    - Stop worker process (of a remote machine)
    - Abort worker process and start somewhere else
    - *Migrate* running worker process

$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ *Which one is fitting?*

Taxonomy of Load Balancing

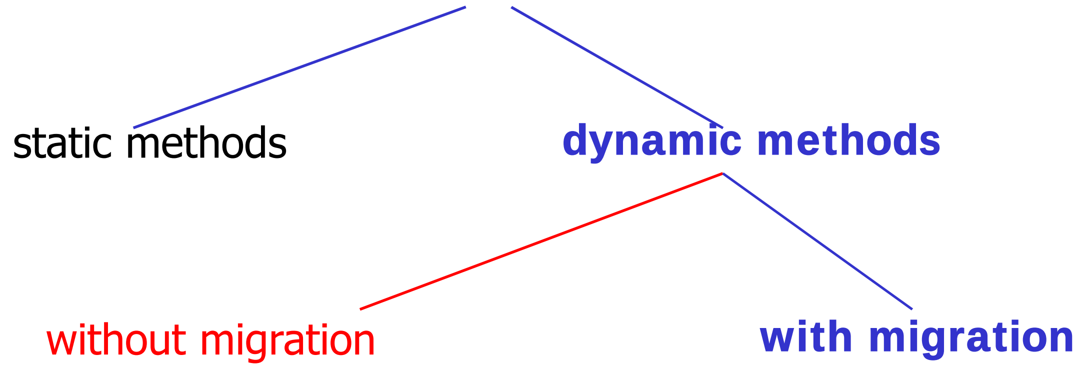
Design Parameters of LB*

- Static versus dynamic
- Deterministic versus probabilistic
- Centralized versus distributed
- Cooperative versus non-cooperative

*LB = Load Balancing

Algorithms for Load Balancing

- Problem
  - w tasks with a given execution and communication behavior
  - *What is the optimal load balancing?*
    - Avoid resource bottlenecks
      - *How to get info on future resource utilization?*
    - Enable efficient execution of requests
- Classification of load balancing algorithms



Static versus Dynamic LB

- Static load balancing
  - Calculate at boot time an optimal distribution of the load
  - Balancing is done whenever a new distributed application will be created
- Dynamic load balancing without migration
  - Whenever you create within a distributed application a new task or process or a new KLT
  - Take into account the current load on all nodes
    - How to get the actual system states?
  - Inform the load balancing node or all other nodes
- Dynamic load balancing with migration
  - Whenever you measure a significant over-/underload try to export/import processes

Static Load Balancing

- Round Robin: whenever a task has to be created, it is created on the next node (chained in a logical ring)
- Randomized: Allocate a new task at random
- Recursive bisection: recursively divide the allocation problem into sub-problems of equal computational effort
  - The problem of allocating tasks to nodes for arbitrary networks is NP-hard

  ⇒ No efficient polynomial time algorithm exists, i.e. we have to live with heuristics. However, there are some interesting static load balancing algorithms

Summary: Static Load Balancing

- When a good mathematical solution exists, static load balancing has the following drawbacks:
  - It is difficult to estimate a-priori [in a accurate way] the executions times of various parts of the program
  - Sometimes there are non negligible communication delays that vary in an uncontrollable way
  - For some problems the number of steps to reach a solution is not known in advance

Dynamic Load Balancing

- Allocating a task or parts of it, is done during the execution of the task

- <u>Features:</u>
  - Drawbacks of static load balancing are taken into account, improving the efficiency of load balancing
  - There is an additional overhead during execution, i.e. how to avoid unnecessary load state messages
  - Termination detection of the tasks is more complicate

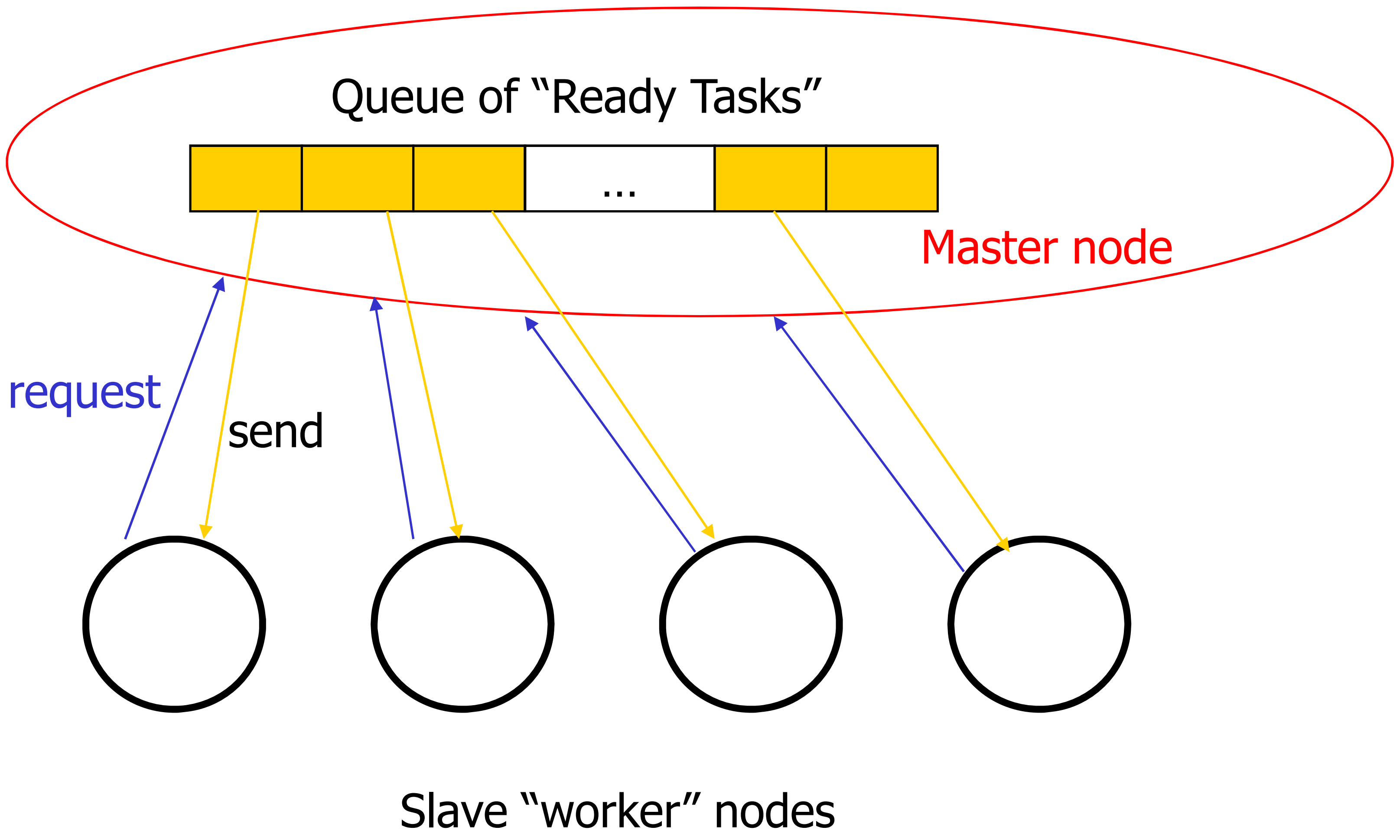
Types of Dynamic Load Balancing

- Centralized load balancing:
  - Tasks are allocated from some master node, master/slave system architecture
  - The centralized master node is again a single point of failure and might become a bottleneck
- Decentralized load balancing:
  - Worker nodes interact among themselves to solve the problem, finally reporting to a single node
  - Tasks are passed between arbitrary nodes, a worker node can receive tasks from any other worker node and can send tasks to any other worker node

Centralized Dynamic LB

- Good, when there is a small number of slaves and the problem consists of computationally intensive tasks
- Basic features:
  - A master node holds the collection of tasks/processes to be performed
  - Tasks are sent to the slave/worker nodes
  - When a slave has completed one task, it requests another one from the master node
- The following terms reflect a centralized load balancing scheme: work pool, replicated worker, processor farm
- Technically, it is more efficient to start with the long runners, i.e. try to do some LPT scheduling in the large

Centralized DLB

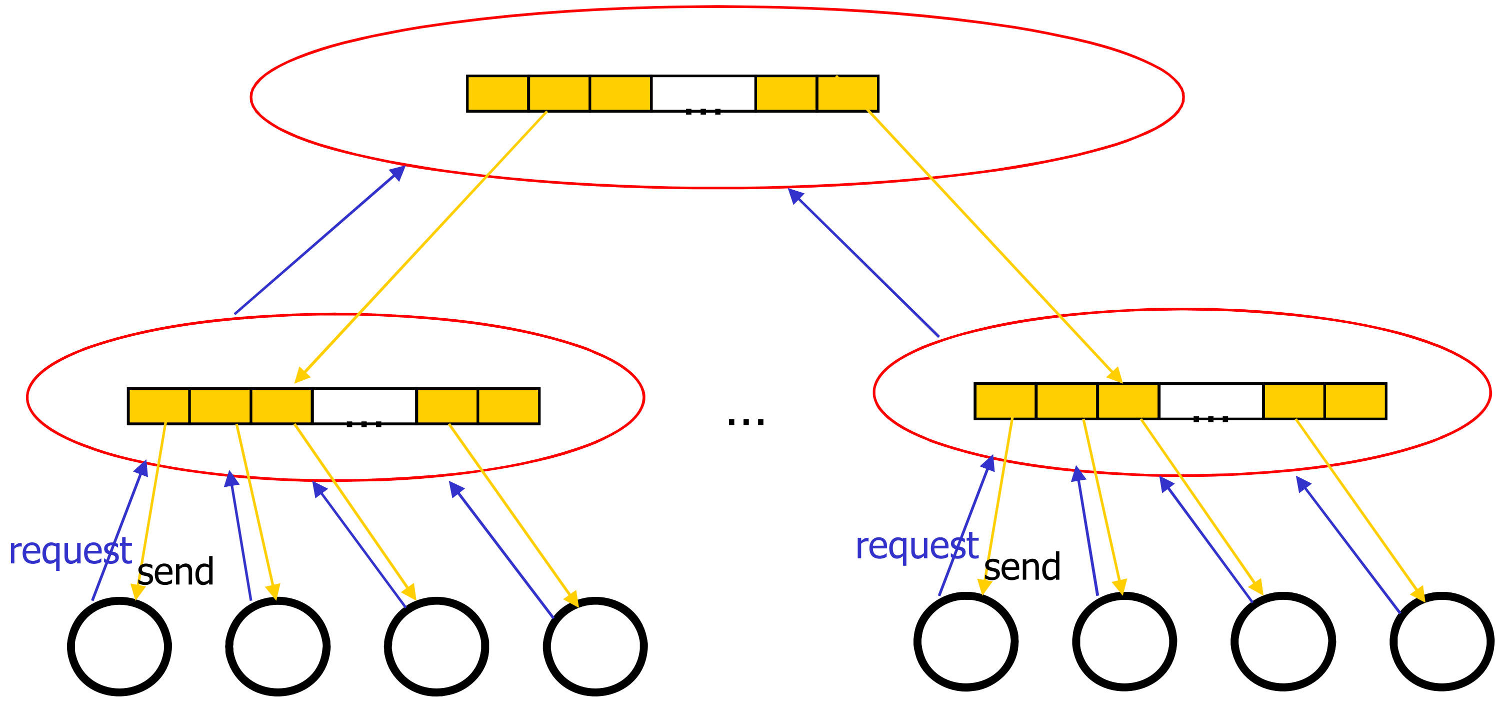


Termination in Centralized DLB

- Stop the computation when the solution has been found
- When the tasks are taken from a task queue, computation terminates when
  - the task queue is empty and
  - every node has made a request for another task without any new tasks being generated
- Note: It is not sufficient to check if the master’s task queue is empty, as long as worker nodes are allowed to put tasks in the task queue
- In some applications a slave can detect the program termination by some local termination, for instance finding an item in a search algorithm

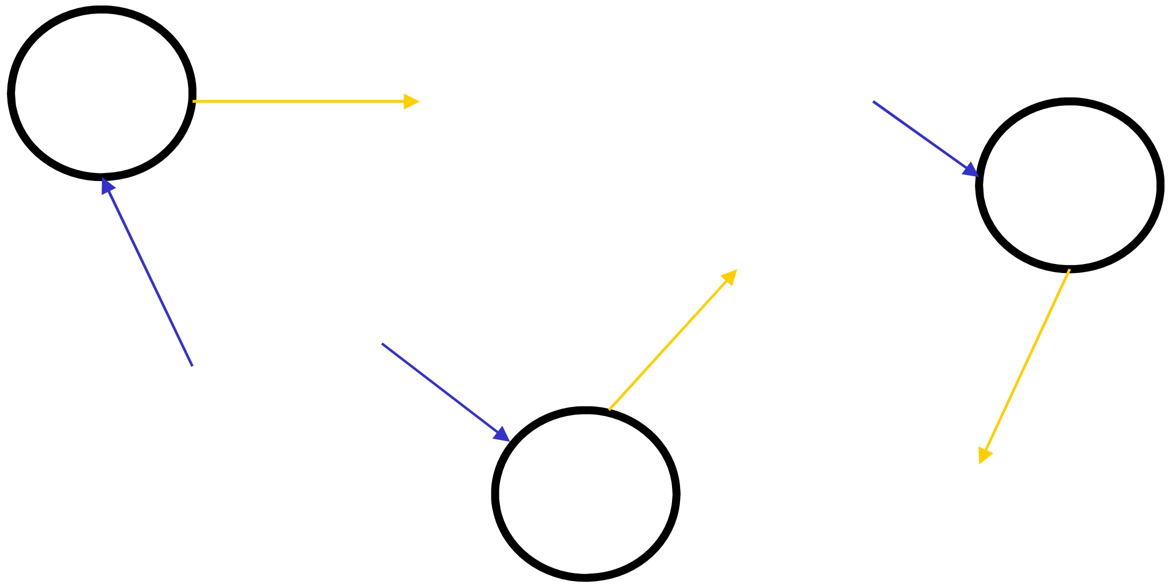
Decentralized DLB (1)

- Tree structured worker pool



Decentralized DLB (2)

- General (fully distributed) worker pool



Triggering Migration

1. Receiver initiated
   - Node requests tasks from another node it selects; typically done when the node has few or no tasks to compute
   - Method works well when there is a high system load
2. Sender initiated
   - Node sends tasks to other nodes it selects; typically done when node has already a heavy load and can find other nodes willing to accept additional load
   - Method works well when there is only light system load

Final comments:

- Above pure approaches can be mixed
- However, whatever method one uses, in very high system loads, load balancing is difficult due to the lack of node capacity

Node Selection in DLB

- Assumption: There are n nodes N_1, … N_n in the DS
- Round Robin: node N_i requests tasks from node N_x, where x is given by a counter that is incremented modulo n, excluding x=i
- Random Polling: Node N_i requests tasks from node N_x, where x is a number that is randomly selected from the set I={1, 2, …i-1, i+1, …, n}

Termination Conditions

- The [application specific] local termination condition are satisfied by all application members on all involved nodes
- There are no messages in transit between these nodes concerning the distributed application
- Note: The second condition is necessary to avoid situations where a message in transit might restart an already terminated task. This case is not easy to check, as long as communication times are not known in advance

Needed Information for LB

Local Load Measuring/Calculating

- You can measure usage patterns of
  - CPU
  - Memory
  - I/O
  - Power
  - …
- With an aging coefficient it's possible to predict the future behavior (see: principle of locality)
- *However, how to decide, that a node, its CPU or any another device is/are overloaded?*
- If there a different nodes we must take into account the different capacities of these nodes

Distinguishable Load States

- Underloaded:
  - New local work can be done
  - New remote work can be done
- Acceptably loaded:
  - No new work can be accepted, i.e.
    - New local work must be postponed or must be exported to another node (e.g. to the least loaded neighbour)
    - Remote work has to be rejected
- Overloaded:
  - New and/or current work has to be migrated

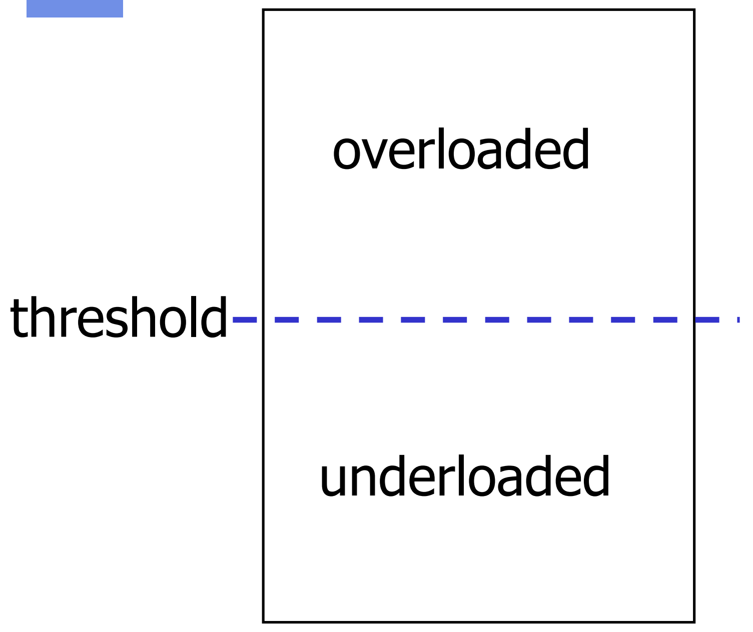
Global Load Calculating

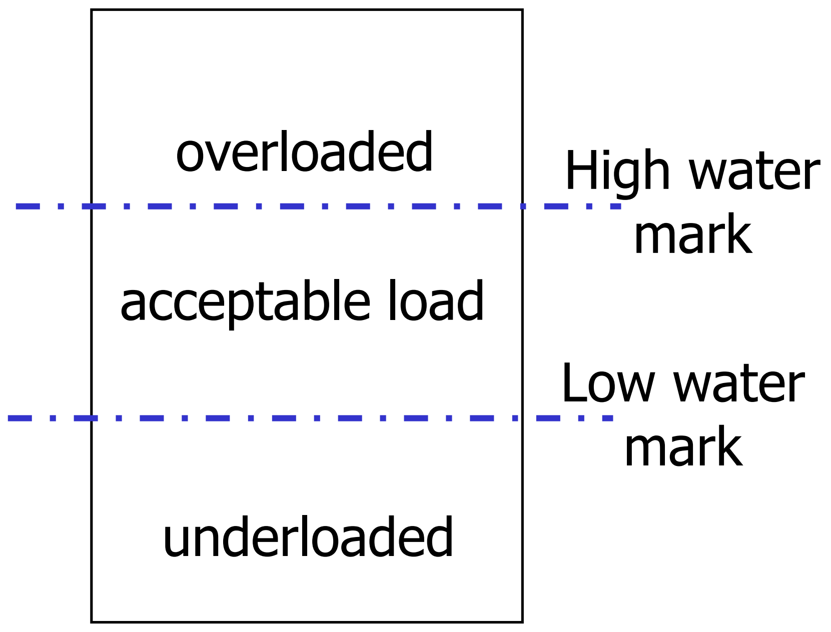
- *How to avoid significant overhead getting the necessary load information of each node in the DS?*
  - Only collect status from the neighbors
  - Broadcast the local status periodically, but not that often (large Δt)
    - As long as N = number of nodes is low and ∃ LAN, these broadcast messages do not cost too much
- *When do we need this information?*
  - Whenever creating a new application you have to decide:
    - Establish it on the local node
    - Postpone it
    - Establish it on a remote node

Local Load Determination

- How to measure the current workload of a node?
- Not an easy problem, up to now there is not yet THE SOLUTION
- Calculating the local load must be fast
- You can measure/estimate the following entities:
  - # of active threads/processes
  - Sum of all resource requirements
  - Instruction mix of the threads/processes
  - Architecture and speed of the node's CPUs
  - Remaining execution times of the threads/processes

Decision for Migration





Load Balancing Policies

Determination of Target Machine

- *Where to migrate a process?*
- Potential policies
  - Threshold policy
  - Shortest policy
  - Bidding policy
  - Pairing policy

Threshold Policy

1. Chose potential target *randomly*
2. Check if migration is accepted, if so migrate
3. Already L>1 potential targets checked?
   - No, go to 1.
   - Yes, don’t migrate, execute process locally, eventually postponing it for a while

Shortest Policy

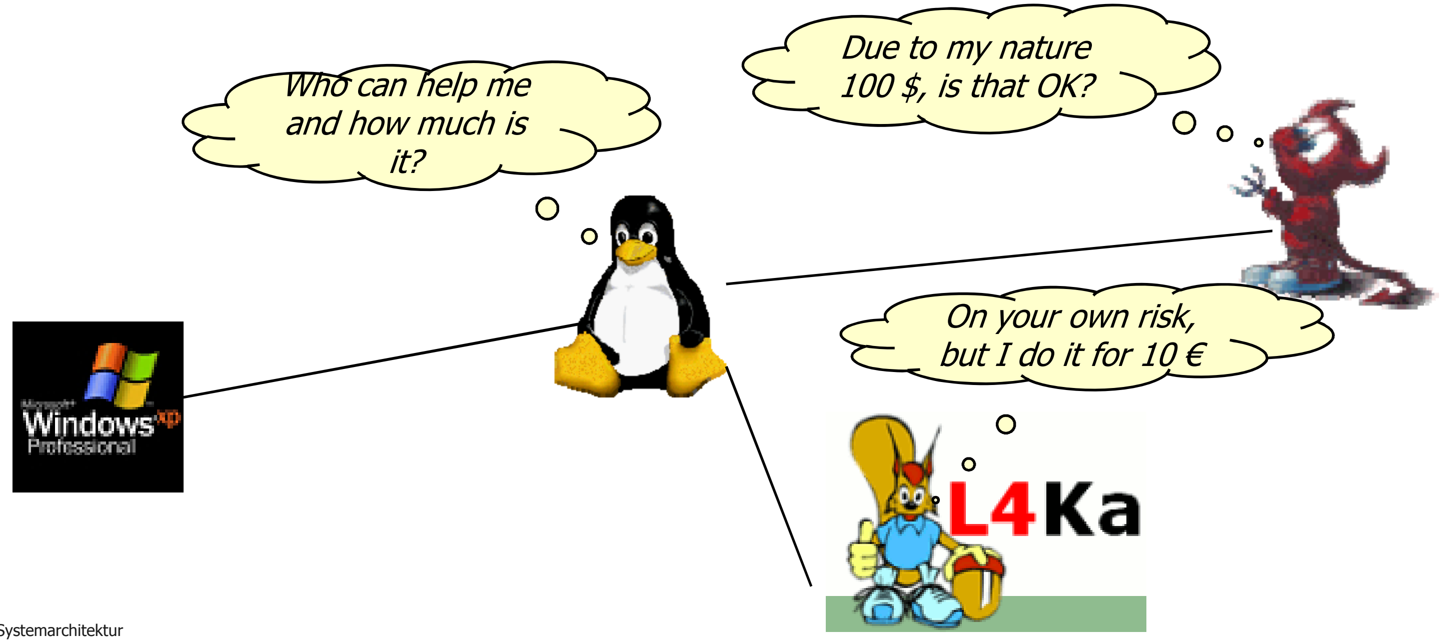
1. Chose L>1 potential targets randomly and ask for their load
2. Migrate to the target with the lowest load, but without danger of overloading this target
3. If there is no such target, execute process locally, eventually postponing it for a while

Analysis:

- Needs often a lot of remote status information
- High communication costs
- Only marginally better than simple threshold policy

Bidding Policy

- DS modeled as big business world consisting of managers and contractors
  - Manager: machine looking for a target machine
  - Contractor: machine acting as the target machine


Who can help me and how much is it?
Due to my nature 100 $, is that OK?
On your own risk, but I do it for 10 €
Windows XP Professional
L4Ka

Bidding Policy

1. Manager broadcasts a “request for bids”
2. Contractors answer with their price
3. Manager chooses the best bid (below its price threshold) and asks the contractor if still interested
4. If so, process is migrated, otherwise …

<u>Analysis:</u>

- Nodes are fully autonomous
- High communication costs
- Difficult price policy

Pairing Policy

Load balance only between 2 machines

1. Overloaded machine randomly looks for a partner
2. Having found a partner they form a couple
3. Only this couple mutually balance its load
4. If no longer ”mutual support” needed they separate

Local Load Information Exchange

- A dynamic load balancing scheme needs current load information
- Too much load information might block the network
- Too few load information might lead to wrong decisions
- We need some convincing compromise

Load Information Exchange (1)

- Periodic broadcast (every Δt)
  - Every node broadcasts its current load state to all other nodes
    - Only in LANs with a limited number of nodes
  - Potentially high communication costs
  - Potentially many superfluous messages
  - Network periodically blocked for application messages

Load Information Exchange (2)

- Broadcast after state changes
  - Every node broadcasts its state changes, e.g.
    - from overload to underload
    - # tasks $\rightarrow$ # tasks + 1
  - Can be combined with a threshold policy

Load Information Exchange (3)

- A node broadcasts that it needs the current load information of all (or of some other related nodes) whenever this node leaves its "acceptable load state"
  - When changing to overload, only the underloaded nodes have to answer
  - When changing to underload, only the overloaded nodes might answer

Using Priorities

- You can distinguish between local (native) and immigrated (foreign) tasks
- Priority rules
  - Selfish
  - Altruistic
  - Hybrid
- “Analysis” of the above priority rules concerning turnaround times
  - Selfish is worst
  - Altruistic is best
  - Hybrid, nearly as good as altruistic

Limiting Migration

How often do you migrate one process?

- Uncontrolled
  - Might lead to a never ending story
- Controlled
  - Each process contains a migration counter
  - Having reached the maximal value, it no longer migrates
  - Maximal value can be static or dynamic