Resource and Process Management

Distributed System
(2021)

Code Mobility : A means of mobile Computing

- **Components : WHAT ?**
  - Computational Components- CC (process)
    - Process
      - Code Segment
      - Resource Segment
        - Resource Components – **RC** (data, file, device driver etc)
      - Execution Segment

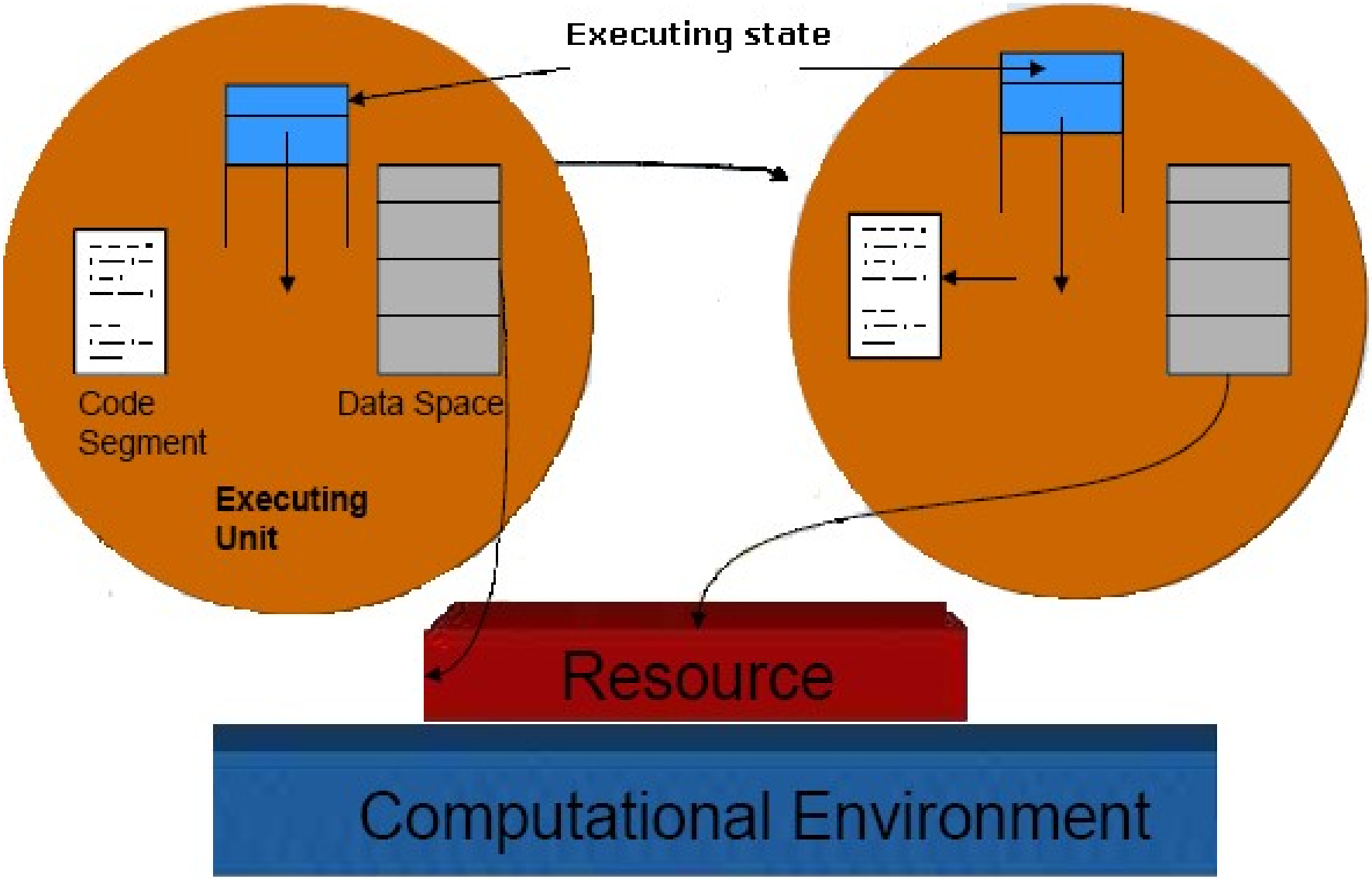
- **Interactions : WHY?**
  - Events between two or more components (messages)

  Load balancing, Improve performance, Flexibility (dynamic configuration)

- **Sites: WHERE?**

  Execution Environment (EE) -- Provide support for execution of the CC (computational components), speed and memory size provided by EE can be different.

Process Migration



Mobile Code Paradigm

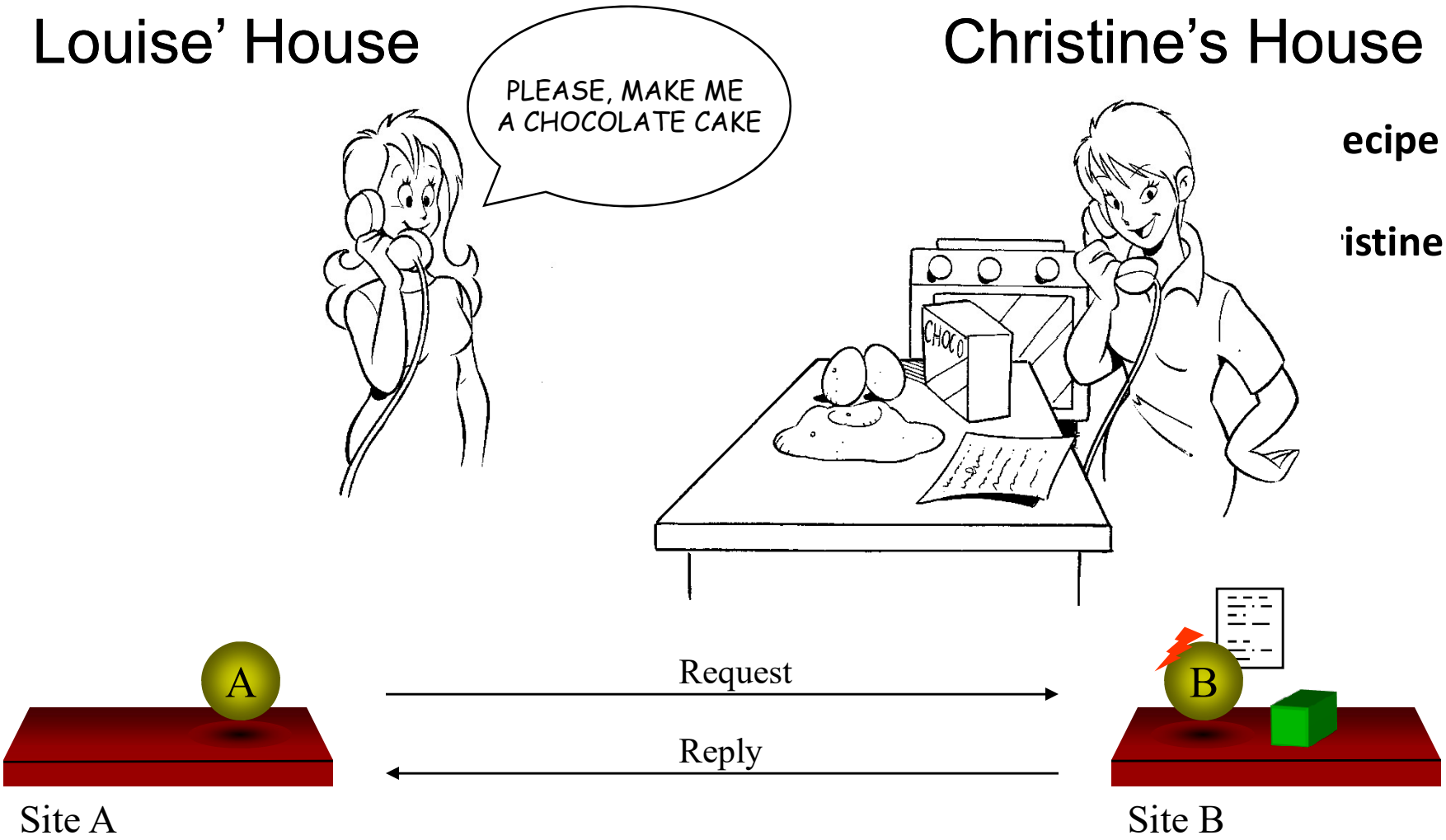
- Client – Server Model
- Remote Evaluation
- Code on Demand
- Mobile Agents

Executing Task	Baking Cake
• **Code (Know-How):**	**Recipe**
• **Resource Components (movable resources) :**	**Ingredients**
• **Resource Components (non-movable resource)**	**Oven**
• **Computational Component responsible: for the execution of the code**	**A person to mix the ingredients in the recipe**

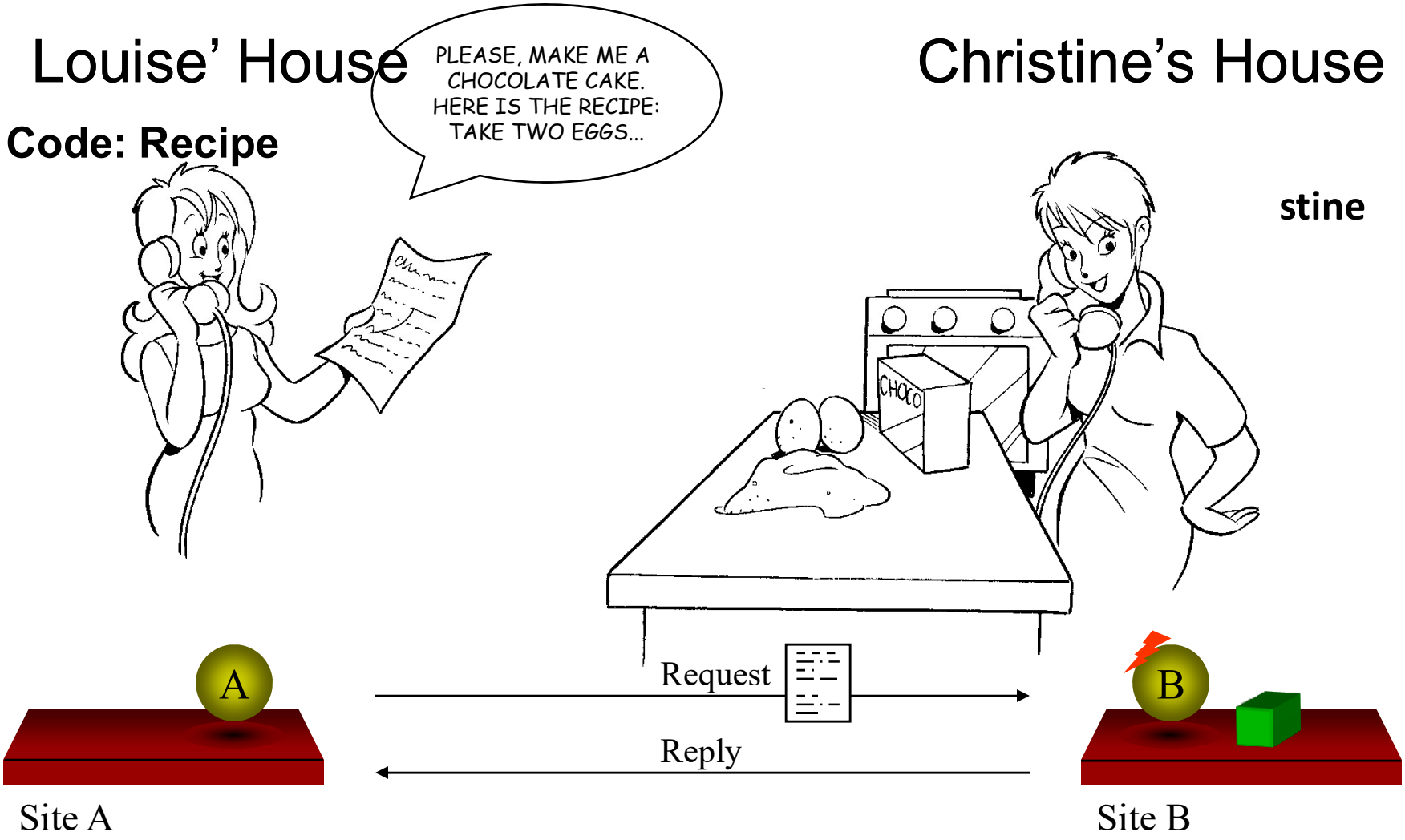
Assumption

To prepare the cake (to execute the service) all these elements must be co-located in the same home (site).

Client Server Paradigm



Remote Evaluation



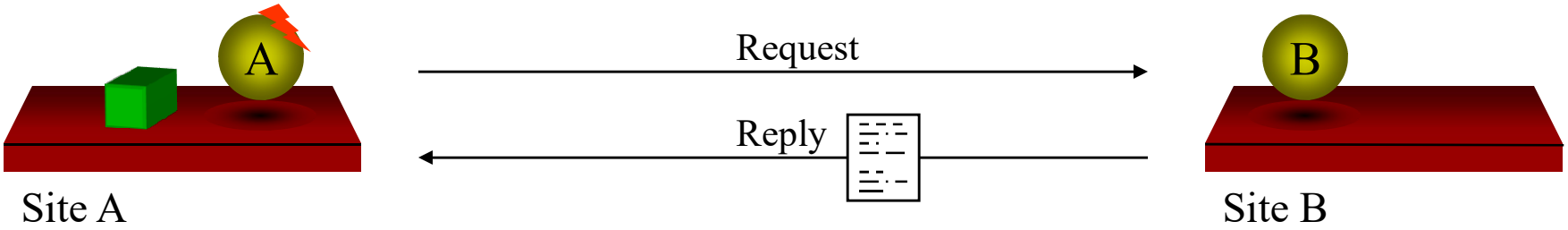
Code On Demand

Louise' House

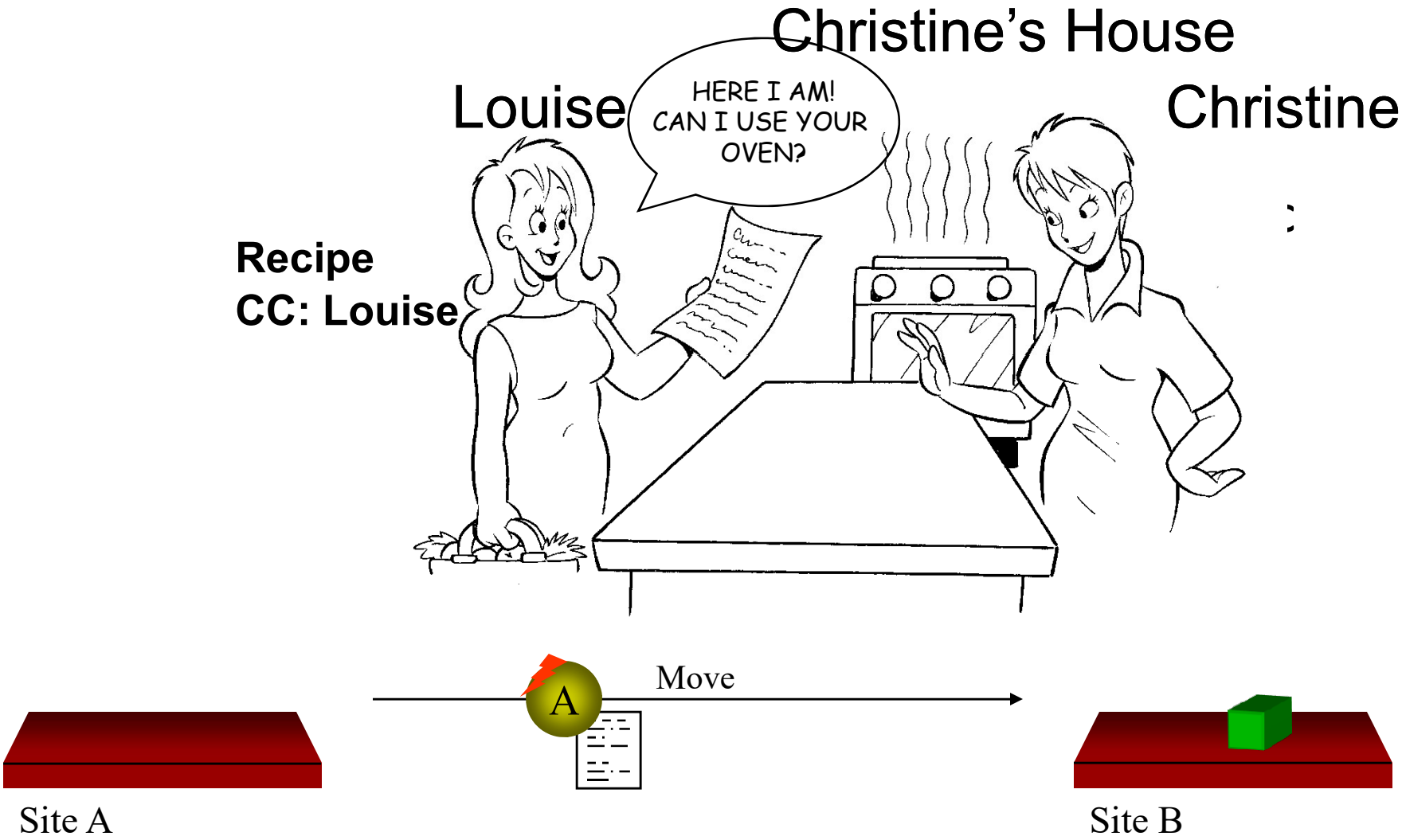
RC
CC: Louise

Christine's House

Recipe



Mobile Agent



Mobile Code paradigm: At a Glance

Paradigm	Site A	Site B	Site A	Site B
Client – Server	A	know-how resources B	A	know-how resources B
Remote Evaluation	know-how A	resources B	A	know-how resources B
Code on Demand	resources A	know-how B	resources know-how A	B
Mobile Agent	know-how A	resources	---	know-how resources A

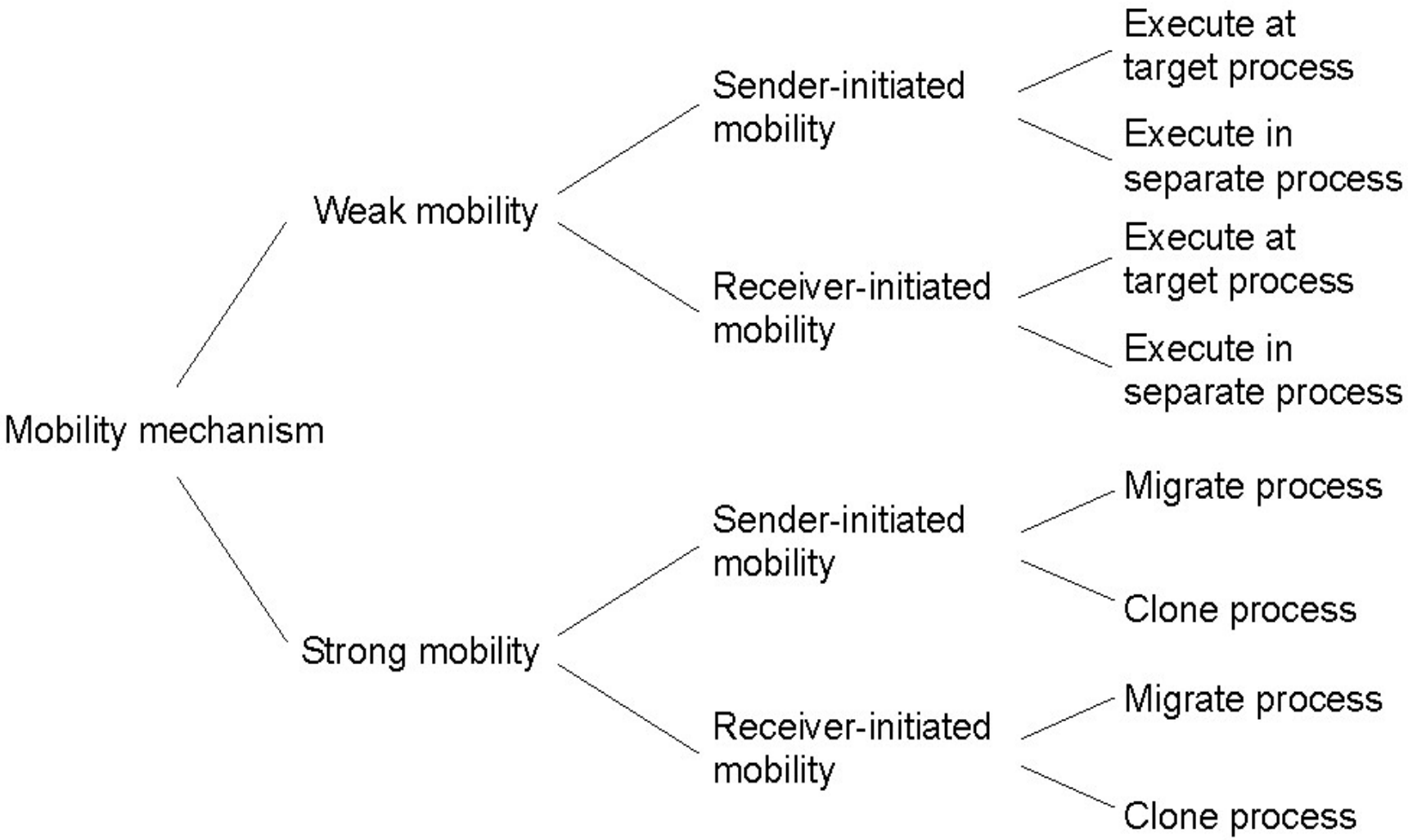
Mobile Code

"Code mobility is the capability to dynamically change the bindings between code fragments and the location where they are executed"

Involves:

- Change in bindings (process to resource & resource-to-machine) dynamically
- Relocation of code

Models of Code Migration



Migration and Local Resources

Resource-to machine binding

Process-to-resource 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)

Actions to be taken with respect to the references to local resources when migrating code to another machine.

GR: establish global system-wide reference
MV: move the resources
CP: copy the resource
RB: rebind process to locally available resource

Migration and Local Resources

- Depends on resource to process binding
  - By identifier: specific web site, ftp server
  - By value: Java libraries
  - By type: printers, local devices
- Depends on type of “attachments”
  - Unattached to any node: data files
  - Fastened resources (moved only at high cost)
    - Database, web sites
  - Fixed resources
    - Local devices, communication end points

Resource Management

- Distributed systems contain a set of resources interconnected by a network
- Processes are migrated to fulfill their resource requirements
- Resource manager are to control the assignment of resources to processes
- Resources can be logical (shared file) or physical (CPU)
- We consider a resource to be a processor

Types of process scheduling techniques

- Task assignment approach
  - User processes are collections of related tasks
  - Tasks are scheduled to improve performance
- Load-balancing approach
  - Tasks are distributed among nodes so as to equalize the workload of nodes of the system
- Load-sharing approach
  - Simply attempts to avoid idle nodes while processes wait for being processed

Desirable features of a scheduling algorithm

- No A Priori Knowledge about Processes
  - User does not want to specify information about characteristic and requirements
- Dynamic in nature
  - Decision should be based on the changing load of nodes and not on fixed static policy
- Quick decision-making capability
  - Algorithm must make quick decision about the assignment of task to nodes of system

Desirable features of a scheduling algorithm II.

- Balanced system performance and scheduling overhead
  - Great amount of information gives more intelligent decision, but increases overhead
- Stability
  - Unstable when all processes are migrating without accomplishing any useful work
  - It occurs when the nodes turn from lightly-loaded to heavily-loaded state and vice versa

Desirable features of a scheduling algorithm III.

- Scalability
  - A scheduling algorithm should be capable of handling small as well as large networks
- Fault tolerance
  - Should be capable of working after the crash of one or more nodes of the system
- Fairness of Service
  - More users initiating equivalent processes expect to receive the same quality of service

Task assignment approach

- Main assumptions
  - Processes have been split into tasks
  - Computation requirement of tasks and speed of processors are known
  - Cost of processing tasks on nodes are known
  - Communication cost between every pair of tasks are known
  - Resource requirements and available resources on node are known
  - Reassignment of tasks are not possible

Inter task Communication and Execution Cost

Inter task communications cost						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	6	4	0	0	12
t_2	6	0	8	12	3	0
t_3	4	8	0	0	11	0
t_4	0	12	0	0	5	0
t_5	0	3	11	5	0	0
t_6	12	0	0	0	0	0

Execution costs		
Task	Nodes	
	n_1	n_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

Task assignment approach

- Basic idea: Finding an optimal assignment to achieve goals such as the following:
  - Minimization of IPC costs
  - Quick turnaround time of process
  - High degree of parallelism
  - Efficient utilization of resources

The following task assignment example highlights the same concepts:

1. **Serial assignment** where tasks t_1, t_2, t_3 are assigned to node n_1 and tasks t_4, t_5, t_6 are assigned to node n_2:

Execution cost

$$x = x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62} = 5 + 2 + 4 + 3 + 2 + 4 = 20$$

Communication cost

$$c = c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36}$$

$$= 0 + 0 + 12 + 12 + 3 + 0 + 0 + 11 + 0 = 38$$

Hence total cost = 58.

2. **Optimal assignment** where tasks t_1, t_2, t_3, $t_4$and t_5 are assigned to node n_1 and task t_6 is assigned to node n_2.

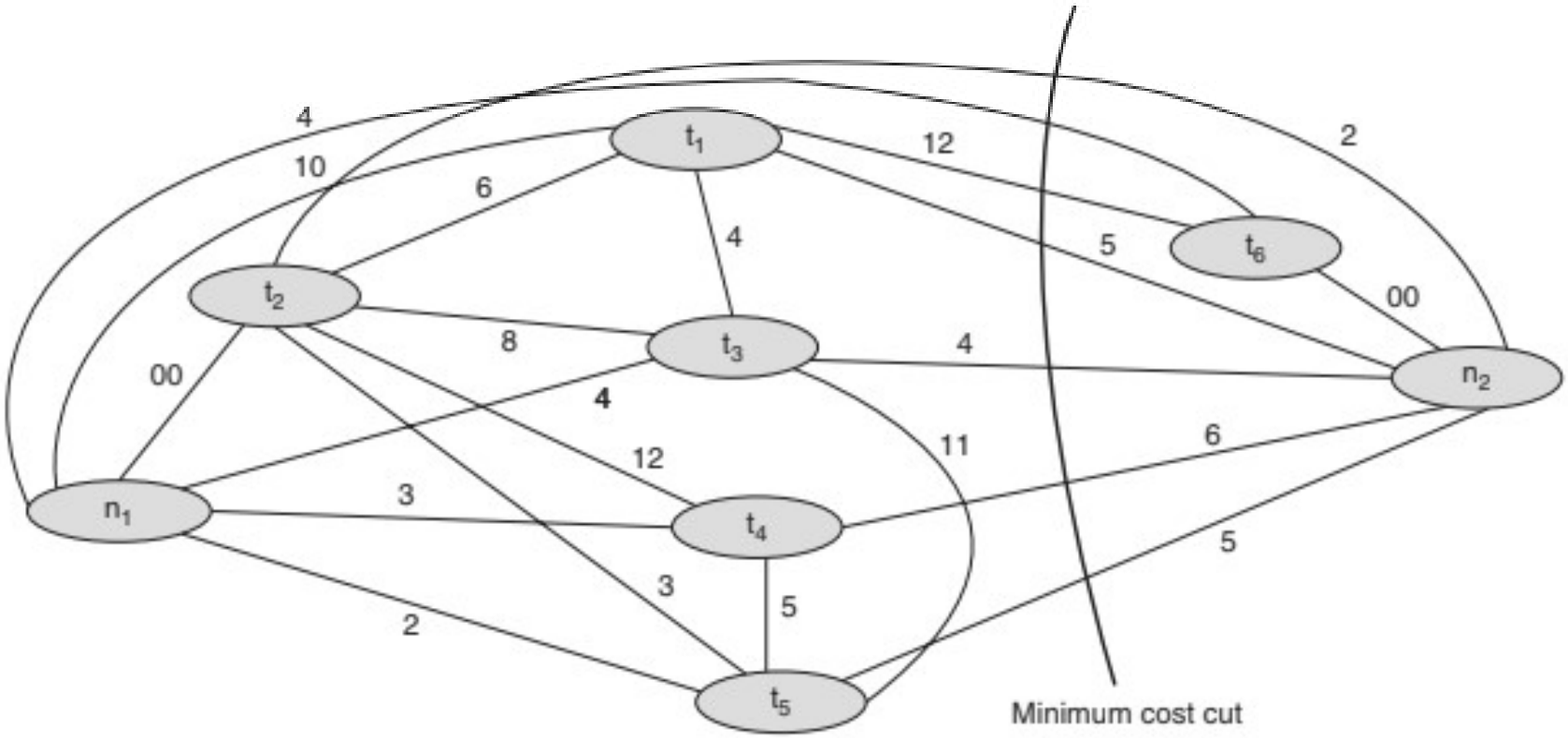
   Execution cost

$$x = x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{62} = 5 + 2 + 4 + 6 + 5 + 4 = 26$$

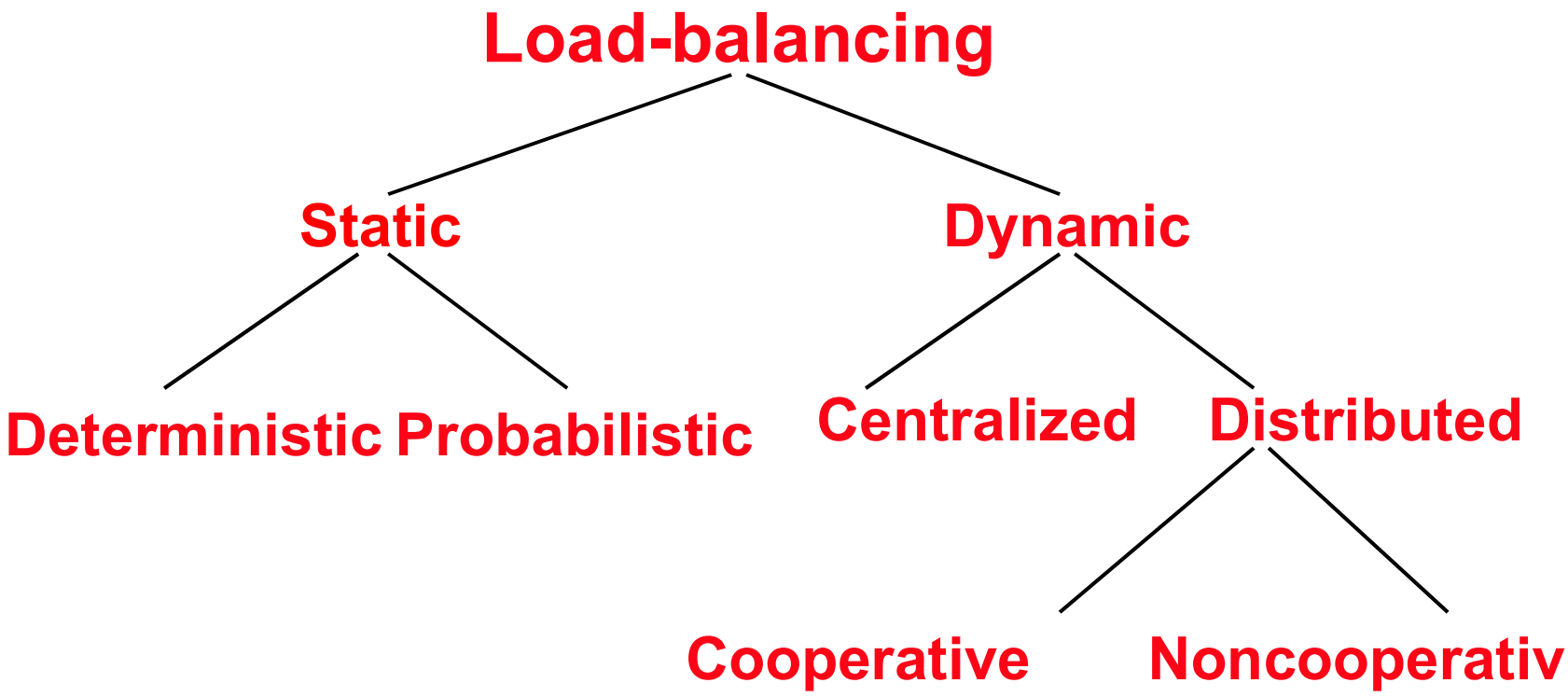
   Communication cost

$$c = c_{16} + c_{26} + c_{36} + c_{46} + c_{56} = 12 + 0 + 0 + 0 + 0 = 12$$

   Therefore, total cost = 38.



Load-balancing approach



Load-balancing approach

Type of load-balancing algorithms

- Static versus Dynamic
  - Static algorithms use only information about the average behavior of the system
  - Static algorithms ignore the current state or load of the nodes in the system
  - Dynamic algorithms collect state information and react to system state if it changed
  - Static algorithms are much more simpler
  - Dynamic algorithms are able to give significantly better performance

Load-balancing approach

Type of static load-balancing algorithms

- Deterministic versus Probabilistic
  - Deterministic algorithms use the information about the properties of the nodes and the characteristic of processes to be scheduled
  - Probabilistic algorithms use information of static attributes of the system (e.g. number of nodes, processing capability, topology) to formulate simple process placement rules
  - Deterministic approach is difficult to optimize
  - Probabilistic approach has poor performance

Load-balancing approach

Type of dynamic load-balancing algorithms

- Centralized versus Distributed
  - Centralized approach collects information to server node and makes assignment decision
  - Distributed approach contains entities to make decisions on a predefined set of nodes
  - Centralized algorithms can make efficient decisions, have lower fault-tolerance
  - Distributed algorithms avoid the bottleneck of collecting state information and react faster

Load-balancing approach

Type of distributed load-balancing algorithms

- Cooperative versus Noncooperative
  - In Noncooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities
  - In Cooperative algorithms distributed entities cooperate with each other
  - Cooperative algorithms are more complex and involve larger overhead
  - Stability of Cooperative algorithms are better

Issues in designing Load-balancing algorithms

- Load estimation policy
  - determines how to estimate the workload of a node
- Process transfer policy
  - determines whether to execute a process locally or remote
- State information exchange policy
  - determines how to exchange load information among nodes
- Location policy
  - determines to which node the transferable process should be sent
- Priority assignment policy
  - determines the priority of execution of local and remote processes
- Migration limiting policy
  - determines the total number of times a process can migrate

Load estimation policy I.
for Load-balancing algorithms

- To balance the workload on all the nodes of the system, it is necessary to decide how to measure the workload of a particular node
- Some measurable parameters (with time and node dependent factor) can be the following:
  - Total number of processes on the node
  - Resource demands of these processes
  - Instruction mixes of these processes
  - Architecture and speed of the node's processor
- Several load-balancing algorithms use the total number of processes to achieve big efficiency

Load estimation policy II.
for Load-balancing algorithms

- In some cases the true load could vary widely depending on the remaining service time, which can be measured in several way:
  - *Memoryless method* assumes that all processes have the *same expected remaining service time*, independent of the time used so far
  - *Pastrepeats* assumes that the remaining service time is equal to the *time used so far*
  - *Distribution method* states that if the distribution service times is known, the associated process's remaining service time is the expected remaining time conditioned by the time already used

Load estimation policy III.

for Load-balancing algorithms

- None of the previous methods can be used in modern systems because of periodically running processes and daemons
- An acceptable method for use as the load estimation policy in these systems would be to measure the CPU utilization of the nodes
- Central Processing Unit utilization is defined as the number of CPU cycles actually executed per unit of real time
- It can be measured by setting up a timer to periodically check the CPU state (idle/busy)

Process transfer policy I.

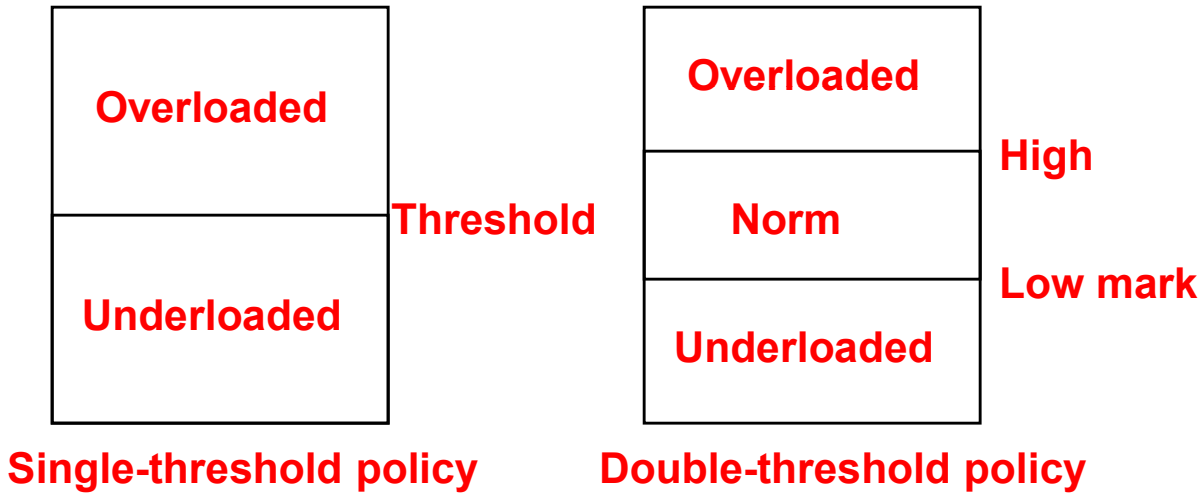
for Load-balancing algorithms

- Most of the algorithms use the *threshold policy* to decide on whether the node is lightly-loaded or heavily-loaded
- Threshold value is a limiting value of the workload of node which can be determined by
  - Static policy: predefined threshold value for each node depending on processing capability
  - Dynamic policy: threshold value is calculated from average workload and a predefined constant
- Below threshold value node accepts processes to execute, above threshold value node tries to transfer processes to a lightly-loaded node

Process transfer policy II.

for Load-balancing algorithms

- Single-threshold policy may lead to unstable algorithm because underloaded node could turn to be overloaded right after a process migration



- To reduce instability double-threshold policy has been proposed which is also known as high-low policy

Process transfer policy III.

for Load-balancing algorithms

- Double threshold policy
  - When node is in overloaded region new local processes are sent to run remotely, requests to accept remote processes are rejected
  - When node is in normal region new local processes run locally, requests to accept remote processes are rejected
  - When node is in underloaded region new local processes run locally, requests to accept remote processes are accepted

Location policy I.

for Load-balancing algorithms

- Threshold method
  - Policy selects a random node, checks whether the node is able to receive the process, then transfers the process. If node rejects, another node is selected randomly. This continues until probe limit is reached.
- Shortest method
  - L distinct nodes are chosen at random, each is polled to determine its load. The process is transferred to the node having the minimum value unless its workload value prohibits to accept the process.
  - Simple improvement is to discontinue probing whenever a node with zero load is encountered.

Location policy II.

for Load-balancing algorithms

- Bidding method
  - Nodes contain managers (to send processes) and contractors (to receive processes)
  - Managers broadcast a request for bid, contractors respond with bids (prices based on capacity of the contractor node) and manager selects the best offer
  - Winning contractor is notified and asked whether it accepts the process for execution or not
  - Full autonomy for the nodes regarding scheduling
  - Big communication overhead
  - Difficult to decide a good pricing policy

Location policy III.

for Load-balancing algorithms

- Pairing
  - Contrary to the former methods the pairing policy is to reduce the variance of load only between pairs
  - Each node asks some randomly chosen node to form a pair with it
  - If it receives a rejection it randomly selects another node and tries to pair again
  - Two nodes that differ greatly in load are temporarily paired with each other and migration starts
  - The pair is broken as soon as the migration is over
  - A node only tries to find a partner if it has at least two processes

State information exchange policy I. for Load-balancing algorithms

- Dynamic policies require frequent exchange of state information, but these extra messages arise two opposite impacts:
  - Increasing the number of messages gives more accurate scheduling decision
  - Increasing the number of messages raises the queuing time of messages
- State information policies can be the following:
  - Periodic broadcast
  - Broadcast when state changes
  - On-demand exchange
  - Exchange by polling

State information exchange policy II. for Load-balancing algorithms

- Periodic broadcast
  - Each node broadcasts its state information after the elapse of every *T* units of time
  - Problem: heavy traffic, fruitless messages, poor scalability since information exchange is too large for networks having many nodes
- Broadcast when state changes
  - Avoids fruitless messages by broadcasting the state only when a process arrives or departures
  - Further improvement is to broadcast only when state switches to another region (double-threshold policy)

State information exchange policy III. for Load-balancing algorithms

- On-demand exchange
  - In this method a node broadcast a State-Information-Request message when its state switches from normal to either underloaded or overloaded region.
  - On receiving this message other nodes reply with their own state information to the requesting node
  - Further improvement can be that only those nodes reply which are useful to the requesting node
- Exchange by polling
  - To avoid poor scalability (coming from broadcast messages) the partner node is searched by polling the other nodes on by one, until poll limit is reached

Priority assignment policy for Load-balancing algorithms

- Selfish
  - Local processes are given higher priority than remote processes. Worst response time performance of the three policies.
- Altruistic
  - Remote processes are given higher priority than local processes. Best response time performance of the three policies.
- Intermediate
  - (L>R) When the number of local processes is greater or equal to the number of remote processes, local processes are given higher priority than remote processes. Otherwise, remote processes are given higher priority than local processes.

Migration limiting policy
for Load-balancing algorithms

- This policy determines the total number of times a process can migrate
  - Uncontrolled
    - A remote process arriving at a node is treated just as a process originating at a node, so a process may be migrated any number of times
  - Controlled
    - Avoids the instability of the uncontrolled policy
    - Use a *migration count* parameter to fix a limit on the number of time a process can migrate
    - Irrevocable migration policy: *migration count* is fixed to 1
    - For long execution processes *migration count* must be greater than 1 to adapt for dynamically changing states

Load-sharing approach

- Drawbacks of Load-balancing approach
  - Load balancing technique with attempting equalizing the workload on all the nodes is not an appropriate object since big overhead is generated by gathering exact state information
  - Load balancing is not achievable since number of processes in a node is always fluctuating and temporal unbalance among the nodes exists every moment
- Basic ideas for Load-sharing approach
  - It is necessary and sufficient to prevent nodes from being idle while some other nodes have more than two processes
  - Load-sharing is much simpler than load-balancing since it only attempts to ensure that no node is idle when heavily node exists
  - Priority assignment policy and migration limiting policy are the same as that for the load-balancing algorithms

Load estimation policies
for Load-sharing algorithms

- Since load-sharing algorithms simply attempt to avoid idle nodes, it is sufficient to know whether a node is busy or idle
- Thus these algorithms normally employ the simplest load estimation policy of counting the total number of processes
- In modern systems where permanent existence of several processes on an idle node is possible, algorithms measure CPU utilization to estimate the load of a node

Process transfer policies
for Load-sharing algorithms

- Algorithms normally use all-or-nothing strategy
- This strategy uses the threshold value of all the nodes fixed to 1
- Nodes become receiver node when it has no process, and become sender node when it has more than 1 process
- To avoid processing power on nodes having zero process load-sharing algorithms use a threshold value of 2 instead of 1
- When CPU utilization is used as the load estimation policy, the double-threshold policy should be used as the process transfer policy

Location policies I.

for Load-sharing algorithms

- Location policy decides whether the sender node or the receiver node of the process takes the initiative to search for suitable node in the system, and this policy can be the following:
  - Sender-initiated location policy
    - Sender node decides where to send the process
    - Heavily loaded nodes search for lightly loaded nodes
  - Receiver-initiated location policy
    - Receiver node decides from where to get the process
    - Lightly loaded nodes search for heavily loaded nodes

Location policies II.

for Load-sharing algorithms

- Sender-initiated location policy
  - Node becomes overloaded, it either broadcasts or randomly probes the other nodes one by one to find a node that is able to receive remote processes
  - When broadcasting, suitable node is known as soon as reply arrives
- Receiver-initiated location policy
  - Nodes becomes underloaded, it either broadcast or randomly probes the other nodes one by one to indicate its willingness to receive remote processes
- Receiver-initiated policy require preemptive process migration facility since scheduling decisions are usually made at process departure epochs

Location policies III.

for Load-sharing algorithms

- Experiences with location policies
  - Both policies gives substantial performance advantages over the situation in which no load-sharing is attempted
  - Sender-initiated policy is preferable at light to moderate system loads
  - Receiver-initiated policy is preferable at high system loads
  - Sender-initiated policy provide better performance for the case when process transfer cost significantly more at receiver-initiated than at sender-initiated policy due to the preemptive transfer of processes

State information exchange policies for Load-sharing algorithms

- In load-sharing algorithms it is not necessary for the nodes to periodically exchange state information, but needs to know the state of other nodes when it is either underloaded or overloaded
- Broadcast when state changes
  - In sender-initiated/receiver-initiated location policy a node broadcasts State Information Request when it becomes overloaded/underloaded
  - It is called broadcast-when-idle policy when receiver-initiated policy is used with fixed threshold value value of 1
- Poll when state changes
  - In large networks polling mechanism is used
  - Polling mechanism randomly asks different nodes for state information until find an appropriate one or probe limit is reached
  - It is called poll-when-idle policy when receiver-initiated policy is used with fixed threshold value value of 1

Mechanism for Handling Coprocesses

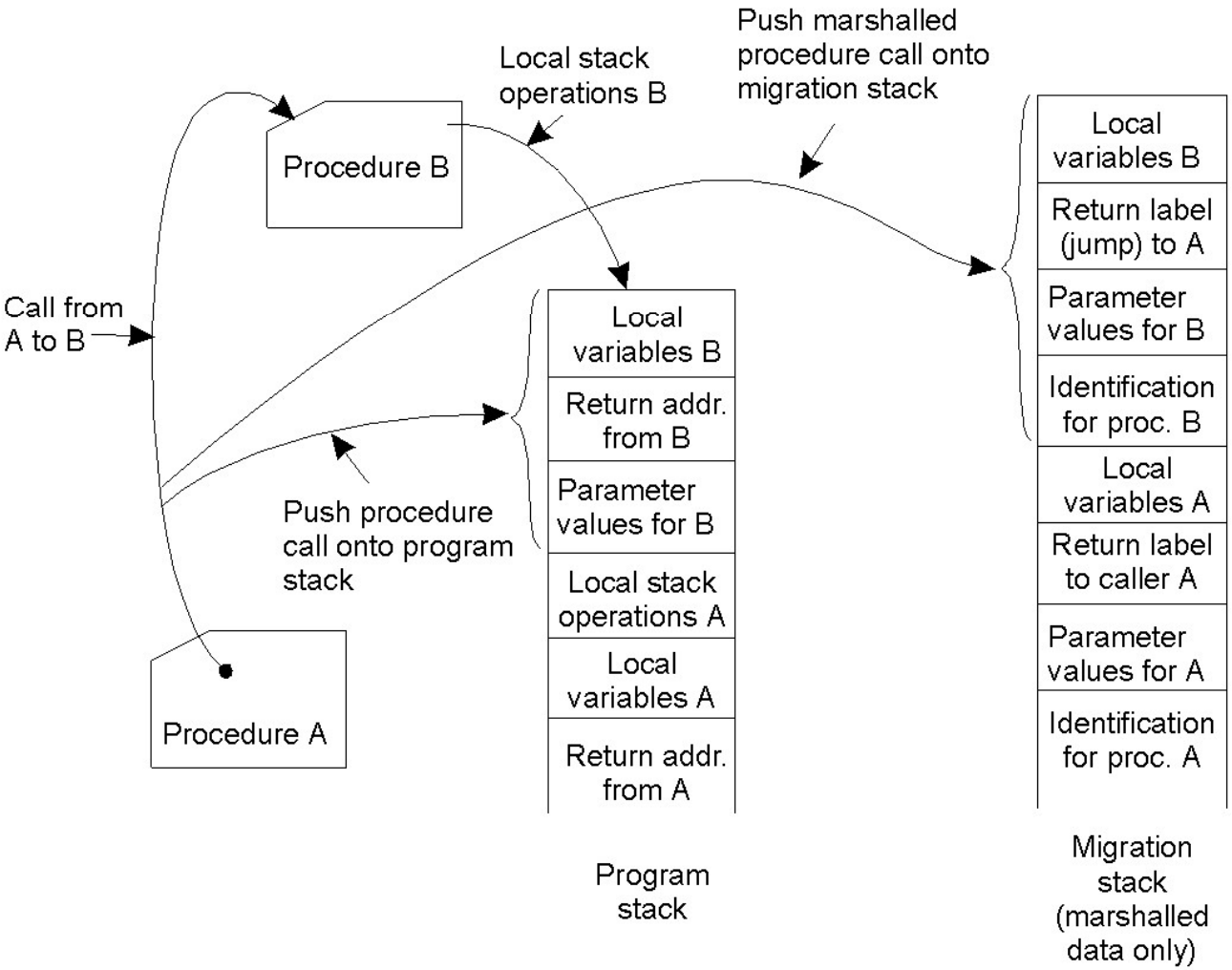
- Disallowing Separation of coprocesses-
  - Disallow the migration of processes that wait for one or more of their children to complete
  - Ensure that when a parent process migrates , its children processes will be migrated along with it
  - Disadvantage- It does not allow the use of parallelism within jobs
  - Home Node or Origin site concept
  - All communication between parent process and child processes take place via home node
  - Disadvantage- Message traffic and communication cost increase considerably

Process Migration in Heterogeneous systems

- Data Translation- when process is migrated in a heterogeneous environment , all the concerned data must be translated from the source CPU format to the destination CPU format before it can be executed on the destination node
- In general, a heterogeneous system having n CPU types must have n(n-1) pieces of translation software in order to support the facility of migrating a process from any node to any other node
- Use of external data representation mechanism- to reduce the software complexity of this translation process
- In this mechanism, a standard representation is used for the transport of data, each processor needs only to be able to convert data to and from the standard form
- The process of converting from a particular machine representation to external data representation format is called Serialization, and the reverse process is called deserialization

- Systems can be heterogeneous (different architecture, OS)
  - Support only weak mobility: recompile code, no run time information
  - Strong mobility: recompile code segment, transfer execution segment [migration stack]
  - Virtual machines - interpret source (scripts) or intermediate code [Java]

Migration on
Only subroutine
Or method
Call
Migrate stack



Cost of migration

- Multiprocessor: nondistributed
  - loss of the lines associated with the process in the processor's instruction a data caches
- Distributed environment
  - Moving a process's virtual memory
  - Forwarding a process's IPC (local and network) messages, informing senders of the process's new contact information.
  - Moving information of files. the open file table, the file descriptor table, the file offset, dirty blocks in the buffer cache, &c
  - Moving the process's user-level state: registers, stack, &c
  - Moving the process's kernel-level state: pwd, pid, signal masks, &c

Cost of migration (partial migration)

- Migration of the whole process too expensive.
- Move certain aspects of a process
  - The remaining portions of the process create *residual dependencies* -- the migrated process still relies on the original host to provide the services that were not migrated.

Migrating Virtual memory

- *Freeze and copy migration:* Suspend or *freeze* the process on the original host, and then to copy all of the pages of memory to the new host. Once all done, process can be resumed on the new host.
  - Simple, clean and easy to implement.
  - Does not create a residual dependency
  - Many pages which are never used may be copied and sent over the network If the process is migrated several times, this cost adds up
  - Do nothing while copying?

Migrating Virtual memory

- *Precopying:* The process runs on the original host, while the pages are being copied.
  - It is clean -- it does not create any residual dependencies.
  - Copying pages that may never be used.
  - Dirty pages must be transferred. Can be more expensive
- *lazy migration:* like demand paging.
  - It creates residual dependencies

Migrating Virtual memory

- Distributed file system: a memory-mapped file. the process's memory can be migrated simply by flushing the dirty blocks and mapping the file from a different host.
  - Isn't as clean as it may seem

Migrating Communication Channels

- If a process migrates, its communications must be able to continue.
  - Inform "interested" processes of the new location of a migrating process.
    - Unclean, unnecessary messages, how to know other communicating clients?
  - *link redirection* or *forwarding* at the original host of the migrating process.
    - Residual dependency and can increase the latency involved in sending messages to the migrated process, but makes the process of migration itself cheaper

Process with open files

- Show up at the new host and re-open the files. But, in truth, there is a great deal of state associated with an open file. Consider the system-wide open file table, the cached inodes, dirty blocks that may live only in the local buffer cache, &c.
  - fork()'d proceses share the same file offset.
- it is often much easier to leave the process dependent on the old host for file service.

Migrate kernel state

- It is often easier to leave a migrating process dependent on a prior (or perhaps first) host for these services.
- Checkpointing and recovery: A process's state to be saved to a file (much like a persistent object) and then a new process to be created (restored) based on this checkpoint file. This checkpoint file contains all of the "goods" including the kernel material.

Migrate? Or not migrate?

- Several things to consider
  - If the home host suffers from a bursty load, it may not make sense to migrate a process -- the home host will be free again, soon.
  - Processes with significant virtual memory or IPC usage or many open files are poor choices for migration.
  - Historical consideration: long running processes are better candidates than recent arrivals – they are likely to continue to run for a long time. Short lived processes are likely to complete shortly after migration, offering little gain to amortize the cost of migration over useful work.

Design Issues

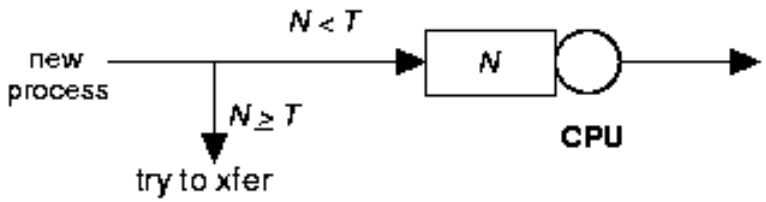
- Measure of load
  - Queue lengths at CPU, CPU utilization
- Types of policies
  - Static: decisions hardwired into system
  - Dynamic: uses load information
  - Adaptive: policy varies according to load
- Preemptive versus non-preemptive
- Centralized versus decentralized
- Stability: $\lambda > \mu$ => instability, $\lambda_1+\lambda_2 < \mu_1+\mu_2$ =>load balance
  - Job floats around and load oscillates

Components

- *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
    - Select processes with long execution times
- *Location policy:* where to transfer the process?
  - Polling, random, nearest neighbor
- *Information policy*: when and from where?
  - Demand driven [only if sender/receiver], time-driven [periodic], state-change-driven [send update if load changes]

Sender-initiated Policy

- *Transfer policy*



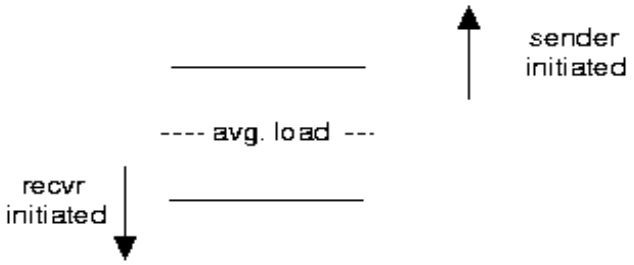
- *Selection policy*: newly arrived process
- *Location policy*: three variations
  - *Random*: may generate lots of transfers => limit max transfers
  - *Threshold:* probe *n* nodes sequentially
    - Transfer to first node below threshold, if none, keep job
  - *Shortest*: poll N_p nodes in parallel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

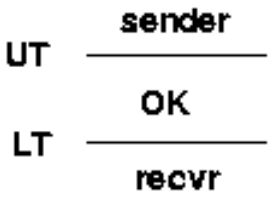
Receiver-initiated Policy

- 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 threshold, if none, do nothing
  - Shortest: poll *n* nodes in parallel, choose node with heaviest load above *T*

Symmetric Policies

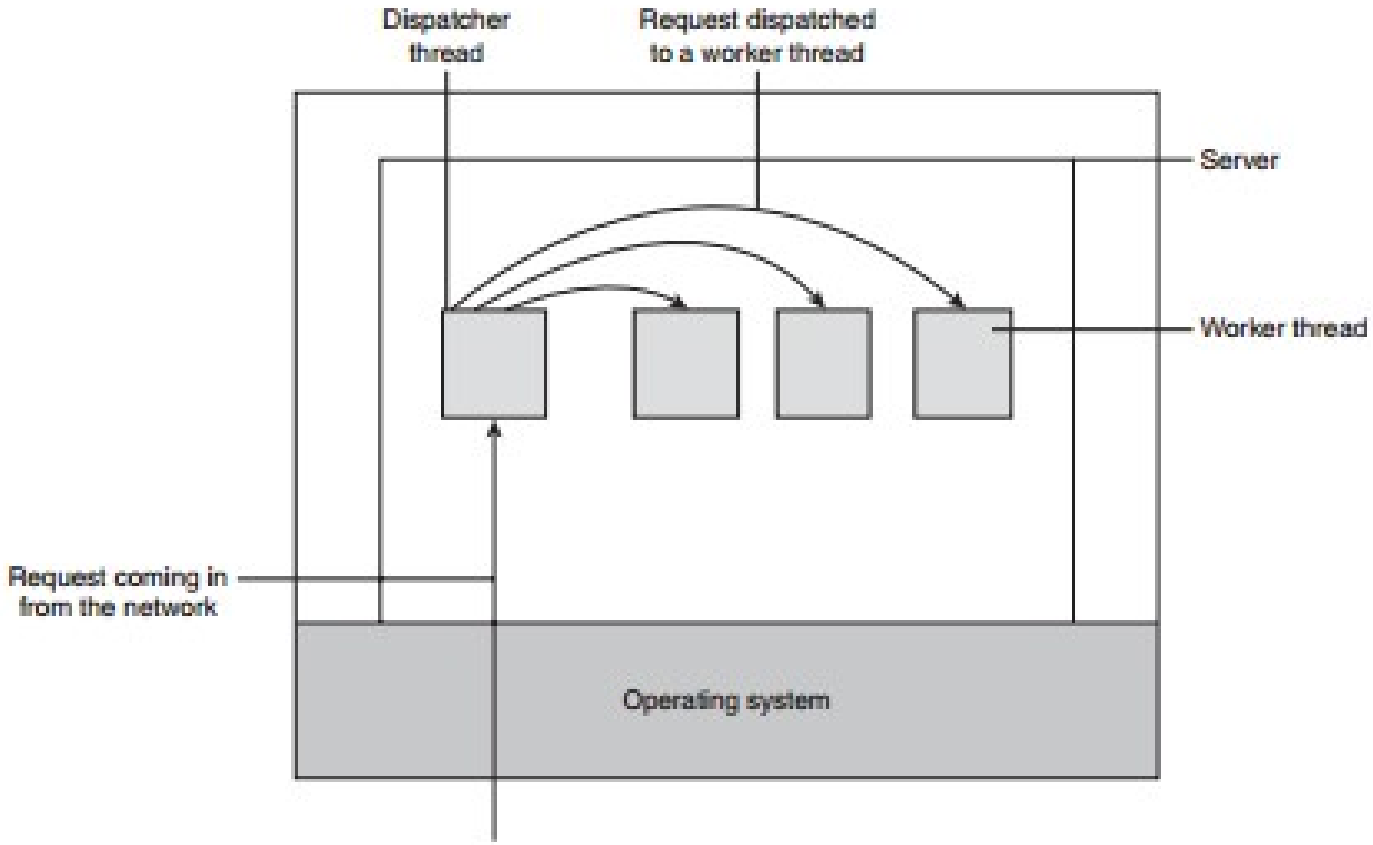
- Nodes act as both senders and receivers: combine previous two policies without change
  - Use average load as threshold
- Improved symmetric policy: explc information
  - Two thresholds: *LT, UT, LT <= UT*
  - Maintain sender, receiver and OK nodes usir info
  - Sender: poll first node on receiver list …
  - Receiver: poll first node on sender list …





Models for organizing Threads

- Dispatcher –worker model

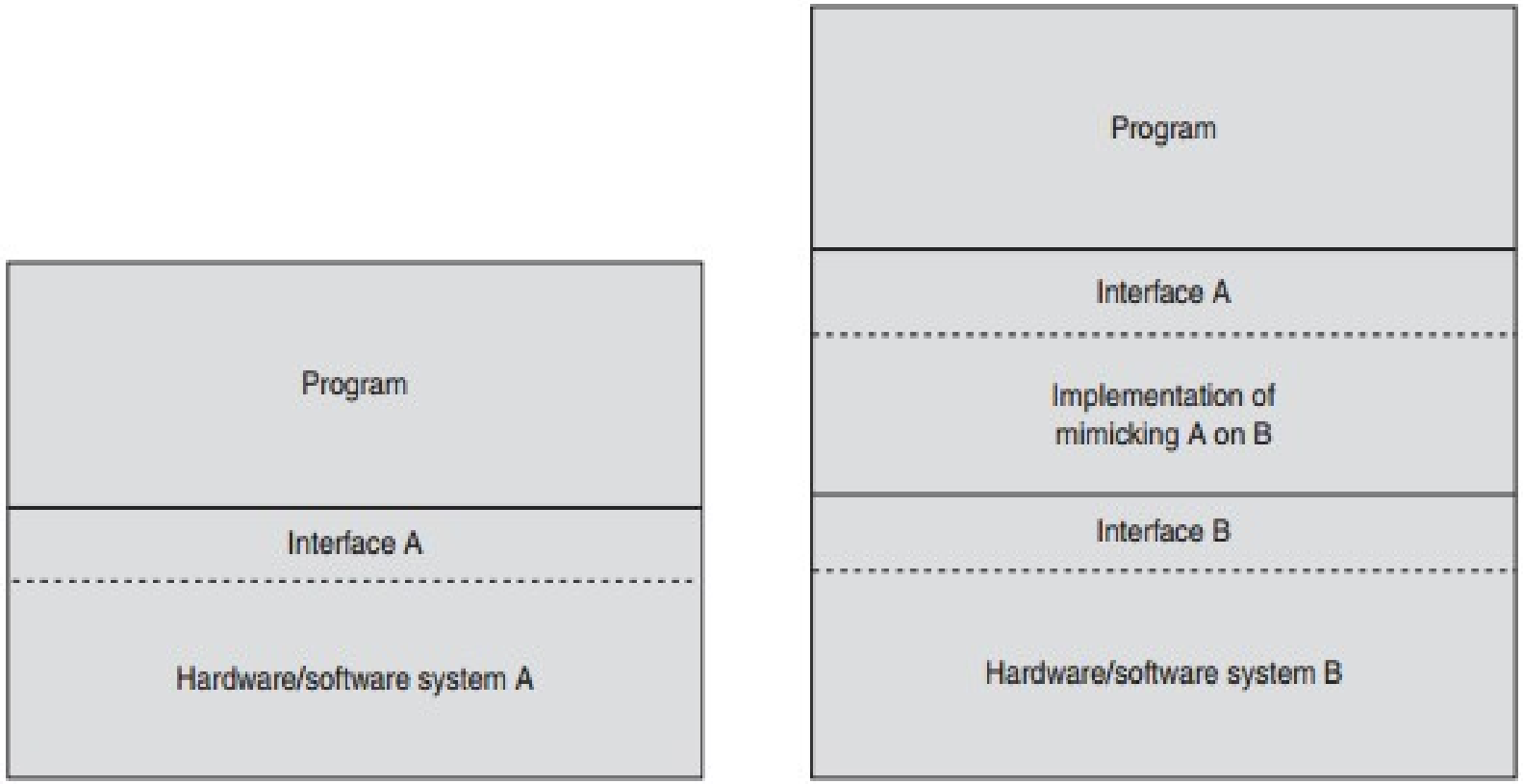


Models for organizing Threads

- Team Model
- In this model all threads behave as equal in the sense that there is no dispatcher-worker relationship for processing clients' request
- Each thread of the process is specialized in serving a specific type of request
- Model is used for implementation of specialized threads within a process
- So multiple types of requests can be simultaneously handled by the process

Models for organizing Threads

- Pipeline Model
- This model is useful for applications based on the producer-consumer model, in which the output data generated by one part of the application is used as input for another part of the application.
- In this model, the threads of a process are organized as a pipeline so that output data generated by the first thread is used for processing by the second thread, the output of the second thread is used for processing by the third thread and so on
- The output of the last thread in the pipeline is the final output of the process to which thread belongs


Program
Interface A
Hardware/software system A
(a)
Program
Interface A
Implementation of
mimicking A on B
Interface B
Hardware/software system B
(b)

Types of Virtualization

1. We build a runtime system that essentially provides an abstract instruction set used for executing applications.

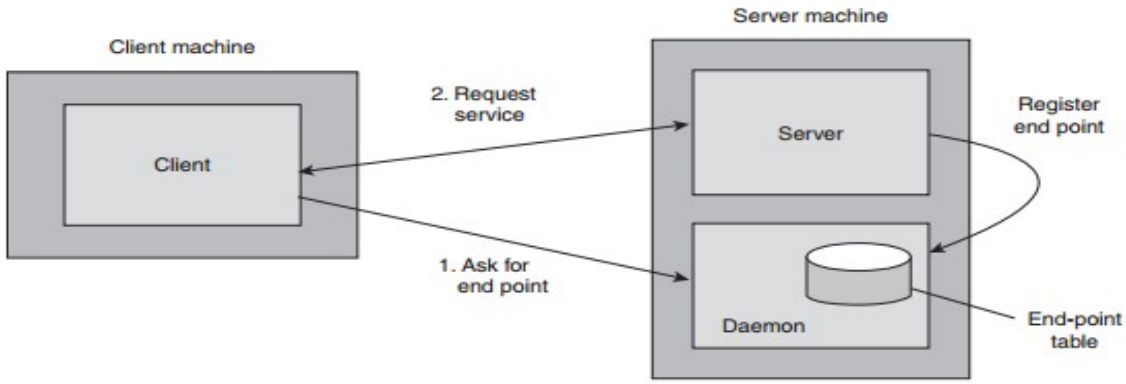
2. Instructions can be interpreted, could also be emulated as is done for running Windows applications on UNIX platforms.

3. This type of virtualization leads to call a process virtual machine, stressing that virtualization is done essentially only for a single process.

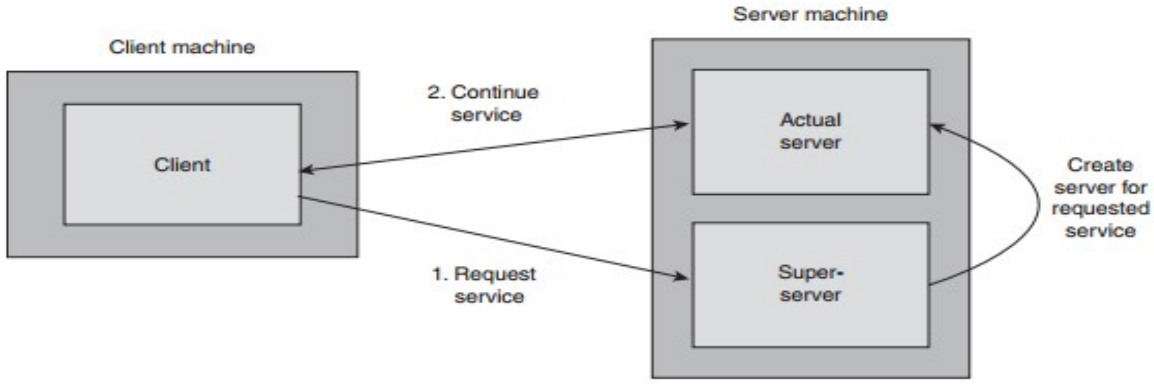
4. An alternative approach towards virtualization is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of the same (or other hardware) as an interface.

5. It is now possible to have multiple and different operating systems run independently and concurrently on the same platform. The layer is generally referred to as a virtual machine monitor (VMM).

Client Server using daemon / Super server

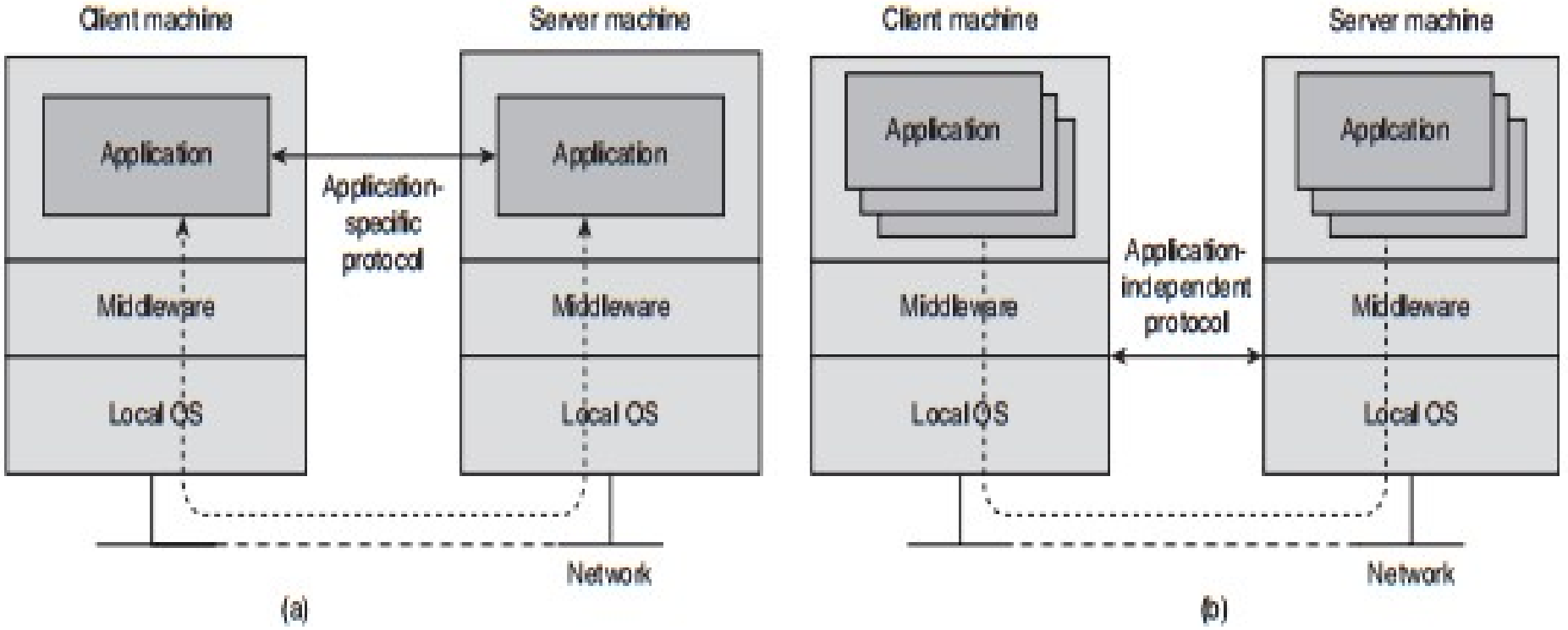


(a)

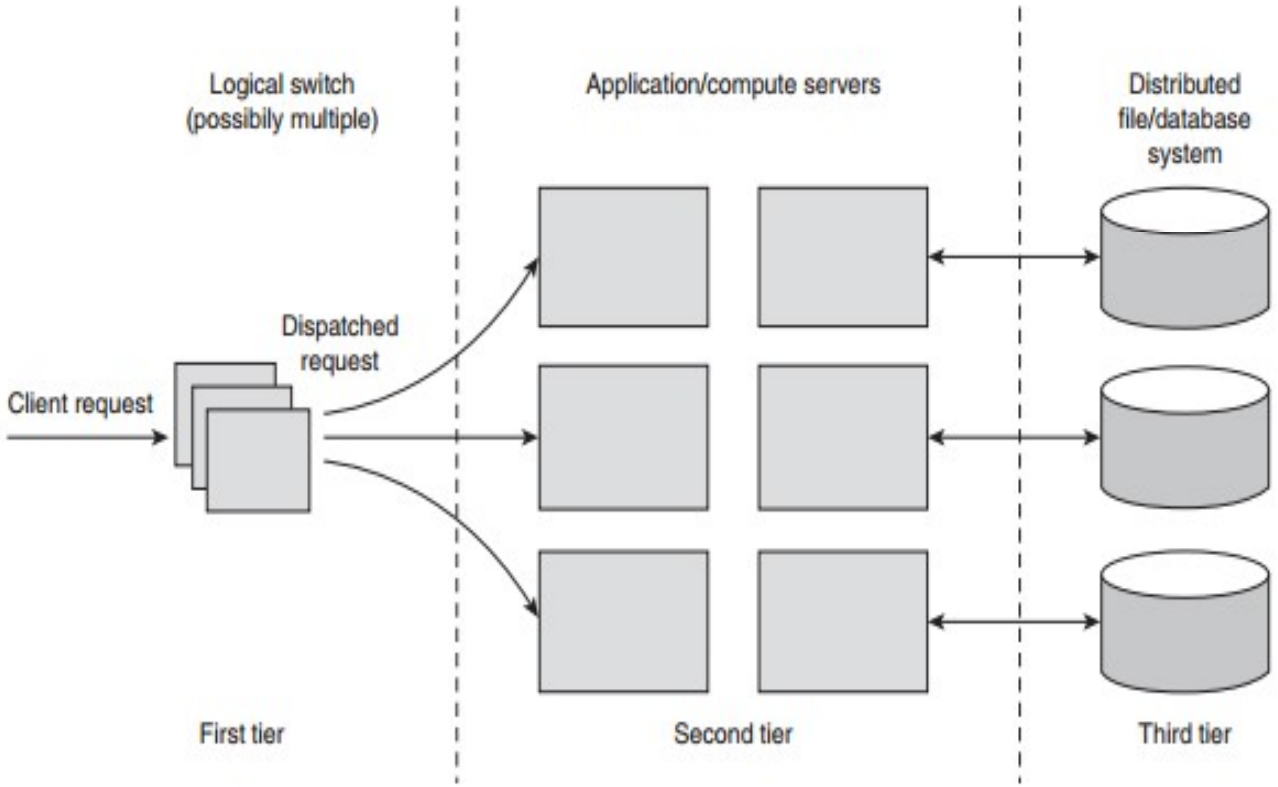


(b)

Virtualization



Cluster


Logical switch
(possibly multiple)
Application/compute servers
Distributed
file/database
system
Dispatched
request
Client request
First tier
Second tier
Third tier