Distributed Scheduling

Distributed Scheduling

Goal: enable transparent execution of programs on networked computing systems

Motivations: reduce response time of program execution through load balancing

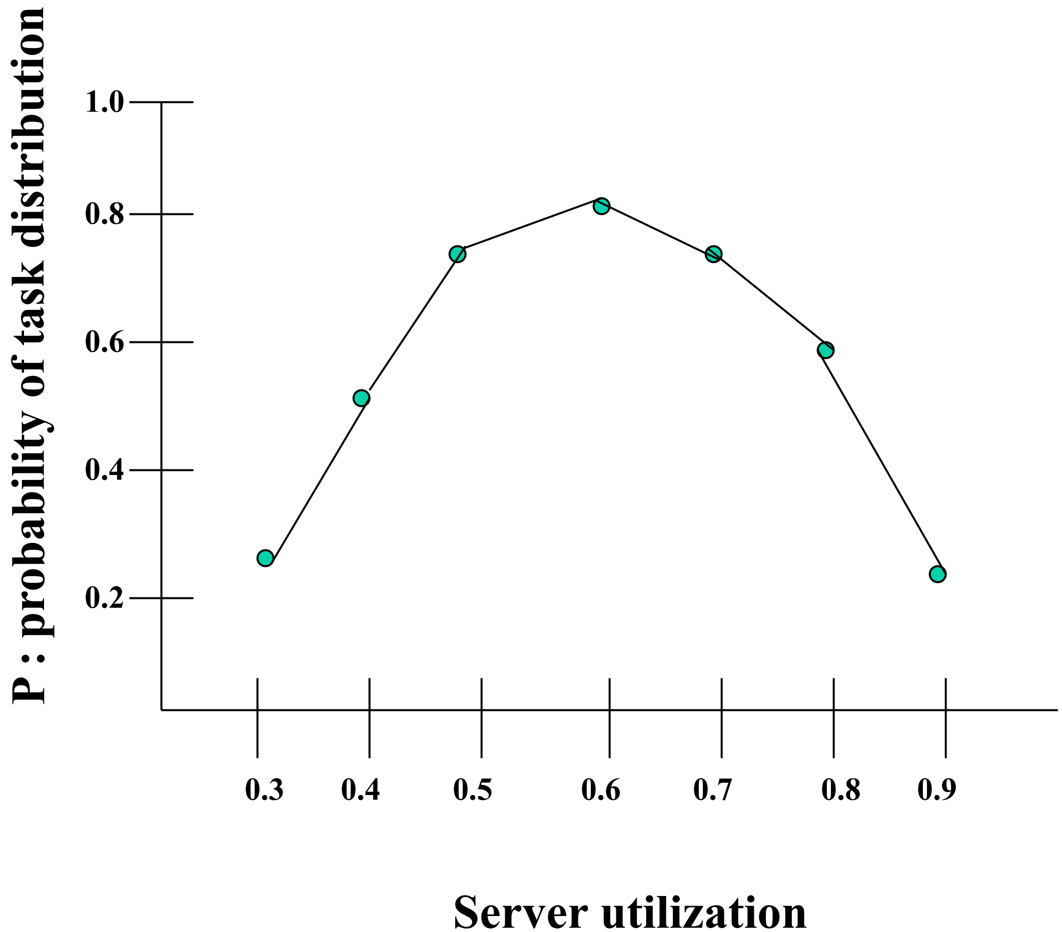
Distributed Scheduling

- Let P be the probability that the system is in a state in which at least 1 task is waiting for service and at least 1 server is idle.

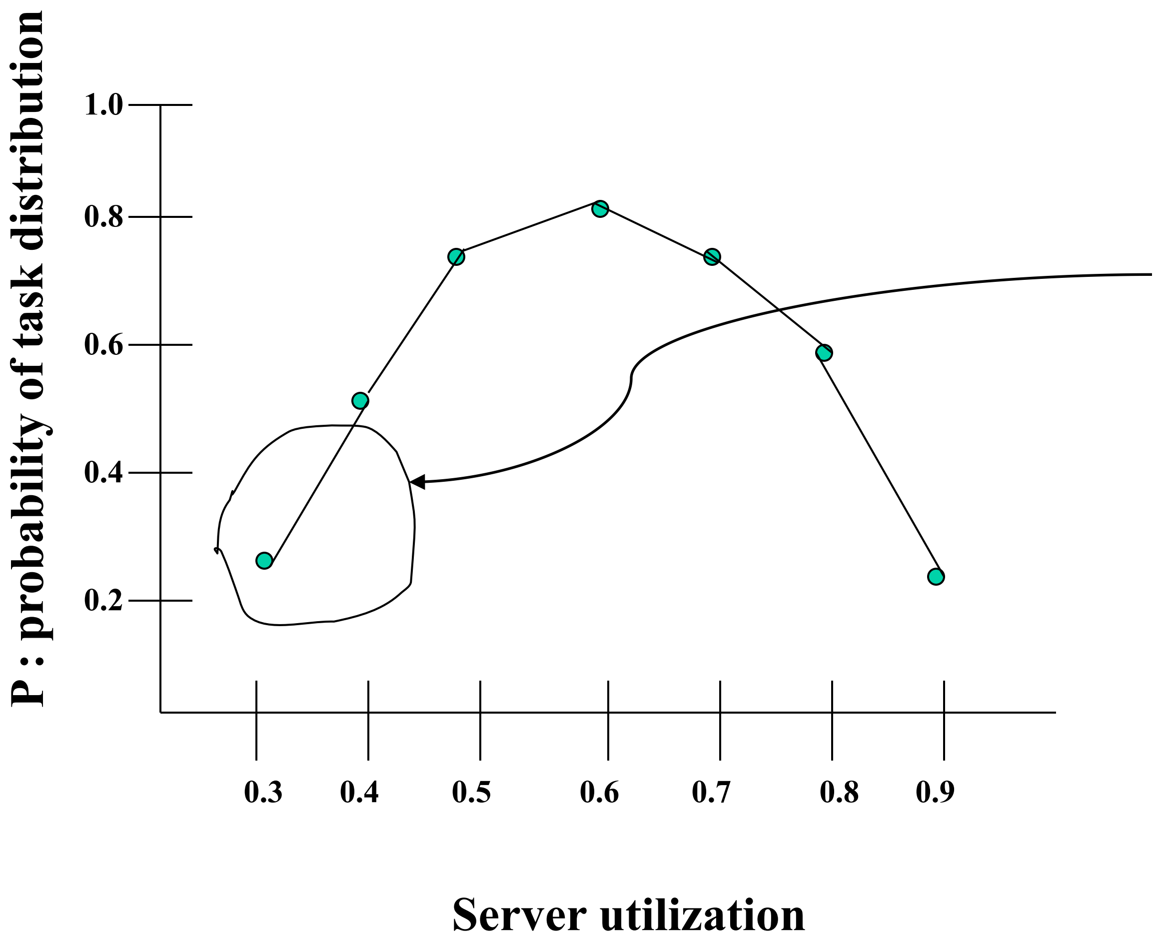
- Let ρ be the utilization of each server.

- We can estimate P using probabilistic analysis and plot a graph against system utilization.

Opportunities for Task Distribution

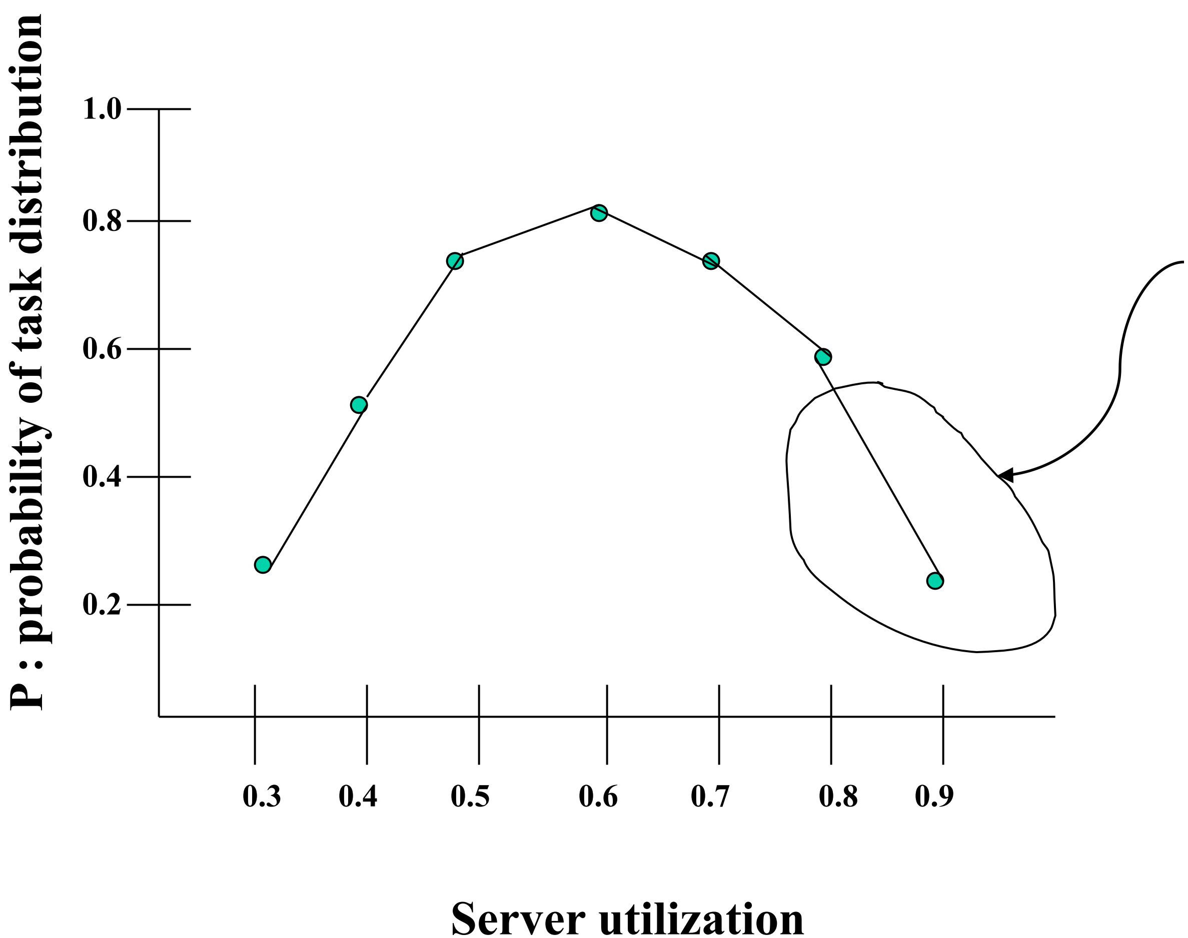


Task Distribution



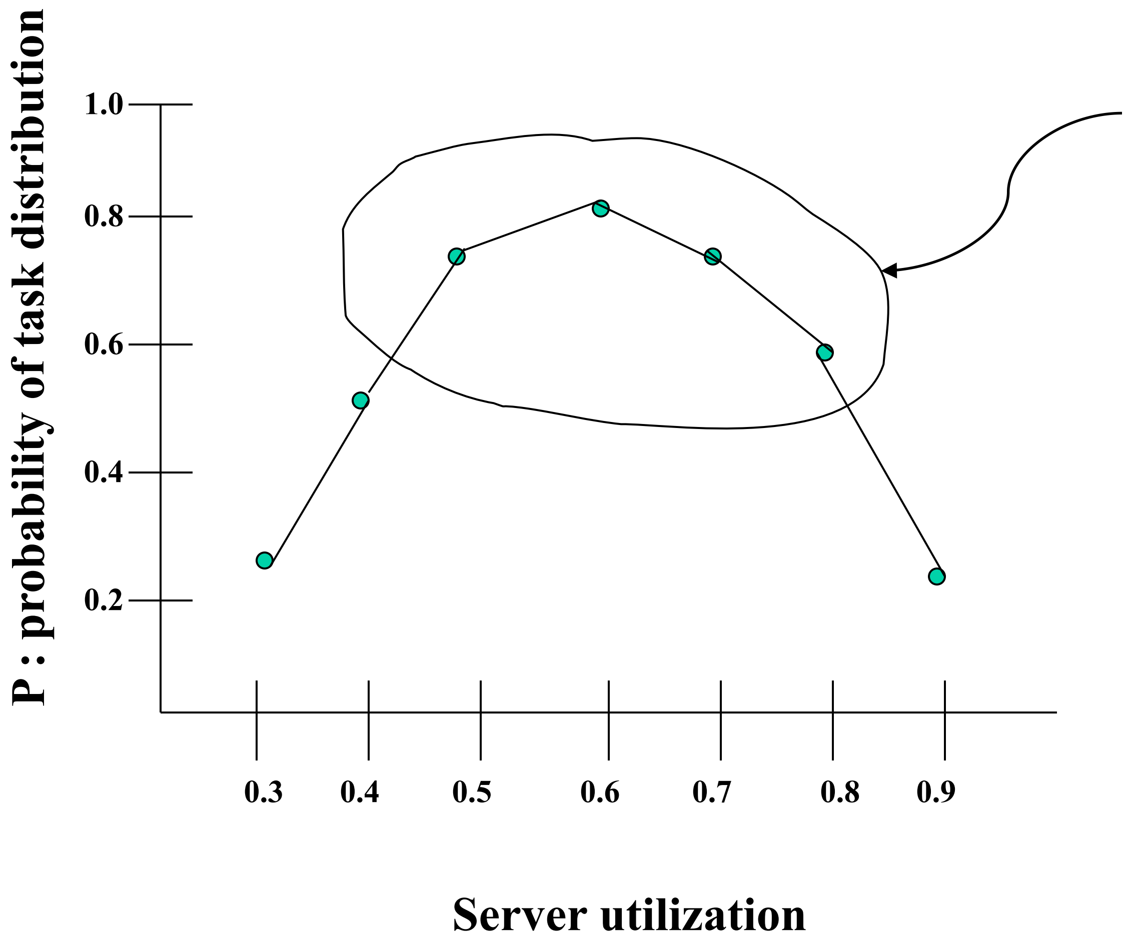
In <u>lightly loaded</u> systems there is not much opportunity for task distribution because most servers are underutilized

Task Distribution



In heavily loaded systems there is not much opportunity for task distribution because no server is free to accept a task

Task Distribution



In moderately loaded systems there are good opportunities to distribute tasks from over-utilized to under-utilized systems

What is Load?

- Load on a system/node can correspond to the queue length of tasks/ processes that need to be processed.
- Queue length of waiting tasks: proportional to task response time, hence a good indicator of system load.
- Distributing load: transfer tasks/processes among nodes.
- If a task transfer (from another node) takes a long time, the node may accept more tasks during the transfer time.
- Causes the node to be highly loaded. Affects performance.

Types of Algorithms

- *Static load distribution algorithms*: Decisions are hard-coded into an algorithm with a priori knowledge of system.

- *Dynamic load distribution*: use system state information such as task queue length, processor utilization.

- *Adaptive load distribution*: adapt the approach based on system state.
  - (e.g.,) Dynamic distribution algorithms collect load information from nodes even at very high system loads.
  - Load information collection itself can add load on the system as messages need to be exchanged.
  - Adaptive distribution algorithms may stop collecting state information at high loads.

Balancing vs. Sharing

- *Load balancing*: Equalize load on the participating nodes.
  - Transfer tasks even if a node is not heavily loaded so that queue lengths on all nodes are approximately equal.
  - More number of task transfers, might degrade performance.
- *Load sharing*: Reduce burden of an overloaded node.
  - Transfer tasks only when the queue length exceeds a certain threshold.
  - Less number of task transfers.
- *Anticipatory task transfers*: transfer from overloaded nodes to ones that are likely to become idle/lightly loaded.
  - More like load balancing, but may be less number of transfers.

Types of Task Transfers

- *Preemptive task transfers*: transfer tasks that are partially executed.
  - Expensive as it involves collection of task states.
  - Task state: virtual memory image, process control block, IO buffers, file pointers, timers, ...
- *Non-preemptive task transfers*: transfer tasks that have not begun execution.
  - Do not require transfer of task states.
  - Can be considered as task placements. Suitable for load sharing not for load balancing.
- Both transfers involve information on user's current working directory, task privileges/priority.

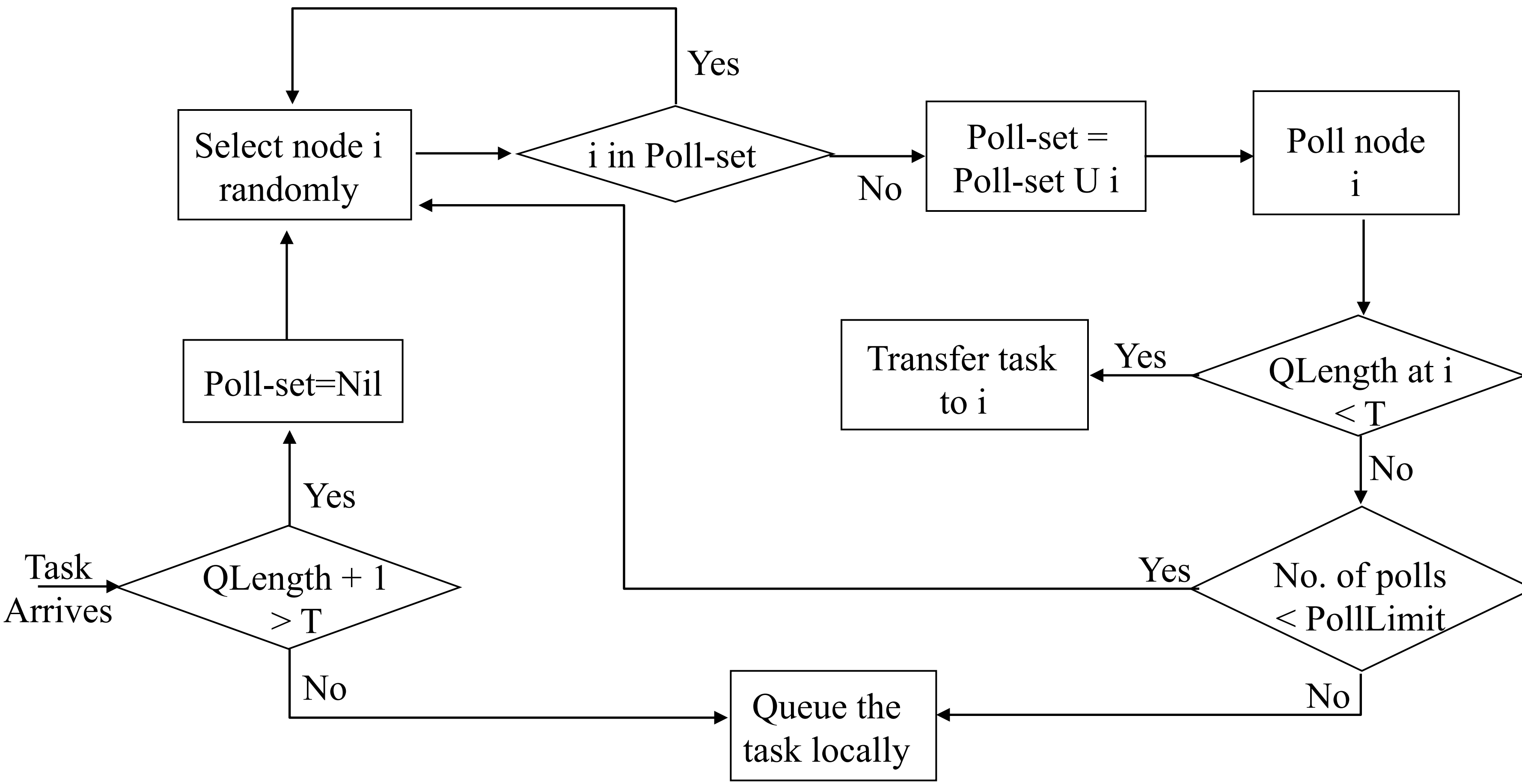
Load Distributing Algorithms

- Sender-initiated: distribution initiated by an overloaded node.
- Receiver-initiated: distribution initiated by lightly loaded nodes.
- Symmetric: initiated by both senders and receivers. Has advantages and disadvantages of both the approaches.
- Adaptive: sensitive to state of the system.

Sender-initiated

- *Transfer Policy*: Use thresholds.
  - Sender if queue length exceeds T.
  - Receiver if accepting a task will not make queue length exceed T.
- *Selection Policy*: Only newly arrived tasks.
- *Location Policy*:
  - Random: Use no remote state information. Task transferred to a node at random.
    - No need for state collection. Unnecessary task transfers (processor thrashing) may occur.
  - Threshold: poll a node to find out if it is a receiver. Receiver must accept the task irrespective of when it (task) actually arrives.
    - *PollLimit,* ie., the number of polls, can be used to reduce overhead.
  - Shortest: Poll a set of nodes. Select the receiver with shortest task queue length.

Sender-initiated



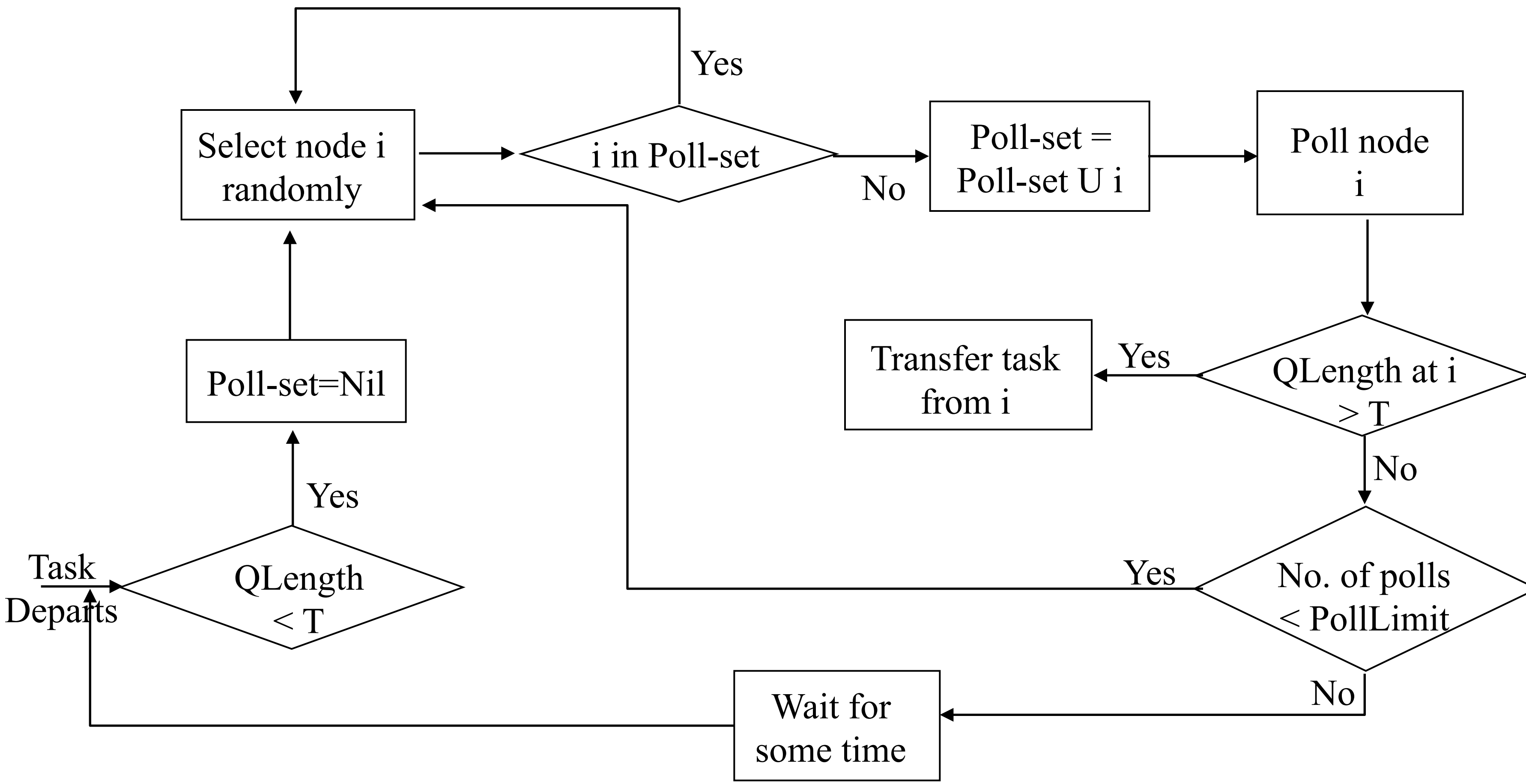
Sender-initiated

- *Information Policy*: demand-driven.
- *Stability*: can become unstable at high loads.
  - At high loads, it may become difficult for senders to find receivers.
  - Also, the number of senders increase at high system loads thereby increasing the polling activity.
  - Polling activity may make the system unstable at high loads.

Receiver-initiated

- *Transfer Policy*: uses thresholds. Queue lengths below T identifies receivers and those above T identifies senders.
- *Selection Policy*: as before.
- *Location Policy*: Polling.
  - A random node is polled to check if a task transfer would place its queue length below a threshold.
  - If not, the polled node transfers a task.
  - Otherwise, poll another node till a static PollLimit is reached.
  - If all polls fail, wait until another task is completed before starting polling operation.
- Information policy: demand-driven.
- Stability: Not unstable since there are lightly loaded systems that have initiated the algorithm.

Receiver-initiated



Receiver-initiated

- Drawback:
  - Polling initiated by receiver implies that it is difficult to find senders with new tasks.
  - Reason: systems try to schedule tasks as and when they arrive.
  - Effect: receiver-initiated approach might result in preemptive transfers. Hence transfer costs are more.
  - Sender-initiated: transfer costs are low as new jobs are transferred and so no need for transferring task states.

Symmetric

- Senders search for receivers and vice-versa.
- Low loads: senders can find receivers easily. High loads: receivers can find senders easily.
- May have disadvantages of both: polling at high loads can make the system unstable. Receiver-initiated task transfers can be preemptive and so expensive.
- Simple algorithm: combine previous two approaches.
- Above-average algorithm:
  - Transfer Policy: Two adaptive thresholds instead of one. If a node's estimated average load is A, a higher threshold *TooHigh* > A and a lower threshold *TooLow* < A are used.
  - Load < TooLow -> receiver. Load > TooHigh -> sender.

Above-average Algorithm

- Location policy:
- Sender Component
  - Node with *TooHigh* load, broadcasts a *TooHig*h message, sets *TooHigh* timer, and listens for an *Accept* message.
  - A receiver that gets the (TooHigh) message sends an *Accept* message, increases its load, and sets *AwaitingTask* timer.
  - If the *AwaitingTask* timer expires, load is decremented.
  - On receiving the Accept message: if the node is still a sender, it chooses the best task to transfer and transfers it to the node.
  - When sender is waiting for Accept, it may receive a *TooLow* message (receiver initiated). Sender sends *TooHigh* to that receiver. Do step 2 & 3.
  - On expiration of *TooHigh* timer, if no Accept message is received, system is highly loaded. Sender broadcasts a *ChangeAverage* message.

Above-average Algorithm...

- Receiver Component
  - Node with *TooLow* load, broadcasts a *TooLow* message, sets a *TooLow* timer, and listens for *TooHigh* message.
  - If *TooHigh* message is received, do step 2 & 3 in Sender Component.
  - If *TooLow* timer expires before receiving any *TooHigh* message, receiver broadcasts a *ChangeAverage* message to decrease the load estimate at other nodes.
- Selection Policy: as discussed before.
- Information policy: demand driven. Average load is modified based on system load. High loads may have less number of senders progressively.
  - Average system load is determined individually. There is a range of acceptable load before trying to be a sender or a receiver.

Adaptive Algorithms

- Limit Sender's polling actions at high load to avoid instability.
- Utilize the collected state information during previous polling operations to classify nodes as: Sender/overloaded, receiver/underloaded, OK (in acceptable load range).
- Maintained as separate lists for each class.
- Initially, each node assumes that all others are receivers.
- Location policy at sender:
  - Sender polls the head of the receiver list.
  - Polled node puts the sender at the head of it sender list. It informs the sender whether it is a receiver, a sender, or a OK node.
  - If the polled node is still a receiver, the new task is transferred.
  - Else the sender updates the polled node's status, polls the next potential receiver.
  - If this polling process fails to identify a receiver, the task can still be transferred during a receiver-initiated dialogue.

Adaptive Algorithms…

- Location policy at receiver
  - Receivers obtain tasks from potential senders. Lists are scanned in the following order.
    - Head to tail in senders list (most up-to-date info used), tail to head in OK list (least up-to-date used), tail to head in receiver list.
    - Least up-to-date used in the hope that status might have changed.
  - Receiver polls the selected node. If the node is a sender, a task is transferred.
  - If the node is not a sender, both the polled node and receiver update each other's status.
  - Polling process stops if a sender is found or a static PollLimit is reached.

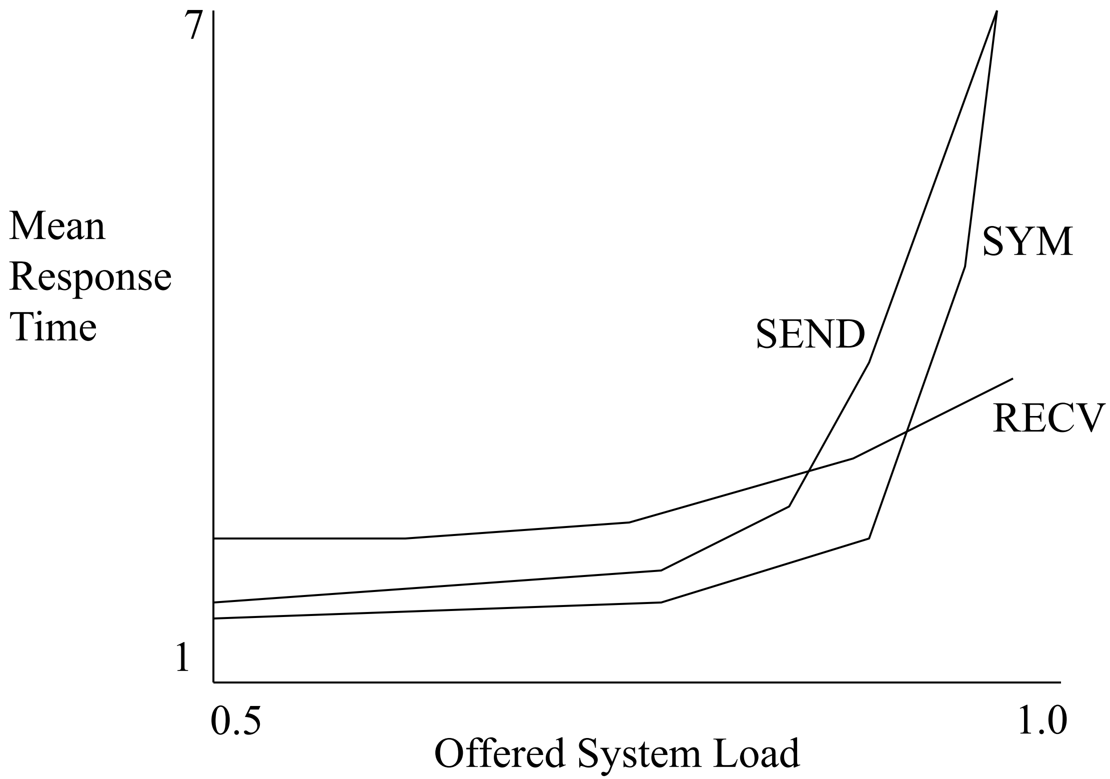
Adaptive Algorithms…

- At high loads, sender-initiated polling gradually reduces as nodes get removed from receiver list (and become senders).
  - Whereas at low loads, sender will generally find some receiver.
- At high loads, receiver-initiated works and can find a sender.
  - At low loads, receiver may not find senders, but that does not affect the performance.
- Algorithm dynamically becomes sender-initiated at low loads and receiver-initiated at high loads.
- Hence, algorithm is stable and can use non-preemptive transfers at low loads (sender initiated).

Selecting an Algorithm

- If a system never gets highly loaded, sender-initiated algorithms work better.
- Stable, receiver-initiated algorithms better for high loads.
- Widely fluctuating loads: stable, symmetric algorithms.
- Widely fluctuating loads + high migration cost for preemptive transfers: stable, sender-initiated algorithms.
- Heterogeneous work arrival: stable, adaptive algorithms.

Performance Comparison



Implementation Issues

- Task placement
  - A task that is yet to begin is transferred to a remote machine, and starts its execution there.
- Task migration
  - *State Transfer*:
    - State includes contents of registers, task stack, task status (running, blocked, etc.,), file descriptors, virtual memory address space, temporary files, buffered messages.
    - Other info: current working directory, signal masks and handlers, references to children processes.
    - Task is frozen (suspended) before state transfer.
  - *Unfreeze*:
    - Task is installed at the new machine, unfrozen, and is put in the ready queue.

State Transfer

- Issues to be considered:
  - Cost to support remote execution including delays due to freezing the task.
    - Duration of freezing the task should be small. Otherwise it can result in timeouts of tasks interacting with the frozen one.
    - Transfer memory as they are referenced to reduce delay?
  - Residual Dependencies: amount of resources to be dedicated at the former host for the migrated task.
    - An implementation that does not transfer all the virtual memory address space at the time of migration, but transfers them as they are referenced.
    - Need for redirection of messages to the migrated task.
    - Location-dependent system calls at the former node.

State Transfer...

- Disadvantages of residual dependencies :
  - Affects reliability as it becomes dependent on former host
  - Affects performance since memory accesses can become slow as pages need to be transferred
  - Affects complexity as task states are distributed on several hosts.
- Location transparency:
  - Migration should hide the location of tasks
  - Message passing, process naming, file handling, and other activities should be transparent of actual location.
  - Task names and their locations can be maintained as hints. If the hints fail, they can be updated by a broadcast query or through a name server.