Distributed Scheduling

- System performance can be improved by distributed load among heavy and light loaded nodes of the system.
- What is performance?
  - response time?
- What is load?
  - CPU queue length or CPU utilization?
  - How is it measured and at what overhead?
- Load:
  - CPU queue length correlates well with task response time and is a simple to measure with low overhead.
  - When a node accepts remote tasks, due to task transfer times, the node must avoid overcommitting.
  - CPU queue length has little correlation with processor utilization.
**Classification of load distribution algorithms**

- static, dynamic, and adaptive.
  - Dynamic use system state to make load transfer decisions. Static do not.
  - Adaptive modify algorithm parameters as the system state changes, dynamic ones do not.
- Load balancing vs Load sharing.
  - Both strive to reduce the likelihood of an unshared state.
  - Load balancing algorithms attempt to equalize loads among nodes.
  - Load-balancing algorithms transfer tasks at a higher rate, thus incurring higher overhead.
- To avoid delays in task transfers, anticipatory task transfers can be used. In this sense, load-balancing is a special case of load-sharing.
- Preemptive vs. non-preemptive task transfers

**Components of a load distributing algorithm**

- Transfer Policy
  - used to determine that a node is in a suitable state to participate in a task transfer. Mostly, threshold-based to classify nodes as senders, receivers, or OK.
  - Alternatively, can use load-imbalance to make that decision.
- Selection Policy
  - which task should be transferred from a node classified as sender?
  - The simplest is to select a newly arrived task that have caused the node to become a sender.
  - The task selected for transfer should be such so that the overhead from task transfer is compensated by a reduction in task's response time. Long-lived tasks usually satisfy this criterion. Other factors to consider include location-dependent calls.
**Components of a load distributing algorithm**

- **Location policy**
  - which node should get the task selected for transfer? Method: usually polling (serially or in parallel). A node can be selected for polling randomly, based on information from previous polls, or on a nearest-neighbor manner.

- **Information policy**
  - triggering collection of system state: when, what, and where.
  - Demand-driven
    - a node collects the state of other nodes when it becomes a sender or receiver. Can be sender-initiated, receiver-initiated, or symmetrically-initiated.
  - Periodic
    - Do not adapt to system state, are slow to respond, and can make the situation worse by increasing system load.
  - State-change driven
    - nodes disseminate info. when their state changes.

---

**Stability of Load Sharing Algorithms**

- **Stability of load-sharing algorithms**
  - Queueing theoretic perspective
    - The sum of load due to tasks and distribution must be less than the system capacity, else unbounded queues can build up.
    - A stable algorithm can still give worse performance than not using it. When an algorithm improves performance is called effective.

  - Algorithmic perspective
    - processor thrashing - ping-pong/hot-potato phenomena
### Sender-initiated algorithms

- **Transfer policy:** CPU queue threshold $T$ for all nodes. Initiated when a new task arrives.
- **Selection policy:** newly arrived tasks only.
- **Location policy:**
  - Random: select any node to transfer the task at random. The selected node $X$ may be overloaded. If transferred task is treated as new arrival, then $X$ may transfer the task again. Limit the number of transfers for a task. Is effective under light-load conditions.
  - Threshold: Poll nodes until a receiver is found. Up to PollLimit nodes are polled. If none is a receiver, then the sender commits to the task.
  - Shortest: Among the polled nodes that where found to be receivers, select the one with the shortest queue. Marginal improvement.
- **Information policy:** demand-driven, initiated by the sender.
- **Stability:** Unstable at high-loads.

### Receiver-initiated algorithms

- **Transfer policy:** when a task departs, node compares its CPU queue length is compared with $T$, and if smaller, node is a receiver.
- **Selection policy:** any approach, preference to non-preemptive transfers.
- **Location policy:** Randomly poll nodes until a sender is found, and transfer a task from it. If no sender is found, wait for a period or until a task completes, and repeat.
- **Information policy:** demand-driven initiated by receiver.
- **Stability:** At high loads, a receiver will find a sender with high-probability with a small number of polls. At low-loads, most polls will fail, but this is not a problem, since CPU cycles are available.
- **Drawback:** CPU scheduling algorithms are mostly round-robin, so a newly arrived task at an overload node is quickly given a slice and this starts execution. Thus, very likely preemptive transfer of tasks will take place.
**Symmetrically initiated algorithms**

- Combine both sender-initiated and receiver-initiated components in order to get a hybrid algorithm with the advantages of both.

- Care must be taken since otherwise, the hybrid algorithm may inherit the disadvantages of both sender and receiver initiated algorithms.

**The above-average algorithm of Krueger and Finkel**

- Maintain node load at an acceptable range of the system average load.

- Transfer policy
  - two thresholds are used, equidistant from the node's estimate of the average load across all nodes.
  - Nodes are classified as senders, receivers, or OK.

- Location policy
  - Has a sender-initiated and a receiver initiated component

- Selection policy: same as before

- Information policy
  - Average system load is determined individually.
  - The thresholds can be adaptive to system state to control responsiveness.
Location Policy of Krueger\&Finkel’s Algorithm

Sender-initiated part
- sender sends TooHigh msg, sets TooHigh timeout, and listens for Accept msgs.
- receiver that gets a TooHigh msg, cancels its TooLow timeout, sends Accept msg, increase its load value, and sets AwaitingTask timeout. If AwaitTask timeout expires, load value is decreased.
- sender receiving Accept msg, transfers task, and cancels timeout.
- if sender receiving a TooLow msg from a receiver, while waiting for an Accept, sends a TooHigh msg to it.
- sender whose TooHigh timeout expires, it broadcasts a ChangeAverage msg to all nodes to increase the average load estimate at the other nodes.

Location Policy of Krueger\&Finkel’s Algorithm

receiver-initiated part:
- a receiver sends a TooLow msg, sets a TooLow timeout, and starts listening for TooHigh msgs.
- a receiver getting a TooHigh msg, sends Accept msg, increase load, and sets AwaitingTask timeout. If it expires, decrease load value.
- receiver whose TooLow timeout expires, sends ChangeAverage to decrease load estimate at other nodes.
A Stable Adaptive Symmetrically Initiated Algorithm

- Instability is typically caused by the sender-component
  - Indiscriminate polling of nodes
  - Try to utilize information gathered through previous polls
- Each node maintains three lists of nodes
  - Sender, receiver, and OK lists
  - Initially it assumes that all other nodes are receivers
  - Update these lists each time information about the state of a node is received
- Transfer policy
  - A node is a receiver, a sender, or an OK node depending on whether its queue length is \( \leq LT \), \( \geq UT \), between \( LT \) and \( UT \), where \( LT \) and \( UT \) are two thresholds

A Stable Adaptive Symmetrically Initiated Algorithm

- Location Policy
- Sender component
  - When a node finds it is a sender
    - Polls up to PollLimit nodes in the Receiver list (head to tail)
    - If a polled is still a receiver, then transfer a task to it
    - If not, move the polled node at the head of the Sender or OK list, depending on its reported status
A Stable Adaptive Symmetrically Initiated Algorithm

Location Policy

Receiver component

- When a node finds it is a receiver
  - Polls up to PollLimit nodes from the
    - Sender List (head to tail)
    - OK list (tail to head)
    - Receiver list (tail to head)
  - If polled node is a Sender, then the polled node transfers a task to the polling node, and the polling moves the polled node at the head of its Sender list
  - Otherwise, move the polled node to the head of the OK or Receiver list depending on its reported status
  - Poll node moves the polling node at the head of its Receiver list

A Stable Adaptive Symmetrically Initiated Algorithm

- Selection policy: same as for sender-initiated and receiver-initiated
- Information policy: demand driven
A stable sender-initiated algorithm

- Use the sender-initiated component of the previous stable symmetric algorithm A as follows
- augmented the information at each node J with a state vector
  - V(I)=sender, receiver, OK depending on whether J knows that node is in I’s Sender, Receiver, or OK list
  - It keeps track to which lists it belongs at each other node
  - The state vector is kept up-to-date during polling
- The receiver component of is as follows
  - Whenever a node becomes a receiver it notifies all other misinformed nodes using its statevector

Selecting a suitable load-sharing algorithm

- Depends on system-load range
  - System never experiences too high-load
    - A sender initiated should perform OK
  - System load ranges wide load fluctuations from low-load to high-load
    - A stable symmetrically initiated algorithm will work OK
    - If preemptive task transfers are expensive
      - A stable sender-initiated algorithm should be preferred
  - In heterogeneous workloads (non-stationary)
    - Adaptive algorithms should be preferred