Load distribution in distributed systems

- Objectives
- How to measure load
- Load distribution algs. classification
  - static/dynamic
  - load sharing/balancing
  - preemptive/non-preemptive
- Quality evaluation
  - Queue theoretic approach
  - Algorithmic approach
- Components
  - Transfer policy
  - Selection policy
  - Location policy
  - Information policy
- Case study: V-System

Features of a good load distribution method

- No *a priori* knowledge about processes
- Dynamic in nature — change with system load, allow process migration
- Quick decision-making capability
- Balanced system performance and overhead — don't reduce system performance by collecting state information
- Stability — don't migrate processes so often that no work gets done (better definition later)
- Scalability — works on both small and large networks
- Fault tolerance — recover if one or more processors crashes

Measuring load (load index)

- number of processes
  - may be inadequate - some are swapped out, dead, etc.
- length of ready queue, length of I/O queues
  - correlates well with response time
  - used extensively
  - does correlate with CPU utilization, particularly in an interactive environment
    - solution — use a background process to monitor CPU utilization (expensive!)
- have to account for delay in task transfer (incoming and outgoing) in CPU utilization

Classifying load distribution algorithms

- How is system state (load on each processor) used?
  - Static / deterministic
    - Does not consider system state; uses static information about average behavior
    - Load distribution decisions are hard-wired into the algorithm
    - Little run-time overhead
  - Dynamic
    - Takes current system state into account
    - Has the potential to outperform static load distribution because it can exploit short-term fluctuations in system state
    - Has some overhead for state monitoring
  - Adaptive
    - Modify the algorithm based on the state
    - For example, stop collecting info (go static) if all nodes are busy so as not to impose extra overhead

Load balancing vs. load sharing

How to distribute the load

- Reduce the chance that one processor is idle while tasks contending for service at another processor, by transferring tasks between processors
  - Load balancing
    - Tries to equalize the load at ***all*** processors
    - Moves tasks more often than load sharing; much more overhead
  - Load sharing
    - Tries to reduce the load on the heavily loaded processors only
    - Probably a better solution
- Transferring tasks takes time
  - To avoid long unshared states, make *anticipatory* task transfers from overloaded processors to ones that are likely to become idle shortly
  - Raises transfer rate for load sharing, making it close to load balancing

Preemptive vs. non-preemptive transfer

- can a task be transferred to another processor once it starts executing?
- non-preemptive transfer (task placement)
  - can only transfer tasks that have not yet begun execution
  - have to transfer environment info
    - program code and data
    - environment variables, working directory, inherited privileges, etc.
  - simple
- preemptive transfers
  - can transfer a task that has partially executed
  - have to transfer entire state of the task
    - virtual memory image, process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc.
  - expensive

Stability of a load-balancing algorithm

- Queuing-theoretic approach
  - When the long-term arrival rate of work to a system is greater than its capacity to perform work, the system is *unstable*
    - Overhead due to load distribution can itself cause *instability*
      - Exchanging state, transfer tasks, etc.
  - Even if an algorithm is stable, it may cause the system to perform worse than if the algorithm were not used at all — if so, we say the algorithm is *ineffective*
  - An effective algorithm must be stable, a stable algorithm can be ineffective (?)
- Algorithmic perspective
  - If an algorithm performs fruitless actions indefinitely with finite probability, it is *unstable* (e.g., processor thrashing)
    - Transfer task from P1 to P2, P2 exceeds threshold, transfers to P1, P1 exceeds…

Components of a load distribution algorithm

- Transfer policy
  - Determines if a processor is in a suitable state to participate in a task transfer
- Selection policy
  - Selects a task for transfer, once the transfer policy decides that the processor is a sender
- Location policy
  - Finds suitable processors (senders or receivers) to share load
- Information policy
  - Decides:
    - When information about the state of other processors should be collected
    - Where it should be collected from
    - What information should be collected

Transfer policy

- Determines whether or not a processor is a sender or a receiver
  - Sender — overloaded processor
  - Receiver — underloaded processor
- Threshold-based transfer
  - Establish a *threshold*, expressed in units of load (however load is measured)
  - When a new task originates on a processor, if the load on that processor exceeds the threshold, the transfer policy decides that that processor is a sender
  - When the load at a processor falls below the threshold, the transfer policy decides that the processor can be a receiver
- Single threshold
  - Simple, maybe too many transfers
- Double thresholds — high and low
  - Guarantees a certain performance level
- Imbalance detected by information policy

Selection Policy

- Selects which task to transfer
  - Newly originated – simple (task just started)
  - Long (response time improvement compensates transfer overhead)
  - small size
  - with minimum location-dependent system calls (residual bandwidth minimized)
  - lowest priority
- Priority assignment policy
  - Selfish – local processes given priority – penalizes processes arriving at busy node
  - Altruistic – remote processes given priority – remote processes may
  - Intermediate – give priority on the ratio of local/remote processes in the system

Location policy

- Once the transfer policy designates a processor a sender, finds a receiver
  - Or, once the transfer policy designates a processor a receiver, finds a sender
- Polling — one processor polls another processor to find out if it is a suitable processor for load distribution, selecting the processor to poll either:
  - Randomly
  - Based on information collected in previous polls
  - On a nearest-neighbor basis
- Can poll processors either serially or in parallel (e.g., multicast)
  - Usually some limit on number of polls, and if that number is exceeded, the load distribution is not done
- Can also just broadcast a query to find a node who wants to be involved

Information policy

- Decides:
  - When information about the state of other processors should be collected
  - Where it should be collected from
  - What information should be collected
- Demand-driven
  - A processor collect the state of the other processors only when it becomes either a sender or a receiver (based on transfer and selection policies)
  - Dynamic — driven by system state
    - Sender-initiated — senders look for receivers to transfer load onto
    - Receiver-initiated — receivers solicit load from senders
    - Symmetrically-initiated — combination where load sharing is triggered by the demand for extra processing power or extra work

Information policy (cont.)

- Demand-driven(cont.)
  - Periodic
    - Processors exchange load information at periodic intervals
    - Based on information collected, transfer policy on a processor may decide to transfer tasks
    - Does not adapt to system state — collects same information (overhead) at high system load as at low system load
- State-change-driven
  - Processors disseminate state information whenever their state changes by a certain degree
  - Differs from demand-driven in that a processor disseminates information about its own state, rather than collecting information about the state of other processors
  - May send to central collection point, may send to peers

Case study: V-System

- developed at Stanford in 80-ies, microkernel-based, UNIX-emulating, binds several workstations on a LAN into a distributed system
- Load index – CPU utilization at the node
- information policy – state change driven (each node broadcasts whenever its state changes significantly, info is cashed by all nodes)
  - State change (rather than demand-driven) scheme is selected because it does not vary as much with load
- Selection policy – only new tasks are scheduled for transfer
- Location policy
  - each machine randomly selects one of the M lightly loaded machines from cache
  - polls it (to verify the cache info) and transfers the task
  - if cache data is not correct, cache is updated and a new machine is selected
  - Usually no more than 3 polls are necessary