Distributed Scheduling

CS239
Advanced Operating Systems

May 7, 1997
Dr. Richard Guy

Lecture Source Materials

Singhal, Ch. 11: **Distributed Scheduling**

Outline

- Introduction and motivation
- Fundamental issues/requirements
- Load distributing algorithm components
- Algorithm classes
- Performance comparison
- Selecting a suitable algorithm
- Task migration

Introduction and Motivation

- How to fully exploit vast processing power of a distributed system?
  - (re)distribute load across system
- What systems are ripe for distributing load?
  - expected imbalance in task submissions
  - known imbalance in processor performance
  - homogenous task arrival/service rate (!)

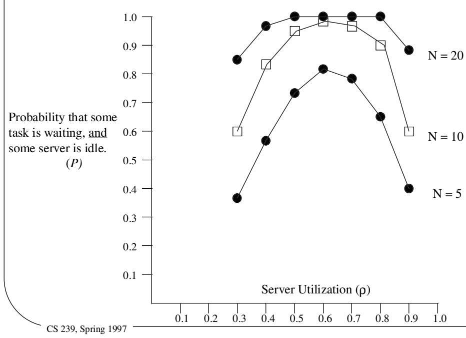
Analytical motivation for DS

- Consider homogenous system with
  - N identical and independent $M/M/1$ servers
    - same task arrival and task service rates (poisson)
  - Let ρ be the utilization of each server
  - $P_0 = 1 - \rho$ is probability that server is idle
  - Let P be probability that total system is in state
    - at least one task is waiting for service, AND
    - at least one server is idle

Homogenous system analysis

- $P = \sum \binom{N}{i} Q_i H_{N-i}$
  - Q_i is probability given set of servers i is idle
  - H_{N-i} is probability given set of servers $N-i$
    - not idle, AND
    - task waiting at one or more of servers
- take Kleinrock's CS212…
- $P = 1 - (1 - P_0)^N(1-P_0^N) - P_0^N(2 - P_0)^N$

Homogenous potential for DS



Homogenous potential notes

- at moderate utilization, *P* is high
- at high utilization, *P* is low
  - – most servers likely to be already busy, so less choice in idle servers
- at low utilization, *P* is low
  - – most servers likely to be idle, so less choice in overloaded servers
- as *N* increases, *P* is high (!) when ρ is high

Fundamental Issues

- Load definition
- Classifiying load distributing algorithms
- Load sharing vs. load balancing
- Preemptive vs. nonpreemptive transfers

Load definition

- standard basis is task reponse time
  - – not so easy to measure/maintain history
- CPU queue length
  - – correlates well with task response time
  - – easy/cheap to measure
  - – care required when transferring tasks
    - artificially increment value when task accepted

Classifying algorithms

- Dynamic
  - – exploits knowledge of current system states
  - – overhead in collecting/storing/analyzing
- Static
  - – ignore system state
- Adaptive
  - – adjust to state
  - – for example, avoid state collection at high load

Load sharing vs balancing

- Load sharing (LS) goal:
  - – reduce likelihood of task waiting on server when another server is idle
- Load balancing (LB) additional goal:
  - – equalize loads at all servers
  - – higher rate of task transfers→higer overhead
- Anticipatory transfers in LS algorithms
  - – reduce idle duration; behaves more like LB

Preemptive vs nonpreemptive

- Preemptive moves partially executed task
  - virtual memory image
  - process control block
  - unread I/O buffers and messages
  - file pointers
  - timers
- Nonpreemptive only moves non-started task
- Environment must move, too...

Additional Requirements

- Scalability
- Execution location transparency
  - a la file location transparency
- Determinism
- Preemption
  - system owner's performance maintained
- Heterogeneity
  - hardware/software services, capabilities

and...

- load sharing overhead cost must not have negative effect on single system performance, especially at high loads
  - must be stable
    - stable: work arrival rate <= service capacity
    - avoid indefinite useless work: processor thrashing
  - must be effective
    - better performance relative to no load distributing

Algorithm components

- Transfer policy
  - OK to participate in transfer?
- Selection policy
  - which task to transfer?
- Location policy
  - where to transfer to?
- Information policy
  - when to get local/global system state

Transfer policy

- Is node in suitable state to participate?
  - as sender
  - as receiver
- Threshold policies
  - when local load > *T*, try to send a task
  - when local load < *T*, accept a task
- Imbalance policies
  - based on global state analysis

Selection policy

- When transfer policy says "OK to send", which task (if any) should be sent?
- Select new task, nonpreemptive
- Transfer overhead vs. response time reduction
  - system overhead amounts to stolen task execution time for system, other tasks
  - small, long-lived tasks often good choice
  - may have historical/type info

Location policy

- Find senders/receivers
- Polling
  - serial vs. parallel (multicast)
  - random selection vs. nearest neighbor
- Broadcast

Information policy

- When/where/what to collect
- Demand-driven
  - collect state when T/S policies make us S/R
  - sender/receiver/symmetrically initiated
- Periodic
  - non-adaptive
  - drives Transfer policy
- State-change-driven (push, vs pull of DD)

Algorithm Classes

- Use component policies + stability to study various classes of demand-driven (cf. information policy) algorithms:
  - Sender-initiated
  - Receiver-initiated
  - Symmetrically initiated
  - Adaptive

Sender-initiated

- activity initiated by overloaded node (sender) wanting to send task to underloaded node (receiver)
- consider three simple, effective algorithms
  - Differ only in location policy
- Transfer policy
  - queue length threshold: receiver < T; sender > T
- Selection policy

Sender-initiated, cont.

- Location policy
  - Random
  - Threshold
  - Shortest

Random

- tranfer without consultation
- possible useless transfers
- must avoid regression to “continuous transfer” state
- place artificial limit on tranfers of one task
- much better performance than no sharing

Threshold

- poll randomly chosen receiver for suitability
- transfer if OK with receiver
- if not OK, re-choose/poll limited times
- possible that task won't move
  - must service locally
- much better performance than Random
  - no useless transfers

Shortest

- attempt to find "best" receiver
- poll limited number randomly chosen nodes
- select shortest queue length < T
- only marginal improvement over Threshold location policy
  - more detailed state info not always helpful(!)

Sender-initiated stability

- all three cause instability at high loads
- polling cost is proportional to workload
- unsuccessful polls + responses consumes all cycles
- eventually, cost of load sharing exceeds benefit

Receiver-initiated

- activity initiated by underloaded node (receiver) wanting to get task from overloaded node (sender)
- consider one algorithm
- Transfer policy
  - queuelength threshold: receiver < T; sender > T
- Selection policy
  - preemptive or nonpreemptive

Receiver-initiated, cont.

- Location policy
  - random
  - if sender not located, wait and retry
  - incentive to be aggressive, to avoid lost cycles
- Stability
  - at high loads, high probability of finding task to transfer in few polls
  - at low loads, spare cycles are available to process useless polls

Receiver-initiated caution...

- most transferred tasks are preemptive!
  - much more expensive than nonpreemptive
- round-robin scheduling disciplines
  - new task quickly gets service quantum
  - receiver poll unlikely to arrive in "time"
- cf. sender-initiated: mostly nonpreemptive

Symmetrically initiated

- Both senders and receivers search
- Has advantages of both send-recv initiated
  - low load, sender more likely to find receivers
  - high load, receiver more likely to find sender
- But, also has disadvantages of both
  - polling at high loads leads to instability
  - preemptive task transfer facility needed
- Simple algorithm: combine S/R-initiated

Above-average algorithm

- Symmetrically initiated
- Acceptable range (fat threshold)
  - avoids most processor thrashing
  - node estimates systemwide load average
  - lower (LT), upper (UT) equidistant to average
  - adaptive range size
  - LT, UT adjusted dynamically at each node

Above-average algorithm, cont.

- Transfer policy
  - receiver, if load < LT
  - sender, if load > UT
- Selection policy
  - standard (see earlier algorithms)
- Information policy
  - demand driven

Above-average Location Policy

- Sender-initiated component
  - sender broadcasts TooHigh, sets timeout, waits
  - receiver hears TooHigh, sends Accept, load++
  - sender sees Accept, sends task to receiver
  - if timeout expires before Accept seen
    - sender infers load average estimates too low
    - broadcasts ChangeAverage++

- Receiver-initiated component
  - receiver broadcasts TooLow, sets timeout, wait
  - receiver hears TooHigh, sends Accept, load++
  - if timeout expires before TooHigh seen
    - receiver infers load average estimates too high
    - broadcasts ChangeAverage--
- Inexpensive, autonomous range adjusting
- Adapts well to network load

Adaptive

- Main cause of instability is sender polling
- Can reduce effect by saving recent history
  - classify nodes as sender, receiver, OK
  - maintain lists of each (cheaply)
- Stable symmetrically initiated algorithm
  - all nodes initially assumed to be receivers

Stable Symmetrically Initiated

- Transfer policy
  - receiver < LT <= OK <= UT < sender
- Selection policy
  - sender only considers new tasks
  - receiver can cause preemptive transfer
- Information policy
  - demand driver

Stable Sym. Location Policy

- Sender-initiated component
  - poll node at head of receiver list
  - polled node updates lists, and responds
  - sender transfers, or updates lists and retries
- Receiver-initiated component
  - poll nodes (up to PollLimit)
    - head to tail on sender list
    - tail to head on OK list, receiver list

Stable Sym. Initiated

- High load
  - useless sender polls are quickly stopped (receiver lists are empty)
- Low load
  - "useless" receiver polls don't degrade performance
  - polls cause receiver lists to be up to date

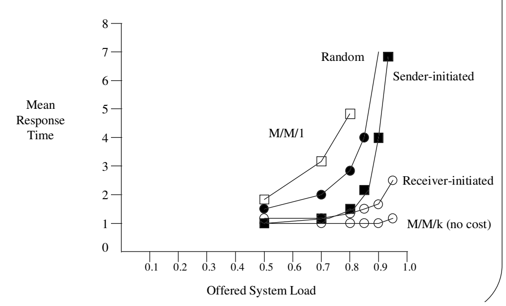
Stable Sender-initiated

- very similar to Stable Sym. Initiated
- limited to nonpreemptive transfers
- add *statevector* to each node
  - which list I'm on at other nodes
  - receivers multicast to "misinformed" nodes

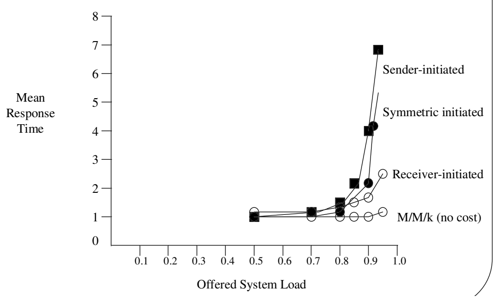
Performance Comparison

- General trends of previous algorithms
- Standard assumptions about load distributions, homogeneity
- Assumes 40 nodes
- Fixed "point" threshold = 1
- PollLimit = 5
- Main result: even simple scheme helps!

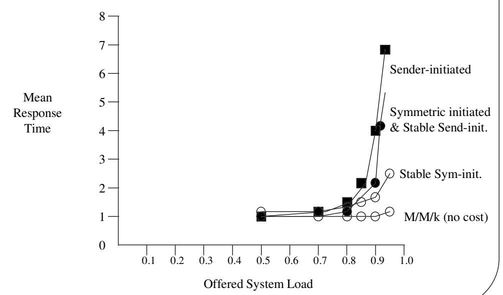
Receiver- vs. Sender-initiated



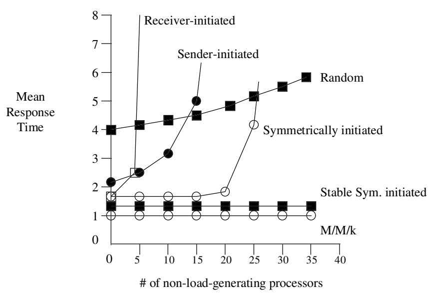
Symmetrically initiated



Stable algorithms



Heterogenous workloads



Selecting a suitable algorithm

- Never high load: sender-initiated helps
- High load: stable better than nonadaptive
  - sender-initiated polls are expensive at high load
  - receiver-initiated preemptions are expensive
- wide fluctuations: stable sym.-initiated
- wide fluc's + costly preemp: stable send-init
- heterogeneous: adaptive stable

Case studies to note

- V-system
- Sprite
- Condor
- Stealth

Task migration

- Hard to avoid preemptive transfer
- Issues
  - State transfer
  - Location transparency
  - Performance

State transfer

- cost to support remote execution
  - delays
    - freeze task, get/transfer state, unfreeze task
    - other tasks timeout/abort, waiting for unfreeze
- residual dependencies...

Residual dependencies

- sender resources still used by migrated task
  - page transfer on demand
  - message redirection
  - location-dependent system calls
- undesirable
  - reliability
  - performance
  - complexity

Location transparency

- Name transparency
  - files
  - processes
  - self-validating name-to-address map caches
- IPC
  - forwarding addresses
  - notify all partners

Task migration performance

- Sprite: 330 ms average to migrate
  - Sparcstation 1 + 10Mbps ethernet
- Charlotte: 242 ms average to migrate simple
  - VAX 11/750 + 10Mbps token ring

Summary

- DS necessary to exploit resource potential
- DS can substantially improve performance
- DS must be done with care to avoid artificially introducing instability
- Preemptive task migration is expensive