# How to deal with uncertainties and dynamicity?

Frédéric Vivien

October 23, 2013

# Outline

1. Sensitivity and Robustness
2. Analyzing the sensitivity: the case of Backfilling
3. Extreme robust solution: Internet-Based Computing
4. Dynamic load-balancing and performance prediction
5. Conclusion

# Outline

# The problem: the world is not perfect!

- Uncertainties
  - On the platforms' characteristics
    (Processor power, link bandwidth, etc.)
  - On the applications' characteristics
    (Volume computation to be performed, volume of messages to be sent, etc.)
- Dynamicity
  - Of network (interferences with other applications, etc.)
  - Of processors (interferences with other users, other processors of the same node, other core of the same processor, hardware failure, etc.)
  - Of applications (on which detail should the simulation focus?)

# Solutions: to prevent or to cure?

To prevent

- Algorithms tolerant to uncertainties and dynamicity.

To cure

- Algorithms auto-adapting to actual conditions.

Leitmotiv : the more the information, the more precise we can statically define the solutions, the better our chances to “succeed”

# Analyzing the sensitivity

Question: we have defined a solution, how is it going to behave "in practice"?

Possible approach

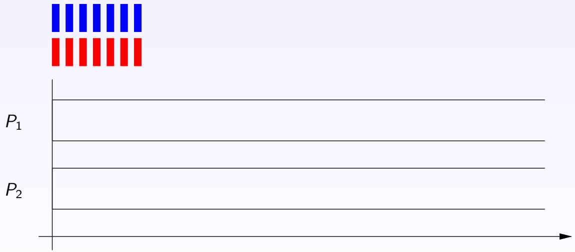
1. Definition of an algorithm $\mathcal{A}$.
2. Modeling the uncertainties and the dynamicity.
3. Analyzing the sensitivity of $\mathcal{A}$ as follows:
    - For each theoretical instance of the problem
        - Evaluate the solution found by $\mathcal{A}$
        - For each "actual" instance corresponding to the given theoretical instance, find the optimal solution and the relative performance of the solution found by $\mathcal{A}$.

    Sensitivity of $\mathcal{A}$: worst relative performance, or (weighted) average relative performance, etc.

# Analyzing the sensitivity: an example

## Problem

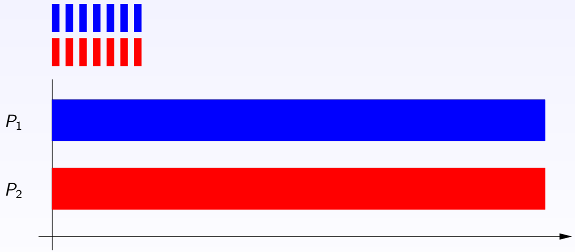
- Master-slave platform with two identical processors
- Flow of two types of identical tasks
- Objective function: maximum minimum throughput between the two applications (*max-min fairness*)



# Analyzing the sensitivity: an example

## Problem

- Master-slave platform with two identical processors
- Flow of two types of identical tasks
- Objective function: maximum minimum throughput between the two applications (*max-min fairness*)

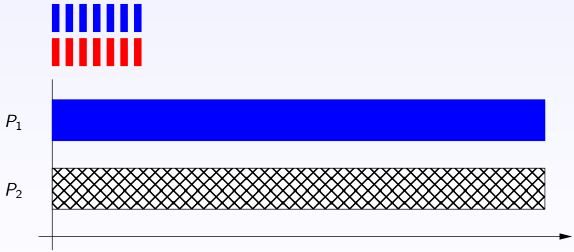



A possible solution...

# Analyzing the sensitivity: an example

## Problem

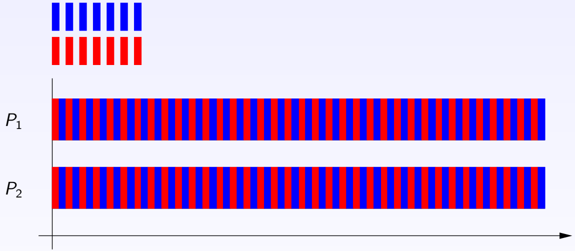
- Master-slave platform with two identical processors
- Flow of two types of identical tasks
- Objective function: maximum minimum throughput between the two applications (*max-min fairness*)




A possible solution... null if processor $P_2$ fails.

# Robust solutions

An algorithm is said to be robust if its solutions stay close to the optimal when the actual parameters are slightly different from the theoretical parameters.




This solution stays optimal whatever the variations in the processors' performance: it is not sensitive to this parameter!

# An example: the problem

Problem:

- A master has an output bandwidth $B$.

# An example: the problem

Problem:

- A master has an output bandwidth $B$.
- $p$ workers; worker $P_i$ is linked with the bandwidth $b_i$ and a computation rate $c_i$.

# An example: the problem

Problem:

- A master has an output bandwidth $B$.
- $p$ workers; worker $P_i$ is linked with the bandwidth $b_i$ and a computation rate $c_i$.
- $n$ task flows; each flow is a set of identical tasks
  A task of flow $i$ requires $\beta_k$ units of communications and $\gamma_k$ units of computations

# An example: the problem

Problem:

- A master has an output bandwidth $B$.
- $p$ workers; worker $P_i$ is linked with the bandwidth $b_i$ and a computation rate $c_i$.
- $n$ task flows; each flow is a set of identical tasks
  A task of flow $i$ requires $\beta_k$ units of communications and $\gamma_k$ units of computations
- $\rho_i^{(k)}$: throughput of application $k$ on processor $i$.
  $\rho^{(k)} = \sum_i \rho_i^{(k)}$ is the overall throughput of application $k$ on the platform

# An example: the problem

Problem:

- A master has an output bandwidth $B$.
- $p$ workers; worker $P_i$ is linked with the bandwidth $b_i$ and a computation rate $c_i$.
- $n$ task flows; each flow is a set of identical tasks
  A task of flow $i$ requires $\beta_k$ units of communications and $\gamma_k$ units of computations
- $\rho_i^{(k)}$: throughput of application $k$ on processor $i$.
  $\rho^{(k)} = \sum_i \rho_i^{(k)}$ is the overall throughput of application $k$ on the platform
- Objective: maximize $\min_k \rho^{(k)}$.

# An example: the problem

Problem:

- A master has an output bandwidth $B$.
- $p$ workers; worker $P_i$ is linked with the bandwidth $b_i$ and a computation rate $c_i$.
- $n$ task flows; each flow is a set of identical tasks
  A task of flow $i$ requires $\beta_k$ units of communications and $\gamma_k$ units of computations
- $\rho_i^{(k)}$: throughput of application $k$ on processor $i$.
  $\rho^{(k)} = \sum_i \rho_i^{(k)}$ is the overall throughput of application $k$ on the platform
- Objective: maximize $\min_k \rho^{(k)}$.

Dynamicity: a processor may fail

# An example: classical solution

1. Resource constraints: processing power

$$\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \le c_i$$

# An example: classical solution

1. Resource constraints: processing power

$$\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$$

2. Resource constraints: workers' bandwidth

$$\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$$

# An example: classical solution

1. Resource constraints: processing power

$$\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$$

2. Resource constraints: workers' bandwidth

$$\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$$

3. Resource constraints: master's bandwidth

$$\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$$

# An example: classical solution

1. Resource constraints: processing power

$$\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$$

2. Resource constraints: workers' bandwidth

$$\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$$

3. Resource constraints: master's bandwidth

$$\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$$

4. Objective

$$\text{Maximize} \min_k \sum_i \rho_i^{(k)}$$

# An example: a robust solution

1. $\forall i, \quad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$
2. $\forall i, \quad \sum_k \rho_i^{(k)} \beta_k \leq b_i$
3. $\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$

# An example: a robust solution

1. $\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$
2. $\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$
3. $\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$
4. Objective when exactly worker $P_p$ fails:

$$\rho_{\bar{p}} = \min_k \sum_{i \neq p} \rho_i^{(k)}$$

# An example: a robust solution

1. $\forall i, \qquad \sum_k \rho_i^{(k)} \gamma_k \leq c_i$
2. $\forall i, \qquad \sum_k \rho_i^{(k)} \beta_k \leq b_i$
3. $\sum_{i,k} \rho_i^{(k)} \beta_k \leq B$
4. Objective when exactly worker $P_p$ fails:

$$\rho_{\bar{p}} = \min_k \sum_{i \neq p} \rho_i^{(k)}$$

5. Objective: Maximize $\min \left\{ \min_p \frac{\rho_{\bar{p}}}{\rho_{\bar{p}}^{(\mathrm{opt})}}, \min_k \sum_i \frac{\rho_i^{(k)}}{\rho^{(\mathrm{opt})}} \right\}$

# Outline

# Analyzing the sensitivity: the case of Backfilling (1)

Context:

- cluster shared between many users
- need for an allocation policy, and a reservation policy
- job request: number of processors + maximal utilization time (A job exceeding its estimate is automatically killed)

Simplistic policies:

- First Come First Served: lead to resource waste
- Reservations: too static (jobs finish usually earlier than predicted)
- Backfilling: large scheduling overhead, possible starvation

# Analyzing the sensitivity: the case of Backfilling (2)

### The EASY backfilling scheme

- The jobs are considered in First-Come First-Served order
- Each time a job arrives or a job completes, a reservation is made for the first job that cannot be immediately started, later jobs that can be started immediately are started.
- In practice jobs are submitted with runtime estimates.
  A job exceeding its estimate is automatically killed.

# Analyzing the sensitivity: the case of Backfilling (3)

The set-up

- 128-node IBM SP2 (San Diego Supercomputer Center)
- Log from May 1998 to April 2000 log: 67,667 jobs
  *Parallel Workload Archive* (www.cs.huji.ac.il/labs/parallel/workload/)
- Job runtime limit: 18 hours.
  (Some dozens of seconds may be needed to kill a job.)
- Performance measure: average slowdown (=average stretch).

Execution is simulated based on the trace: enable to change task duration (or scheduling policy).

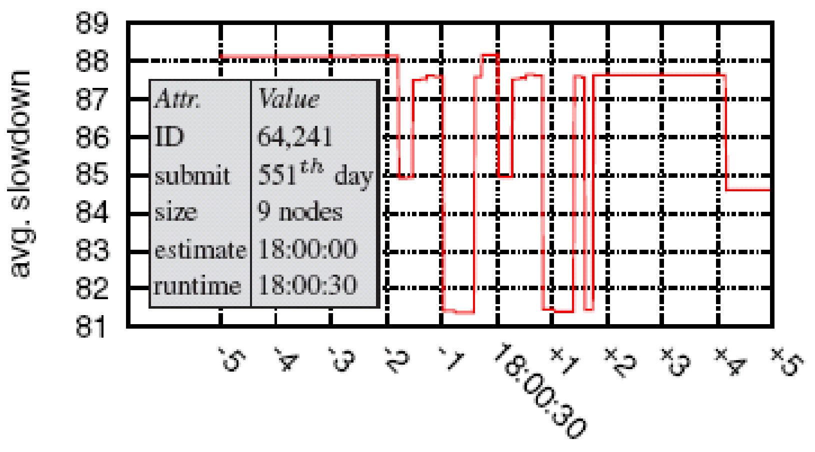
# Analyzing the sensitivity: the case of Backfilling (3)

The set-up

- 128-node IBM SP2 (San Diego Supercomputer Center)
- Log from May 1998 to April 2000 log: 67,667 jobs
  *Parallel Workload Archive* (www.cs.huji.ac.il/labs/parallel/workload/)
- Job runtime limit: 18 hours.
  (Some dozens of seconds may be needed to kill a job.)
- Performance measure: average slowdown (=average stretch).
  Bounded slowdown: $\max\left(1, \frac{T_w + T_r}{\max(10, T_r)}\right)$

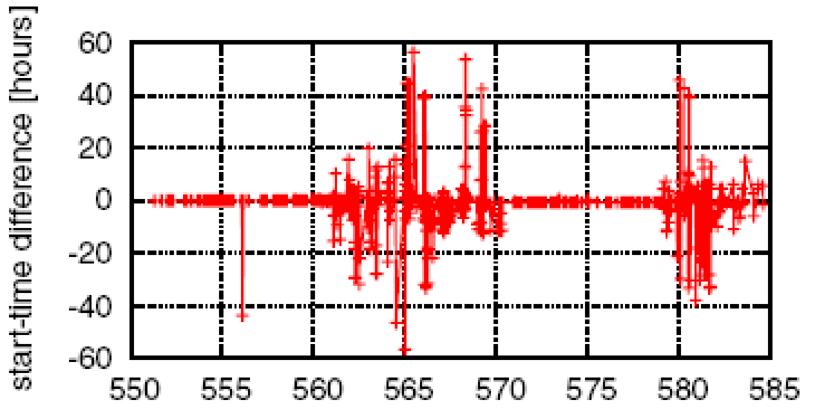
Execution is simulated based on the trace: enable to change task duration (or scheduling policy).

# Analyzing the sensitivity: the case of Backfilling (4)

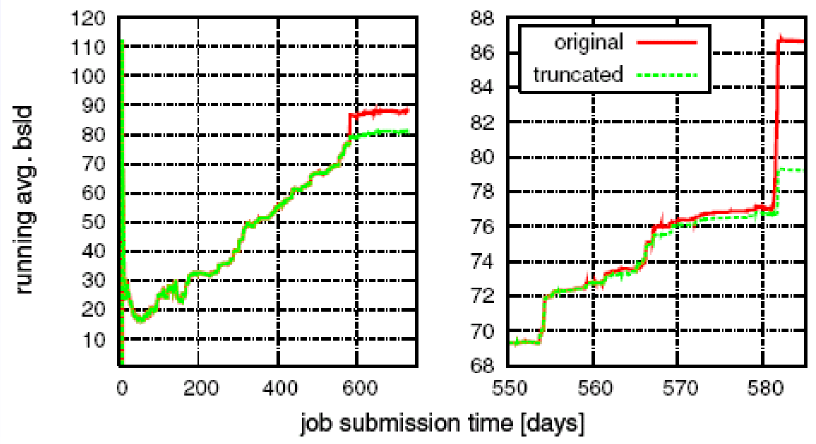



The length of a job running for 18 hours and 30 seconds is shorten by 30 seconds.

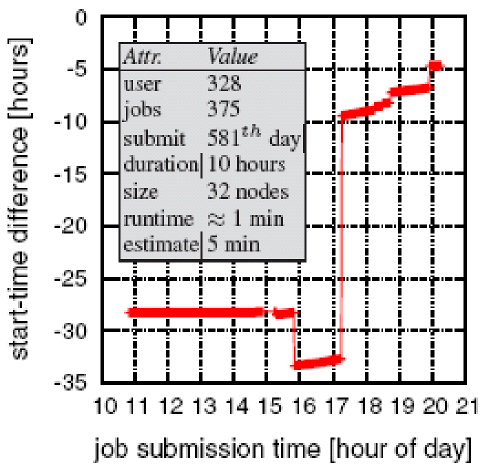
# Analyzing the sensitivity: the case of Backfilling (4)



# Analyzing the sensitivity: the case of Backfilling (4)



# Analyzing the sensitivity: the case of Backfilling (4)



# Outline

# Internet-Based Computing

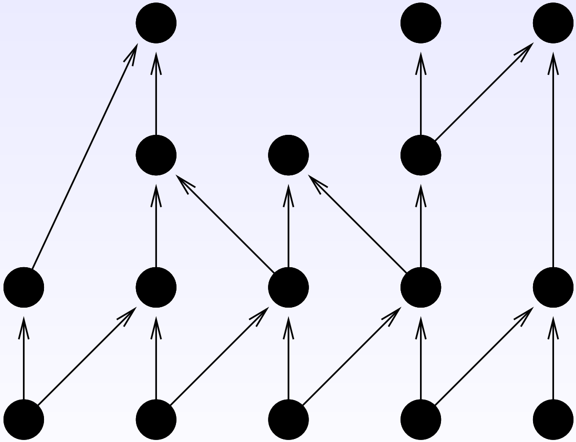
## Context

- Volunteer computing (over the Internet)
- Processing resources unknown, unreliable
- Application with precedence constraints (task graph)

## The principle

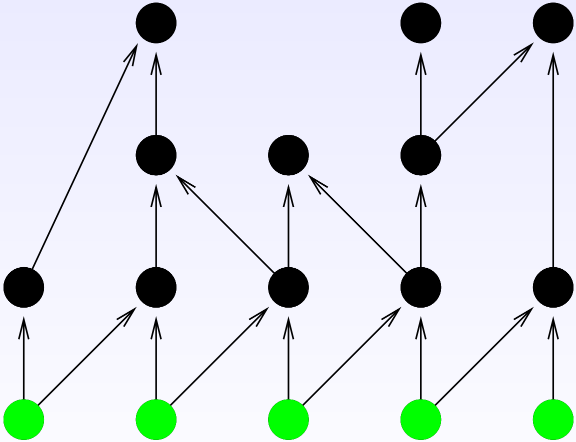
- Motivation: lessening the likelihood of the "gridlock" that can arise when a computation stalls pending computation of already allocated tasks.

# Internet-Based Computing: example



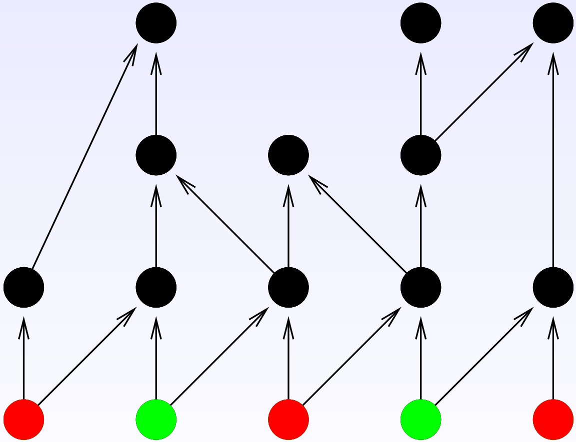
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



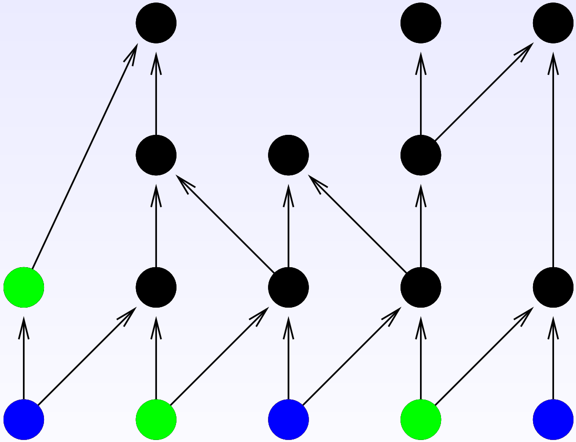
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



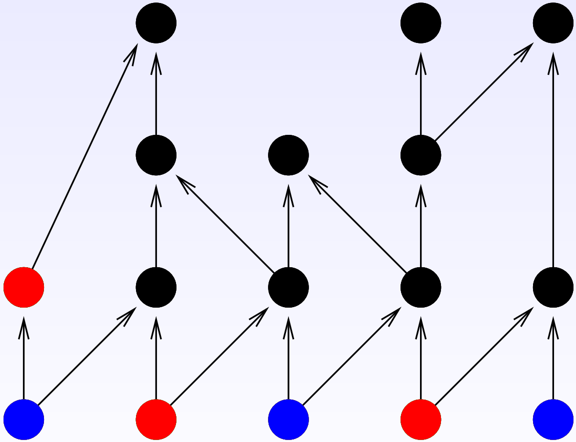
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



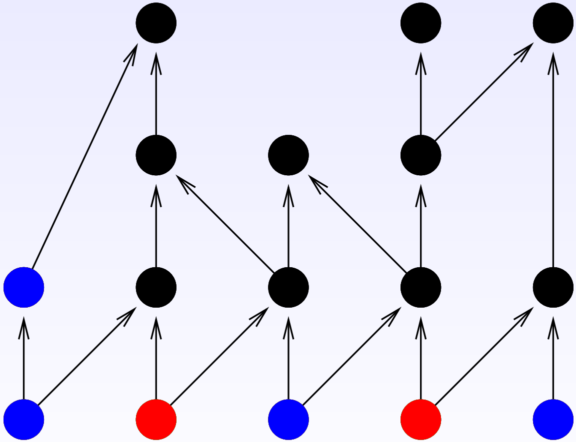
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



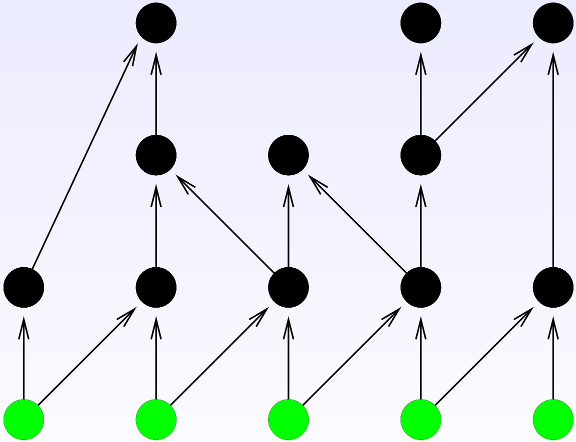
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



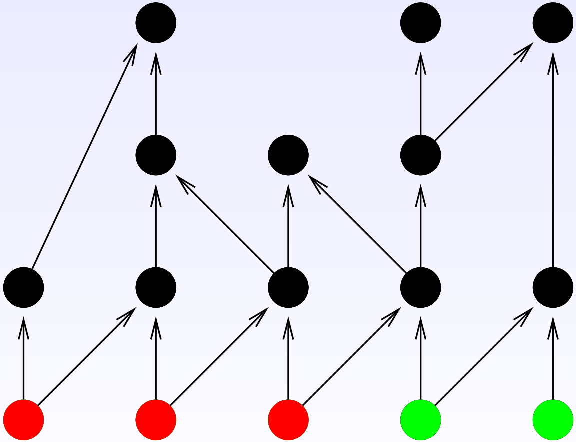
A possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



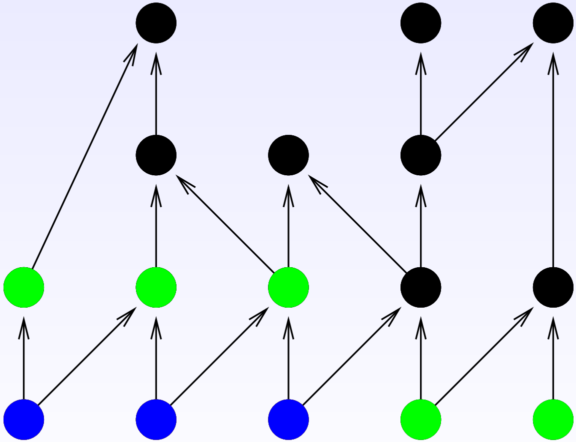
Another possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



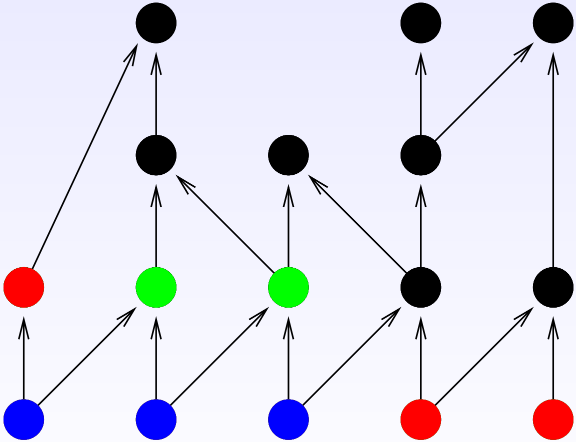
Another possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



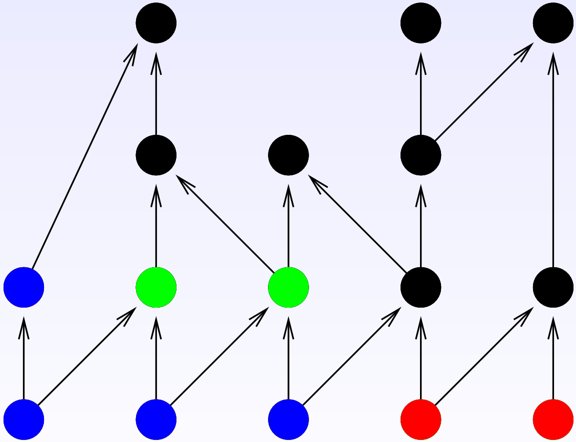
Another possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



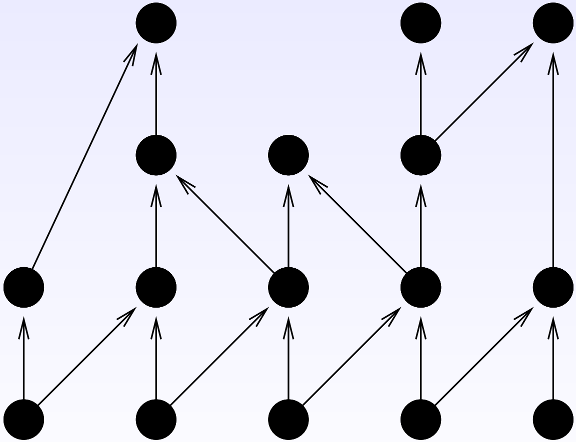
Another possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



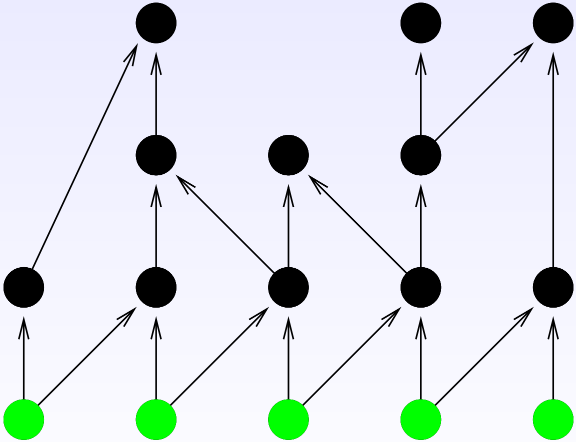
Another possible schedule
(enabled, in process, completed)

# Internet-Based Computing: example



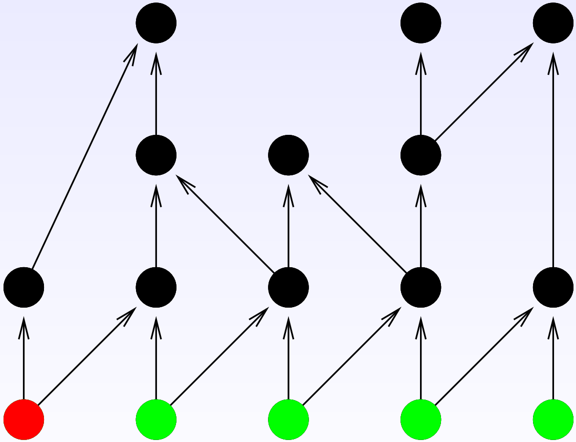
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



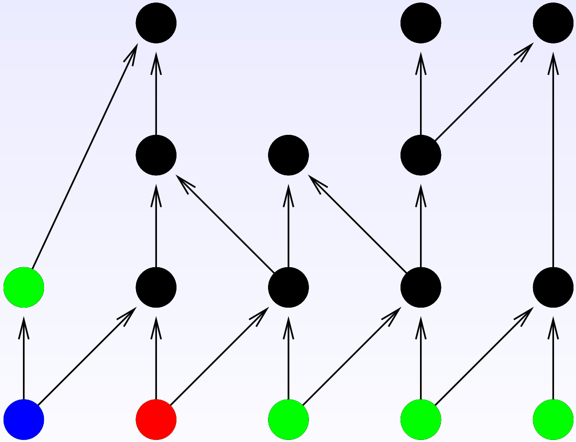
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



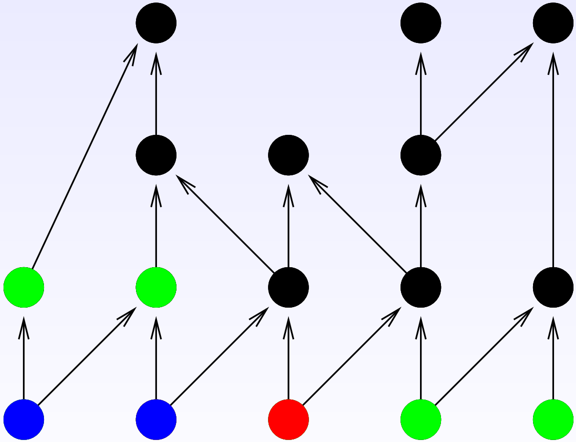
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



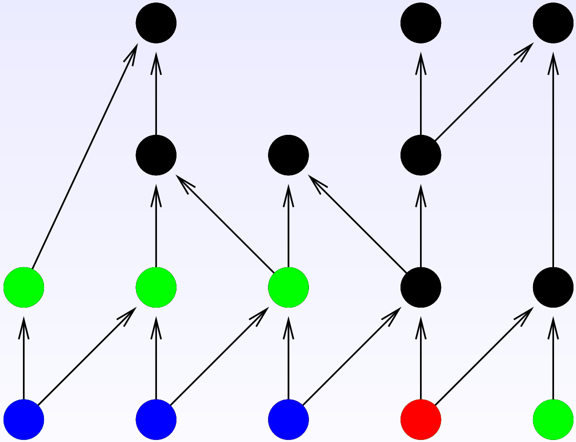
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



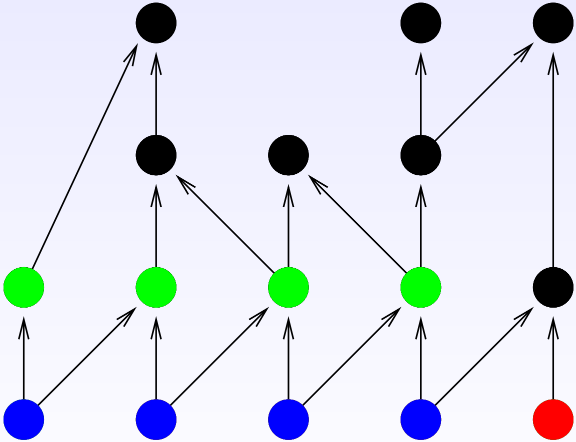
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



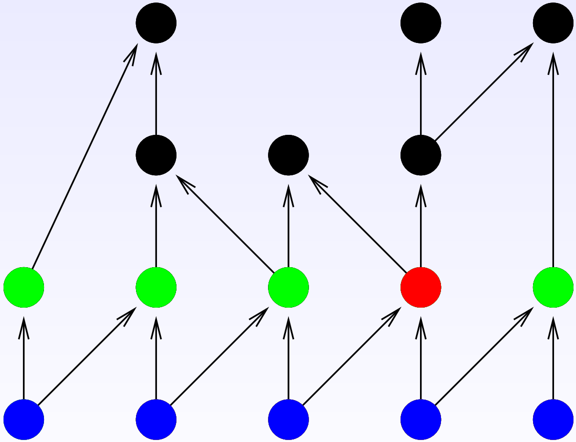
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



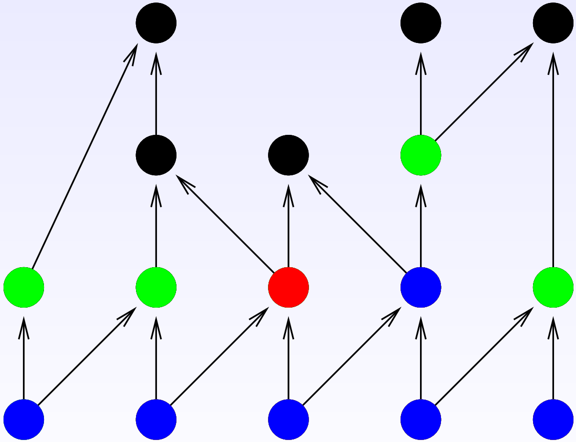
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



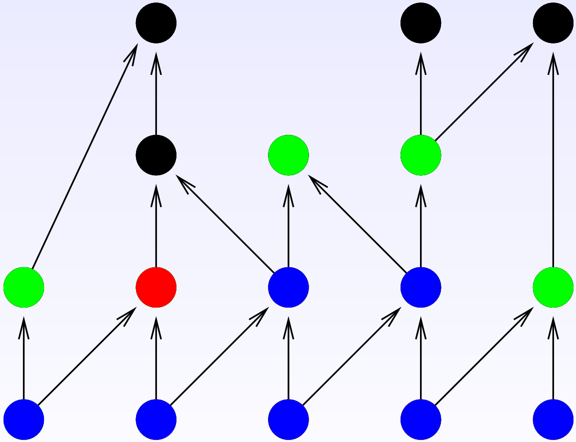
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



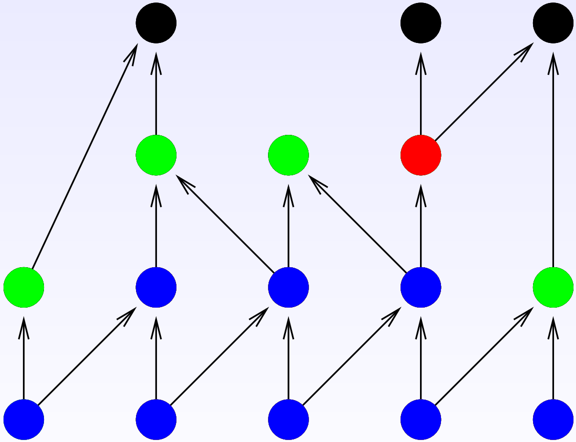
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



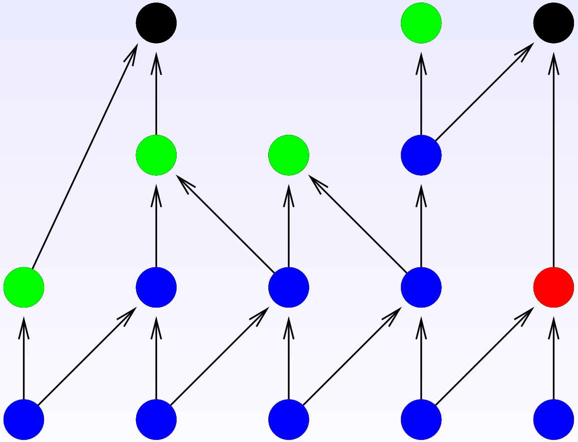
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



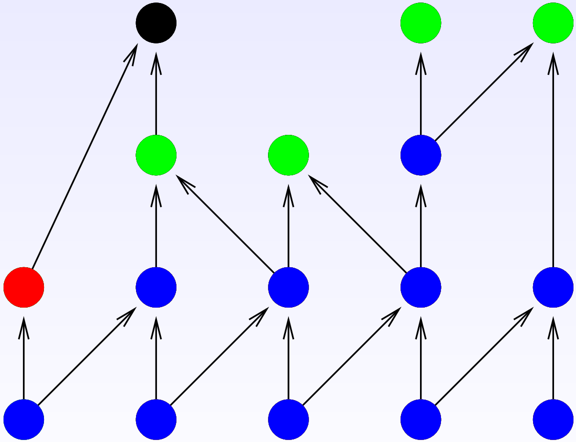
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



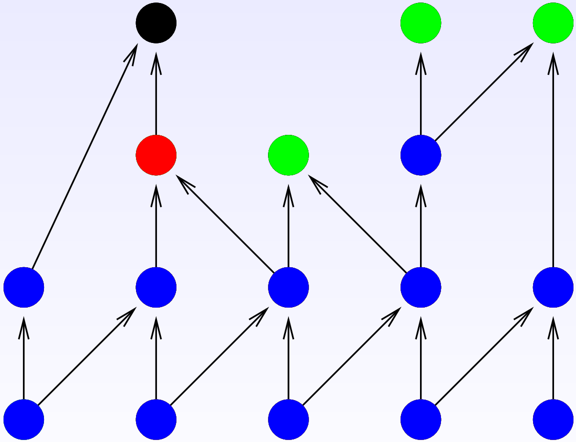
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



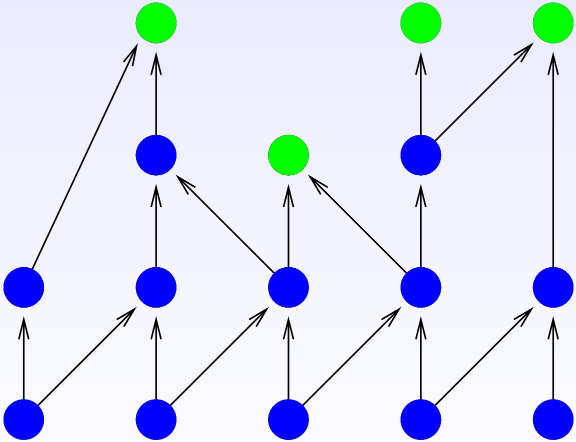
The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: example



The IC-optimal schedule: after $t$ tasks have been executed, the number of eligible (=executable) tasks is maximal (for any $t$)

# Internet-Based Computing: results

Results:

- IC-optimal schedule for basic DAGs (forks, joins, cliques, etc.)
- Decomposition of DAGs into basic building blocks
- IC-optimal schedules for blocks compositions

Shortcomings:

- No IC-optimal schedules for many DAGs (even trees)
- Move from "maximize number of eligible tasks at all times" to "maximal average number of eligible tasks"

# Outline

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:

    - Compute a good solution using the observed parameters.

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:
  - Compute a good solution using the observed parameters.
  - Evaluate the cost of balancing the load

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:
    - Compute a good solution using the observed parameters.
    - Evaluate the cost of balancing the load
    - If the gain is larger than the cost: load-balance

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:
  Each invocation has a cost: the invocations should only take place at "useful" instants
  - Compute a good solution using the observed parameters.
  - Evaluate the cost of balancing the load
  - If the gain is larger than the cost: load-balance

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:
  Each invocation has a cost: the invocations should only take place at "useful" instants
  - Compute a good solution using the observed parameters.
  - Evaluate the cost of balancing the load
  - If the gain is larger than the cost: load-balance
    If the objective is to minimize the running time, the comparison is obvious. How do we compare a time and some QoS?

# General scheme

To cure (rather than to prevent): the algorithm balance the load to take into account uncertainties and dynamicity.

- From time to time, do:
  Each invocation has a cost: the invocations should only take place at "useful" instants
  - Compute a good solution using the observed parameters.
    How do we predict the future from the past?
  - Evaluate the cost of balancing the load
  - If the gain is larger than the cost: load-balance
    If the objective is to minimize the running time, the comparison is obvious. How do we compare a time and some QoS?

# Performance monitoring

Distributed system which periodically monitors/records network and processor performance.

Also, allows to predict the future performance of the network and of the processors.

# Performance monitoring

Distributed system which periodically monitors/records network and processor performance.

Also, allows to predict the future performance of the network and of the processors.

Does the past enable to predict the future?

# How useful is old information?

### The problem

- The values used when taking decisions have already "aged".
- Is it a problem? Should we take this ageing into account?

# Framework: the platform

- A set of $n$ servers.
- Tasks arrive according to a Poisson law of througput $\lambda n$, $\lambda < 1$.
- Task execution time: exponential law of mean 1.
- Each server executes in FIFO order the tasks it receives.
- We look at the time each task spent in the system (=flow).

# Framework: information

There is a *bulletin board* on which are displayed the loads of the different processors.

This information may be wrong or approximate.

We only deal with the case in which this information is *old*.

This is the only information available to the tasks: they cannot communicate between each other and have some coordinated behavior.

# The obvious strategies

- Random and uniform choice of the server.
  - Low overhead, finite length of queues.

# The obvious strategies

- Random and uniform choice of the server.
  - Low overhead, finite length of queues.
- Random and uniform choice of $d$ servers, the task being sent on the least loaded of the $d$ servers.
  - Better than random, practical in distributed settings (poll a small number of processors)

# The obvious strategies

- Random and uniform choice of the server.
    - Low overhead, finite length of queues.
- Random and uniform choice of $d$ servers, the task being sent on the least loaded of the $d$ servers.
    - Better than random, practical in distributed settings (poll a small number of processors)
- Task sent on the least loaded server.
    - Optimal in a variety of situations, need for centralization.

# First model: periodic updates

- Each $T$ units of time the bulletin board is updated with correct information.

- $P_{i,j}(t)$: fraction of queues with true load $j$ but load $i$ on the board, at time $t$
- $q_i(t)$ rate of arrivals at a queue with size $i$ on the board at time $t$

System dynamics:

$$\frac{dP_{i,j}(t)}{dt} = P_{i,j-1}(t) \times q_i(t) + P_{i,j+1}(t) - P_{i,j}(t) \times q_i(t) - P_{i,j}(t)$$

# First model: specific strategies

fractions of servers with (apparent) load $i$: $b_i(t) = \sum_j Pi, j(t)$

- choose the least loaded among $d$ random servers

$$q_i(t) = \lambda \frac{\left(\sum_{j \geq i} b_j(t)\right)^d - \left(\sum_{j > i} b_j(t)\right)^d}{b_i(t)}$$

# First model: specific strategies

fractions of servers with (apparent) load $i$: $b_i(t) = \sum_j Pi, j(t)$

- choose the least loaded among $d$ random servers

$$q_i(t) = \lambda \frac{\left(\sum_{j \geq i} b_j(t)\right)^d - \left(\sum_{j > i} b_j(t)\right)^d}{b_i(t)}$$

- choose the shortest queue (assume there is always a server with load 0)

$$\begin{aligned} q_0(t) &= \frac{\lambda}{b_0(t)} \\ q_i(t) &= 0 \qquad i \neq 0 \end{aligned}$$

# Three possible resolutions

1. Theoretical:
   - fixed point $when \frac{dP_{i,j}(t)}{dt} = 0$?
   - fixed cycle on $[kT, (k+1)T]$
   - can be solved using waiting queue theory (close form, but complex)

# Three possible resolutions

1. Theoretical:
   - fixed point $when \frac{dP_{i,j}(t)}{dt} = 0$?
   - fixed cycle on $[kT, (k+1)T]$
   - can be solved using waiting queue theory (close form, but complex)
2. Practice with the above differential system:
   - simulations, on truncated version of the system (bounding $i$ and $j$)

# Three possible resolutions

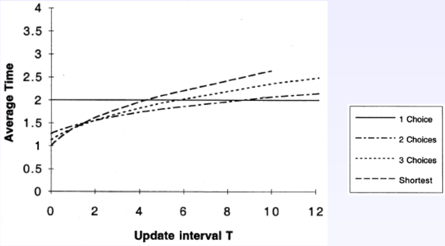
1. Theoretical:
   - fixed point $when \frac{dP_{i,j}(t)}{dt} = 0$?
   - fixed cycle on $[kT, (k+1)T]$
   - can be solved using waiting queue theory (close form, but complex)
2. Practice with the above differential system:
   - simulations, on truncated version of the system (bounding $i$ and $j$)
3. Practice without the differential system:
   - simulate 100 queues
   - can use every distribution you want

# Three possible resolutions

1. Theoretical:
   - fixed point $when \frac{dP_{i,j}(t)}{dt} = 0$?
   - fixed cycle on $[kT, (k+1)T]$
   - can be solved using waiting queue theory (close form, but complex)
2. Practice with the above differential system:
   - simulations, on truncated version of the system (bounding $i$ and $j$)
3. Practice without the differential system:
   - simulate 100 queues
   - can use every distribution you want

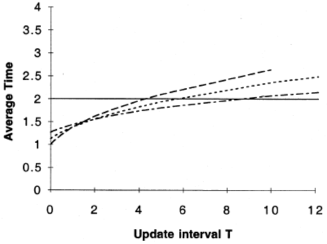
After using 2 and 3: comparable results on the same set of parameters.

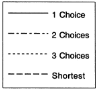
# First model: results

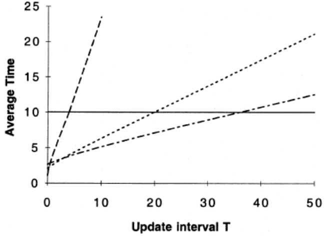



$n = 100$ and $\lambda = 0.5$

# First model: results



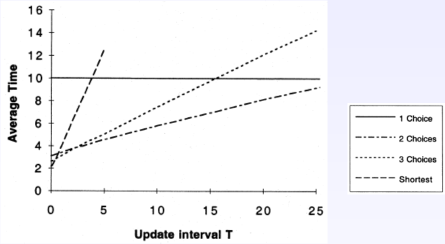



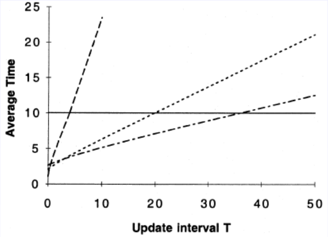

$n = 100$ and $\lambda = 0.5$

$n = 100$ and $\lambda = 0.9$

# First model: results



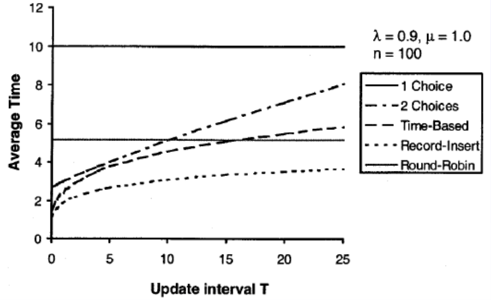

$n = 8$ and $\lambda = 0.9$




$n = 100$ and $\lambda = 0.9$

# First model: more elaborated strategies

- *Time-based*: random choice among the servers which are supposed to be the least loaded.
- *Record-Insert*: centralized service in which each task updates the bulletin board by indicating on which server it is sent.

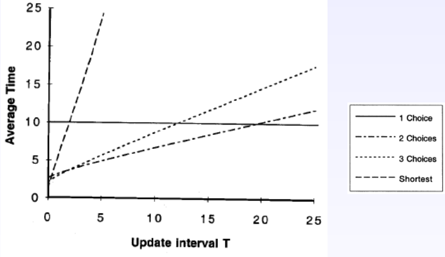



$n = 100$ and $\lambda = 0.9$

# Second model: continuous updates

Model: continuous updates, but the information used is $T$ units of time old.

$n = 100$ and $\lambda = 0.9$



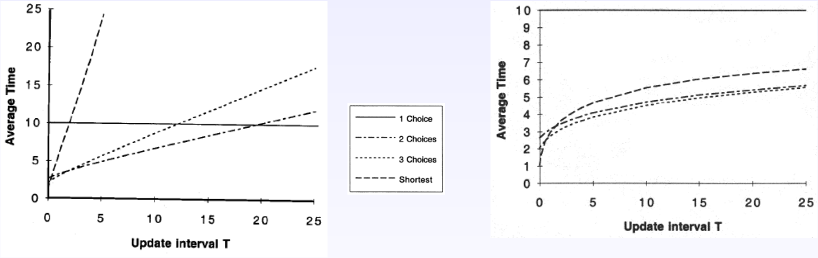

Age of information:

exactly $T$

# Second model: continuous updates

Model: continuous updates, but the information used is $T$ units of time old.

$n = 100$ and $\lambda = 0.9$



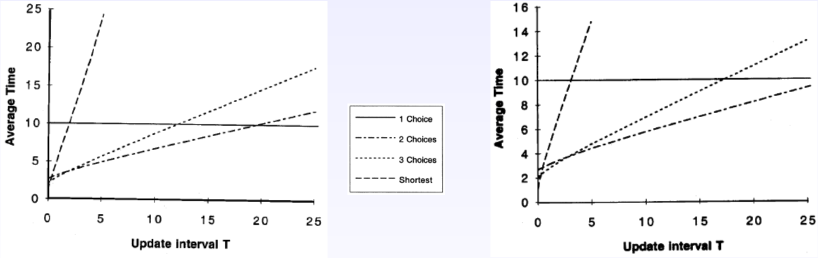

Age of information:

exactly $T$ | exponential distribution,average $T$

# Second model: continuous updates

Model: continuous updates, but the information used is $T$ units of time old.

$n = 100$ and $\lambda = 0.9$



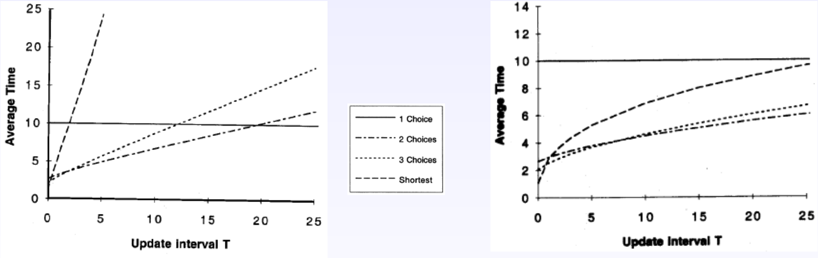

Age of information:

exactly $T$ | uniform distribution on $[\frac{T}{2}; \frac{3T}{2}]$

# Second model: continuous updates

Model: continuous updates, but the information used is $T$ units of time old.

$$n = 100 \text{ and } \lambda = 0.9$$



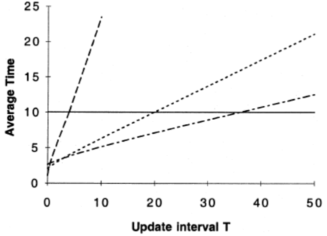

Age of information:

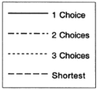
exactly $T$ uniform distribution on $[0; 2T]$

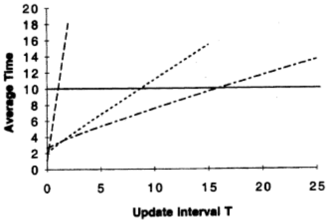
# Third model: de-synchronized updates

The different servers update their information in a de-synchronized manner, each following an exponential law of average $T$.

$$n = 100 \text{ and } \lambda = 0.9$$




Regular updates

De-synchronized updates.

# And if some were cheating?

With a probability $p$ a task does not choose between two randomly determined servers, but takes the least loaded of all servers.

| $T$ | $p$ | Avg. Time All Tasks | Avg. Time 2 Choices | Avg. Time Shortest | Variance All Tasks | Variance 2 Choices | Variance Shortest |
|---|---|---|---|---|---|---|---|
| 1 | 0.00 | 3.23286 | | | | | |
| 1 | 0.01 | 3.21072 | 3.22093 | 2.19877 | 5.73117 | 5.74186 | 3.63718 |
| 1 | 0.05 | 3.17061 | 3.21389 | 2.34814 | 5.62948 | 5.67621 | 4.02956 |
| 1 | 0.10 | 3.14132 | 3.20978 | 2.52474 | 5.58554 | 5.65450 | 4.54205 |
| 1 | 0.25 | 3.20098 | 3.25693 | 3.03311 | 6.05849 | 5.94553 | 6.35980 |
| 5 | 0.00 | 4.94051 | | | | | |
| 5 | 0.01 | 4.95386 | 4.95677 | 4.66575 | 13.8029 | 13.8821 | 11.8131 |
| 5 | 0.05 | 5.05692 | 5.05668 | 5.06154 | 14.4591 | 14.5105 | 13.4837 |
| 5 | 0.10 | 5.21456 | 5.17956 | 5.52974 | 15.6083 | 15.6597 | 15.7552 |
| 5 | 0.25 | 6.06968 | 5.70758 | 7.15609 | 23.6380 | 22.0182 | 26.9240 |
| 10 | 0.00 | 6.74313 | | | | | |
| 10 | 0.01 | 6.80669 | 6.80588 | 6.88703 | 26.4946 | 26.5391 | 22.0827 |
| 10 | 0.05 | 7.00344 | 6.97692 | 7.50776 | 28.4836 | 28.6189 | 25.6448 |
| 10 | 0.10 | 7.36957 | 7.26152 | 8.34185 | 32.7326 | 32.7395 | 31.6201 |
| 10 | 0.25 | 8.91193 | 8.23577 | 10.9422 | 54.8097 | 52.0265 | 57.6721 |

# Complete vs. incomplete information

Complete information

- Requires some centralization (or total replication);
- Communications of the most remote elements to the "center";
- Obsolescence of the information.

Decentralized schedulers

- The local data are more up-to-date;
- A local optimization does not always lead to a global optimization...

# Outline

# Conclusion

- An obvious need to be able to cope with the dynamicity and the uncertainties.
- Crucial need to be able to model the dynamicity and the uncertainty.
- The static world is already complex enough!
- Where is the trade-off between the precision of the models and their usability?
- Trade-off between static and dynamic approaches?