Resource Management in Parallel and Distributed Systems

Paolo Romano

Outline

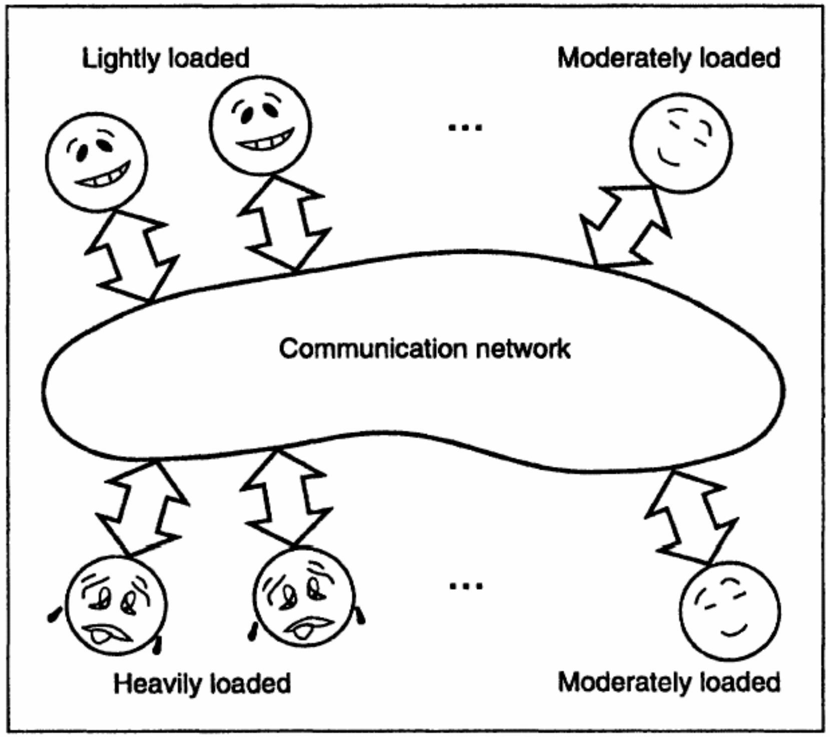
- **Scheduling & Load Balancing**:
  - Introduction to Scheduling
  - Problem definition & some hints on computational complexity
  - A Taxonomy of (General) Scheduling Approaches
    - Ortogonal classification attributes
  - Components of load distribution algorithms
  - Case Studies:
    - Workstation clusters
    - Scheduling in locally & geographically distributed Web server systems

- **Distributed Cache Management**:
  - Introduction to Distributed Caching Schemes
  - Case Studies:
    - WWW Caching
    - Survey on Transactional Caches

- **Multi-path Approaches in Geographically Distributed Systems**:
  - Content Delivery
  - Transactional Systems

Introduction to Scheduling

- **Allocation of system resources relative to the system computational load is fundamental to exploit the potential power of distributed computations.**

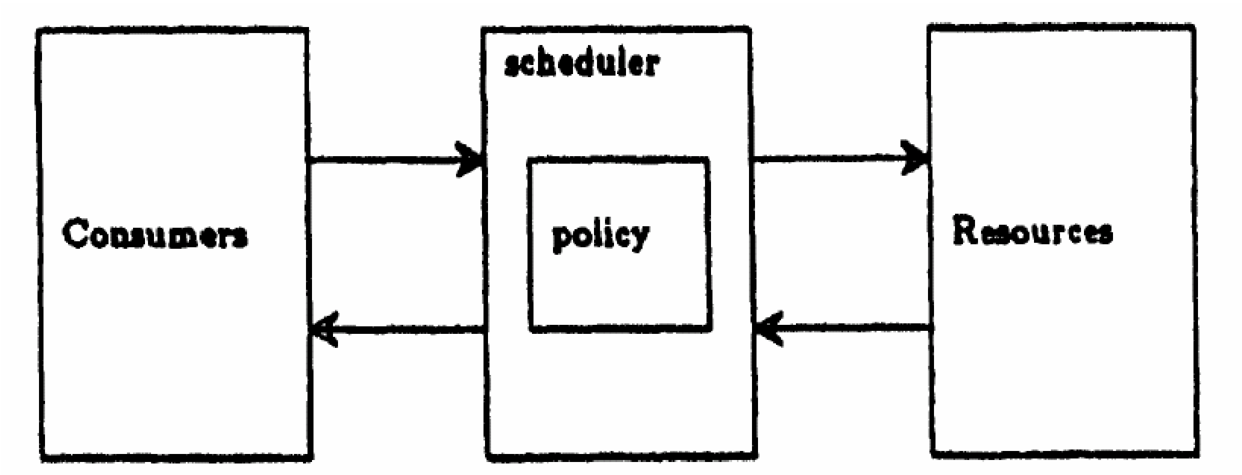


A distributed system with no load distribution

Introduction to Scheduling

"how to distribute (or schedule) the processes among processing elements to achieve some performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization"

- **A restatement of the classical problem of job sequencing in production management**
- **A scheduling problem consists of three (logical) components:**



Even simple scheduling problem instances (static) are NP-complete…

- Consider this simple scenario:
  - 2 processors, same capacity
  - N processes, different execution times, known a-priori (static scheduling)
  - No dependencies, no communication cost…
  - GOAL: minimize completion time of the processes set.

- If a schedule exists in which the processing loads are equal with no unnecessary delay, this is optimal!

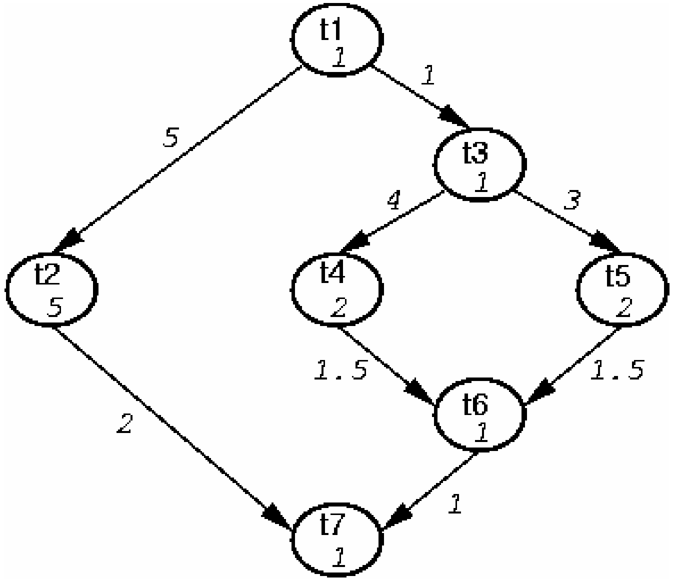
- Finding such a schedule maps straightforwardly to the NP-complete “set-partitioning” problem:
  - Given a set of integers A and an integer size s(a), find A’ subset of A s.t.:

$$\sum_{a \in A'} s(a) = \sum_{b \in (A-A')} s(b).$$

…and scheduling problems can be much more complex than that!

Essentially with additional constraints, e.g. :

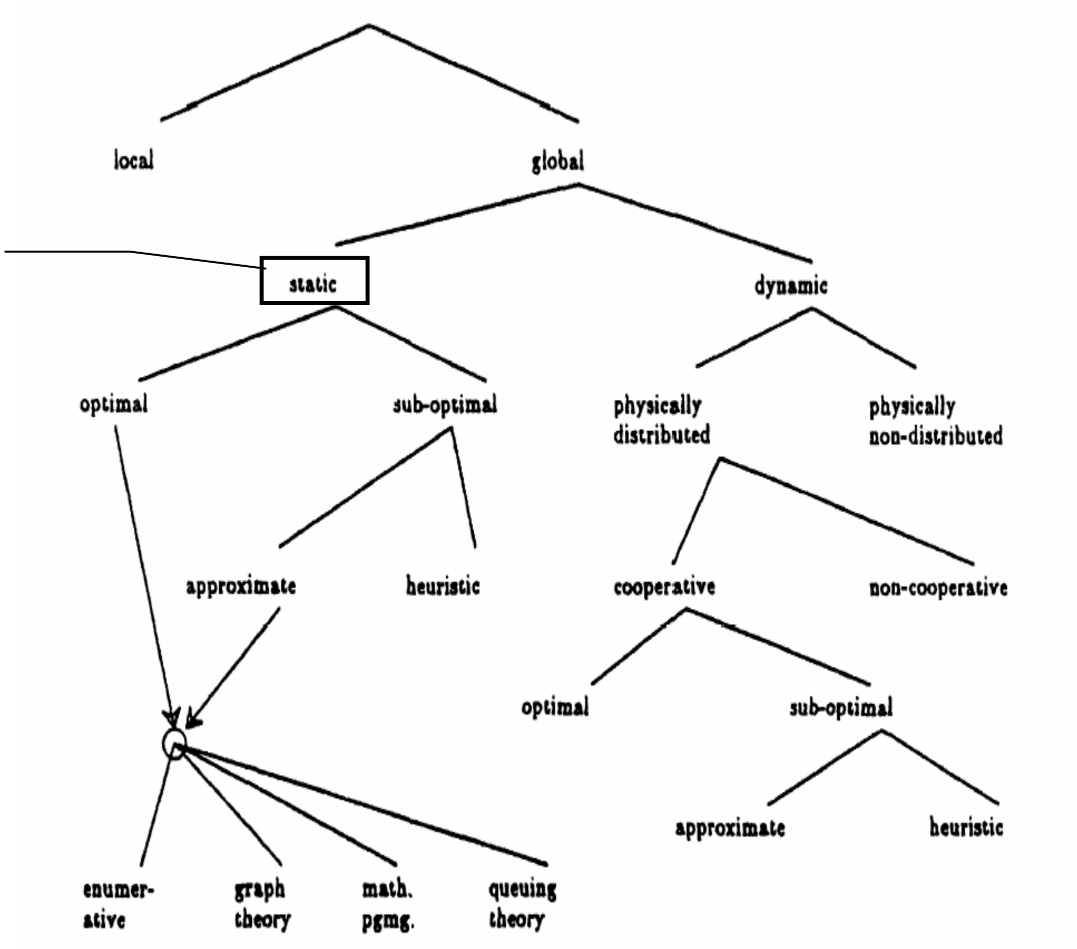
- Task dependencies:
  - Example: parallel programming
  - Modelled by Direct Acyclic Graphs (DAGs)
    - Vertex is a task, its weight representing the computational cost
    - Arc expresses causal dependency, its weight representing the communication cost



- Meeting task deadlines:
  - Example: Real Time Systems
  - In this case a feasible scheduling may even not exist!

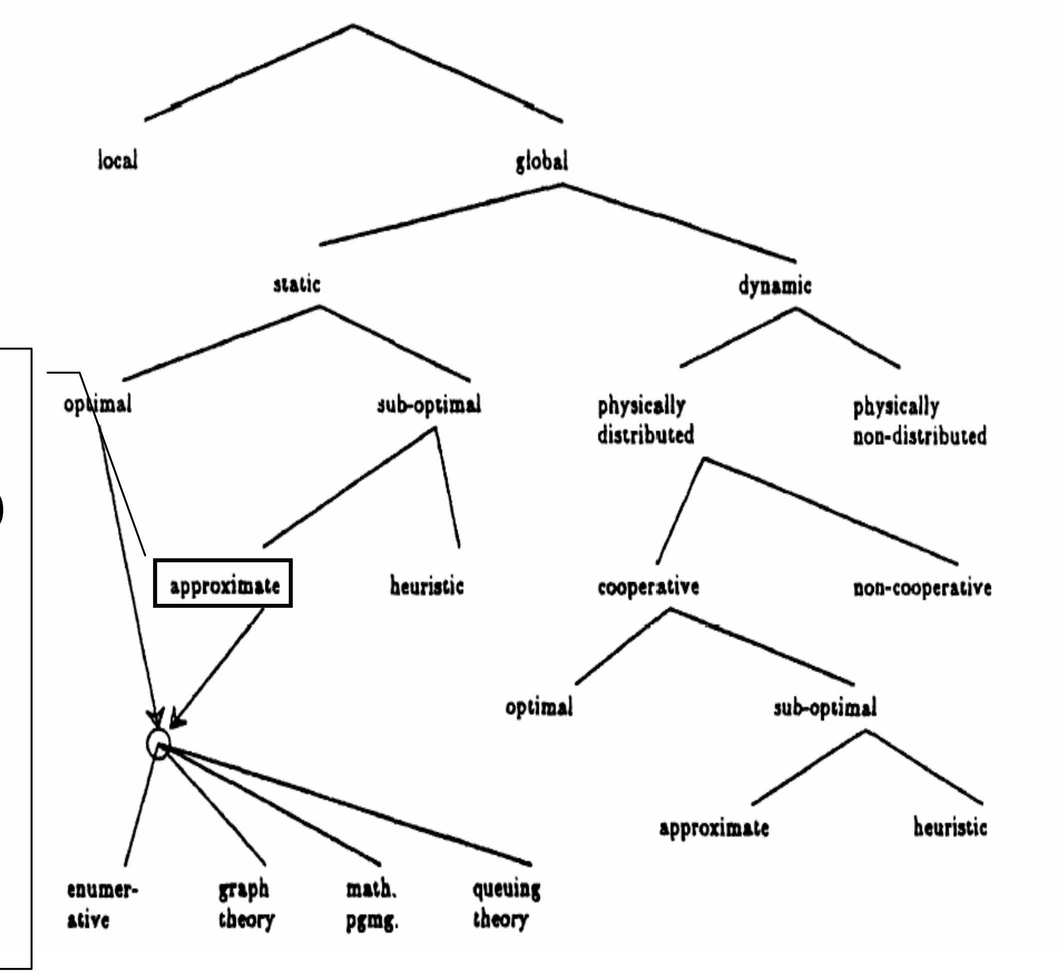
A taxonomy of (general purpose) scheduling approaches

- a-priori info on the task demands is known.
- topology may change, in which case a new assignment has to be determined.



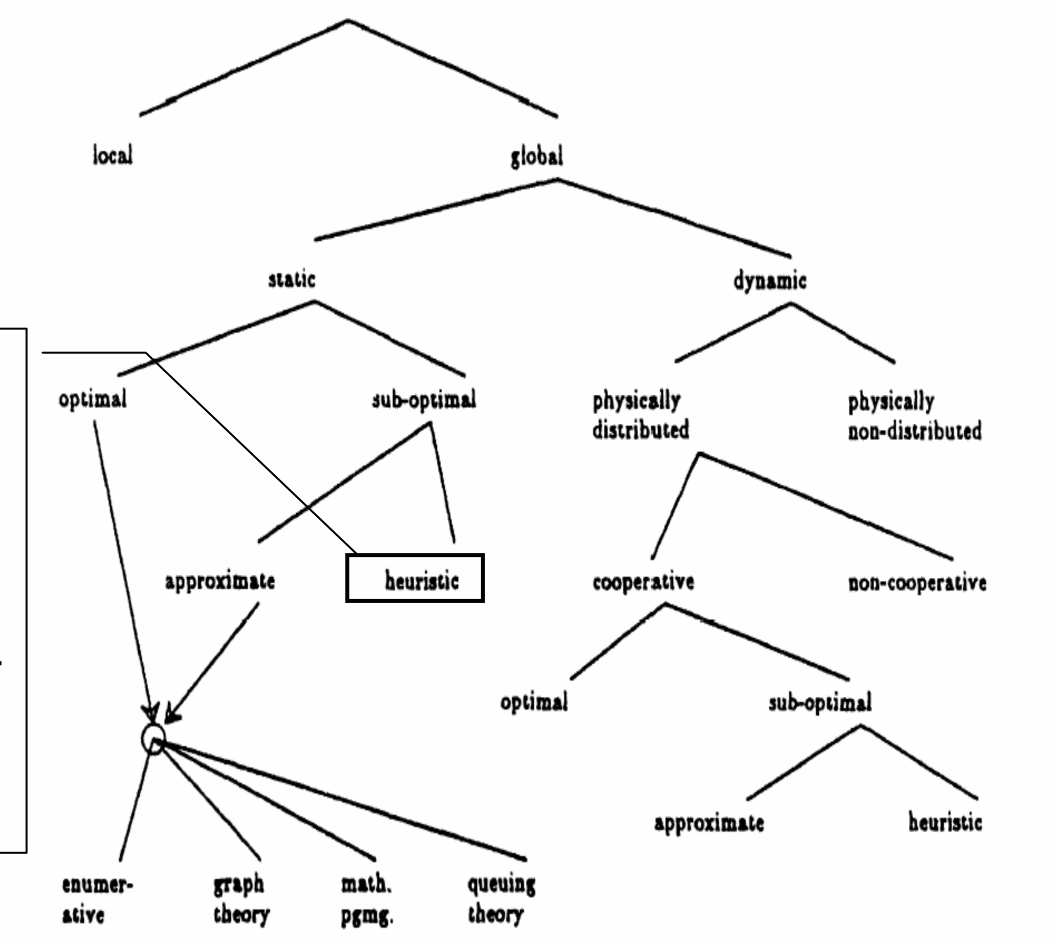
A taxonomy of (general purpose) scheduling approaches

- based on approximate solving techniques of the associated mathematical model (e.g. CSP)
- exploration of the solution space is stopped when a “good” result is found:
- how to evaluate a solution?
- how expensive is it?
- how to prune the solution space?



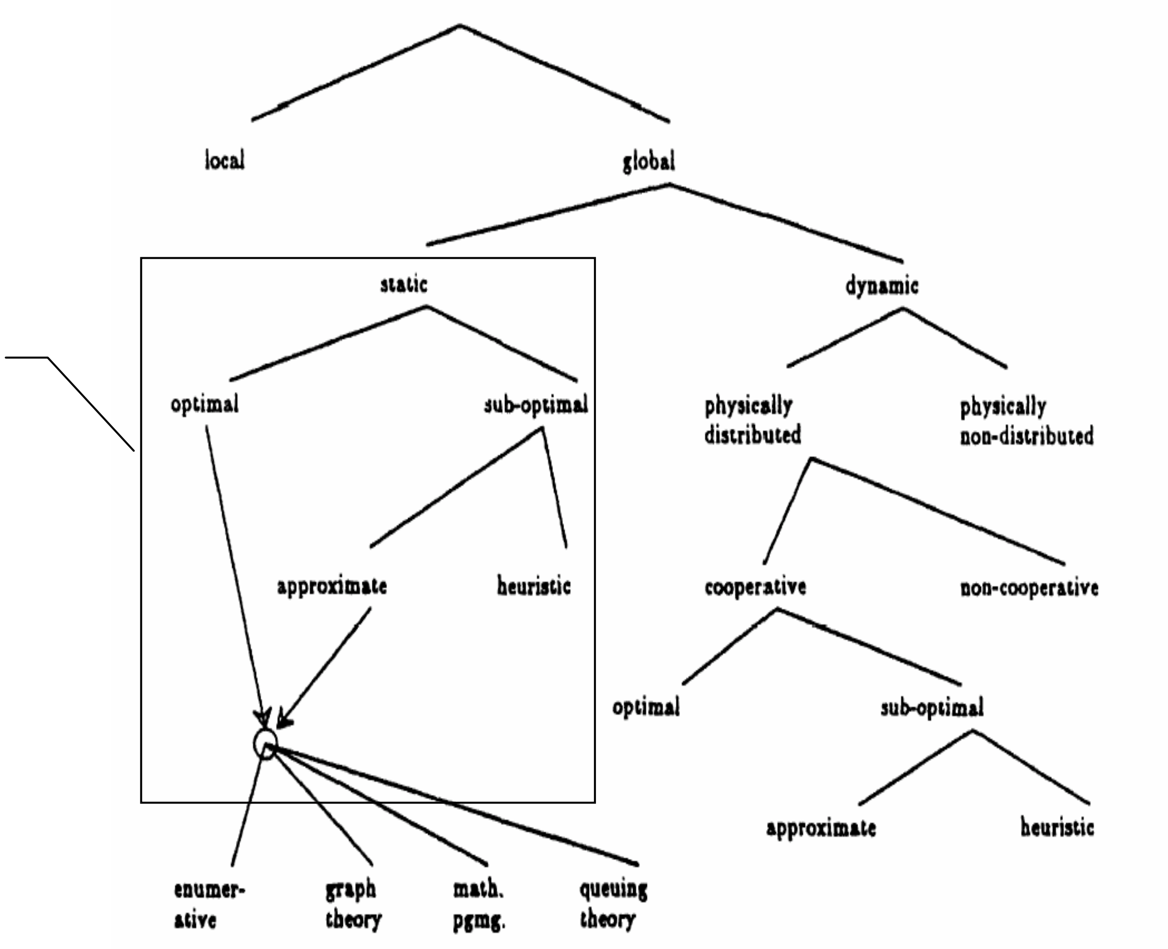
A taxonomy of (general purpose) scheduling approaches

- rely on rule-of-thumbs that “should” guide the selection process towards a near optimal solution, e.g.:
  - optimize the critical-path
  - cluster heavy communica-ting processes
  - …

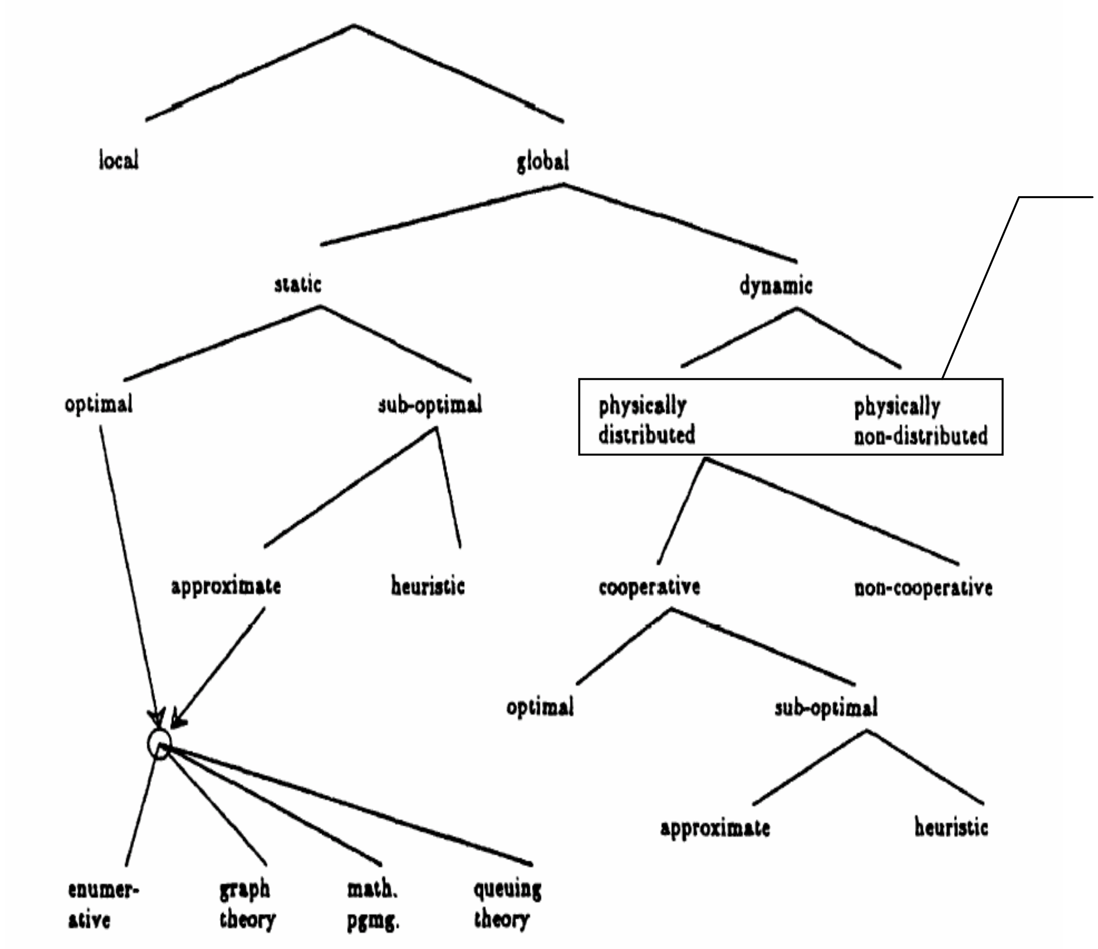


A taxonomy of (general purpose) scheduling approaches

- won’t be further addressed during this course…
- typically studied in the IA context

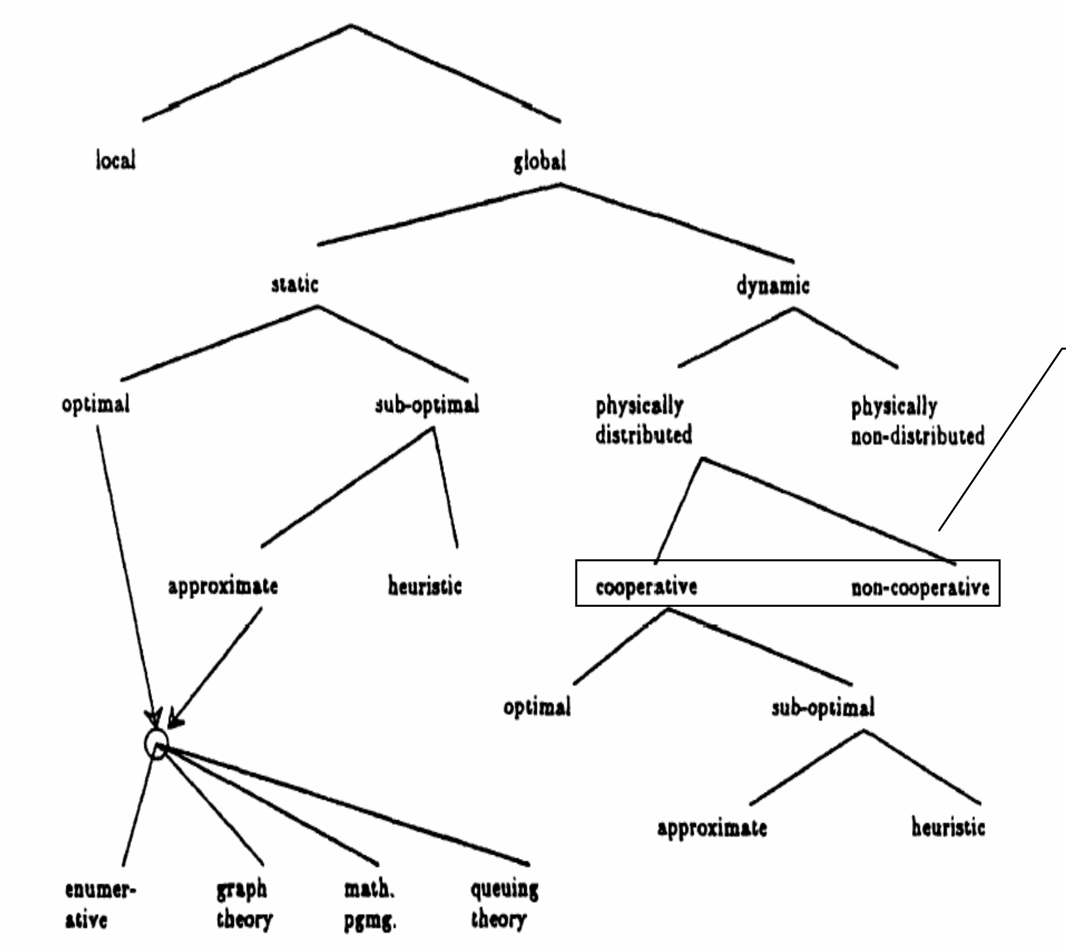


A taxonomy of (general purpose) scheduling approaches



•is the scheduling functionality residing on a single node or is distributed among the proces-sors?

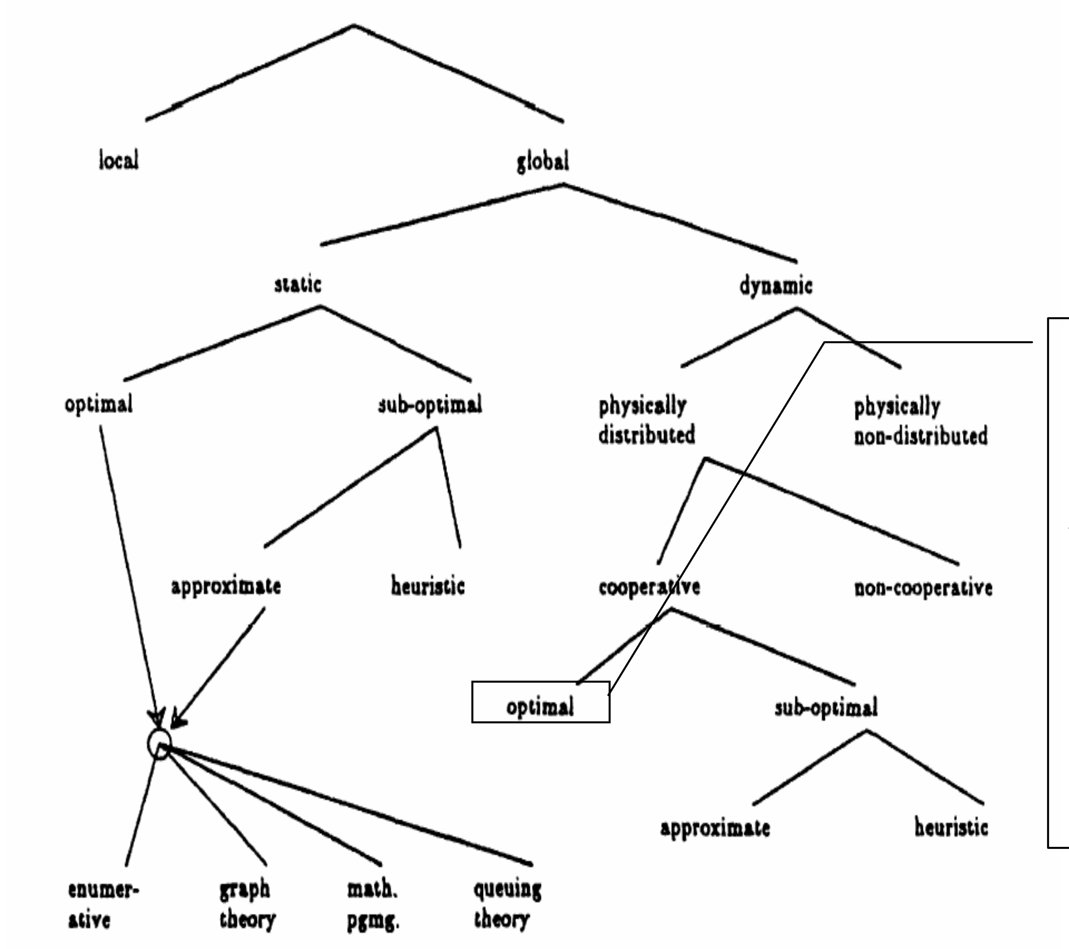
A taxonomy of (general purpose) scheduling approaches



- •choice clearly affects autonomy of the processors in the scheduling decisions
- •cooperation has an inherent cost:

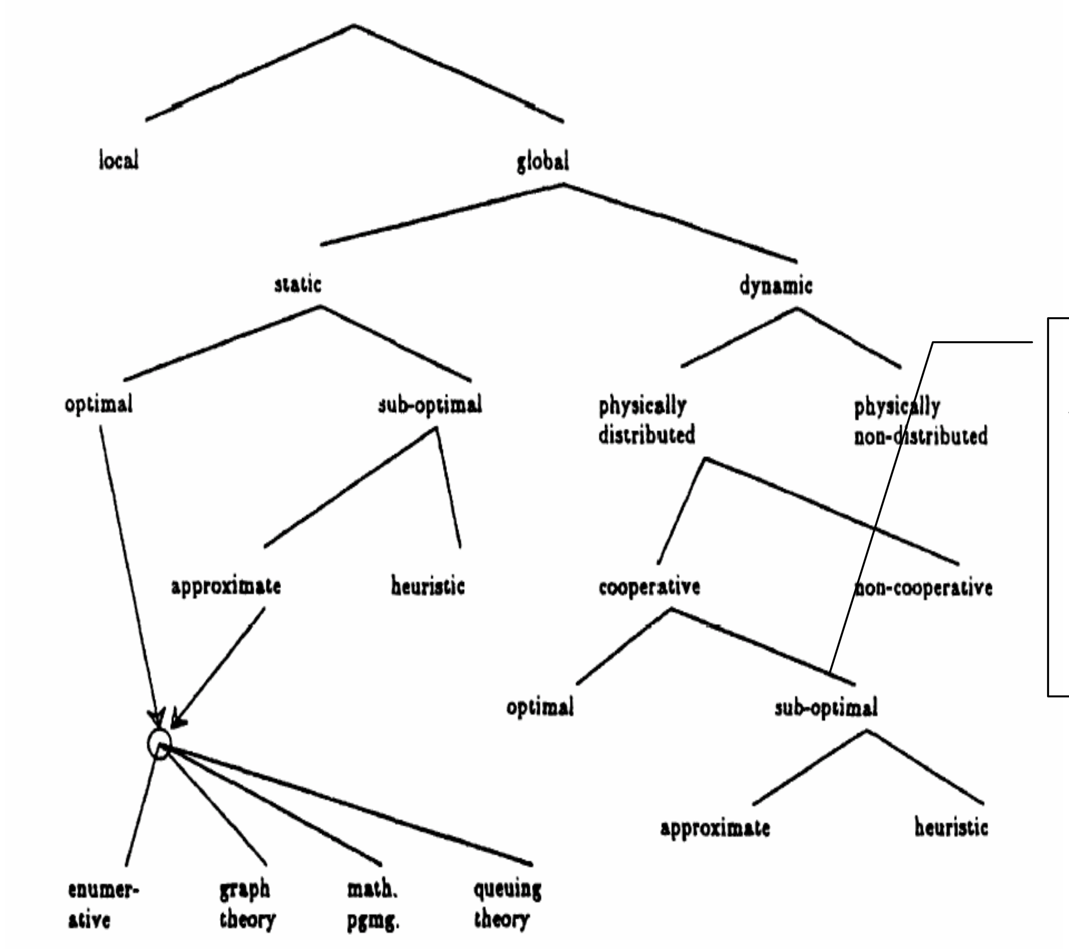
network
+
processing
OVERHEADS

A taxonomy of (general purpose) scheduling approaches



- •very rare solutions due to high complexity of dynamic scheduling
- •typically restrict the dynamicity of the system, e.g.:
  - •known number of processors and process resource demands
  - •unknown number of process instances
- •…and still not tractable.

A taxonomy of (general purpose) scheduling approaches



- better performing and simpler: are the only ones to be empoyed in practice
- we’ll ovierview and evaluate the fundamental classes of the proposed approaches

Orthogonal Classification Attributes

- adaptive *vs* non adaptive
- load balancing *vs* load sharing
- sender- *vs* receiver- *vs* symmetrically-initiated
- preemptive *vs* non preemptive

adaptive *vs* non adaptive

- the algorithms and parameters used to implement the scheduling policy change dynamically adapting to previous and current system behavior:
  - change of the considered system parameters
  - enabling/disabling of algorithm subfunctions
  - history based choice of the most appropriate scheduler (distributed) algorithm

load balancing *vs* load sharing

- The goal of a *load-sharing algorithm* is to maximize the rate at which a distributed system performs work when work is available.

- The goal of a *load-balancing algorithm* is to equalize the loads at all resources.

- Both strive to avoid *unshared states:* not trivial due to the time required to migrate tasks, communication latencies…

- *load-balancing algorithms* can potentially reduce the mean and standard deviation of task response times wrt *load-sharing* ones, but typically impose higher communication overheads.

sender- *vs* receiver- *vs* symmetrically- initiated

- In a sender-initiated scheduling algorithm, load-distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver).

- Viceversa, in the receiver-initiated case, a node that goes idle tries to get tasks from overloaded nodes

- In symmetrically-initiated ones, both idle and busy nodes activate the load distribution mechanisms.

preemptive *vs* non preemptive

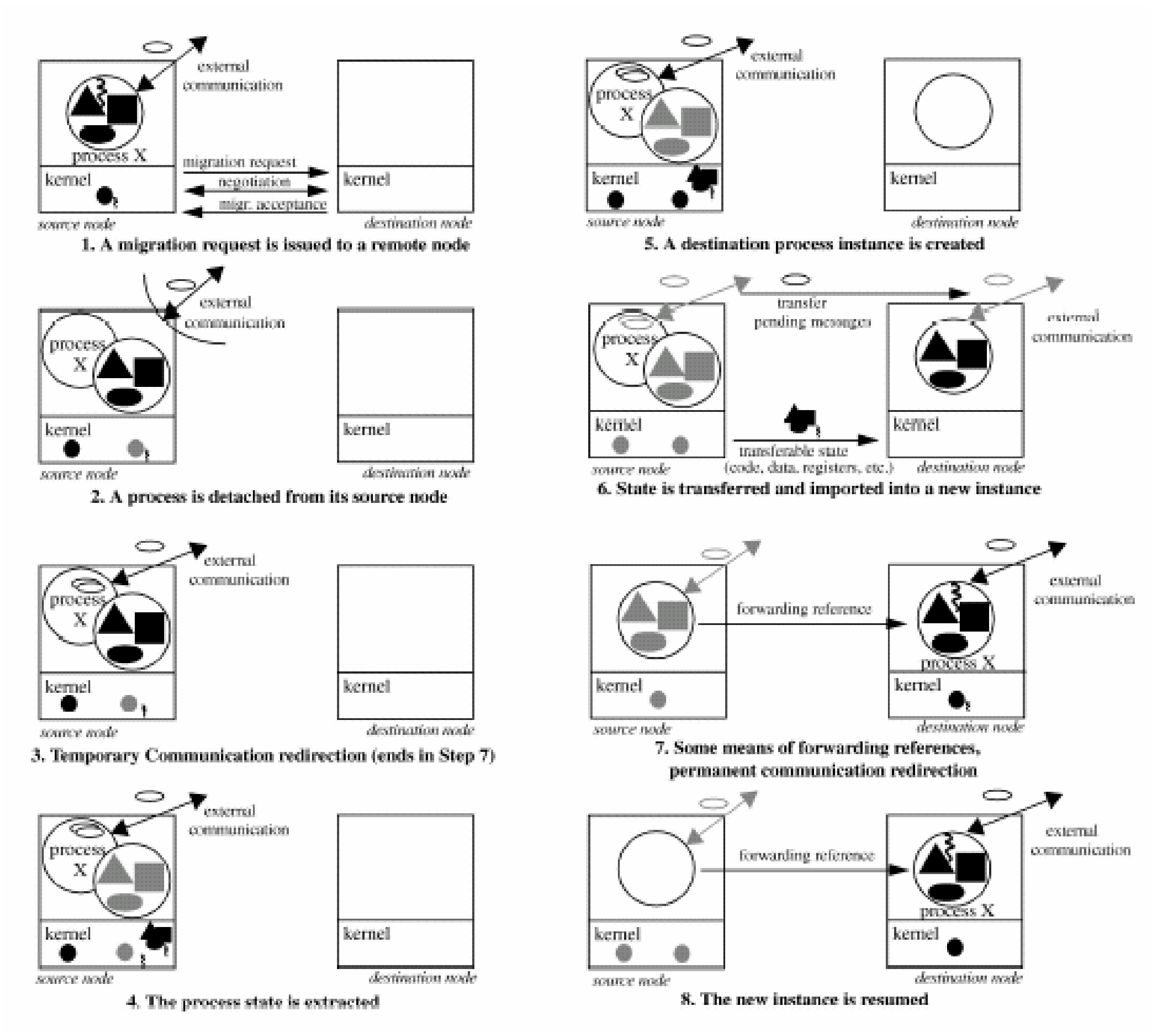
- *Preemptive* schedulers require transferring a partially executed tasks on a different processor.

- *Non preemptive schedulers* involve only tasks that have not begun execution and hence do not require transferring the task's state

- Both policies require that information about the environment in which the task will execute is transferred to the remote node,…

- …but migrating an executing process is far more complex:
  - (generally) expensive, since process state can be quite large:
    - virtual memory image, process control block, file pointers, timers…
  - difficult, requires non conventional kernels and raises additional difficulties (shared memory?)
  - not always possible, e.g. migrated process may not have access to the same devices

Process migration



1. A migration request is issued to a remote node

2. A process is detached from its source node

3. Temporary Communication redirection (ends in Step 7)

4. The process state is extracted

5. A destination process instance is created

6. State is transferred and imported into a new instance

7. Some means of forwarding references, permanent communication redirection

8. The new instance is resumed

Components of a load distributing algorithm

- *transfer policy:*
  - determines if a node is suitable for partipicating in a task transfer as a sender (overloaded) or a receiver (underloaded)
  - based on,e.g, local load thresholds, relative unbalances wrt other nodes…
- *selection policy:*
  - if the node is a suitable sender, which task should be sent? Preferably:
    - not yet started tasks (to avoid preemption costs)
    - small, long-lived, tasks, having minimal location-dependence

Components of a load distributing algorithm

- *location policy:*
  - Who’s the ideal transfer “partner”?
    - Distributed approach:
      - (serial vs multi-cast vs broad-cast) polling
      - purely random (it’s very light-weight!)
      - history based
      - nearest neighbors
    - Centralized approach:
      - load state collection and coordination role at a single node
      - single point of failure and possible bottleneck

Components of a load distributing algorithm

- *information policy:*
  - when, what and from where is information about the states of other nodes in the system to be collected?
  - choice of representative load indexes
  - load measurement mechanisms with clear trade-off accurateness vs cost. Three classes:
    - Demand-driven
      - distributed scheme, when a node becomes a sender or receiver, it collects the state of other nodes
      - Sender-, receiver- or symmetrically-iniated
    - Periodic:
      - Both centralized and distributed
    - State-change-driven:
      - Both centralized and distributed
      - Upon certain state changes, a node disseminates its state information

Case Study:
Workstations clusters

System model

- Fixed set of interconnected general-purpose workstations (nodes)
- Nodes have a-priori knowledge about each other existence
- Fully decentralized task scheduling:
  - each node takes part in the scheduling process,
  - no central coordinator
- Tasks are generated at any node and there’s no a priori knowledge on the workload

Sender-initiated Algorithms (i)

Basic Idea:

An overloaded node (sender) tries to send a task to an underloaded node (receiver)

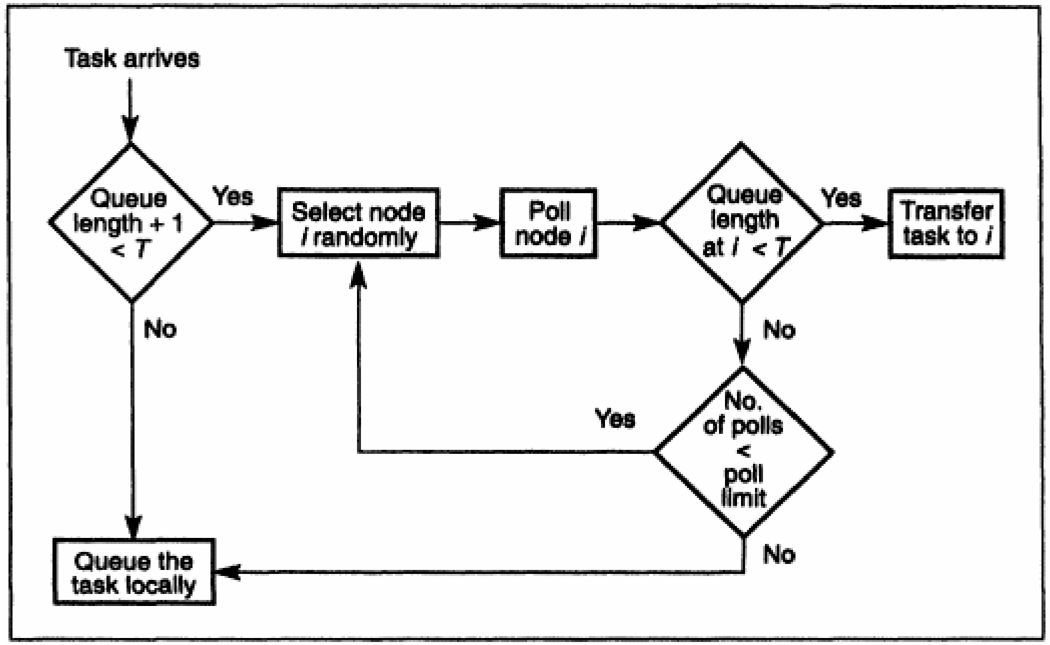
- *Information policy:*
  - Demand-driven: upon identification of a node as a sender, load information from other nodes is collected
  - Among the possible load indexes, we'll consider only CPU queue length
- *Transfer policy*:
  - Threshold based:
    - A sender is a node originating a task that makes its local CPU queue length exceed a threshold T.
    - A receiver is a node that, by accepting a given task, would not exceed T.
- *Selection policy*:
  - For simplicity let's consider only newly started processes =>not requiring preemption

Sender-initiated Algorithms (ii)

- *Location policy:*
  1. Random:
     - No information exchange
     - Useless transfer are possible: if an overloaded node is chosen as receiver, re-transfer to an other random node (for a limited number of times to avoid trashing)
     - Despite its simplicity we’ll see that it’s still better than nothing…

Sender-initiated Algorithms (iii)

- *Location policy:*
  2. Threshold:



•Nodes are sequentially polled in random order.

•Note that by the time the task is transferred the queue length may have become > G…

•…in this case, accept the job anyway – or we may incur trashing!

Sender-initiated Algorithms (iii)

- *Location policy:*
  3. Shortest:
     - A random subset of nodes is polled and the one with the shortest queue length is chosen:
       - » aims at choosing the best receiver node rather than the first suitable one.
     - Quite surprisingly, in practice, despite the additional complexity, it provides only marginal performance improvement:

“LESSON”

MORE DETAILED STATE INFORMATION
DOES NOT NECESSARILY
IMPROVE SYSTEM PERFORMANCE

Sender-initiated Algorithms (iv)

- *A note on the stability of these algorithms:*
  - Can the proposed algorithms make the system unstable in case of heavy load?
  - In high load scenarios there won’t (likely) be receivers!
  - But polling would still keep on, triggered more and more frequently as task arrival rate increases….
  - the algorithm overhead can become higher and higher, eventually overcoming the service capacity of the system, causing system instability.

Receiver-initiated Algorithms (i)

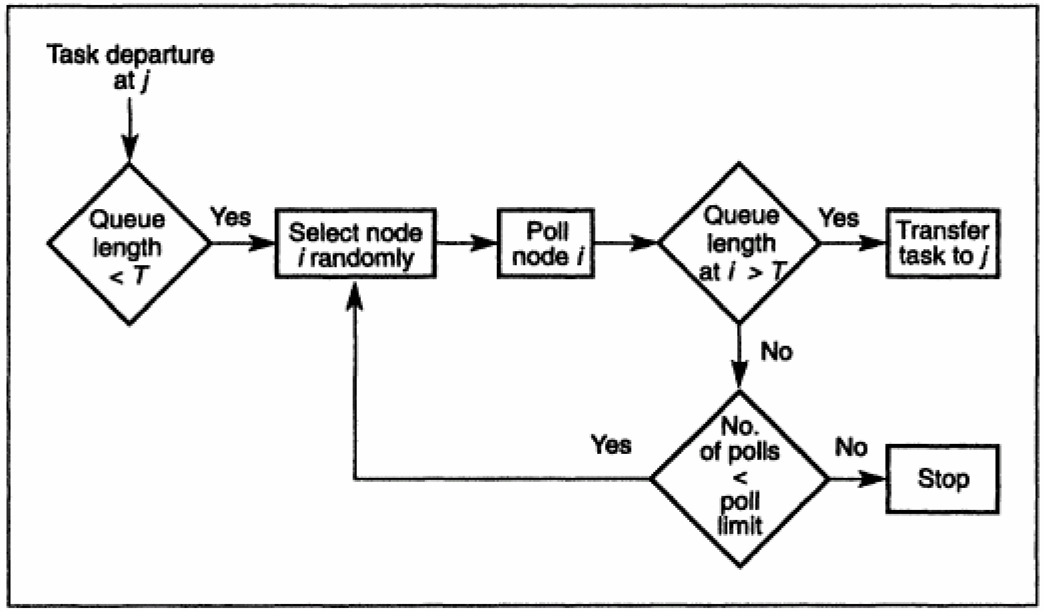
Basic Idea:

An underloaded node (sender) tries to get a task to an overloaded node (receiver)

- *Information policy:*
  - Demand-driven
- *Transfer policy*:
  - Threshold based
- *Selection policy*:
  - In this case let's consider also preemption based approaches, since task migration is asynchronous wrt task generation as we'll see….

Receiver-initiated Algorithms (ii)

- *Location policy*:
  - Threshold based



•<u>Problem</u>: if no sender is found, the receiver resources are not employed till next local task creation:

•Not good in case of heterogenously loaded nodes, where polls may even miss loaded nodes!!

•<u>Remedy</u>: poll anyway after a given timeout

Receiver-initiated Algorithms (iii)

- *A drawback*:
  - In receiver-initiated algorithms, the polling starts when a node becomes a receiver.
  - However, these polls seldom arrive at senders just after new tasks have arrived at the senders but before these tasks have begun executing:
    - after creation but before assignment of first quantum service.
  - Most task transfers are preemptive!
  - Possible work-around:
    - resource reservation for the next arriving task at a sender.
    - poor performances due to increased resource under-utilization…

Symmetrically initiated algorithms

- both senders and receivers initiate load-distributing activities for task transfers.
- they come with both pros and cons of the two algorithms families:
  - Pros:
    - At low loads, senders will likely find receivers
    - At high loads, receivers will likely find senders
  - Cons:
    - As in sender-initiated strategy, polling at high system loads may compromise stability
    - As in receiver-initiated strategy, Preemption is needed.

Adaptive algorithms

A stable symmetrically-initiated adaptive algorithm

Observation:

- Indiscriminate polling by the sender's negotiation component can lead to instability.

Basic idea:

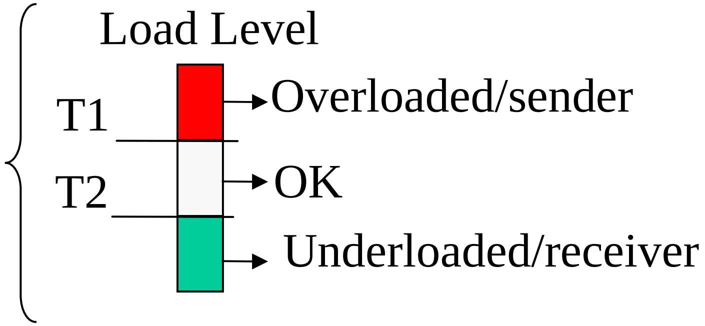
- Make nodes keep track of relative load states and, during high loads, stop uselessly polling nodes that are known to be overloaded

A stable symmetrically-initiated adaptive algorithm (i)

Each node keeps a list reflecting its knowledge of the other nodes load state. Three possible states:

Overloaded / Underloaded / OK

- *Information policy:*Demand-driven
- *Selection policy:*any
- *Transfer policy:*
  - Based on two thresholds: T1, T2



- *Location policy:*
  - Two behaviors according to the process state:
    - Sender vs Receiver

A stable symmetrically-initiated adaptive algorithm (ii)

Initially all nodes are set in the receivers’ list.

Sender-iniated behavior:

- The sender polls the receivers’ list head.
- The receiver puts the polling node at the sender list head, and sends out his current state
- If the polled nodes says it’s a receiver, the send transfers the job. Else, remove the node from the receiver’s list, insert in the sender/OK list head, and poll an other receiver until:
  1. A receiver is found
  2. No receiver is found after a given number of tries
  3. The receiver list gets empty.

- If no receiver is found the task is locally executed (but later may be still possibly preemptively transferred)

A stable symmetrically-initiated adaptive algorithm (iii)

Receiver-iniated behavior:

- Receiver polls nodes to obtain tasks in the following order:
  - Head to tail in the senders list, most accurate info first
  - Tail to head in the ok list, after a longer time it's more likely it has become a sender
  - Tail to head in the receiver list, idem
- If the polled node is a sender it transfer a task and informs the counter-part of its state after the transfer; else it sets the polling node in the receiver list head.
- Upon receipt of the poll reply, the polled node is inserted in the head of the list reflecting its load state.
- Polling stops in case:
  1. A sender is found
  2. No sender is found after a given number of tries

- If no receiver is found the task is locally executed (but later may be still possibly preemptively transferred)

A stable symmetrically-initiated adaptive algorithm (iv)

At high loads:

- There will be no (or few) idle nodes:
  - Likely failure of sender-initiated pollings => cleaning of receivers list => no more sender-initiated pollings
  - Only receiver-initiated load-sharing activities are carried on

At low loads:

- There will be no (or few) overutilized nodes:
  - Likely failure of received-initiated pollings => wastage of resources due to polling is not an issue as extra processing capacity is available at low loads
  - Received-initiated pollings by updating receivers list increase probability of successfully completing a sender-iniated polling

A sender-initiated adaptive algorithm (i)

Observation:

Previous solution requires task preemption, which may not always be feasible or desirable.

Basic Idea:

Avoid receiver-initiated task tranfers, while still preventing unneeded sender-initiated pollings at high loads.

A sender-initiated adaptive algorithm (ii)

- Each node maintains a list to keep track of which list it belongs to (OK, senders, receivers) at any other node.

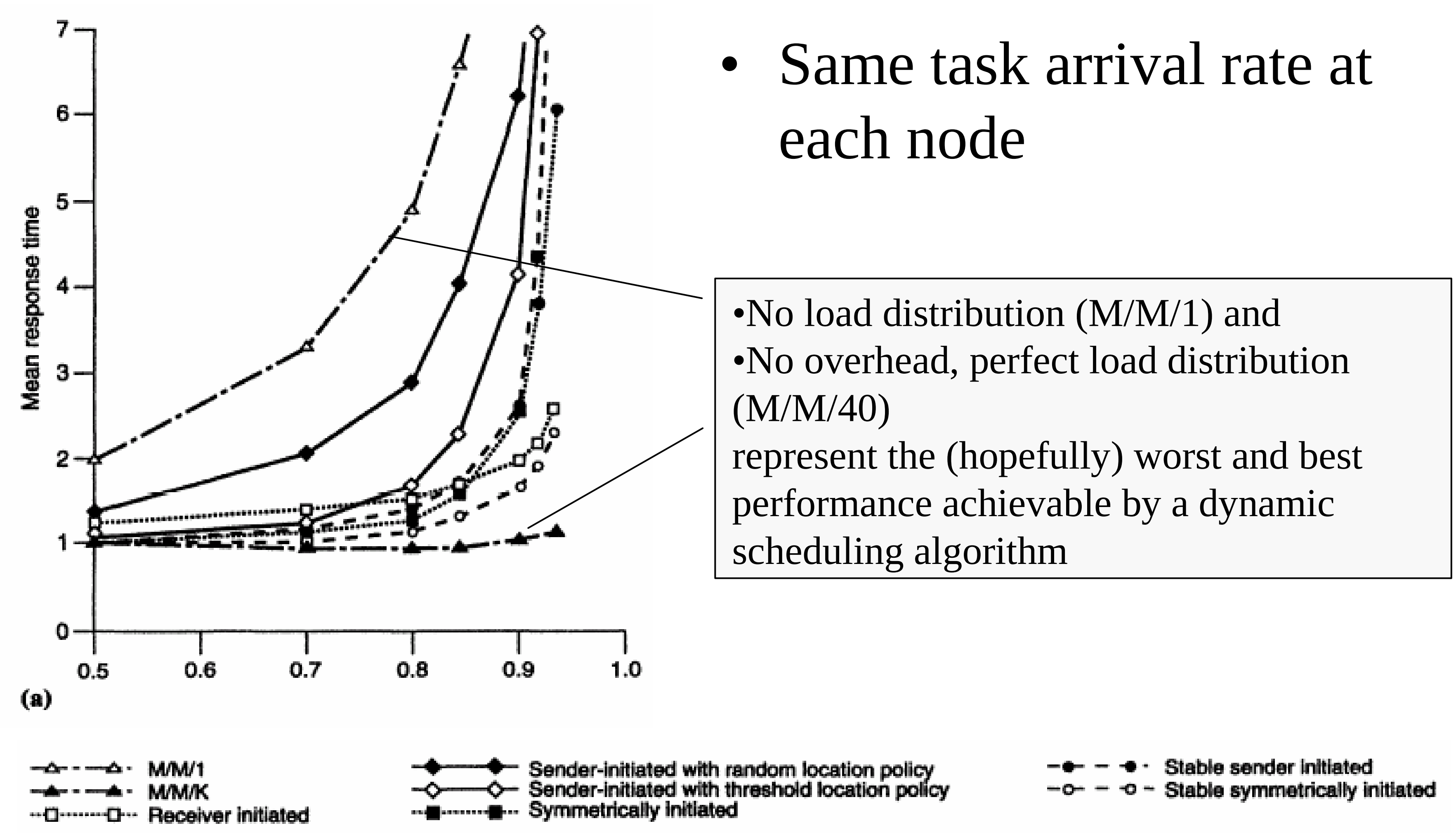
- When a sender polls a node, the latter updates its local list, putting the sender node in the sender list. The same does the polling node.

- During receiver-initiated protocol only the nodes that are mis-informed about the receiver load state are informed. No task transfer takes places at this phase.

Performance Evaluation

- System settings:
  - 40 processors on local network.
  - both preemptive and non preemptive task selection policies were considered
  - comparison with:
    - a system composed of 40 independent machines with no load distribution(M/M/1)
    - an ideal system of 40 machines performing perfect load distributing with no overhead (M/M/K)
  - Single Threshold
  - Small fixed poll limit =5

Performance Evaluation: Homogenous workload

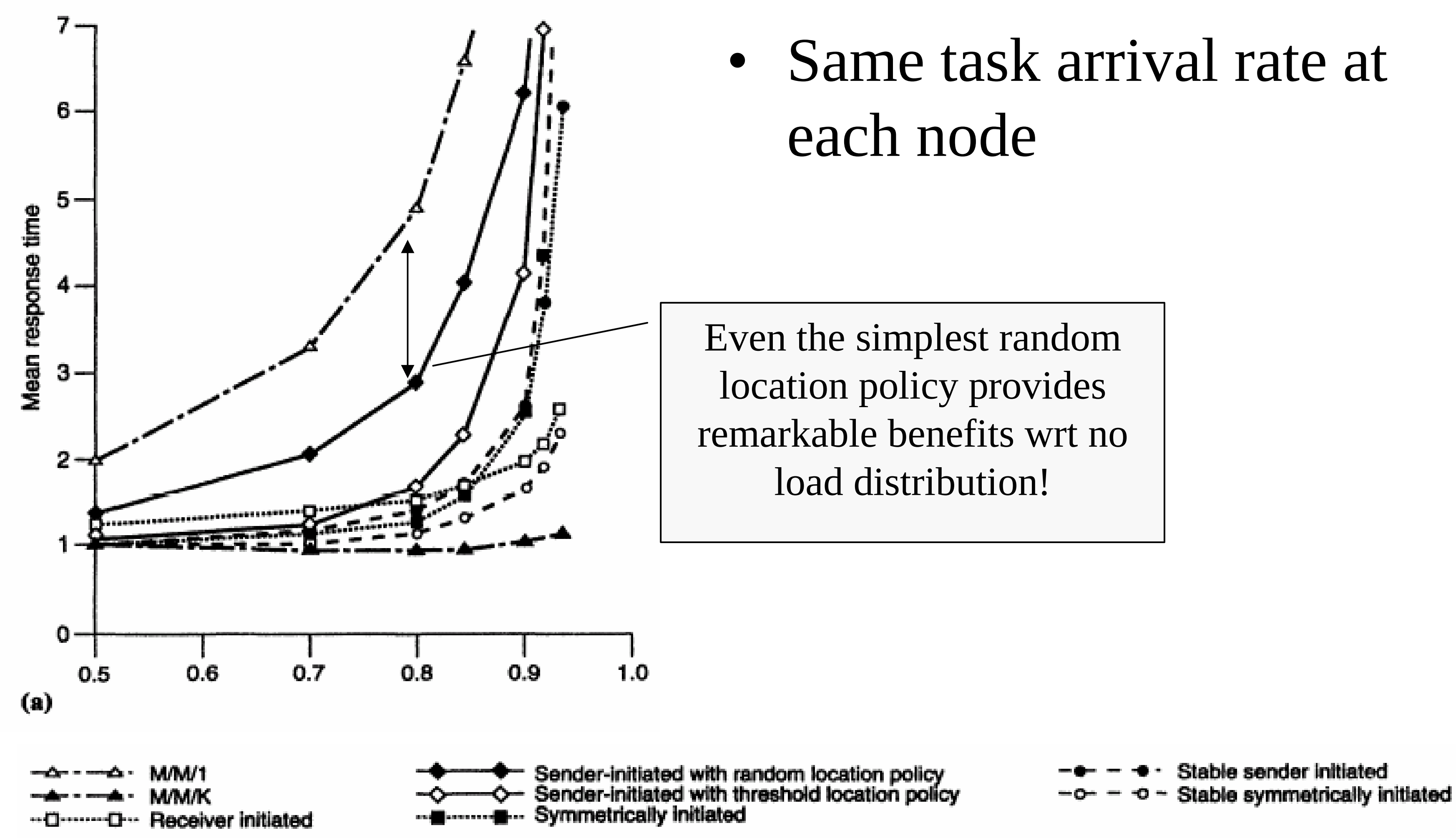


- Same task arrival rate at each node

•No load distribution (M/M/1) and
•No overhead, perfect load distribution (M/M/40)
represent the (hopefully) worst and best performance achievable by a dynamic scheduling algorithm

Performance Evaluation: Homogenous workload

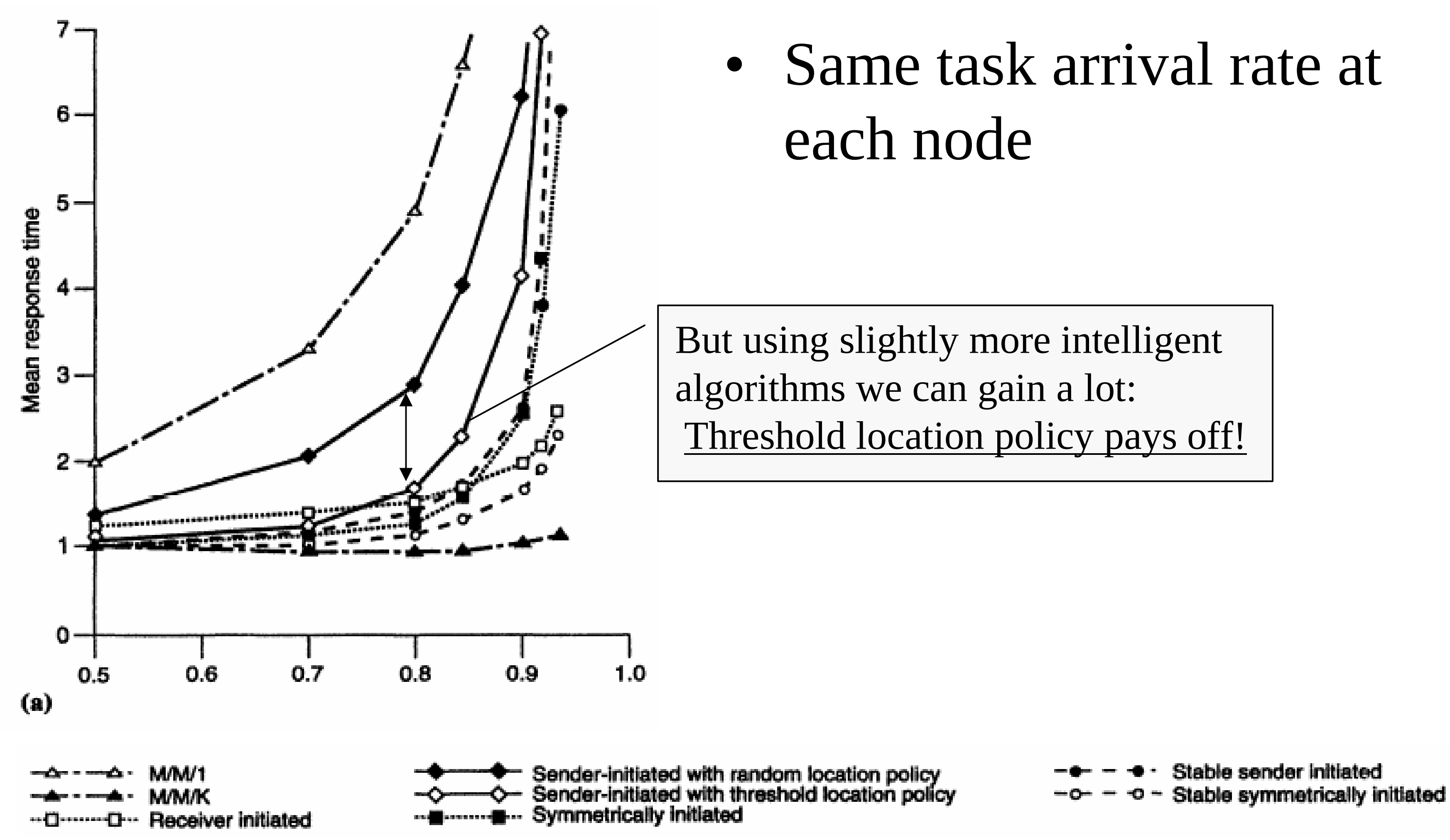
- Same task arrival rate at each node



Even the simplest random location policy provides remarkable benefits wrt no load distribution!

Performance Evaluation: Homogenous workload

- Same task arrival rate at each node

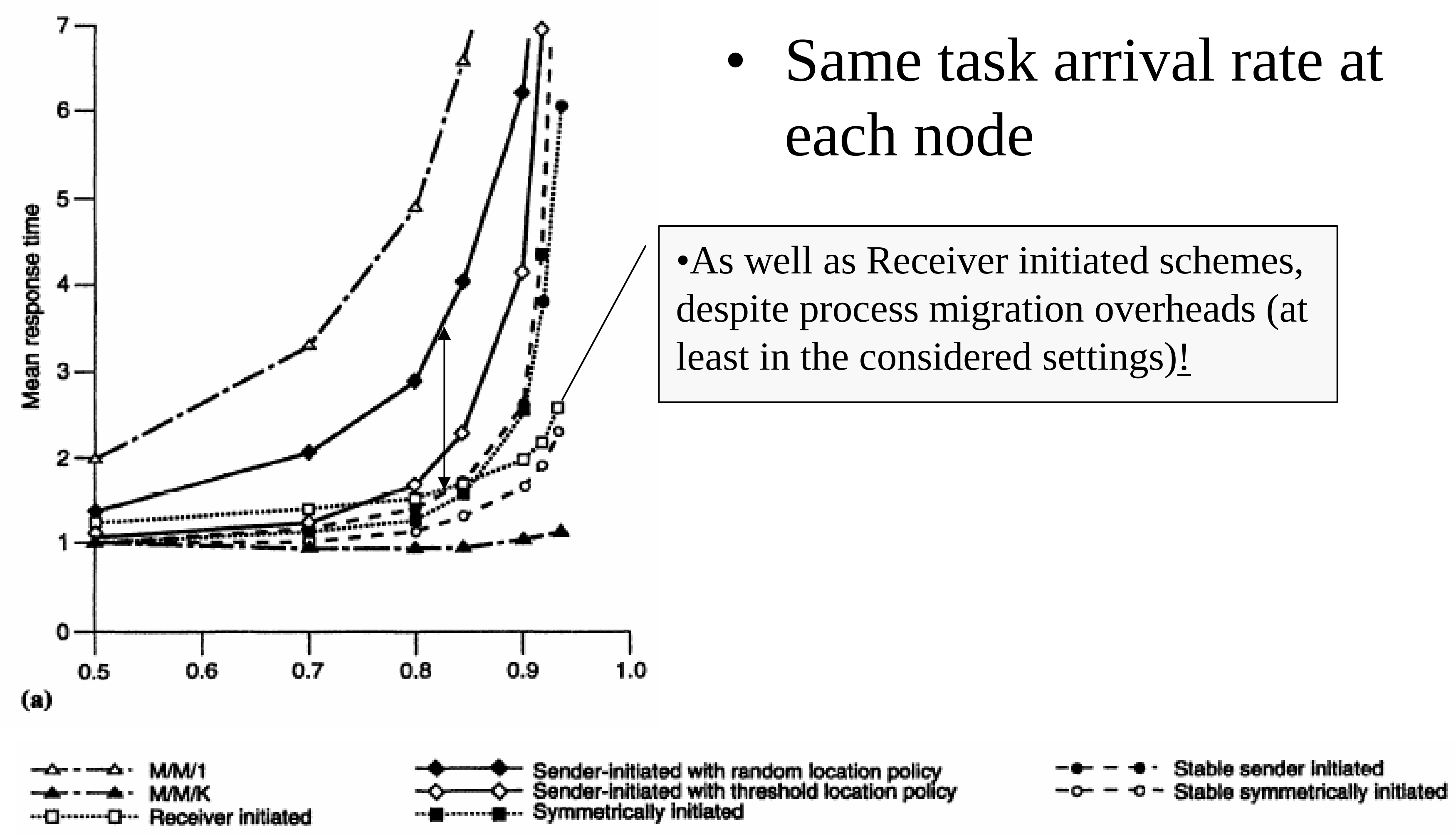


But using slightly more intelligent algorithms we can gain a lot:
Threshold location policy pays off!

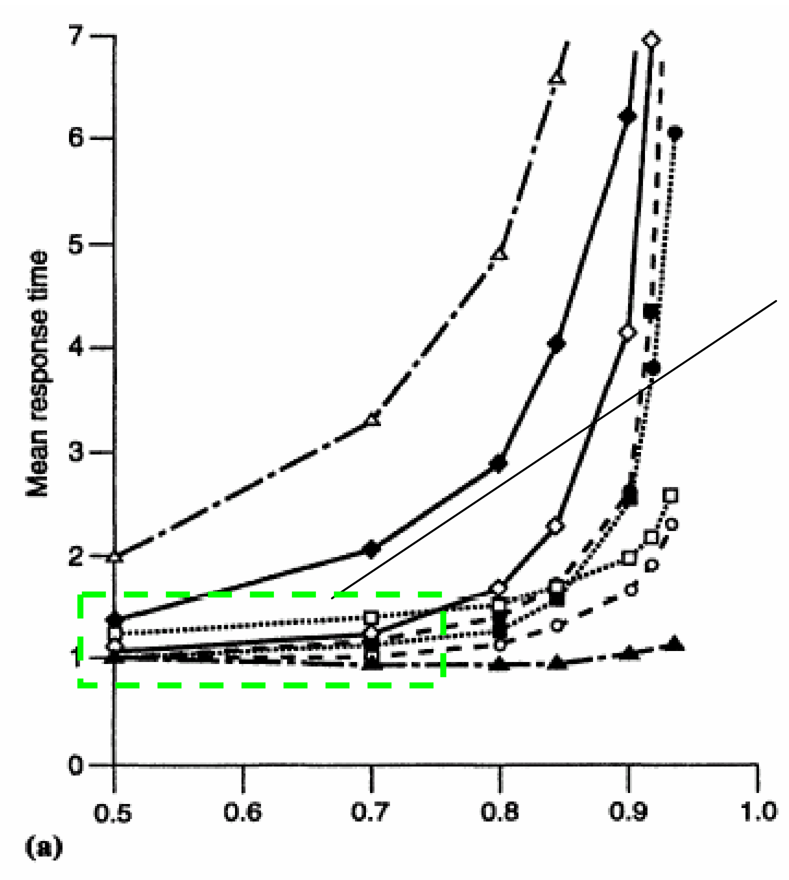
Performance Evaluation: Homogenous workload

- Same task arrival rate at each node

•As well as Receiver initiated schemes, despite process migration overheads (at least in the considered settings)!



Performance Evaluation: Homogenous workload



- Same task arrival rate at each node

Sender-initiated vs Receiver-initiated

LOW-LOAD

•At low loads sender-initiated rapidly finds receivers, whilst receiver-initiated is unlikely to find senders!

Performance Evaluation: Homogenous workload

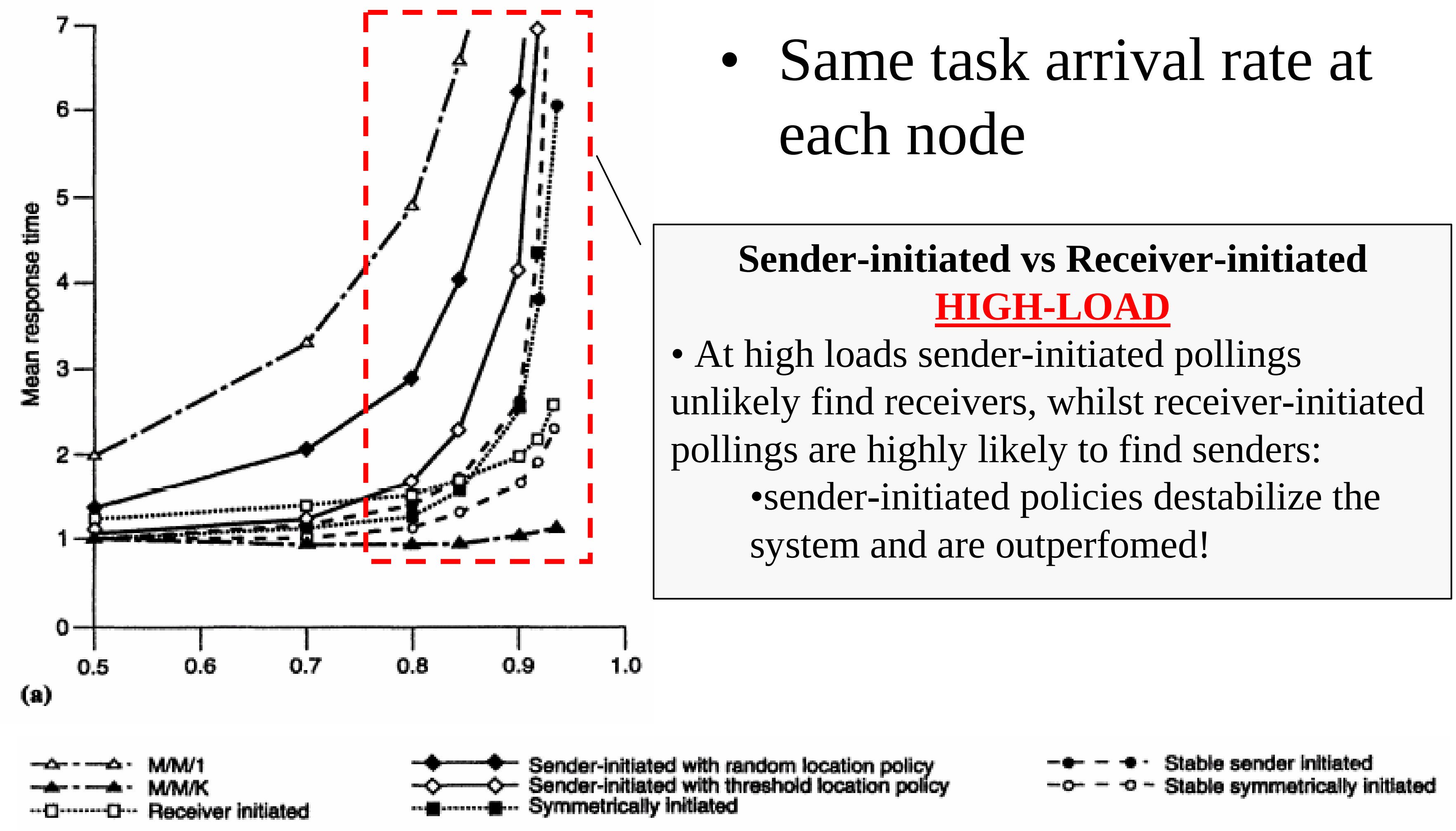
- Same task arrival rate at each node

Sender-initiated vs Receiver-initiated

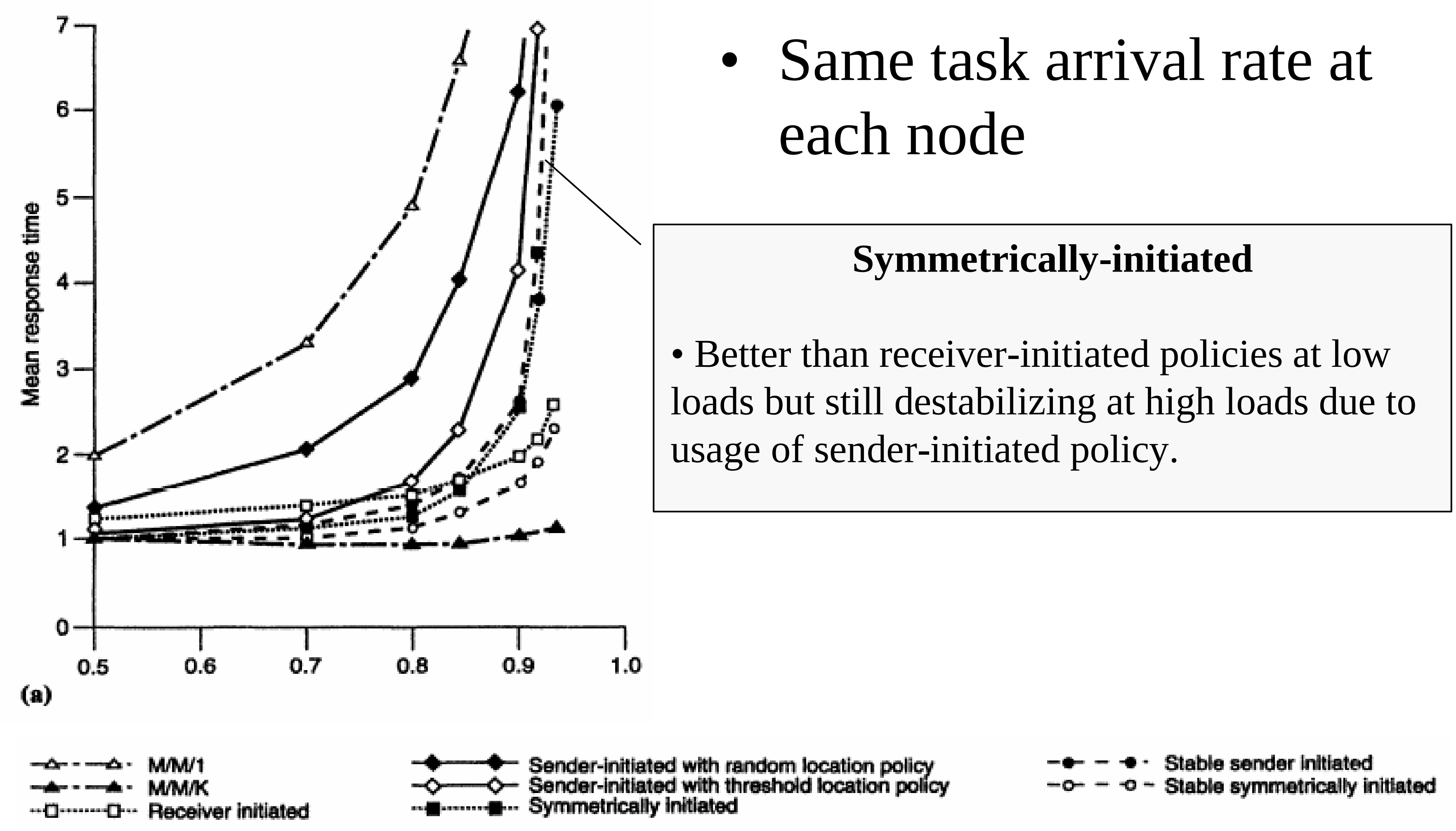
HIGH-LOAD

• At high loads sender-initiated pollings unlikely find receivers, whilst receiver-initiated pollings are highly likely to find senders:

•sender-initiated policies destabilize the system and are outperfomed!



Performance Evaluation: Homogenous workload

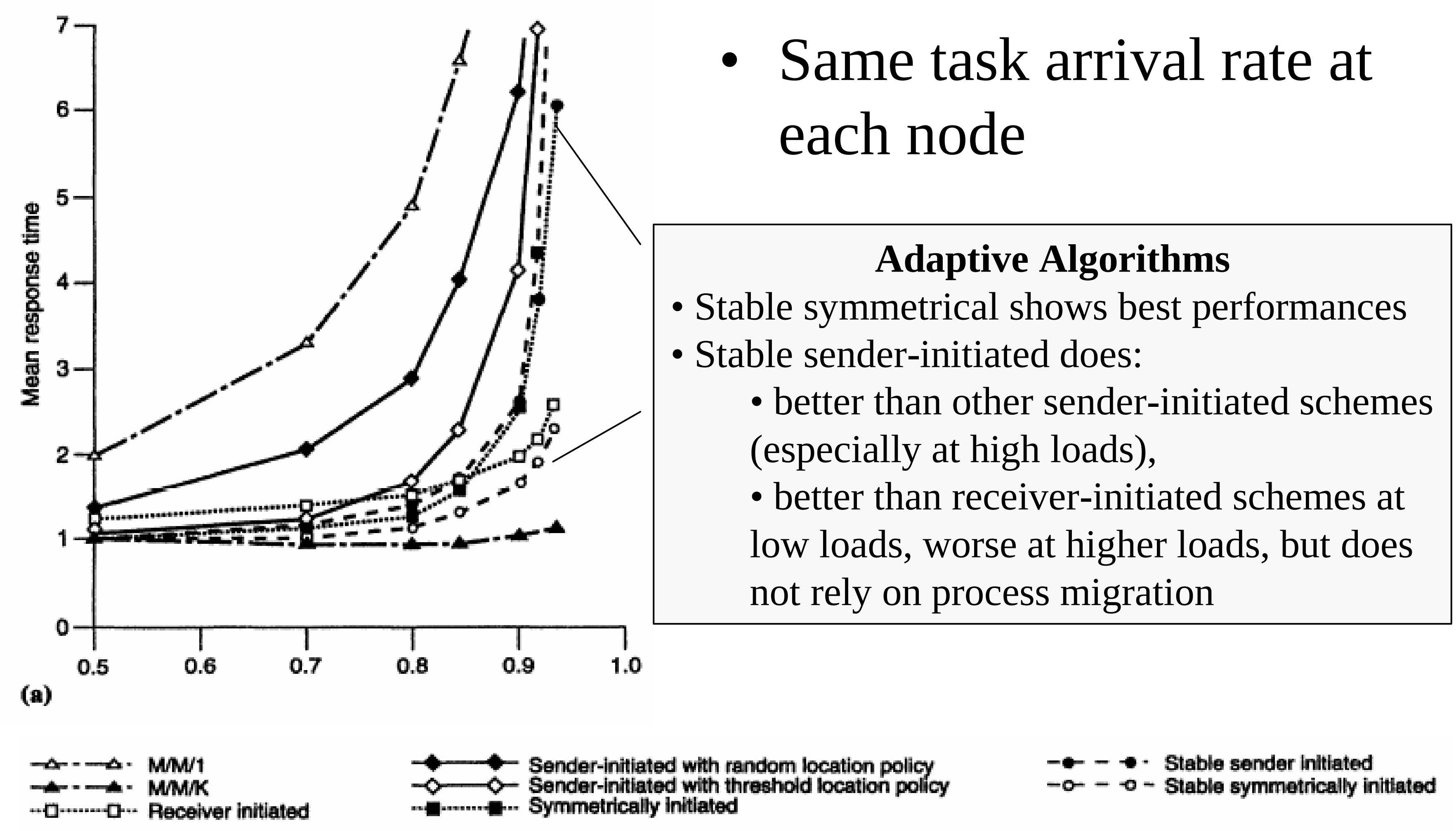


- Same task arrival rate at each node

Symmetrically-initiated

• Better than receiver-initiated policies at low loads but still destabilizing at high loads due to usage of sender-initiated policy.

Performance Evaluation: Homogenous workload

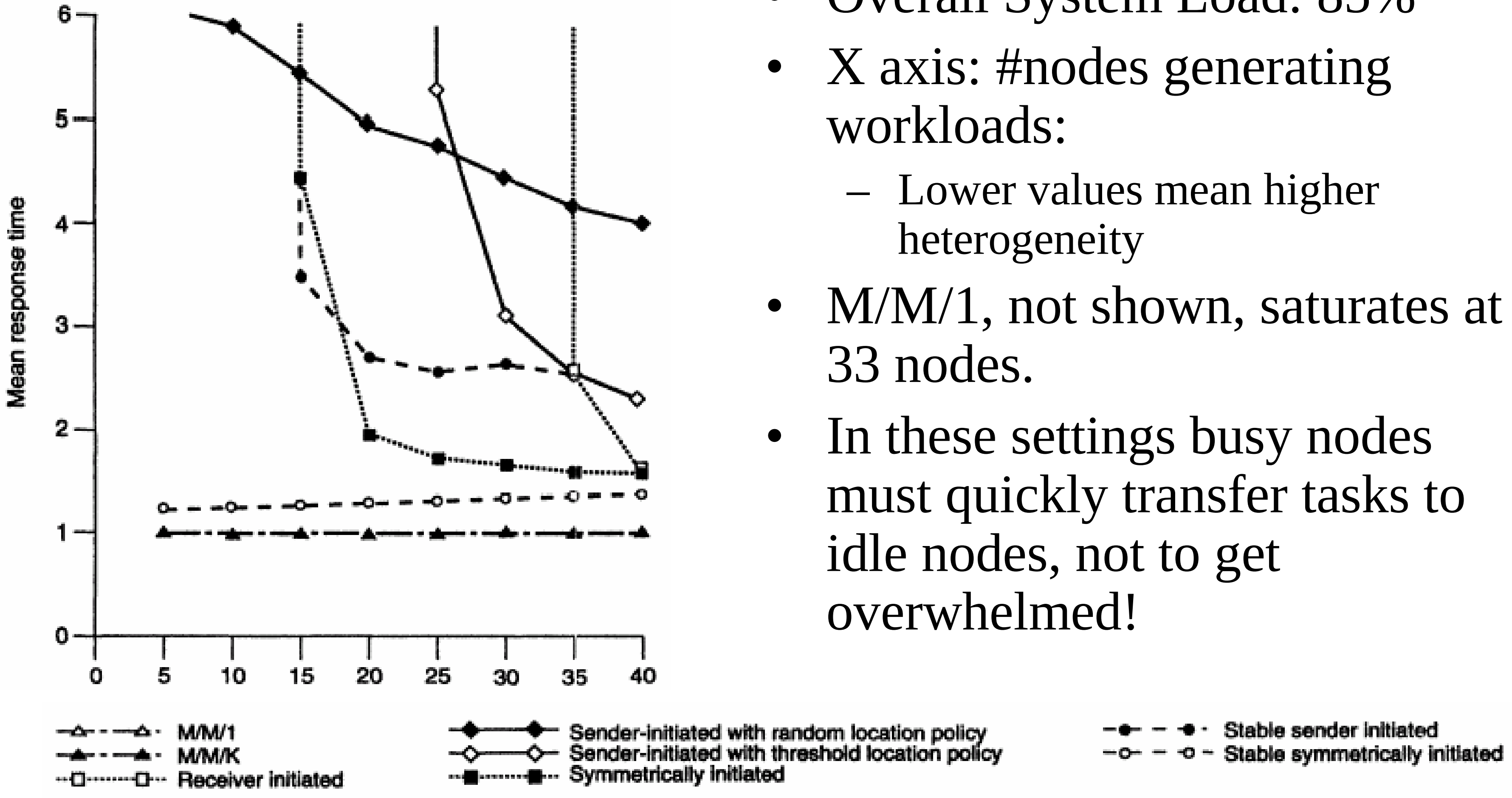


- Same task arrival rate at each node

Adaptive Algorithms

- Stable symmetrical shows best performances
- Stable sender-initiated does:
  - better than other sender-initiated schemes (especially at high loads),
  - better than receiver-initiated schemes at low loads, worse at higher loads, but does not rely on process migration

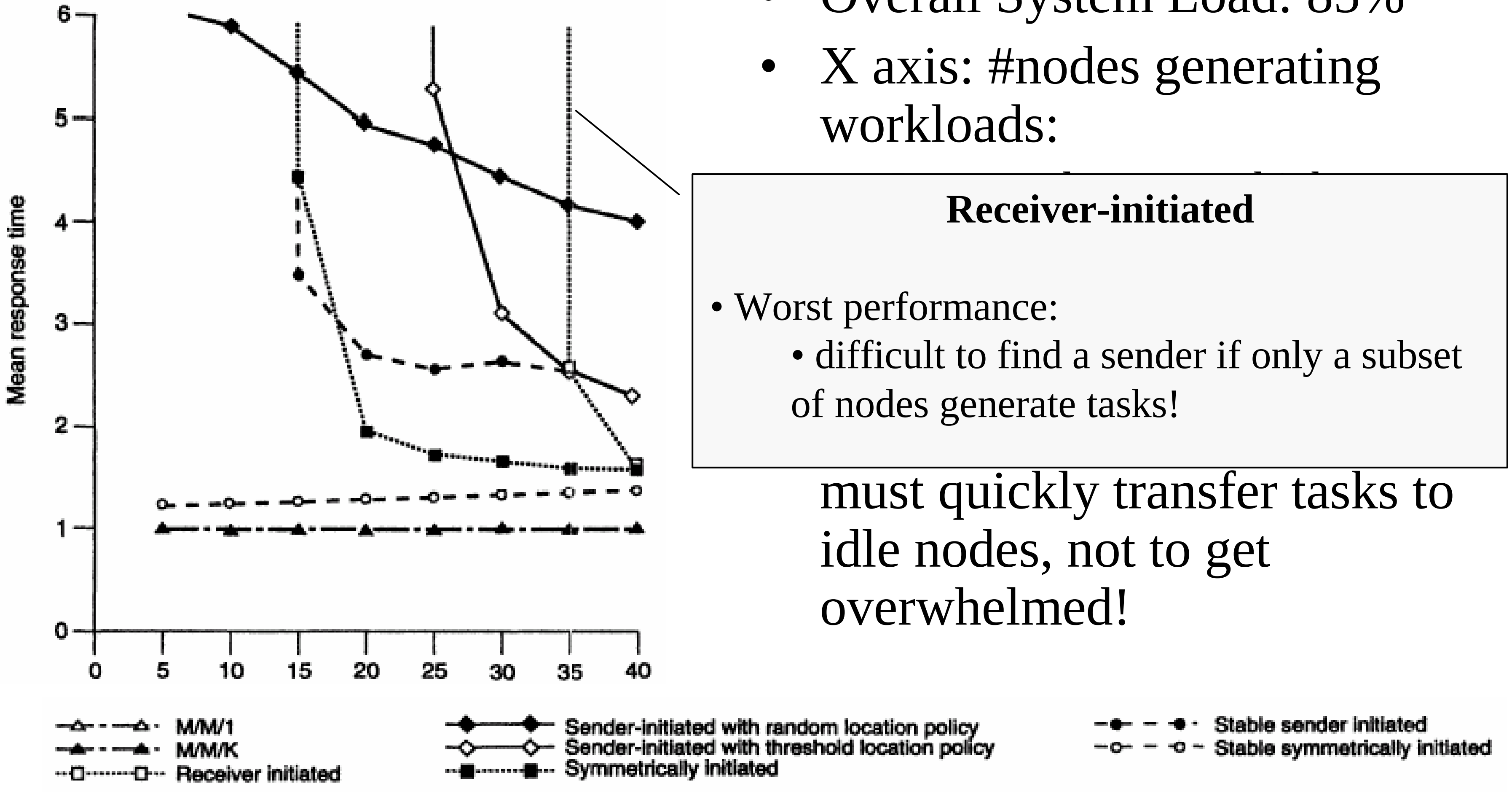
Performance Evaluation: Heterogeneous workload



- Overall System Load: 85%
- X axis: #nodes generating workloads:
  - Lower values mean higher heterogeneity
- M/M/1, not shown, saturates at 33 nodes.
- In these settings busy nodes must quickly transfer tasks to idle nodes, not to get overwhelmed!

Performance Evaluation: Heterogeneous workload

- Overall System Load: 85%
- X axis: #nodes generating workloads:

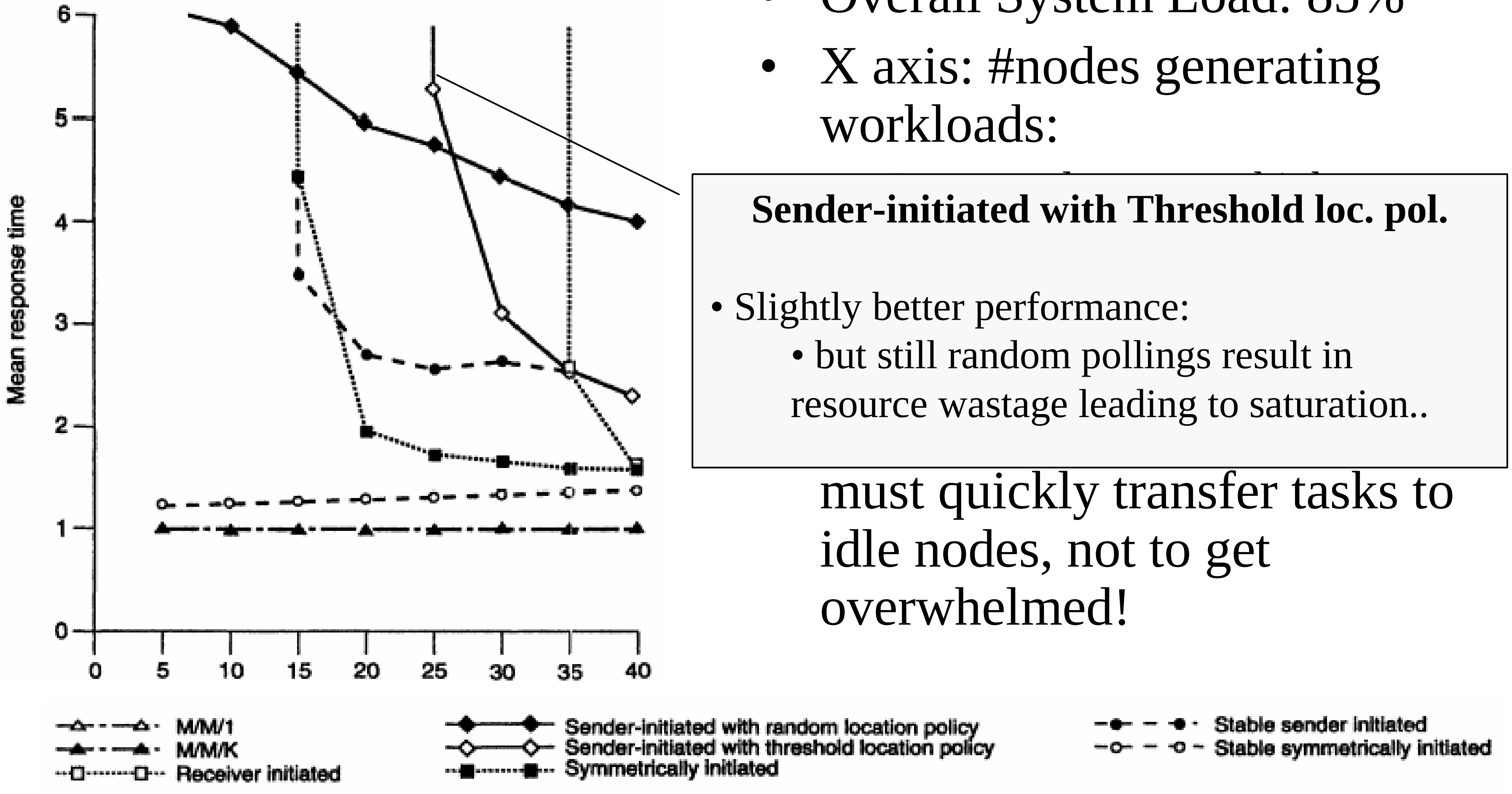


Receiver-initiated

- Worst performance:
  - difficult to find a sender if only a subset of nodes generate tasks!

must quickly transfer tasks to idle nodes, not to get overwhelmed!

Performance Evaluation: Heterogeneous workload



- Overall System Load: 85%
- X axis: #nodes generating workloads:

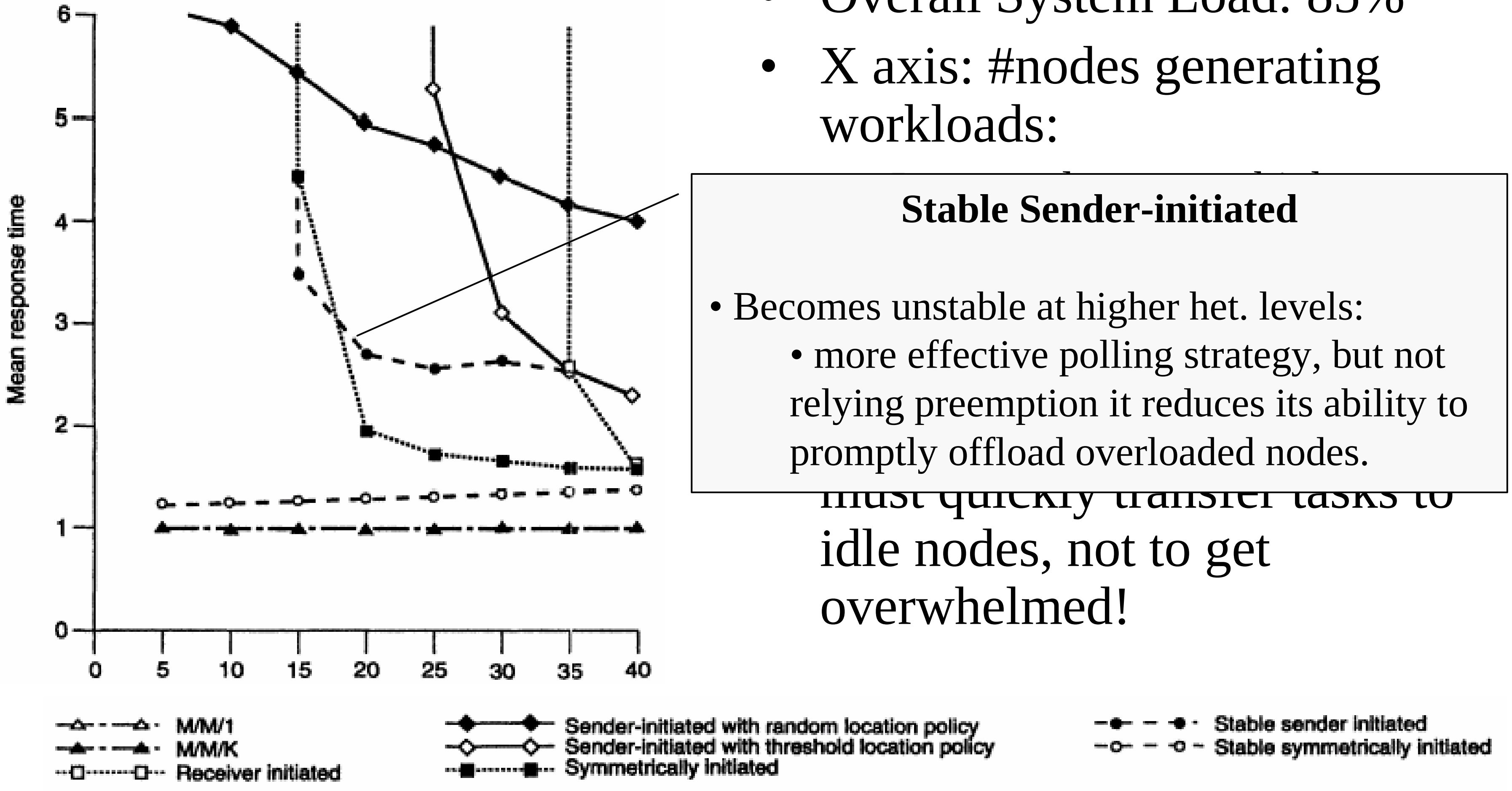
Sender-initiated with Threshold loc. pol.

- Slightly better performance:
  - but still random pollings result in resource wastage leading to saturation..

must quickly transfer tasks to idle nodes, not to get overwhelmed!

Performance Evaluation: Heterogeneous workload

- Overall System Load: 85%
- X axis: #nodes generating workloads:

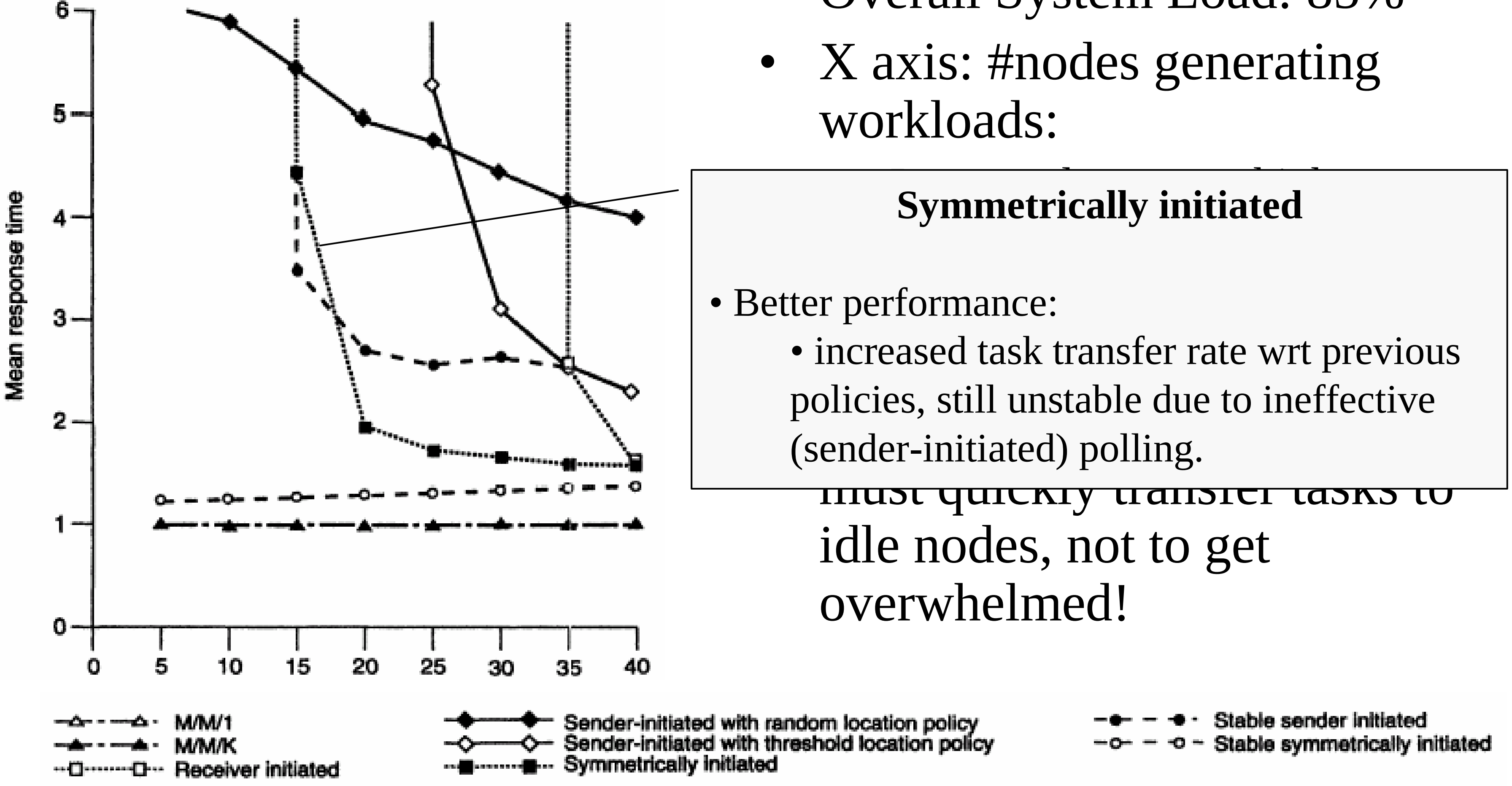


Stable Sender-initiated

- Becomes unstable at higher het. levels:
  - more effective polling strategy, but not relying preemption it reduces its ability to promptly offload overloaded nodes.

must quickly transfer tasks to idle nodes, not to get overwhelmed!

Performance Evaluation: Heterogeneous workload



- Overall System Load: 85%
- X axis: #nodes generating workloads:

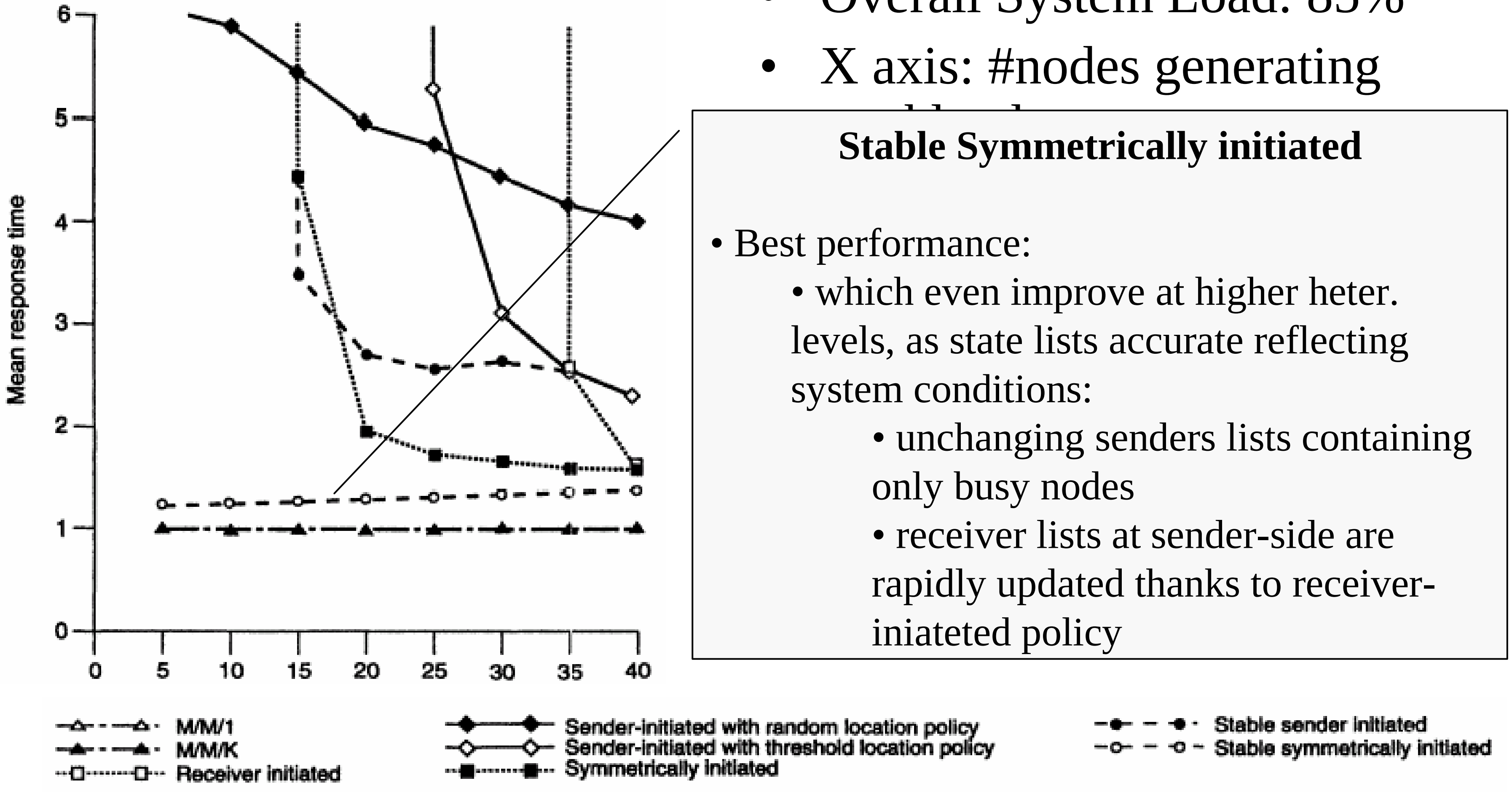
must quickly transfer tasks to idle nodes, not to get overwhelmed!

Symmetrically initiated

- Better performance:
  - increased task transfer rate wrt previous policies, still unstable due to ineffective (sender-initiated) polling.

Performance Evaluation: Heterogeneous workload

- Overall System Load: 85%
- X axis: #nodes generating



Stable Symmetrically initiated

- Best performance:
  - which even improve at higher heter. levels, as state lists accurate reflecting system conditions:
    - unchanging senders lists containing only busy nodes
    - receiver lists at sender-side are rapidly updated thanks to receiver-iniateted policy