INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Adaptive load sharing based on a broker module

M. Avvenuti *, L. Rizzo, and L. Vicisano

TR-96-036

August 1996

Abstract

This paper describes a dynamic, symmetrically-initiated load sharing scheme which adapts to changing load condition by varying the algorithm's dependency on system's status information. The scheme is hybrid in that it relies on a a fully distributed algorithm when the system is heavily loaded, but resorts to a centrally coordinated location policy when parts of the system become idle. The simplicity of the algorithms proposed makes it possible to use a centralized component without incurring in scalability problems and presenting instabilities. Both algorithms are very lightweight and do not need any tuning of parameters, so that they are extremely easy to implement to the point that an inexpensive hardware implementation of the centralized component is capable of handling millions of requests per second. Simulations show that the hybrid approach outperforms existing dynamic algorithms under all load conditions and task generation patterns, it is weakly sensitive to processing overhead and communication delays, and scales well (to hundreds of nodes) despite the use of a centralized component.

*Email: `marco@iet.unipi.it`

1 Introduction

The overall performance of a distributed system depends on the way by which the nodes are made to cooperate and share the workload. Load sharing strategies are aimed at reducing the average response time of tasks by avoiding *unshared states*, i.e. situations in which some nodes lie idle while tasks contend for service at some other nodes [10]. When no a priori assumption can be made on the task service time and the system's behavior, load distributing has to be performed at run-time by some *dynamic* algorithm that arranges for task transfer between nodes based on information on the system's status [8].

A distributed system may experience highly variable load conditions, including periods during which a few nodes generate very heavy workload. This entails for dynamic load sharing algorithms to behave adaptively, i.e. to dynamically change their parameters and/or policies to suit the system's changing status [11]. Adaptive behavior also copes with the problem of avoiding algorithm's instability, which may occur with particular load configurations because the algorithm enters a state where it spends all of the system's time in doing useless activities (such as looking for an idle node on a busy system, or looking for a busy node in an almost idle system) [2].

In this paper we describe a hybrid, adaptive load sharing scheme able to act quickly and efficiently in situations where some nodes are becoming idle, and to achieve a weaker but very cheap load balancing when the system is saturated. The scheme is hybrid in the sense that it behaves in a fully distributed manner when the system is heavily loaded, and is centrally coordinated by a *broker* component when the system is lightly loaded, smoothly moving between the two extremes as the system load decreases/increases. The information dependency of the algorithm is inversely proportional to the system load level, and collecting/disseminating status information does not rely on polling of other nodes to determine whether they are suitable as receivers or senders. This makes our approach able to regulate its demand of computing resources, turning down overheads and avoiding stability problems.

For comparison purposes, we begin by reviewing the most representative dynamic algorithms. We then describe our algorithms and give the design of a hardware realization of the broker module suitable for tightly coupled systems. Next, we evaluate the hybrid approach through extensive performance simulations, including the effects of processing overhead and communication latency, and discuss stability and scalability issues. Finally, we show comparisons with other algorithms, including the behavior in response to bursty workload.

1.1 Dynamic Load Sharing

A large amount of load sharing algorithms have been devised in the last years [15, 16, 17]. The references quoted in this paper constitute a small but hopefully representative part of them. In this Section we briefly describe three of the most well known dynamic load sharing algorithms: the *sender initiated* [6], the *receiver initiated* [5, 14] and the *stable symmetrically initiated* [14], with the aim to provide terms of comparison to evaluate the behavior of our algorithm. The three algorithms under consideration work in a fully distributed manner. They trigger a task transfer when the workload at a certain node exceeds/drops below some threshold (*threshold transfer policy*), and locate suitable nodes to take part to the transfer by polling them (*polling location policy*). Stable symmetrically initiated algorithms keep memory of the system status, while the other algorithms are memoryless.

The *sender-initiated algorithm* (SI) is started by an overloaded node when a newly arrived task increases the local task-queue length beyond a certain threshold T. This node starts polling sequentially a limited number of randomly selected nodes, looking for a potential receiver. A node is a receiver if its task-queue length is below T. If the overloaded node does not succeed in finding a receiver among the probed nodes, it will process the task locally. SI algorithms have the advantage of causing only non-preemptive task transfer, but have the drawback of being unable to cope effectively with unshared states occurrence. Moreover these algorithms present an instability problem that can lead the system to an early saturation. In fact, when the system workload is high, senders keep hopelessly looking for receivers adding to the actual system load the overhead due to this useless polling activity.

In *receiver-initiated algorithms* (RI) the load sharing activity is triggered by a node in which the departure of a task makes the task-queue length drop below T. Such a node starts looking for an overloaded node (with queue length greater than T) to get a task. As in the previous algorithm, this is performed by randomly polling an upper-limited number of nodes, until a suitable node is found or the polling limit is reached. RI algorithms are able to react quickly to the occurrence of unshared states but, in non batch systems, they trigger preemptive task transfers only.

A stable *symmetrically-initiated algorithm* (SSI) was proposed by Shivaratri and Krueger as an adaptive solution to the problems of the SI and RI algorithms. It embodies both sender and receiver initiative, adding the capability for the nodes to make use of the information gathered during polling to settle further polls. Each node, using this information, classifies other nodes either as *sender* (overloaded), *receiver* (under-loaded) or *OK* (correctly loaded). This way, an overloaded node that needs to offload some tasks polls only receiver nodes. Vice versa, an under-loaded node preferably polls sender nodes. Nodes update their knowledge on system's status on each poll they perform and every time they are polled. The presence of this memory allows the algorithm to avoid sender-initiated instability at high system load, and to save a number of unsuccessful polls in normal operating condition.

2 Design Rationale and Overview

Our load sharing scheme applies to distributed settings where tasks are logically independent of one another and can be generated and processed by any of the system's nodes. The load index is the number of pending tasks at nodes; this can be correlated to the CPU queue length, which is generally considered a good workload descriptor [9, 18], and is very cheap to evaluate. The performance index is the average response time for tasks. The scheme deals with tasks eligible for migration, provided that a task migration facility able to move tasks either non-preemptively and preemptively is available in the system.

Our design is based on two guiding ideas:

- when the system is heavily loaded there is no need to strive for accurate load sharing, because all the nodes are likely to be busy in servicing tasks and the probability of having unshared states is low. Experimental studies have shown that simple load distributing policies such as random placement and cyclic splitting yield good performance improvement, and that using more informed algorithms with high load levels might even be detrimental to performance, because of the waste of resources for maintaining information on system's status [6, 15, 17];

- in a lightly loaded system, i.e. a system where the probability of having idle nodes is high, it can be worth spending additional resources for running more information-dependent, possibly centrally coordinated algorithms. This is because the performance of a lightly loaded system is more sensitive to the effects of inappropriate task transfers which may result from uncoordinated location policies [13].

The solution we propose consists in a symmetrically-initiated algorithm that adapts to changing load condition by varying the algorithm's dependency on system's status information. This means that, as the system load increases, the algorithm becomes more lightweight by degrading the accuracy with which it balances the load.

Basically, we designed a distributed load sharing algorithm with a centralized location policy: By default, each task transfer between a sender node and a receiver one is coordinated by a *broker* module. The idea is that a broker, kept informed about the system's status, should allow a node to save time in finding its party on a task transfer. This means that a node, whether a sender or a receiver, will initiate a task transfer by first asking the broker for advice. Note that such a solution does not require explicit polling of potential senders and receivers and introduces only the overhead required to contact the broker.

As long as the system's load is low, computing and communication resources are likely to be available for the broker's service, that in turn allows the nodes to share the workload fairly and efficiently. However, as the system load increases, the probability of finding idle nodes becomes lower and the number of potential senders higher: Then, consulting the broker becomes likely to introduce a bottleneck, and the sender-initiative to cause system's instability. We deal with this problem by making sender nodes stop turning to the broker when the latter sees each node has enough work to be kept busy. Afterwards, senders start using a simple, fully distributed task placement algorithm where decisions are only based on the local knowledge each node has about how it contributed to the load at other nodes. The broker still remains available for possible receiver-initiated task transfers, which cause the involved nodes to revert to using the centralized broker algorithm.

An interesting feature of our approach is that it does not depend on any tunable parameter, while still being fully adaptive. Nodes switch between the centrally coordinated and the fully distributed algorithms asynchronously, the transition being triggered by the type of communication each node has been having with the broker.

3 Detailed Algorithms

3.1 The Broker Algorithm

In this Section we provide details on the centrally coordinated algorithm based on the broker. The full algorithms run by the broker and the nodes are shown, in pseudo-C code, in Figures 19 and 20.

As shown in Figure 1, on each node we can schematize a task generator, the *source*, a task processor, the *server*, and a process which performs the local computation related to the load sharing algorithm and dispatch tasks to servers, the *manager*. A node in the system is designated to run a *broker process* that maintains some data structures representing a view of the system's status. In such a view, *the status of each node is recorded in terms of the load index.*

In theory, the broker could keep the exact load index of each node in order to be able to take always the best decision on task transfers: This would have the drawback of requiring a great number of messages to be exchanged, both to keep consistent and up-to-date the broker's view of the system, and to query the broker about task destination. Also, it is well known by literature that complex policies collecting large amounts of information do not sensibly improve the performance yielded by simpler policies [6]. Thus, we decided to allow only three different states for each node, which entails the broker to maintain three data structures: a *idle queue* containing idle nodes, a *busy queue* containing nodes that are processing the unique task in their task queue, a *loaded queue* recording nodes that have more than one task queued. The information in the broker, as we keep it, is *likely* to be correct, although the exact status of each node is only known to the node itself.

Task transfers are symmetrically-initiated. The sender-initiated component of the algorithm (Figures 1a and 2) is executed when a new task originates at a node. The local manager sends a request to the broker asking for a candidate server. If the broker does not know of under-loaded nodes, the manager switches to a de-centralized task allocation policy and selects the candidate autonomously. Either way, the task is sent to the selected candidate server. To deal properly with scalability issues, the broker algorithm is enriched with the following adaptive *timeout* mechanism: if a new task arrives when there are pending responses from the broker, the manager will immediately switch to the de-centralized algorithm, thus avoiding further delays and further demand to the broker. Turning to the broker will restart when the local server becomes idle again.

All the above is implemented as synthesized in Figures 2 and 3. Note that, normally, a node starts using the de-centralized algorithm when all nodes are in the *loaded* queue, i.e. they are likely

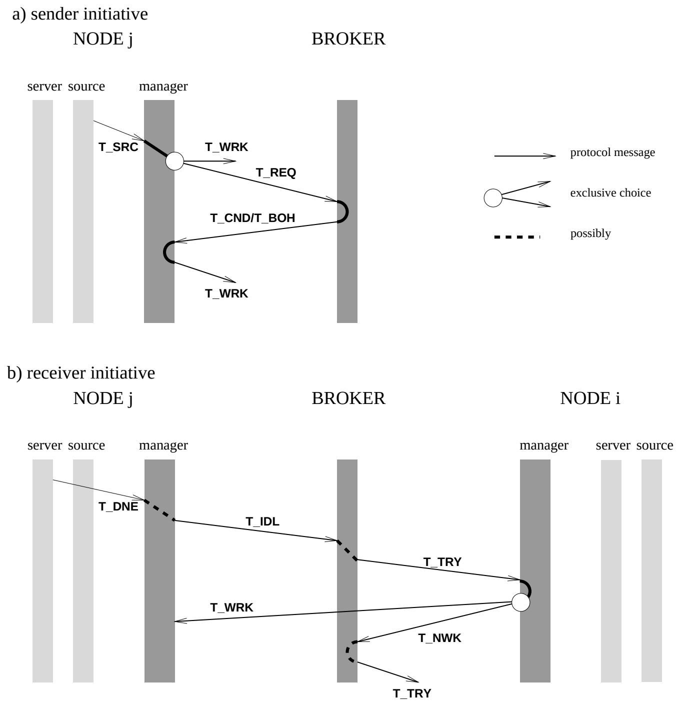

Figure 1: Node-broker-node interaction.

to be busy processing one task and have at least one more task in their local queue.

The receiver-initiated component of the algorithm (Figures 1b and 4) is executed when the local server becomes idle. In this case, the manager restarts turning to the broker for future allocations, and notifies to the broker that some work can be offloaded from a busy node. On its behalf, the broker will forward requests for a task transfer to the nodes, if any, which it knows to be loaded. This is implemented as synthesized in Figures 4 and 5.

3.2 The Ticket Algorithm

Although making use of little knowledge of system's status, the de-centralized task placement policy still strives to keep the workload at nodes roughly balanced, in order to postpone the point in which we have to make use of the (more expensive) centrally coordinated algorithm. We avoided aiming at exact balancing as it requires extra communication and computation, a situation that is undesirable when the system is heavily loaded.

The algorithm relies on a local view of the system's status: Each node keeps count of the number of pending tasks it has submitted to every node in the system. We can think of it as if a source must spend one *ticket* to send the task to a server (including the local one); tickets are returned to the source node when the task completes. The selection of the candidate server is done so as to spend approximately the same number of tickets for each server. In practice, this means choosing at random among nodes with the smallest number of tickets spent for, privileging the local server in order to minimize the number of task transfers. For stability reasons, servers always accept incoming tasks regardless of their status. This makes the algorithm overhead and transfer delay to incur at most once per task.

This simple algorithm (the *ticket algorithm*) works better than random or cyclic splitting, in that it keeps count of the fact that different tasks have possibly different service times. The ticket

Sender-initiated component (see Fig. 1a)

(Triggered by a request (T_SRC) coming from a *source* to the local *manager*)

manager side

The *manager* behaves as follows:

- If enabled to use the broker, it asks the latter for a candidate server to process the task (T_REQ), and waits for a reply (T_CND/T_BOH);
- If not enabled to use the broker, or the latter cannot suggest a candidate (T_BOH), or does not reply promptly (i.e. before another request from the source arrives), it disables further uses of the broker, and chooses a candidate basing on local information only.
- It sends the task to the selected candidate *server* (T_WRK).

broker side

The *broker*, upon receiving T_REQ, behaves as follows (see Fig. 3):

- If *idle* and *busy queues* are empty, it replies with a T_BOH and stops.
- If the *idle queue* is non empty, it selects the last node in that queue moving it to the *busy queue*; otherwise it selects the first node in the *busy queue* moving it to the *loaded queue*. The selection strives to promote the locality of task.
- It designates the selected node as candidate (T_CND).

Figure 2: The broker algorithm: sender-initiated component.

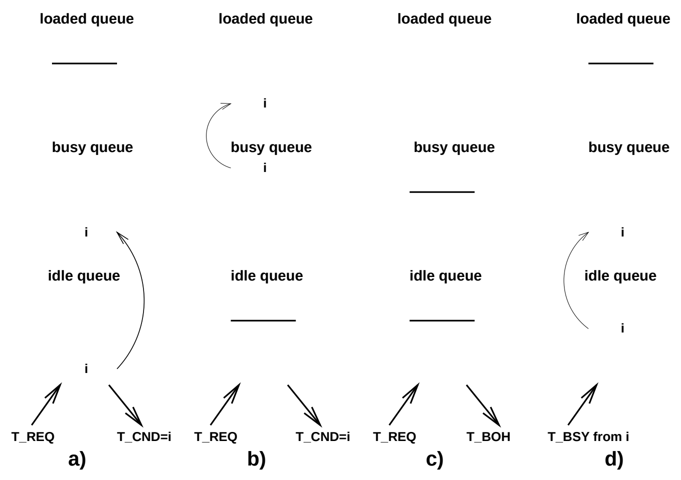

Figure 3: Broker's data management, increasing load.

Receiver-initiated component (see Fig. 1b)

(Triggered by a node becoming idle)

manager side

The *manager* behaves as follows:

- It re-enables further use of the broker;
- It sends a message (T_IDL) to the broker informing that the node has become idle.

Upon receiving T_TRY, it behaves as follows:

- It picks one element (if available) from its task queue, and sends it to the requesting node. If no tasks are available, then it notifies the broker (T_NWK).

broker side

The *broker*, upon receiving T_IDL, behaves as follows (see Fig. 5):

- If the *loaded queue* is empty, it inserts the sender node in the *idle queue* and stops.
- Otherwise it inserts the sender node in the *busy queue*, selects the first node in the *loaded queue*, moves it to the tail of that queue and send to it a T_TRY to see if it can offload some work to the idle node.

Upon receiving T_NWK, it behaves as follows:

- It inserts the sender node in the *busy queue*.
- If the *loaded queue* is empty, it inserts the idle node in the *idle queue*.
- Otherwise it selects the next node in the *loaded queue* and sends it a T_TRY.

Figure 4: The broker algorithm: receiver-initiated component.

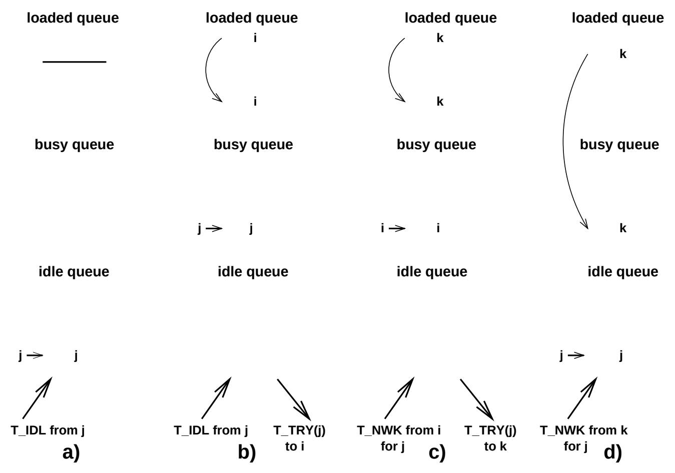

Figure 5: Broker's data management, decreasing load.

algorithm can be run at almost no cost: in fact, the amount of date structures required on each node is reasonably small (a N integer vector), and usually no additional communication is required to return tickets, as the execution of a task will very likely produce a result to the source node, and this can be implicitly assumed as the return of a ticket. Also, managing the tickets' vector requires only (small) constant time operations.

As we already noticed before, the system does not change its behavior all at once from the centralized algorithm to the distributed one, but transitions are performed independently on each node. This makes it possible that a source, using the ticket algorithm, sends a task to a node marked as idle by the broker, thus leading to temporarily incorrect information on the broker. Such an occurrence can be detected and corrected by means of the T_BSY message, whose reception causes the broker to move the sender from the *idle* to the *busy* queue (Figure 3d).

3.3 Comments

The simulation results shown in Section 5 prove that our hybrid algorithm performs well under varying operating conditions. Nevertheless some notes are worthwhile before going on.

The first issue is that each sender-initiated task allocation is delayed by a full message round trip (T_REQ/T_CND). That can be inefficient especially when the sender node itself is idle and must wait the response from the broker before starting processing the task. This situation is very frequent when the system is lightly loaded. Section 5.2.2 discusses such issue and proposes a modification to the broker algorithm in order to cope with this inefficiency source.

A second important issue, which arises whenever a centralized solution is proposed, is the scalability of the approach with respect to the number of nodes. Section 5.3 considers scalability issues and shows that the algorithm is able to work properly with a considerable number of nodes.

As far as the reliability is concerned, note that with our hybrid approach the system *does not* depend on the broker in order to work. In practice, the timeout mechanism seen above can be slightly modified to deal with failures in the system that cause breakdown in communication with the broker. In such situations, the nodes experiencing the failure start resorting to the ticket algorithm and possibly carry out the election of a new broker.

4 The Hardware Broker

We introduced our hybrid load sharing algorithm aiming at a loosely coupled distributed system such as a workstation cluster. In this Section we want to show that this algorithm is suitable as well for a tightly coupled parallel system based on a — at least logical — shared memory architecture. This working environment poses some new constraints, first of all the need for the algorithm to work faster and with less overhead, due to a much finer task granularity. That can be achieved with the hardware implementation of the broker component briefly described in the following (for a more comprehensive discussion see [1]).

In order to adapt the broker to a hardware implementation, the algorithm must be changed slightly from the description given in Section 3.1. The reason is twofold: First, our system model is not anymore a message-passing one, but rather a shared-memory one. What was before implemented as a message exchange, can be now implemented more efficiently as reads or writes to some port in a shared address space. Second, in order to keep the broker simple, it must be designed as a "slave" peripheral device, with some internal registers and the ability of sending interrupts to nodes of the system. The broker should only respond to requests from the managers, and ask their attention through an interrupt when necessary.

As a result, we designed the broker as a peripheral device seen by the processors through three ports, as shown in Figure 6:

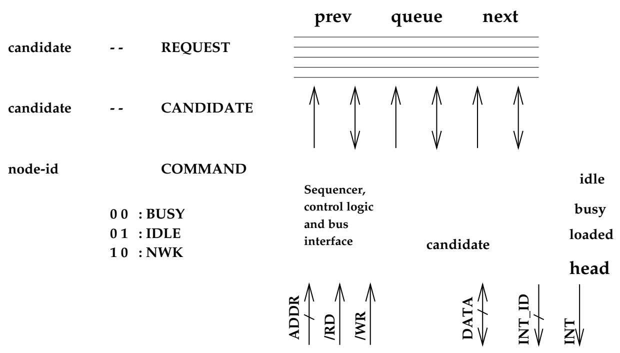

Figure 6: The architecture of the fast broker

REQUEST (read only) The REQUEST port is accessed by a node when it wants to allocate a new task. The broker should return the (precomputed) identifier of the node (the *candidate*) which is able to process the new task, or a special identifier indicating that the broker is unable to respond. An access to this port replaces the exchange of the `T_REQ+T_CAND/T_BOH` messages. In response to this access, the broker has to manipulate its queues as shown in Figures 3a-c, that is:

- extract node *candidate* from the queue it belongs to;
- insert node *candidate* into the queue corresponding to the next load level;
- compute a new *candidate* by reading the head of the IDLE or BUSY queue.

CANDIDATE (read only) The CANDIDATE port is accessed by a node which has received an interrupt (generated by the broker in response to a `T_IDL` or `T_NWK` message). The node expects to find here the identifier of a node (the *candidate*, again) which is idle and thus available to offload some work. The access to this port replaces the delivery of the `T_TRY` message, and requires the queue manipulations of Figures 5b-c, that is:

- extract node *candidate* from the queue it belongs to;
- insert node *candidate* into the queue corresponding to the next load level;
- compute a new *candidate* by reading the head of the IDLE queue;

Note the similarity with the operations performed in response to accesses to the REQUEST port. The only difference lies in the fact that a valid *candidate* for this request must be an IDLE node, while it can also be a BUSY node in the previous case.

COMMAND (write only) The command port is used to send the `T_BSY`, `T_IDL` and `T_NWK` messages, which also contain the identifier of the node sending the command. Upon reception of these commands, the broker executes the appropriate actions (Figures 3d and 5a-d), which consist at most in extracting one node from the queue it belongs to, inserting it into the proper queue and possibly sending an interrupt to a second node and preparing a new candidate for the next request.

As we aim to build the broker in hardware, we have to identify which data structures support most efficiently the desired operations, and the dependencies among them, so that we can possibly improve the performance by doing some actions in parallel. There are three basic operations that we must support inside the broker: insertion and extraction from a circular list, and determining which list a node belongs to. All these operations can be executed in constant time if we represent the lists as an array of records, indexed with the node identifier.

Extracting and inserting elements in a list requires several steps, which can be parallelized somewhat if we store each field of the record (the *next* and *prev* pointers, and the data field), and the pointers to the head of the queues in separate memory blocks. In this way, both insertion and extraction require two sequential accesses to the memories, as follows:

Extraction of node s from a queue

1. x $\leftarrow$ prev[s], y $\leftarrow$ next[s], q $\leftarrow$ queue[s], h $\leftarrow$ head[queue[s]];
2. head[q] $\leftarrow$ if h == s then y else h, prev[y] $\leftarrow$ x, next[x] $\leftarrow$ y.

Insertion of node s into queue q

1. y $\leftarrow$ next[head[q]], next[head[q]] $\leftarrow$ s, prev[s] $\leftarrow$ head[q], queue[s] $\leftarrow$ q;
2. next[s] $\leftarrow$ y, prev[x] $\leftarrow$ s.

The other operations that must be performed require accessing/modifying the pointers to the head of the lists, and can be performed in parallel with the updates to the lists.

4.1 Implementation issues

We can design a semi-custom chip for the broker with the structure shown in Figure 6. The device will require some RAM to store the queues, three registers used as queue pointers, one register to hold the node id that will be supplied in the next read from the REQUEST/CANDIDATE port, plus a sequencer which controls accesses to the various data structures. With such a structure, we can achieve maximum speed with little additional cost if we use separate memory blocks to store each field of the record (the *next* and *prev* pointers, and the *queue* field). This way, any insertion or extraction requires two sequential steps where the memories are accessed in parallel. Each access to the broker's registers requires at most one extraction, one insertion, and possibly some modifications to the pointers, which can be done in parallel with the previous operations. Thus, we can estimate that one access to the broker requires little more than the time to perform four accesses to its internal memories. A further improvement could be achieved in principle by further parallelizing the extraction and insertion, but this would require dual port memories and add much greater complexity to the circuit.

To get some realistic numbers, we have to size the RAM, and determine its access speed using state-of-the art technology, such as the one made available by ES2 [4]. The *queue* field of the data structure is 2-bit wide. All the other dimensions depend on the number of nodes that must be supported by a single broker. If we "limit" ourselves to up to 256 nodes, we need 8-bit pointers in the link fields and in the pointers, and a total of 256 entries. The total amount of RAM, about 4Kbits, can easily fit in a semi-custom design. The worst case access times for library memory modules of this size lie in the $15ns$ range; by adding some $5 \div 10ns$ for control logic and multiplexer we end up with a system which can perform all the processing required for a message in about $100ns$. This has two important consequences:

- running the centralized part of our load sharing algorithm has roughly the cost of a bus access;
- there is no need to queue requests to the broker. The broker can sustain a load of 10Mrequests/second, and its performance is only limited by the interconnection between the nodes and the broker.

Note that a task allocation involves only two access to the broker (`T_REQ, T_IDL`). Thus the broker is (in principle) able to allocate tasks with a granularity of $.2n$ μs, where n is the total number of nodes in the system.

4.2 Performance

The response and cycle times of the broker are both equal to t_{bus}. However, nodes compete to access the bus where the broker is connected. Thus the actual service time of the broker depends on the traffic on this communication channel. We can model this system, with good precision, as an M/D/1 queue. The expected waiting time is thus [3]

$$E(w) = \frac{\rho}{1-\rho}\frac{t_{bus}}{2}$$

where t_{bus} is the bus service time, λ is the frequency of requests, and $\rho = \lambda t_{bus}$. The actual service time thus becomes $E(w) + t_{bus}$. As we can see, the waiting time remains below $2t_{bus}$ when the bus is used for less than 80% of its capacity.

5 Performance Simulation

The performance of our hybrid load sharing algorithm (named HBT henceforth) has been simulated thoroughly under a variety of operating conditions, to determine its effectiveness in achieving an even distribution of workload, and its sensitivity to parameters such as the number of nodes, processing overheads and communication latencies.

Simulation has been carried out with `psim` [12], a software that allows us to simulate the execution of a parallel program on a distributed memory system. `psim` lets the user define system features such as the number of processors, the processing speed of individual processors, the execution time of sections of code, the communication time. `psim` allows writing code in high level language, and supports a set of synchronous and asynchronous communication primitives very similar to those available in the Unix environment. Thus the simulation code is essentially a working implementation (in C language in our case) of the algorithm to be tested. Another significant advantage of `psim` is that simulations are completely reproducible, and that the code can be instrumented with no perturbations at all.

The simulated system consists of homogeneous processing nodes, able to communicate with each other via an unspecified connection network, incurring in a deterministic communication latency. We have assumed exponentially distributed task service times with unit average value ($S = 1$). As far as task arrivals at nodes are concerned, we used both Poisson arrivals and IPP arrivals [7] (to model burst workload). In addition to processing tasks, each node also has to process messages related to the broker algorithm, and to do some small amount of work to handle tickets (no additional communication is required by the ticket algorithm). Each protocol message requires some processing on the sender and on the receiver, plus incurs some delay in the network. To model that, we have adopted the following simulation parameters:

- *offered system load per node* ($\rho = \lambda S$), computed as the average task service time (S) multiplied by the global task arrival rate divided by the number of nodes ($\lambda = \Lambda/N$);
- *communication latency* (γ_m), to model systems with different communication media;
- *message-processing time* (η_m). Although protocol messages require a small, constant service time, which can be reasonably thought of as negligible with respect to the average task service time, we have to keep count of the fact that the node hosting the broker has to process a larger number of messages.

As far as task transfers are concerned, in a first analysis we have assumed the transfer overhead/latency being equal to the transfer overhead/latency for protocol message. This overhead does not affects the broker, and influences more or less linearly the task execution time. Subsequently, in order to evaluate the penalty of preemptive task transfers, we have assigned different values to preemptive and non-preemptive task transfer overhead/latency (η_{t-p}, η_{t-np}, γ_{t-p}, γ_{t-np} respectively).

Unless otherwise specified, the number of nodes (N) is 40, and all nodes run a task source, a task server and a manager process. The time values reported are normalized to the task service time.

5.1 Preliminary Evaluation

Simulations in absence of overheads allow us to compare the performance of HBT algorithm with that of M/M/1 and M/M/k systems. This is shown in Figure 7, which plots the average response time versus system load for zero message-processing overhead and several communication latencies; Independent Poisson task generators are applied in each node. It is to note that with zero latency the HBT performs as the ideal M/M/k system does (lower curve in Figure 7). This can be explained by the fact that the HBT algorithm behaves essentially as its broker component for not too-heavy loads (see Figure 9 and related discussion). In turn, in the case of zero processing overhead and communication latency, the broker component of HBT can be viewed as a single queue model with a *non-biased queuing discipline* [3], so we obtained the same performance of the M/M/k system.

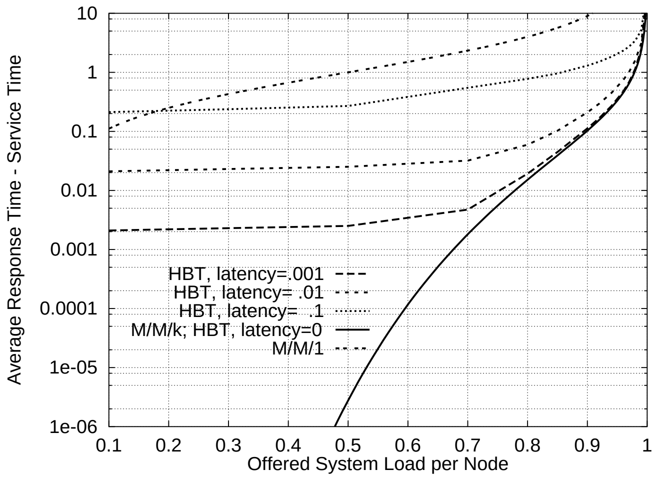

Figure 7: Average response time vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = 0$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = latency$.

Even the behavior with a variable number of task sources is satisfactory: Figure 8 shows the response to a variable number of source nodes, for a fixed global system load and variable communication latency. As we can see, there is no appreciable sensitivity to the number of sources, which means that tasks are distributed evenly among nodes and thus the algorithm fulfills its purposes.

The separate behaviors of the ticket, the broker and the whole HBT algorithm versus system load are shown in Figure 9 for high load levels. The curves have been computed with a moderate processing overhead ($\eta_m = \eta_{t-p} = \eta_{t-np} = 0.001$), but their shape does not change significantly with the overhead. As the load increases, the broker component, which determines the behavior of the system at low-to-medium load levels, becomes more influenced by the processing overhead, up to the point where it is outperformed by the ticket component. The exact value of the system load where the two curves intersect (in our case is $\rho = 0.99$) depends on the overheads.

A relevant result is that the hybrid approach yields better performance than that obtainable from the separate ticket and broker components. This can be explained as follows: At low load levels, nodes will occasionally recognize that the broker cannot give a useful reply and avoid contacting it, thus reducing the broker's load and slightly improving the overall performance; conversely, at high

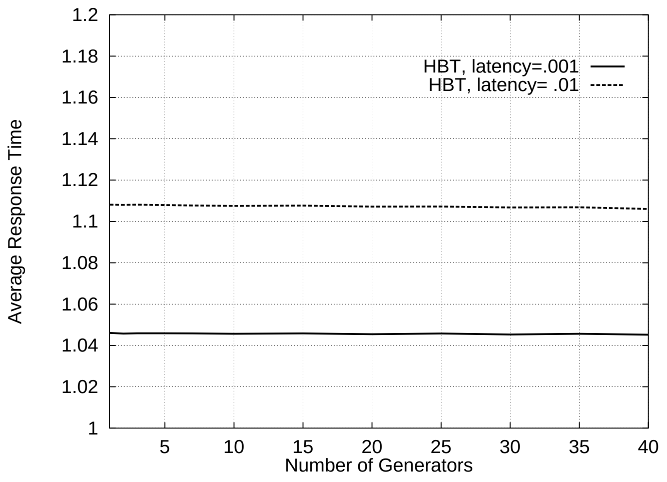

Figure 8: Average response time vs. number of task sources, $N = 40$, $\rho = .85$, $\eta_m = \eta_{t-p} = \eta_{t-np} = 0$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = latency$.

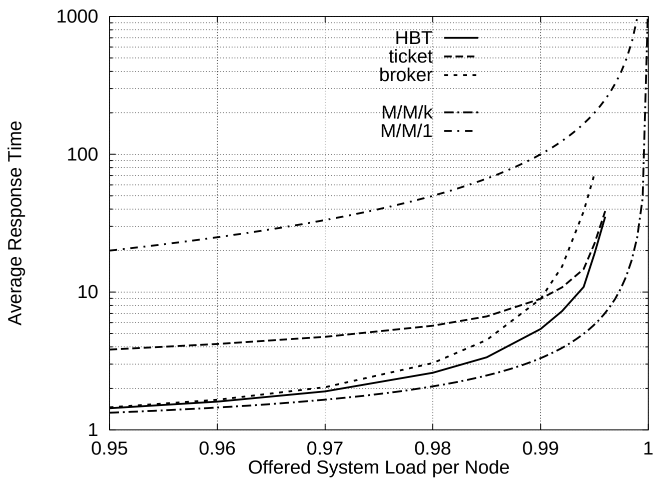

Figure 9: Comparison between separate behaviors of HBT and its broker and ticket components, average response time vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = .001$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = 0$.

load levels, nodes that become idle will be able to exploit the presence of the broker to get some work from loaded nodes.

5.2 Overhead Analysis

Communication among nodes incurs in some delays, which can be considered as made of three parts: i) the time spent by the sender of the message to prepare and transmit the command; ii) the time spent by the message to travel through the network from source to destination; iii) the time spent by the receiver to collect the message and process it.

Parts i) and iii) can be further divided into a communication overhead (related to the activities performed by the operating system to move the message between the user process and the network hardware), and a message-processing overhead, related to the processing of the message (in our case, the load sharing algorithm). In modern systems, the communication overhead is small and relatively independent of the size of the message. For our purposes, we can consider these overheads as small amounts of CPU time consumed by the nodes involved in the communication. The message processing overhead depends, of course, on the particular processing that must be done on the message. However, in our case, the operations required to implement the load sharing algorithm are extremely simple, and require a constant time which is probably negligible with respect to the communication overhead. As a reference, the sum of all these processing times even on current, low-end Unix workstations is well below one millisecond.

The overhead in part ii) is of a different kind: It does not consume CPU time, but delays messages which have to flow through the communication hardware. The actual value of this delay can be anywhere between a few microseconds (in a tightly connected system) and the few seconds that can be experienced between very far-apart sites on the Internet. Of course, we cannot expect to achieve good performance when communication latencies are of the same or greater order of magnitude of the task service time.

Another important overhead is incurred whenever we decide to transfer a task from a source to a server. As in the case of ordinary messages, this overhead can be characterized in terms of processing time plus communication delays. However, the amount of data that needs to be transferred (hence the CPU time spent in moving them) can be much different – and is in general larger – than in the case of the simple messages which are needed to implement the load sharing algorithm. When non-preemptive task transfers are concerned (task placement), the amount of data is made of the input data for the task, plus the output data produced by the task. It is straightforward to devise some technique that makes this overhead incur at most once per task[1], independently of the number of hops between queues that the task is subject to. Of course, privileging the local server when making task allocations can eliminate this overhead and improve performance. Preemptive task transfers incur in some additional overhead/delay mainly due to the need to collect–transmit-restore the task status (process memory image). For the time being we neglect this additional overhead assuming the same overhead/latency for task and protocol message transfers. In Section 6 a larger cost for task transfers will be introduced.

5.2.1 Processing Overheads

From the previous discussion on the nature of processing overheads, we can see that it suffices that our algorithm performs well when the processing overhead is a small fraction of the average task service time. The effect of such a kind of overhead is that the actual system load is greater than in the ideal case. This is revealed by the translation to the left of the curves in Figure 10 and is more visible at high system loads when the overhead takes away resources from task processing.

[1]This can be achieved by transferring only the name of a task between nodes, and moving the input data for a task when it has actually started.

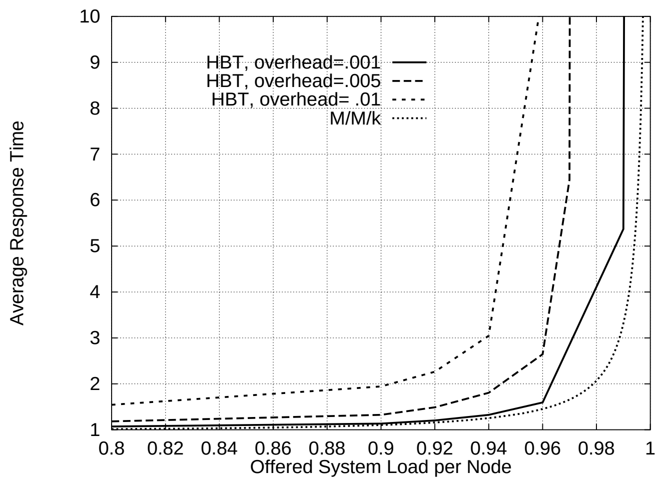

Figure 10: Average response time vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = overhead$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = 0$.

To understand better the dependency of our algorithm on the task transfer time, we have measured the average number of times a task is transferred. Figure 11 shows the average number of times it happens that a task is serviced on a different node from the one where it was generated. There are three reasons why a task is transferred to a different node: Because of a suggestion from the broker (*sender-initiative*), because of a local decision based on the ticket component (*ticket policy*), or because of a request from an idle node (*receiver-initiative*). The number of transfers for each of these reasons is also shown in the plots of Figure 11, which are computed with zero processing overhead. The dependency of these curves on the processing overhead would show up as an asymptote for a load slightly lower than unity, but would not change the shape of the curves.

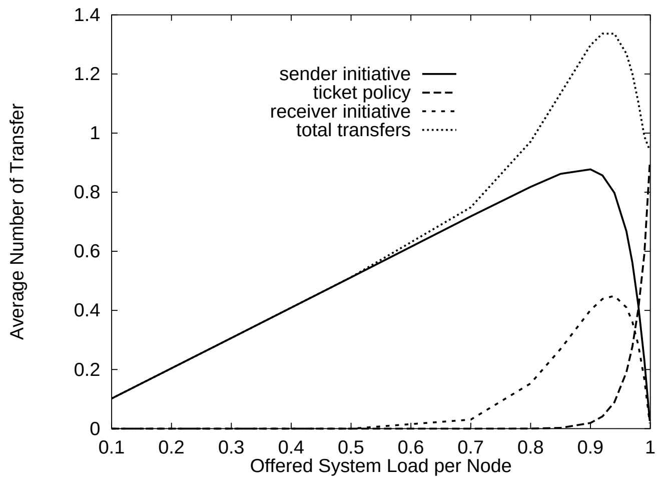

Figure 11: Average number of transfers per task vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = 0$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = .01$.

As we can see, at low load levels the average number of transfers is essentially equal to the system load (which is also the probability of finding the local queue non-empty), and all of the transfers are caused by the broker. As the load increases, the overall action of the sender-initiative and the ticket policy still causes a more or less linear number of transfers, but globally there is a deviation from linearity. This is mainly determined by the fact that the greater the load the more frequently the receiver-initiative succeeds in finding a task to transfer. When the system load gets even higher, the number of transfers due to the presence of idle nodes drops to zero. Thus, from the graph we can conclude that each task incurs an additional overhead which is more or less proportional to the task transfer time and system load, but surely lower than twice the transfer time.

5.2.2 Communication Latency

The effect of communication latency is visible from Figure 7, where the curves show the dependency of the average response time on the communication latency. The behavior of the system at low load levels is easy to explain: the latency appears as a simple additive term in the response time, because we incur on a round trip delay every time we have to contact the broker for allocating a newly generated task (which happens essentially all the times). As the load increases, non linear phenomena occur. To understand them better, we have to look again at Figure 11. We can see that at moderate-high load levels there is a peak of transfers caused by nodes becoming idle; unfortunately, retrieving a task to be transferred to an idle server makes the server itself to remain idle for a time proportional to the communication latency, with an effect similar to an increase of the effective load for the system. Thus, there is a range of system loads where the response time increases more than linearly in the communication latency. At even higher loads, transfers are essentially caused by the use of the ticket algorithm, which means that the system is able to use more effectively its resources, thus the asymptote for all the curves in Figure 7 occurs at a load level of 1 (in absence of processing overheads, of course).

Any delay in assigning a task to a server can be detrimental to performance when the receiving node is idle. In order to optimize the algorithm, avoiding unnecessary task transfer delay, we serve locally the tasks produced by an idle node. There are two way to do that: The first one (the local approach) consists in skipping the request to the broker and selecting the local server instead; a T_BSY message is then to be sent to update the information on the broker. The second one (the remote approach) consists in moving the optimization to the broker, which can try to keep tasks on the originating node whenever possible. The local approach allows us to save the time needed to contact the broker but it might trigger the following race condition: A node i sends a T_IDL message to the broker which starts looking for some work to be sent to the sender. In the meantime, the source of node i sends a task to the local server, which is known to be idle. The net effect is that node i is likely to receive two tasks while other nodes in the system might lie idle.

Simulations have been done in order to evaluate the effect of both the approaches and the results are shown in Figure 12. We can see that, at low load levels, the local approach yields better performance, while the remote approach performs better at higher load levels. This is easily explained by the fact that the higher the system load, the higher is the probability of having an idle node assigned with tasks by both the broker and the local manager in a uncoordinated manner. Simulations shown hereafter use the remote approach.

Another source of loss of performance arises when a node becomes idle. In this case, there is a minimum of two messages (T_IDL followed by T_TRY and zero or more T_NWK/T_TRY rounds) followed by a task transfer. In principle, it is possible to hide such a delay by sending the T_IDL message in advance, so that the selection of a loaded node is carried on in parallel with the final processing of the current task. However, this only makes sense if we know how much processing is left on the current task, otherwise we risk to steal a task from a node which might complete processing it before the local one. Moreover, the broker usually has very up-to-date information about the system-status, so that the chance of requiring several rounds to determine a potential candidate is likely to be low.

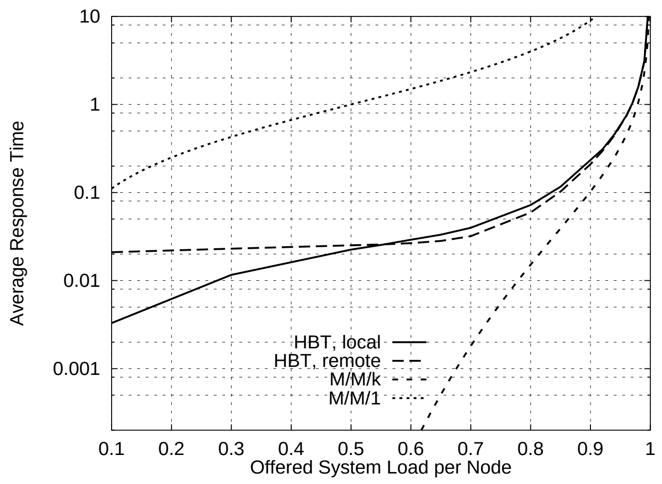

Figure 12: Comparison between the *local* and the *remote* approach to local task allocation, average response time vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = 0$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = .01$.

5.3 Stability and Scalability Issues

The usual concern in using centralized algorithms is the introduction of a potential bottleneck in the system. As said before, however, the CPU overhead involved with the load sharing algorithm is usually negligible with respect to the actual workload. In addition to this, the system is self-compensated because of the following properties: i) the broker becomes less and less used as the load increases; ii) the load sharing algorithm indirectly acts for the broker's task as well, so that the node hosting the broker appears slower than other nodes and thus processes fewer tasks than others. Still in our case there might be concern on the number of messages that the broker has to process.

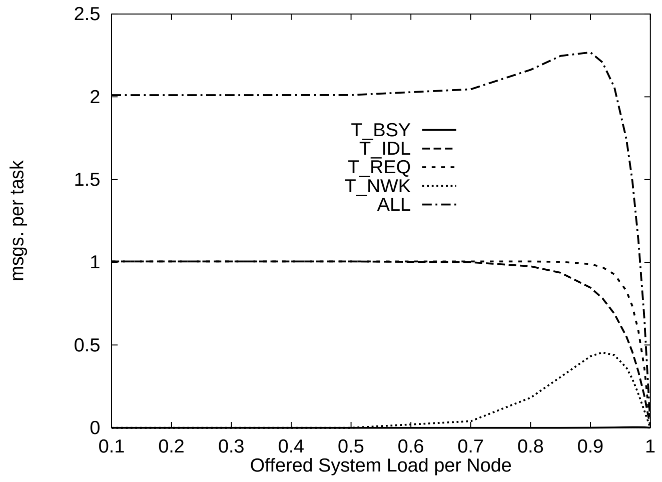

Figure 13: Average number of messages to the broker per task vs. offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-p} = \eta_{t-np} = 0$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = .01$.

Figure 13 shows the average number of messages that are received by the broker for each task. Even these curves show the same dependency on processing overhead and communication latency as

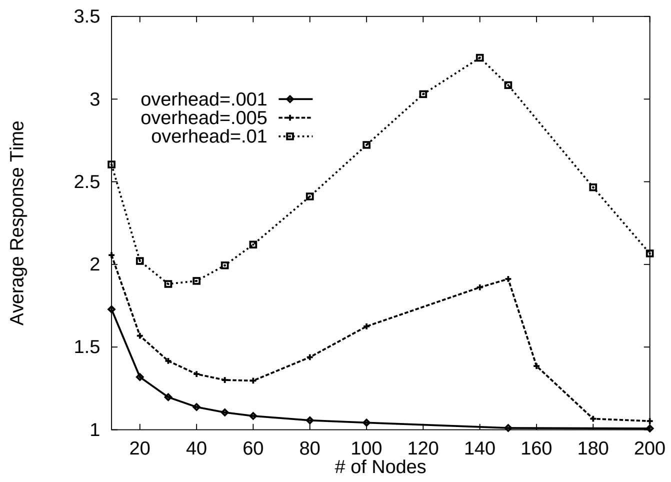

Figure 14: Response time versus number of nodes (N) for different overheads, $\rho = .9$, $\eta_m = \eta_{t-p} = \eta_{t-np} = overhead$, $\gamma_m = \gamma_{t-p} = \gamma_{t-np} = 0$.

those in Figure 11. As we expected, at low load levels, two messages are received per task (T_REQ when the task must be initially allocated, T_IDL when the task completes and the server becomes idle). As the load increases, there are fewer T_REQ and T_IDL, but the receiver-initiated part of the algorithm causes the exchange of T_NWK messages. The stability of the algorithm is clearly shown by the fact that as the load increases further, the number of messages sent to the broker decreases towards zero.

The worst case occurs when the broker receives the highest number of messages, which, for a reasonable value of the latency, occurs at a load around 0.9 (Figure 13). Thus we made a number of simulations with such a system load and a variable number of nodes to get more information on the scalability of the algorithm. This is shown in Figure 14, for different processing overheads. We have to remark that the reasonable curve in the figure is the bottom one, with a overhead of .001 time units. The other curves have been drawn to show the pathologies that can occur when the broker approaches saturation, and to prove the effectiveness of the "timeout" mechanism described in Section 3.1.

Theoretically, the curves in Figure 14 should decrease as the number of nodes increases. In practice, the increase of load on the broker causes a change of slope at some point: The response time increases as the broker approaches saturation, until the timeout brings the system back to stability. The effect of the "timeout" mechanism is to actually disable the access to the broker by a subset of nodes when the operating condition becomes critical (large number of nodes, high load, slow broker), thus allowing the system to keep running without incurring in instabilities.

By putting realistic numbers in the system we can safely assume that the HBT algorithm is able to perform efficiently with a number of nodes in the hundreds. With such a system size, it is likely that scalability issues would rather be of concern for other parts of the system first – the communication subsystem comes to mind.

6 Compared Performance Evaluation

To show of the effectiveness of the HBT algorithm we compared its performance with that of the SSI algorithm described in Section 1.1. We have chosen the SSI algorithm because it appears to be one of the most effective among those with the same characteristics (dynamic load sharing with both preemptive and non-preemptive transfers, and memory of system status). As we did before we used a system composed of 40 nodes, exponentially distributed task processing time and Poisson generators, but this time we assigned different values to task transfer overhead and latency. Namely we used $\eta_m = \eta_{t-np} = .001$, $\gamma_m = \gamma_{t-np} = .001$, $\eta_{t-p} = .03$ and $\gamma_{t-p} = .01$. Overhead and latency for preemptive task transfer have been chosen to be larger than those used for non-preemptive transfers in order to evaluate the worthiness of preemptive transfers.

Figure 15 shows the average task response time versus the offered system load (ρ) for HBT and SSI algorithms.

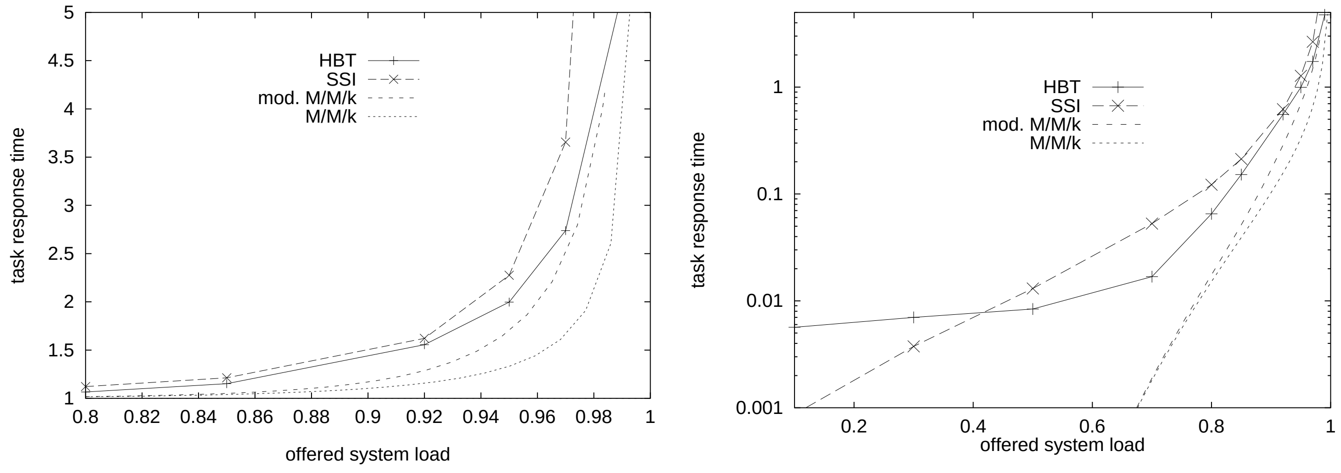

Figure 15: Average response time versus offered system load per node (ρ) for different algorithms, $N = 40$, $\eta_m = \eta_{t-np} = .001$, $\eta_{t-p} = .03$, $\gamma_m = \gamma_{t-np} = .001$, $\gamma_{t-p} = .01$.

In order to have an idea of the absolute performances of algorithms, figure 15 also shows the behavior of the ideal M/M/k system and that of a modified M/M/k (M/M/k_{mod}). M/M/k_{mod} takes into account transfer overheads for those tasks that have to be necessarily moved. It is based on a similar model proposed by Eager et al. in [6]. M/M/k_{mod} is obtained introducing in M/M/k the task transfer cost, as an increase in task processing time. To do that, we evaluate the minimum number of task transfers required to perform load sharing regardless the algorithm used. As pointed out in [10], a task must be necessarily transferred when a completion occurs in a system having more than k task (receiver-initiated transfer), and when a new task arrive in a system with less than k task (sender-initiated transfer). Thus we can evaluate the receiver and sender-initiated minimum transfer rate (λ_R, λ_S) as:

$$\lambda_R = \sum_{i=1}^{k-1} \frac{1}{S}(k-i)P[k+i] \ , \quad \lambda_S = \sum_{i=1}^{k-1} \lambda i P[i]$$

where $P[i]$ is the probability that there are i tasks in the system, obtained from the analytic results available for M/M/k. After evaluating λ_R and λ_S, we use them to compute the transfer overhead to be used as an increase in processing time S.

$$S^{'} = S + (2\eta_{t-p} + \gamma_{t-p})\lambda_R + (2\eta_{t-np} + \gamma_{t-np})\lambda_S$$

As λ_R and λ_S depend on S and vice-versa, some iteration are used to compute the final values with the needed accuracy degree. Finally S is used to compute the average task service time.

Note that the HBT algorithm approaches very close the M/M/k_{mod} system at high workload, that is an indication of its effective and light weighted behavior. At medium-high system load

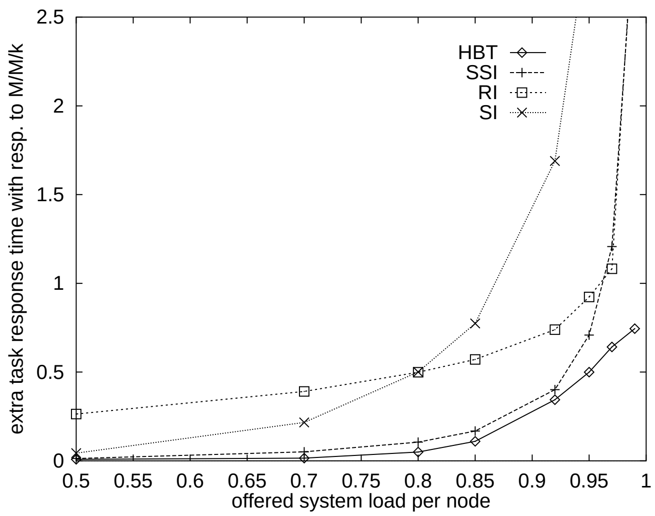

Figure 16: Normalized extra response time with respect to M/M/k versus offered system load per node (ρ), $N = 40$, $\eta_m = \eta_{t-np} = .001$, $\eta_{t-p} = .03$, $\gamma_m = \gamma_{t-np} = .001$, $\gamma_{t-p} = .01$.

($\rho = .85 \div .95$), either the HBT and the SSI algorithms appreciably move away from M/M/k_{mod}, that is due to the large protocol overhead present in both the algorithms at such workload level.

Figure 16 shows how the performance of algorithms deviate from the ideal case. It plots the extra response time of HBT, SSI, SI and RI algorithms with respect to the ideal M/M/k case, normalized to its value. We can notice that SI, first, and RI and SSI, after, asymptotically deviate from the ideal case as the workload exceeds some amount. SI early saturation is due to its already noted instability, whereas saturation of SI and SSI is due to the overhead related to the great number of preemptive task transfers triggered at high system load. HBT algorithm, thanks to its double adaptive behavior, does not presents this problem yet at the computed load value of .99. Actually, as the workload increases, it switches to the ticket component that performs with small overhead and moves task non-preemptively.

6.1 Bursty workload

The Poisson process is statistically well known and, sometimes, allows to obtain an analytical prediction of algorithm performance, once a model for the system is devised; moreover it is something like a standard de facto in the literature that allows authors to compare their results with the ones obtained by their predecessors. In spite of its advantages, the Poisson process, being very far from the actual workload a real system might experience, is not able to test some characteristics of algorithms, that might be important in their actual usage. These characteristics are, for example, the ability of the system to react to load peaks and its capability to spread workloads applied in a non uniformly distributed way (time and space workload fluctuations). Workload peaks are very common in workstations. We may think, for example, of a programmer, developing an application, that usually edits source files (CPU non-intensive task), and, every so often, runs the make command, submitting a number of compilations (CPU intensive tasks). To have an idea of the behavior of algorithms in response to bursty workload we first observed the response of the system to a single workload peak, then we evaluate its response to a IPP independent workload applied in all nodes.

Figure 17 shows the protocol messages exchanged versus the time, as consequence of the load distributing activity in the case of HBT and SSI algorithms, when a single burst of 800 tasks is applied on 20 nodes of the system. The amount of messages exchanged gives a quite accurate idea of the protocol overhead. The most important thing to note in figure 17 is that HBT generates much less messages than SSI. Both HBT and SSI curves presents two main steps: the first one is due to the initial task placement and the second one to the task redistribution. HBT involves about 240 messages `(2*40*(T_REQ+T_CND) + 40*(T_REQ+T_BOH))` to place two tasks in each node, and to make each node realize that the system is heavy loaded. After that, HBT, working with the ticket algorithm, does not generate additional messages until a final task redistribution is needed. On the contrary SSI needs much more messages to perform the initial placements, and keeps generating messages for the whole execution time.

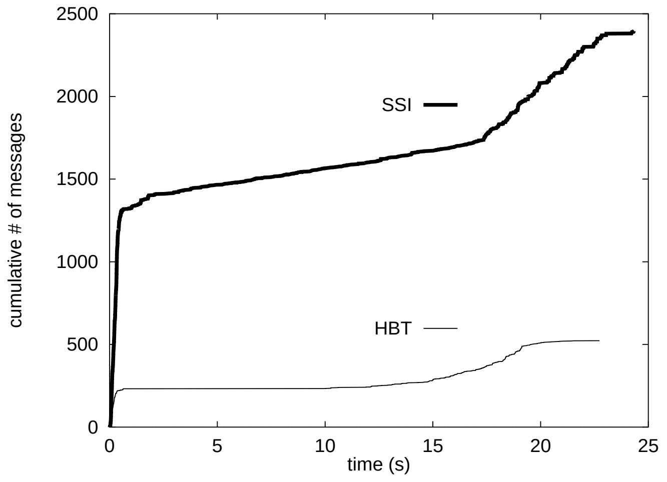

Figure 17: Number of protocol messages versus time, in response to a workload peak applied to 20 nodes (800 tasks altogether) $N = 40$, $\eta_m = \eta_{t-np} = .001$, $\eta_{t-p} = .03$, $\gamma_m = \gamma_{t-np} = .001$, $\gamma_{t-p} = .01$.

An alternative model for characterizing the offered system workload is an interrupted Poisson process (IPP) applied in each node of the system. An IPP is a Poisson process which is alternately turned on for an exponentially distributed length of time (with mean T_{on}) and then turned off an other exponentially distributed time (with mean T_{off}) [7]. For our simulations we have chosen the IPP total period ($T_p = T_{on} + T_{off}$) to be 1000 times the mean task service time S and the IPP-on generation rate $\lambda_{on} = \lambda/\Delta$, being $\Delta = T_{on}/(T_{on} + T_{off})$ the average IPP duty-cycle. This way the average generation rate per node is still λ regardless the actual value of Δ.

Figure 18 shows the average task response time versus the average peak duty-cycle Δ for a fixed offered workload $\rho = .8$. The lower curve is the response time of the ideal MMPP/M/k system whose input process (MMPP) is obtained from the superposition of all the IPPs at nodes [7]. As we can see the response time of HBT algorithm is essentially that of the ideal case plus a (nearly constant) overhead. SSI algorithm presents a slowly increasing overhead as the duty-cycle decreases, while RI and SI quickly diverge from the ideal behavior. That might be explained as follows: using IPP with small duty-cycle, nodes experiences either quite long inactivity periods and periods in which they are highly loaded (during T_{on} IPP generates at much higher rate than a Poisson process with the same average rate does), thus they are well suitable to be characterized either as heavily loaded or as lightly loaded; HBT and SSI algorithms, that keep memory of the node status, can exploit that to address better the searches for suitable nodes. The better performance of HBT algorithm gives evidence of the effectiveness of the centralized status information, that can be kept up to date even in the face of quickly varying system condition. As a matter of fact distributed information kept at nodes in the SSI algorithm is updated only when a node polls an other node or when it is polled, whereas information kept in the broker is updated each time a node contacts it.

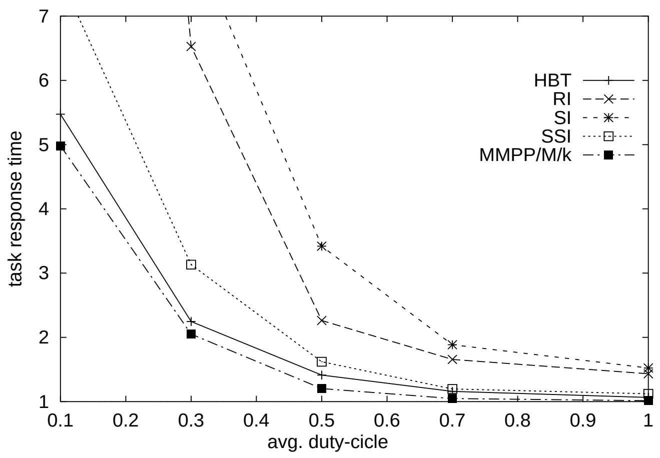

Figure 18: Average task response time versus average peak duty-cycle (Δ) with IPP input, $\rho = .8$, $N = 40$, $\eta_m = \eta_{t-np} = .001$, $\eta_{t-p} = .03$, $\gamma_m = \gamma_{t-np} = .001$, $\gamma_{t-p} = .01$.

7 Conclusions

A load sharing algorithm must be simple, in order not to consume otherwise useful processing power. It must be effective, particularly when parts of the system tend to become idle. Operation in a distributed environment must also consider that communication has a cost and a latency, so that it must be minimized and possibly moved out of the critical paths of algorithms.

We have described a hybrid algorithm for adaptive load sharing on a distributed system which fulfills both requirements. Good performance at high loads is achieved by a fully distributed task placement algorithm, while at low system loads a centrally coordinated location policy guarantees an efficient use of idle nodes. The transition between the two ways of operation occurs smoothly and adaptively as the system's load changes. Simulations show that the hybrid scheme performs well under all load conditions and task generation patterns, it is weakly sensitive to communication delays (which is an interesting feature when the processing nodes are not tightly connected), provides reasonable scalability under realistic system sizes, and does not present instabilities at all load conditions. Among the interesting features of our approach there are the extreme simplicity of algorithms and the absence of tunable parameters, which make the load sharing strategy extremely easy to implement and use.

References

[1] M.Avvenuti, L.Rizzo, and L.Vicisano, "Hardware support for load sharing in parallel systems", *Journal of Systems Architecture*, (to appear).

[2] T.L.Casavant, and J.G.Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems", *IEEE Trans. on Software Engineering*, Vol 14, No 11, pp.1578-1587, Nov 1988.

[3] R.B.Cooper, *Introduction to Queuing Theory, 2^{nd} ed.*, London, E. Arnold, 1981.

[4] *ES2 ECPD10 Library Databook*, European Silicon Structures Inc., 1994.

[5] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Performance Evaluation*, Vol 6, No 1, pp.53-68, March 1986.

[6] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Trans. on Software Engineering*, Vol 12, No 5, pp.662-675, May 1986.

[7] W.Fischer and K.Meire-Hellstern, "The Markov-modulated Poisson process (MMPP) cookbook", *Performance Evaluation North-Holland*, Vol 18, No 2, pp 149-171, 1993.

[8] P. Krueger, and M.Livny, "The Diverse Objectives of Distributed Scheduling Policies", *Proc. of the 7th International Conference on Distributed Computing Systems*, pp.242-249, Sept 1987.

[9] T.Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme", *IEEE Trans. on Software Engineering*, Vol 17, No 7, pp.725-730, July 1991.

[10] M.Livny, and M.Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems", *ACM Performance Evaluation Review*, Vol 11, No 1, pp.47-55, July 1982.

[11] R. Mirchandaney, D.Towsley, and J.A.Stankovic, "Adaptive load sharing in heterogeneous distributed systems", *J. Parallel & Distributed Computing*, Vol 9, No 4, pp.331-346, Aug 1990.

[12] L. Rizzo, "Simulation and performance evaluation of parallel software on multiprocessor systems", *Microprocessors & Microsystems*, Vol 13, No 1, pp.39-46, Jan/Feb 1989.

[13] C.G. Rommel, "The Probability of Load Balancing Success in a Homogeneous Network", *IEEE Trans. on Software Engineering*, Vol 17, No 9, pp.922-933, Sept 1991.

[14] N.G. Shivaratri, and P. Krueger, "Two Adaptive Location Policies for Global Scheduling", *Proc. 10th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, pp.502-509, 1990.

[15] N.G. Shivaratri, P. Krueger, M. Singhal, "Load Distributing for Locally Distributed Systems", *IEEE Computer*, Vol 25, No 12, pp.33-45, Dec 1992.

[16] M.Singhal, N.G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw-Hill, 1994, Chap. 11.

[17] Y.Wang, R.J.T.Morris, "Load Sharing in Distributed Systems", *IEEE Trans. on Computers*, Vol C-34, No 3, pp.204-217, March 1985.

[18] S. Zhou, "An Experimental Assessment of Resource Queue Lengths as Load Indices", *Proc. of the 1987 Winter USENIX Conference*, Washington, D.C., pp.73-82, Jan 1987.

```
PROC broker() {
    put all nodes into IDLE queue
    do {
        receive a message from node s
        switch (message-type) {

        case T_BSY:
            move s to BUSY queue
            break;

        case T_REQ:
            if (IDLE queue not empty) {
                if (node s is in IDLE queue)
                    candidate = s;
                else
                    candidate = tail node in IDLE queue;
                move candidate node to BUSY queue
                send T_CND to node s
            } else if (BUSY queue not empty) {
                if (node s is in BUSY queue)
                    candidate= s;
                else
                    candidate = head node in BUSY queue;
                move candidate node to LOADED queue
                send T_CND to node s
            } else
                send T_BOH to node s
            break;

        case T_IDL:
            if (LOADED queue is empty)
                move node s to IDLE queue
            else {
                move node s to BUSY queue
                candidate = head node of LOADED queue;
                move candidate to the tail of LOADED queue
                send T_TRY to candidate
            }
            break;

        case T_NWK: /* s' is the originator of T_IDL */
            if (node s is in LOADED queue)
                move node s to BUSY queue
            if (LOADED queue is empty)
                move node s' to IDLE queue
            else {
                candidate = head node of LOADED queue;
                move candidate to the tail of LOADED queue
                send T_TRY to the candidate
            }
            break;
        } /* end switch */
    } while (TRUE);
}
```

Figure 19: The algorithm run by the broker.

```
PROC manager() {
    call_broker = TRUE; pending = NULL;
    initialize spent_ticket vector to 0
    do {
        receive message from node s
        switch (message-type) {

        case T_BOH: /* broker has no answer */
            call_broker = FALSE;
            behave as if T_SRC had arrived

        case T_SRC: /* brand new task from generator */
            if (pending != NULL) {
                send pending task to local server;
                call_broker = FALSE;
            }
            if (call_broker) {
                pending = current task;
                send T_REQ to broker
            } else {
                choose a candidate basing on ticket vector
                spent_ticket[candidate]++;
                send T_WRK message to candidate
            }
            break;

        case T_CND:
            if (call_broker) {
                spent_ticket[candidate]++;
                send T_WRK message to candidate
            } /* otherwise ignore the (late) reply */
            pending = NULL;
            break;

        case T_WRK:
            queue_len++;
            if (queue_len == 1) {
                send task to local server /* which was idle */
                send T_BSY to broker
            } else
                enqueue task on local task queue
            break;

        case T_DNE: /* local server returns a task done */
            queue_len--;
            if (queue_len > 0)
                send new task to local server
            else { /* local server is now idle */
                call_broker = TRUE;
                send T_IDL to broker
            }
            send T_TKT to the originator of the task
            break;

        case T_TKT: /* remote server returns a ticket */
            spent_ticket[remote_server]--;
            break;

        case T_TRY: /* broker requests a task */
            call_broker = TRUE;
            if (queue_len > 1) {
                take first task from queue
                queue_len--;
                send T_WRK to requestor
            } else
                reply T_NWK to broker
            break;
        } /* end switch */
    } while (TRUE);
}
```

Figure 20: The algorithm run by the manager.