Implementation of Decentralized Load Sharing in Networked Workstations Using the Condor Package[1]

Chao–Ju Hou

Computer Engineering Division
Dept. of Elect. and Comp. Eng.
The University of Wisconsin
Madison, WI 53706-1691
jhou@ece.wisc.edu

Kang G. Shin

Real-Time Computing Laboratory
Dept. of Elect. Eng. and Comp. Science
The University of Michigan
Ann Arbor, MI 48109-2122
{*kgshin,thomast*}*@eecs.umich.edu*

ABSTRACT

In recent years a number of load sharing (LS) mechanisms have been proposed or implemented to fully utilize system resources. We have designed and implemented a decentralized real-time LS mechanism based on the Condor package [17, 18]. Two important features of our design are use of region-change broadcasts in the information policy to provide each workstation with timely state information at minimum communication cost, and use of preferred lists in the location policy to avoid task collisions. With these two features, we remove the central manager workstation in Condor, configure its functionalities into each participating workstation, transform Condor into a decentralized LS mechanism, and equip Condor with the capability to tolerate single workstation failures. Also discussed are the experiments on the proposed LS mechanism and the off-the-shelf Condor package and our observations of empirical data.

Index Terms: distributed systems, adaptive load sharing, region-change broadcasts, preferred lists, performance evaluation.

[1]The work reported in this paper was supported in part by the National Science Foundation under Grant MIP–9203895, and the Office of Naval Research under Grants N00014–94–1–0229 and N00014–91–J–1226. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

1 Introduction

The availability of inexpensive, high-performance processors and memory chips has spurred considerable interests in using a network of workstations for a wide range of applications. However, since tasks may arrive unevenly and randomly at these workstations and/or computation power may vary from workstation to workstation, some workstations may get overloaded while others are left idle or under-loaded. Livny and Melman [1] showed that in a network of autonomous workstations, with large probability, at least one workstation is idle while many jobs are being queued at other workstations. Consequently, some jobs may suffer an extremely long response time while the system capacity is being under-utilized. Thus, an effective "load sharing" (LS) method is needed to enable idle/under-loaded workstations to share the loads of over-loaded ones.

As was discussed in [2, 3], a LS mechanism can be designed by developing the *transfer policy* which determines when to transfer a job, the *information policy* which determines how workstations communicate with one another to exchange state information [4, 5, 6, 7, 8, 9], and the *location policy* which determines where to transfer the job [10, 11, 2, 12, 4, 6, 13, 7]. On the other hand, the implementation issues commonly considered include where to place the LS mechanism (i.e., inside or outside the OS kernel), how to transfer process state (virtual memory, open files, process control blocks) during job transfer/migration, how to support LS transparency, and how to reduce the effect of residual dependency[2] [14]. A few LS mechanisms have been implemented, such as the V-system [15], the Sprite OS [14], the Charlotte OS [16], and the Condor software package [17, 18]. They are designed using different policies for transferring jobs/processes, collecting workload statistics used for LS decisions, and locating target workstations. Also, they are implemented using different strategies to detach a migrant process from its source environment, transfer it with its context (the per-process data structures held in the kernel), and attach it to a new environment on the destination workstation.

In this paper, we design and implement, based on the Condor software package, a decentralized LS mechanism with each LS policy carefully re-designed. As reported in [17, 18], Condor is a software package for executing long-running tasks on workstations which would otherwise be idle. It is designed for a workstation environment in which the workstation's resources are guaranteed to be available to the owner of the workstation. The reason for choosing Condor as our "base system" is because Condor is implemented entirely outside the OS kernel and at the user level. This eliminates the need to access/change the internals of OS. On the other hand, there are several design drawbacks of Condor: it uses a central manager workstation to allocate queued tasks to idle workstations. That is, the location policy is realized by the central manager. This centralized component makes the LS mechanism susceptible to single workstation failures. Another drawback is that Condor uses a periodic information policy; that is, each workstation reports periodically to the central manager

[2]Residual dependency is defined as the need for the source workstation to maintain data structures or provide functionality for a remote process.

regarding its (workload) state and task-queue status. This makes the central manager a potential bottleneck of network traffic from time to time. Determination of a reporting period also becomes crucial to the LS performance, and has to make a tradeoff between the communication overhead of frequent reporting and the possibility of using out-of-date state information resulting from infrequent reporting. As a result, we decided to enhance performance and reliability by configuring and dispatching the functions of the central manager workstation to all the participating workstations, and "transforming" Condor to a decentralized LS mechanism.

Two important design issues must be considered in achieving the above goal. First, each workstation has to collect/maintain elaborate and timely state information *on its own* in the decentralized mechanism, and hence, a policy is needed to provide each workstation with updated state information at minimum communication overhead. Second, each workstation has to determine, for each task, the best target workstation if there are several workstations available, and more importantly, each workstation has to reduce the possibility of multiple workstations sending their tasks to the same idle workstation. We deal with the former issue by using *region-change broadcasts* as the information policy, and the latter issue by using the *preferred lists* in our location policy. Both strategies are detailed in Section 3.

The rest of the paper is organized as follows. In Section 2, we give an overview of Condor software package and discuss how Condor daemons collaborate to manage the task queue and locates target idle workstations. In Section 3, we present our decentralized LS mechanism. In particular, we discuss the transfer, information, and location policies used in our LS mechanism. Then, we discuss how to get rid of the central manager by reconfiguring Condor component daemons. In Section 4, we highlight the implementation features adopted in the decentralized mechanism. In Section 5, we present experimental measurements and Section 6 discusses alternative design approaches used in other existing LS mechanisms. This paper concludes with Section 7.

2 Overview of Condor Software Package

In this section, we summarize the functionality of, and the interactions among, Condor's daemons. Especially, the task distribution process is described in a step–by–step manner.

As shown in Fig. 1, there are two daemons, Negotiator and Collector, running on the central manager workstation. In addition, there are two other daemons, Schedd and Startd, running on each participating workstation. Whenever a task is executed, two additional processes, Shadow and Starter, shall run on the submitting workstation and on the executing workstation, respectively (whether or not these two workstations are actually identical).

The Condor task relocation mechanism works as follows (Fig. 2). A user invokes a submit program to submit a task. The submit program takes the task description file, constructs the corre-

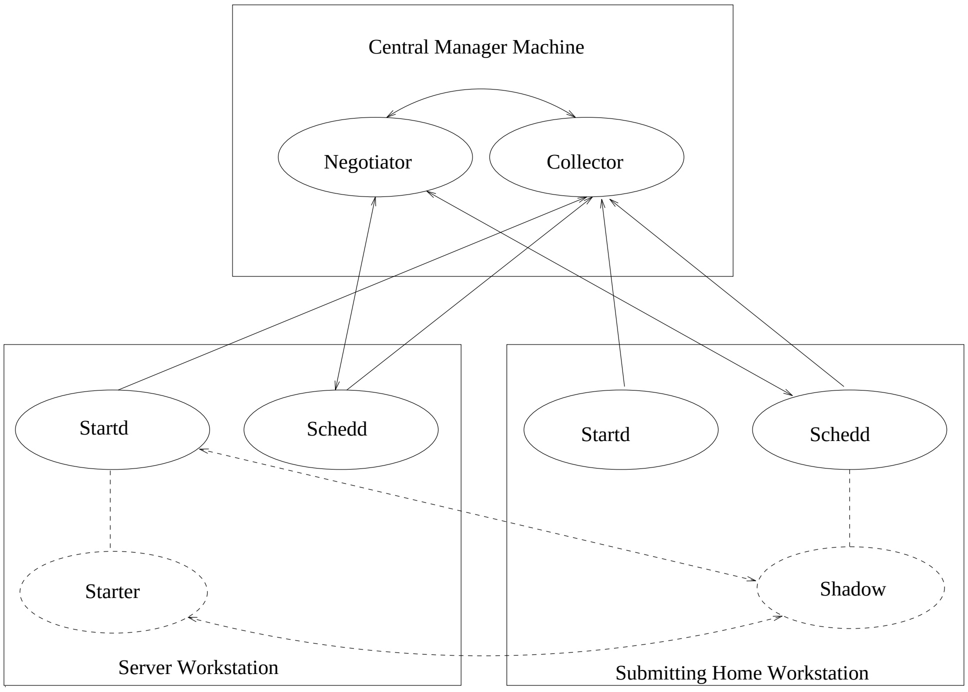


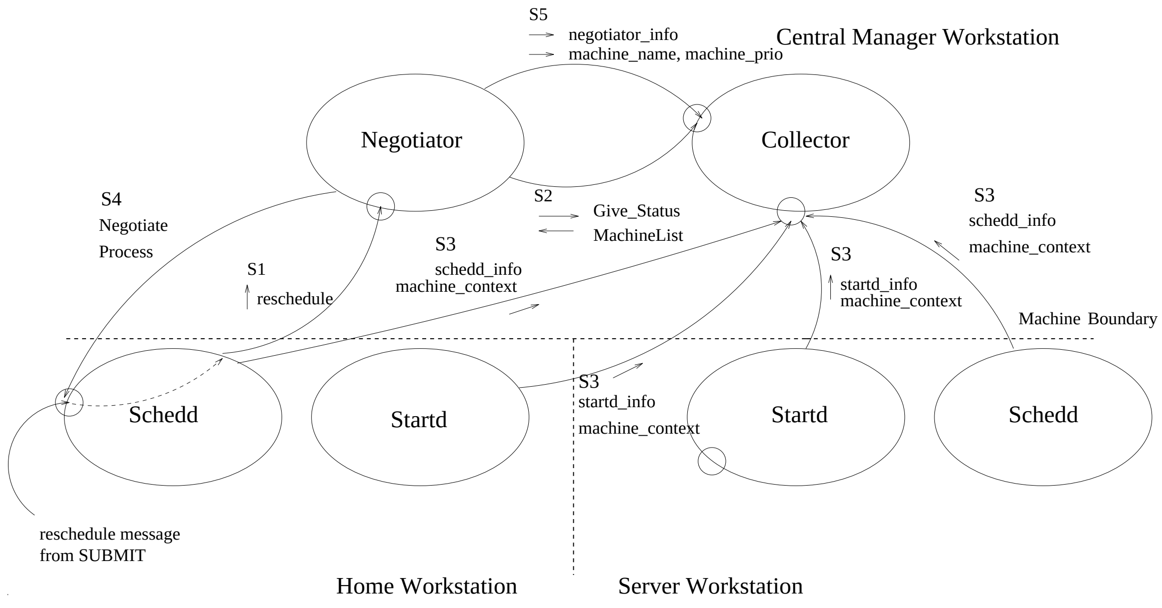
Figure 1: Daemons in Condor.

sponding data structures, and sends a reschedule message to Schedd on the home workstation. Schedd then asks Negotiator on the central manager workstation to relocate tasks to idle workstations by sending a reschedule message to Negotiator (**S1** in Fig. 2).

Upon receiving a reschedule message from any of Schedds on the participating workstations, or upon periodic schedule timeout, Negotiator gets from Collector a list of machine records which contains the workload and task queue of all participating workstations (**S2** in Fig. 2). Collector receives periodically from Schedd and Startd on each participating workstation updated information of task queue and workload, respectively (**S3** in Fig. 2), and updates accordingly its list of machine records.

After receiving the list of machine records, Negotiator first prioritizes the participating workstations: the priority of a workstation is incremented by the number of individual users with tasks queued on that workstation, and decremented by the number of tasks which are submitted to that workstation and are currently running (either remotely or locally). Negotiator then contacts each workstation with queued tasks, one at a time, starting with the workstation with the highest priority, and inquires to relocate the task(s) queued on the workstation. If the swap space on the workstation being inquired is enough for Shadow processes[3], the workstation supplies Negotiator with the

[3]As will be discussed below, each executing task will have associated Shadow processes running on the home workstation.



(a) Negotiation process

Figure 2: Interactions among Condor daemons.

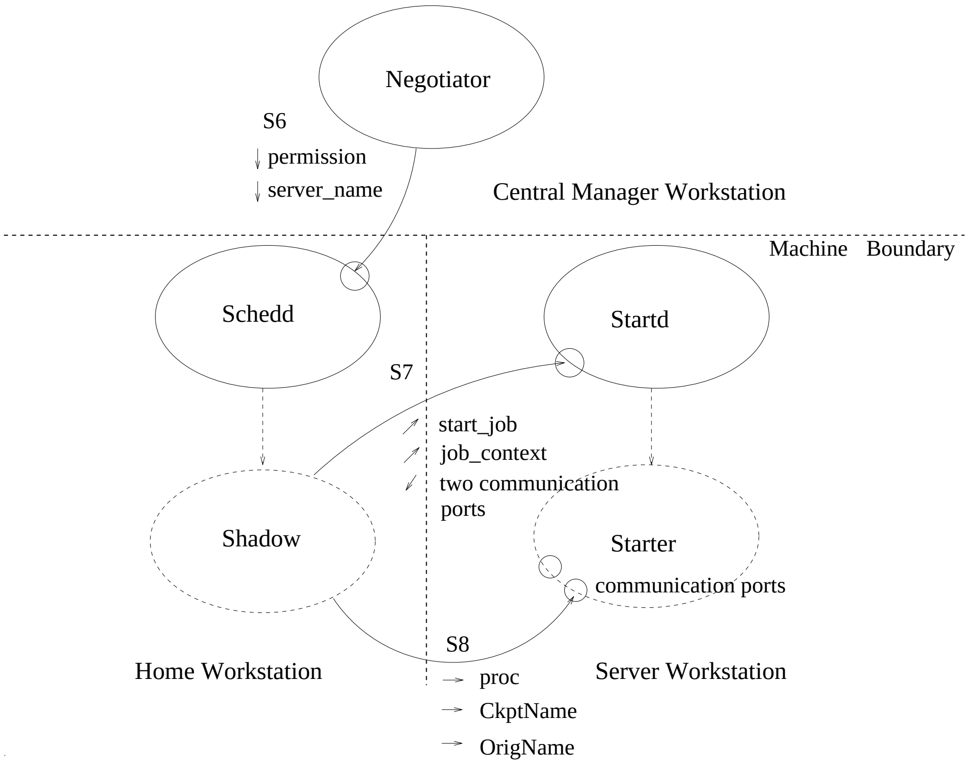
information on the required OS, architecture, and the size of a queued task, with which Negotiator finds a server workstation for the task. A workstation is qualified as a server if (i) both its CPU and keyboards are idle; (ii) it satisfies the task requirement specified; and (iii) no other task is currently running on it. The negotiation process will be repeated for each queued task[4] until either Negotiator finds for all queued tasks their server workstations, or no server can be located (**S4** in Fig. 2). At the end of the negotiation process, Negotiator sends back the updated record of machine priorities to Collector (**S5** in Fig. 2).

For each of the servers located, the task transfer process is collaborated among (a) Negotiator on the central manager workstation, (b) Schedd and Shadow process on the home workstation, and (c) Startd and Starter on the server workstation in the following steps: Negotiator sends a permission message followed by the name of the server workstation to Schedd on the home workstation (**S6** in Fig. 2). Schedd on the home workstation then spawns off a Shadow process which connects to Startd on the located server workstation (**S7** in Fig. 2) and will henceforth take care of remote system calls[5] from the server workstation.

Startd on the server workstation, upon being notified by Shadow on the home workstation

[4]The tasks in a local queue are also prioritized with respect to the user-specified priority and the order in which they are queued.

[5]More on remote system calls will be elaborated in Section 4.



(b) Task transfer process

Figure 2: (continued) Interactions among Condor daemons.

of the task transfer decision, re-evaluates its workload situation and memory space available. If the situation has not changed since the last time Startd reported to the central manager, Startd creates two communication ports, then sends the port numbers back to Shadow. Shadow acknowledges the receipt of the port numbers. Startd then spawns off a Starter process (which inherits these communication ports and is responsible for executing the task), and notifies Collector on the central manager of the workload change in the server workstation. Startd henceforth keeps track of **Task_state** of Starter, and signals Starter to suspend, checkpoint, or vacate the executing process whenever necessary to ensure that workstation owners have the workstation resources at their disposal.

For example, if during the execution of a task (i.e., **Task_state** is TaskRunning), and if either the average workload increases (e.g., **AvgLoad** > 0.3) or the workstation owner returns (e.g., **KeyboardIdle** < 5 minutes), then a SIGUSR1 (suspend) signal is sent to Starter, **Task_state** enters the Suspend state, and Starter will temporarily suspend the task. If the task has been suspended for more than a certain period (e.g., 10 minutes), a SIGTSTP (vacate) signal is sent to Starter, Startd enters the **Checkpointing** state, and Starter will abort the task and return the latest checkpoint

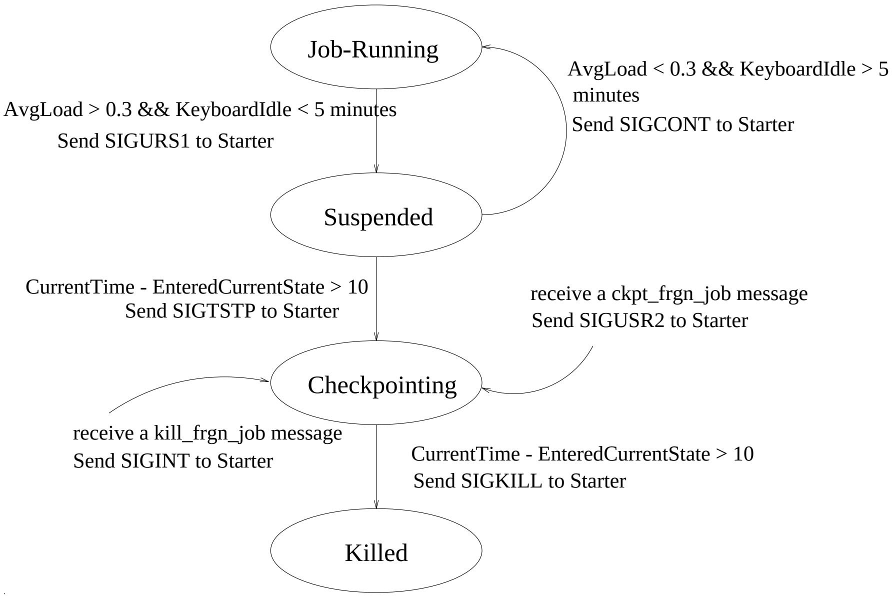


Figure 3: **Task_state** transition process.

file to Shadow on the home workstation. Fig. 3 gives a complete description of how Startd keeps track of the execution status of Starter and the associated **Task_state** transition process.

The newly–spawned Starter is responsible for (a) getting the executable[6] and other relevant process information from Shadow via either NFS or RPC whichever available and spawning off a child process to execute the task; (b) communicating (via remote system calls) with Shadow on the home workstation for environments/devices–related operations; and (c) suspending, resuming, or checkpointing the executing process upon being requested by Startd (Fig. 3). Both Starter and Shadow exit when the task completes/stops execution.

3 Design of a Distributed Mechanism Based on Condor

As mentioned in Section 1, there are several design drawbacks in Condor:

- The central manager component makes Condor susceptible to a single-workstation crash;
- The information policy periodically invoked introduces a potential traffic bottleneck while suffering the effect of using out-of-date state information if the report period is not fine-tuned;
- The location policy is so designed that it is possible for a task arrived at an idle workstation to be transferred to other idle workstations for execution (Section 2), since the central manager

[6]which is itself a checkpoint file without stack information.

takes the full responsibility of locating a server workstation.

To remedy the above deficiencies, we eliminate the central manager, and configure the functionality of Negotiator and Collector into every participating workstation. Specifically, each participating workstation collects and maintains state information on its own. Moreover, if a workstation is not idle upon arrival of a task, it chooses the best server workstation among several candidate workstations, and coordinates with other workstations to reduce the probability of multiple workstations sending their tasks to the same idle workstation and to distribute tasks as evenly as possible in the system.

3.1 LS Policies Used

We now discuss how to incorporate our proposed LS policies into Condor to achieve the above objectives.

Transfer Policy: Upon submission/arrival of a task, Schedd on the home workstation determines whether or not the task can be executed locally. That is, the transfer policy is invoked upon arrival of a task, and hence a task transfer, if it ever takes place, will occur during an `exec` system call and the new address space will be created on the server workstation. This significantly reduces the process state necessary to be transferred. A task is executed locally on the home workstation if **AvgLoad** is less than or equal to 0.3 and the **KeyboardIdle** time (the smallest keyboard idle time observed among all terminals) is greater than 15 minutes, and no other tasks are currently running on the workstation. If the task cannot be executed locally, a transfer decision is made and the location policy is invoked to select a server workstation, if possible, for the task. Also, the workstation rescans its task queue periodically, treats each queued task[7] as if it were newly-arrived, and repeats the transfer policy.

Information Policy: The state space is divided into several regions, and a workstation broadcasts a message, informing all the other workstations of the state change whenever its state switches from one region to another. In contrast to a periodic information policy, a region-change broadcast occurs only when the state of a workstation changes significantly, thus reducing the communication overhead. Moreover, the state information kept at each workstation is more likely to be up–to–date. The state defined in our current version is the combination of three quantities: **AvgLoad**, **KeyboardIdle**, and the **State** (NoTask, TaskRunning, Suspended, Vacating, or Killed) of the workstation. The state space for the current implementation is divided into two state regions: runnable and unrunnable. The workstation is said to be in the runnable state region if **AvgLoad** ≤ 0.3, **KeyboardIdle** > 15 minutes, and **State** is NoTask. Extension to multiple state regions is straightforward.

[7]The task which fails to locate a server workstation at the time of its arrival.

Order of preference	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
node 0	1	2	4	8	6	10	12	3	5	9	14	13	11	7	15
node 1	0	3	5	9	7	11	13	2	4	8	15	12	10	6	14
node 2	3	0	6	10	4	8	14	1	7	11	12	15	9	5	13
node 3	2	1	7	11	5	9	15	0	6	10	13	14	8	4	12
node 4	5	6	0	12	2	14	8	7	1	13	10	9	15	3	11
node 5	4	7	1	13	3	15	9	6	0	12	11	8	14	2	10
node 6	7	4	2	14	0	12	10	5	3	15	8	11	13	1	9
node 7	6	5	3	15	1	13	11	4	2	14	9	10	12	0	8
node 8	9	10	12	0	14	2	4	11	13	1	6	5	3	15	7
node 9	8	11	13	1	15	3	5	10	12	0	7	4	2	14	6
node 10	11	8	14	2	12	0	6	9	15	3	4	7	1	13	5
node 11	10	9	15	3	13	1	7	8	14	2	5	6	0	12	4
node 12	13	14	8	4	10	6	0	15	9	5	2	1	7	11	3
node 13	12	15	9	5	11	7	1	14	8	4	3	0	6	10	2
node 14	15	12	10	6	8	4	2	13	11	7	0	3	5	9	1
node 15	14	13	11	7	9	5	3	12	10	6	1	2	4	8	0

Figure 4: Preferred lists in a 4–cube system.

Location Policy: Based on the topological property of the system, each workstation orders all the other workstations into a *preferred list* [8]:

P1. A workstation is the k-th preferred workstation of *one and only one other* workstation, where k is some integer.

P2. If workstation i is the k-th preferred node of workstation j, then workstation j is also the k-th preferred node of workstation i.

For example, Fig. 4 gives the preferred lists in a 4–cube system. When a workstation is unable to execute a task, it will contact the first "runnable workstation" found in its preferred list, and tries to transfer the task to that workstation. It is important to note that although the preferred list of each workstation is generated *statically*, the actual preference of the workstation in transferring a task may change dynamically with the states of workstations in its preferred list. If a workstation's most preferred workstation becomes unrunnable, this fact will be known to the workstation via a state-region change broadcast and its second preferred workstation will become the most preferred. (This workstation will change back to the second most preferred whenever the original most preferred becomes runnable, which will again be informed via a state-change broadcast.)

The preferred list provides each workstation with a means of selecting a server workstation among several runnable candidates. Moreover, using (**P1**) and (**P2**), we formally proved in [19] that the probability of more than one node simultaneously sending their tasks to the same node is minimized, and transferred tasks are evenly distributed among workstations.

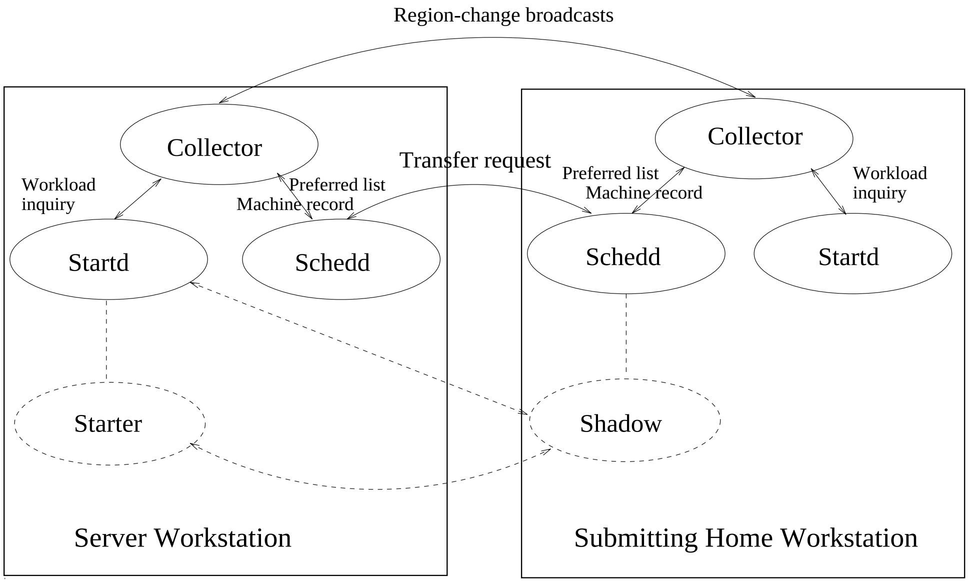


Figure 5: Daemons in Modified Condor.

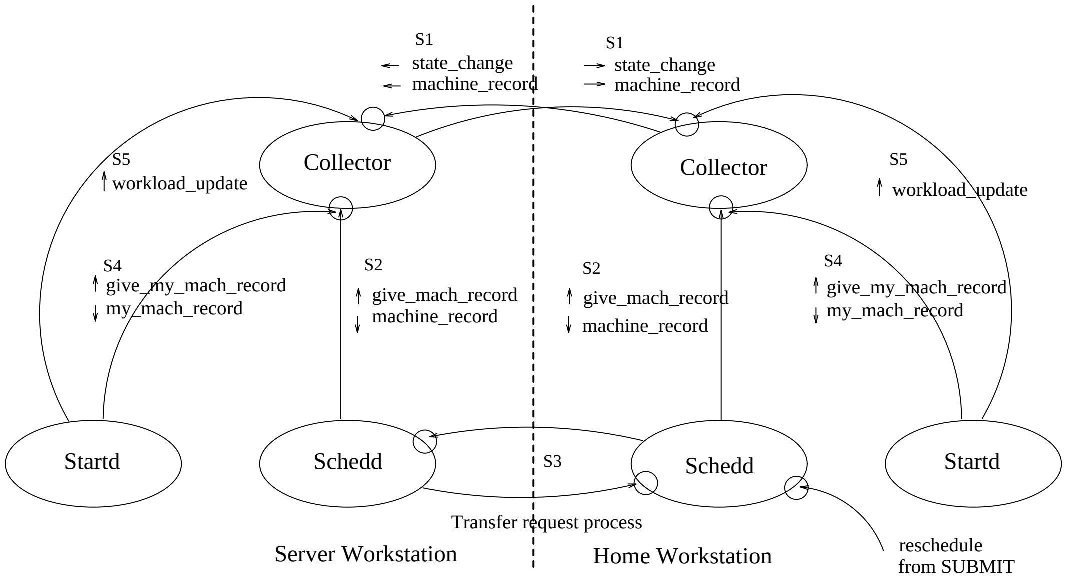
3.2 Daemon Configuration

We come up with three daemons, Collector, Schedd, and Startd, which reside constantly on each participating workstation for the decentralized LS mechanism (Fig. 5). Similarly as in Condor, two additional processes, Shadow and Starter, run on the home workstation and on the server workstation whenever a task is executed. Note that we carefully configure the transfer, information, and location policies only into Schedd, Startd, and Collector, and leave Shadow and Starter which deal with process transfer, execution, and checkpoint unchanged for the distributed LS mechanism. The functionality of, and the interactions among, daemons are depicted in Fig. 6, and are described below.

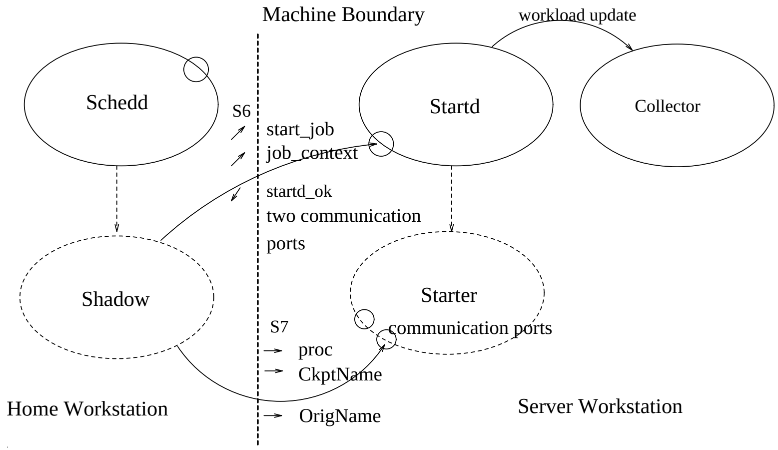
3.2.1 Collector

Collector is responsible for gathering local workload information, broadcasting a region-change message whenever necessary, updating the workload information of other workstations in its preferred list upon receiving a broadcast message, and responding to Schedd and Startd for information inquiries. The pseudo code for Collector is listed in Fig. 7.

The local task queue, the average workload (in terms of **AvgLoad**, **KeyboardIdle**, and the **Task_state** of the workstation), and the disk/memory space available are measured upon Collector



(a) Negotiation process



(b) Task transfer process

Figure 6: Interactions among daemons in the distributed mechanism.

Collector

```
upon Collector timeout or upon receiving workload_update from Startd:
     workload_measurement(&machine_record.AvgLoad, &machine_record.KeyboardIdle,
       &machine_record.Task_state, &machine_record.machine_context.VirtualMemory,
       &machine_record.machine_context.DiskSpace);
     workload_evaluation(&Busy);
     if (Busy != machine_record.Busy)
       region_change_broadcast();
     machine_record.Busy = Busy;

upon receiving give_machine_record_for_ith_preferred_runnable_machine message from Schedd:
     send_message(machine_record_of_ith_preferred_runnable_machine,local_Schedd);

upon receiving give_my_machine_record message from Startd:
     send_message(my_machine_record,local_Startd);
```

Figure 7: Pseudo code for Collector.

timeout, or upon receiving a workload_update message from the Startd.[8] The parameters measured are then used to evaluate whether or not a workstation is runnable. A workstation is evaluated to be runnable (i.e., **Busy** = false) if the function

$$\textbf{START} : (\textbf{AvgLoad} \leq 0.3) \quad \&\& \quad \textbf{KeyboardIdle} > 15 \text{ minutes}$$

is true and **Task_state** of the workstation is NoTask.

A state-region change message is broadcast to Collectors on other workstations in the preferred list whenever the state switches from runnable to unrunnable (because of the increase in average workload, return of the workstation owner, or receipt of a task), or vice versa (**S1** in Fig. 6). The message contains, among other things,

(I1) the hostname, the network address, and the network address type,

(I2) the indicator variable of whether or not a task is runnable, **Busy**, along with other workload-related parameters, **AvgLoad**, **KeyboardIdle**, **Task_state**,

(I3) the operating system, **OpSys**, and the architecture, **Arch**, of the workstation,

(I4) the swap space, **VirtualMemory**, available in virtual memory, and the disk space, **DiskSpace**, available on the file system where foreign checkpoint files are stored. Note that **VirtualMemory** is only calculated at the time of state-change broadcasts (but not periodically at every timeout), because its calculation is expensive.

(I5) a time-stamp.

[8] When a task starts or exits/dies, the Startd notifies Collector to update workload situation.

As will become clear in the discussion of Schedd, **(I3)** is used to verify whether or not a workstation's OS and architecture satisfies the user-specified task requirements; **(I4)** is used to verify whether or not a workstation has enough memory/disk space for running foreign tasks; and **(I5)** is used to indicate the degree of a record being obsolete. Upon receiving a state-change broadcast from one of Collectors on other workstations, the machine record corresponding to the broadcasting workstation in the preferred list is updated.

There are two possible situations Schedd will ask information from Collector (**S2** in Fig. 6): (i) when Schedd receives a new task and asks for its own machine record; (ii) when Schedd decides to transfer the task and asks for the machine record of the first runnable workstation available in the preferred list. On the other hand, Startd asks Collector for the machine record which contains workload-related and memory/disk space-related parameters (**S4** in Fig. 6), when it wants to check whether a running task should be suspended, checkpointed, resumed, or vacated. (More on this will be discussed in Section 3.2.3.)

3.2.2 Schedd

Schedd determines whether or not a (local or remote) task can be executed on the workstation, and, in the case of not executing an arrived task, initiates the location policy to identify a candidate workstation for task transfer. Also, Schedd invokes the location policy periodically for tasks that did not find their servers upon their arrival and are currently queued on the workstation. The pseudo code of Schedd is listed in Fig. 8.

There are three major events Schedd handles: the arrival of a task, the receipt of a transfer request, and the periodic timeout:

Upon arrival of a task: upon receiving a reschedule message from the submit program, Schedd gets the local machine record from Collector (**S2** in Fig. 6), evaluates the parameter **Busy**, and checks whether or not the task requirement is satisfied. The task requirement includes the system configuration and the disk/memory space needed for executing the task.

If the task can be executed locally, a Shadow process is spawned off which contacts the local Startd. Startd then creates two communication ports, sends the port numbers back to Shadow, and spawns off a Starter. Here Starter inherits the two communication ports and will actually fork off a child process to execute the task. Shadow and Starter then communicate through the communication ports, and the task execution/checkpoint process proceeds as in Condor. Note that by carefully "reconfiguring" the daemons, we leave the "low-level" implementation mechanism (for task transfer and checkpointing) unchanged in the distributed LS mechanism.

If the task cannot be executed locally (either **Busy** is true, or the task requirement is not satisfied), then Schedd checks if there is enough swap space for a new Shadow process. If the swap

Schedd

```
upon arrival of a task (i.e., receiving a reschedule message):
     /* the 0-th machine record is the record for the local machine */
     get_machine_record(local_Collector, 0th, &machine_record);
     requirement_satisfied = evaluate_task_requirement(machine_record);
     if (requirement_satisfied)
        forks_off_Shadow_process();
     else { /* task must be remotely executed */
        if (!enough_swap_space())
           queue_task();
        else { /* enough swap space */
           preferred_list_not_exhausted = true;
           server_not_found = true; i = 1;
           while (preferred_list_not_exhausted && server_not_found) {
                 get_machine_record(local_Collector, ith, &machine_record);
                 if (machine_record == NULL)
                    preferred_list_not_exhausted = false;
                 else { /* preferred list is not exhausted */
                    requirement_satisfied = evaluate_task_requirement(machine_record);
                    if (!requirement_satisfied)
                       i = i + 1;
                    else {
                       send_message(task_transfer_request, ith_preferred_runnable_machine_Schedd);
                       wait_for_response(ith_preferred_runnable_machine_Schedd, &response);
                       if (response == ``transfer_ok'') {
                          fork_off_Shadow_process();
                          wait_for_response(local_Shadow, &response);
                          /* Shadow in turns gets the response from remote Startd */
                          if (response == ``startd_not_ok'')
                             i = i + 1;
                          else server_not_found = false;
                       } /* if transfer_ok is received */
                       else i = i + 1;
                    } /* else requirement is satisfied */
                 } /* else preferred list is not exhausted */
           } /* while */
           if (!preferred_list_not_exhausted)
              queue_task();
        }
     }

upon receiving a transfer request from workstation k:
     get_machine_record(local_Collector, 0th, &machine_record);
     requirement_satisfied = evaluate_task_requirement(machine_record);
     if (!requirement_satisfied)
        send_message(transfer_not_ok, workstation_k_Schedd);
     else { /* requirement is satisfied */
        send_message(transfer_ok, workstation_k_Schedd);

upon schedule timeout:
     prioritize_queued_tasks();
     for (i=0; i<queue_length; i++)
         /* treat each task in the queue as if it were a newly--arrived task */
         invoke_location_policy; /* execute the code executed upon the arrival of a task.  */
```

Figure 8: Pseudo code for Schedd.

space is not enough, the task is queued and will be attempted for execution/transfer upon next schedule timeout. If the swap space is sufficient, Schedd gets from Collector the machine record of the first runnable workstation in the preferred list, and checks whether or not the task requirement can be satisfied on that workstation. If not, the machine record of the next runnable workstation available in the preferred list is fetched from Collector and checked against the task requirement. The process repeats itself until either a target server workstation is found or the preferred list is exhausted. In the latter case, the task is queued for later execution/transfer.

If a target server workstation is located, Schedd sends a transfer request to Schedd on the target server workstation (**S3** in Fig. 6). Either a transfer_ok or a transfer_not_ok message will be received from the target server workstation, depending on whether or not the target workstation is truly runnable: if a transfer_ok message is received, a Shadow process is spawned off on the home workstation which notifies the Startd on the target server workstation of its responsibility to execute the task. If the workload situation has not changed on the target server workstation since its last region-change broadcast, a startd_ok message, along with two communication ports, is received. The communication and task transfer/execution operations between Shadow and Starter then proceed as in Condor. If the workload situation has changed and is not runnable anymore, a startd_not_ok message is received, in which case Schedd gets from Collector the machine record of the next runnable workstation available in its preferred list, and repeats the transfer-request process until either a target server workstation is found or the preferred list is exhausted. On the other hand, if a transfer_not_ok message is received, Schedd gets from Collector the machine record of the next runnable workstation, and repeats the transfer-request process as described above.

To deal with a possible machine failure, the ioctl system call is used to designate the sockets as non-blocking: an I/O request that cannot be completed is not performed, and return is made immediately. Moreover, a timer is set for each connection: if no response has ever come back until the timer expires, return is also immediately made. In either case, Schedd repeats the transfer-request process for the next runnable workstation available in the preferred list.

Upon receipt of a transfer request: upon receiving a transfer request, Schedd gets from Collector the local machine record and evaluates the function **Busy**. In terms of the four-component task requirements, Schedd only needs to check **VirtualMemory**, because

- **OpSys** and **Arch** have already been checked by the home workstation who initiated the transfer request;
- The **Disk** space available under the directory where checkpoint files are saved will not change if no task is executing on the workstation. So, it suffices to assure the **Disk** space has not changed by checking if the workstation is non-**Busy**;

- Since **VirtualMemory** is calculated at the time of state-region change broadcast, the **VirtualMemory** information collected (via state-change broadcasts) by the requesting workstation may differ from the actual **VirtualMemory** information currently kept if either a broadcast message is lost or not yet received by the requesting workstation before the transfer request was made. Hence, **VirtualMemory** needs to be re-checked.

If **Busy** is false and **VirtualMemory** is enough, Schedd responds with a transfer_ok message. The Shadow process on the requesting workstation will then contact Startd on the server workstation (which honors the transfer request) to handle the low-level mechanism of task execution/transfer and checkpoint process. Otherwise, the Schedd replies a transfer_not_ok message.

Upon schedule timeout: upon schedule timeout, Schedd first prioritizes the tasks currently queued on the local workstation based on their user-specified priority, queueing time, and whether or not a task was ever executed. Higher priority is given to tasks with higher user-specified priority, longer queueing time and/or tasks which were vacated from server workstations because of the return of the server workstation owner or some abnormal situation on the server workstation. Schedd then initiates the location process for each queued task, starting from the task with the highest priority.

3.2.3 Startd

Upon being notified by a Shadow process of the responsibility to execute a task, Startd generates two communication ports, spawns off a Starter to execute the task, keeps track of the execution status of the task, and signals the Starter, whenever necessary, to suspend, resume, checkpoint, or vacate the executing task. There are four events Startd will handle: the receipt of a start_task message from the Shadow on a requesting workstation, the receipt of a SIGCHLD signal (at the exit of Starter), the periodic startd timeout, and the receipt of a checkpoint_task message from Shadow on the home workstation. The pseudo code for Startd is listed in Fig. 9.

Upon receipt of a start_task message: upon receiving a start_task message from a requesting Shadow, Startd gets from Collector its machine context (**S4** in Fig. 6), and re-evaluates the **Busy** function. If the **Busy** function is false, two communicating ports are created and returned (along with a **startd_ok** message) to the Shadow on the requesting home workstation. Startd then waits for acknowledgement from Shadow to these two ports. When this connection is made, Startd spawns off a Starter, closes the two communication ports, changes the **Task_state** of the workstation to TaskRunning, and notifies Collector of its state_change (**S5** in Fig. 6; in which case Collector updates workload). If the **Busy** is true, a **startd_not_ok** message is returned.

Startd

```
upon receiving a start_task message from remote Shadow:
     /* the 0-th machine context is for the local machine */
     get_machine_record(local_Collector,0th,&machine_record);
     workload_evaluation(&Busy);
     if (!Busy) {
        create_communication_ports();
        send_message(startd_ok,remote_Shadow);
        wait_for_acknowledgement_from_remote_Shadow();
        fork_off_Starter_process(&pid);
        if (pid != 0) { /* Startd process */
           close_communication_ports();
           Task_state = TaskRunning;
           send_message(workload_update, local_Collector);
        } /* if */
        else /* Starter process */
           /* code for Starter */
     }
     else
        send_message(startd_not_ok,remote_Shadow);

upon receiving a SIGCHLD signal:
     clean_up_checkpoint_files();
     Task_state = NoTask;
     send_message(workload_update,local_Collector);

upon startd timeout:
     get_machine_record(local_Collector,0th,&machine_record);
     /* signal Starter (if necessary) based on Task_state, AvgLoad, and KeyboardIdle (Fig. 3)*/
     task_state_transition(machine_record.Task_state, machine_record.AvgLoad,
         machine_record.KeyboardIdle);

upon receiving a checkpoint_task message from Shadow:
     kill(Local_Starter, SIGUSR2);
     Task_state = Checkpointing;
```

Figure 9: Pseudo code for Startd.

Upon receipt of a SIGCHLD signal: upon receiving a SIGCHLD signal, Startd clears up the checkpoint files in the directory where the checkpoint files are stored, changes the **Task_state** of the workstation to NoTask, and notifies Collector of its state_change (**S5** in Fig. 6).

Upon periodic startd timeout: upon periodic startd timeout, Startd gets from Collector the parameters **AvgLoad** and **KeyboardIdle** (specified in the machine_record), and properly signals Starter based on these workload–related parameters to assure that workstation owners have the workstation resources at their disposal.

Upon receipt of a checkpoint_task message: Upon receiving a checkpoint_task from Shadow, Startd sends a SIGUSR2 signal to Starter, and enters the **Checkpointing** state.

4 Implementation Issues

In this section, we discuss how we handle some of the implementation issues, such as where to place the LS mechanism (inside or outside the OS kernel), how to transfer the process state (virtual memory, open files, and process control blocks) during task transfer/migration, and how to support location transparency and reduce the effects of residual dependency.

Where to place the LS mechanism: We follow Condor's principles, and implement the LS mechanism outside the OS kernel in trusted daemon processes. Placing the mechanism outside the kernel incurs execution overhead and latency (e.g., in the form of kernel calls) in passing statistics (from kernel to daemon processes) and LS decisions (in the other direction). However, as discussed in [16], the dominating factor in assessing LS performance lies more in the global communication overhead and aggregate resource management than in (small) delays incurred by kernel calls. Moreover, placing the mechanism outside the kernel facilitates later expansion or generalization of other LS strategies to deal with large communication latency [20], excessive task transfer [21], and node/link failure [22, 23, 24]. One inherent limitation resulted from placing the LS mechanism outside the OS kernel is that inter-process communication and signal facilities cannot be easily implemented, and are not supported in the current implementation. We plan to reconfigure some of the low-level process and memory management functions into a *kernel server* that resides inside the OS kernel to handle IPC and signal facilities.

Approach to transferring process state: Process state typically includes virtual memory, open files, message channels, and other kernel states contained in the process control block. In Condor, the state of a process is transferred in the form of checkpoint files. Before a process is executed for the first time, its executable file is augmented to a checkpoint file with no stack area, so that every checkpoint file may henceforth be handled in the same way. Moreover, every process is periodically checkpointed, and a new checkpoint file is created from pieces of the previous checkpoint (which contains the text segment) and a core image (which contains the data and stack segments) as follows: the LS mechanism (i.e., the Starter) causes a running task to checkpoint by sending itself the signal SIGTSTP. When a task is linked, a special version of "crt()" is included which sets up CKPT() as the SIGTSTP signal handler. Information about all open files which the process currently has is kept in a table by the modified version of the open system call routine. When CKPT() is called, it updates the table of open files by seeking each one to the current location and recording the file position. Next a setjmp is executed to save key register contents (e.g., stack pointer and program counter) in a global data area, then the process sends itself a SIGQUIT signal which results in a core dump. Starter then combines the original executable file, and the core file to produce a checkpoint file.

When the checkpoint file is restarted, it starts from the special "crt()" code, and the "crt()" code will set up the restart() routine as a SIGUSR2 signal handler with a special signal stack (in the data segment), then send itself the SIGUSR2 signal. When restart() is called, it will operate in the temporary stack area and read the saved stack in from the checkpoint file, reopen and reposition all files, and execute a longjmp back to CKPT(). When the restart routine returns, all the stacks have been restored, and CKPT() returns to the routine which was active at the time of the checkpoint signal, not "crt()".

Location transparency and residual dependency: *Location transparency* is one of the most important goals in implementing load sharing. By transparency, we mean a process's behavior should not be affected by its transfer. Its execution environment should appear the same, it should have the same access to system resources such as files, and it should produce exactly the same results as if it had not been transferred [16, 14]. To maintain location transparency, sometimes the home workstation has to provide data structure or functionality for a process after it is transferred from the workstation [14]. This need for a home workstation to continue to provide some services for a process remotely-executed is termed *residual dependency*. In Condor's and our implementation, location transparency is achieved at the expense of residual dependency in the following manner: the LS mechanism preserves the home workstation's execution environment for the remote process by using "remote system calls" in which requests for file/device access are trapped and forwarded to the Shadow process on the home workstation. As was discussed in Section 3, whenever a workstation is executing a task remotely, it also runs a Shadow process on the home workstation. The Shadow acts as an agent for the remotely-executing task in doing system calls. Specifically, each task submitted to the LS mechanism is linked with a special version of the C library. The special version contains all of the functions provided by the normal C library, but the system call stubs have also been changed to accomplish remote system calls. The remote system call stubs package up the system call number and arguments and send them to the Shadow via the network. The Shadow, which is linked with the normal C library, then executes the system call on behalf of the remotely-running task in a normal way. The Shadow then packages up the results of the system call and sends them back to the system call stub (in the special C library on the submitting machine) which then returns its result to the calling procedure.

5 Experimental Results

We evaluate our LS mechanism by comparing its performance with the "off-the-shelf" Condor package, and discussing empirical measurements over a period of one week. Three sets of measure-

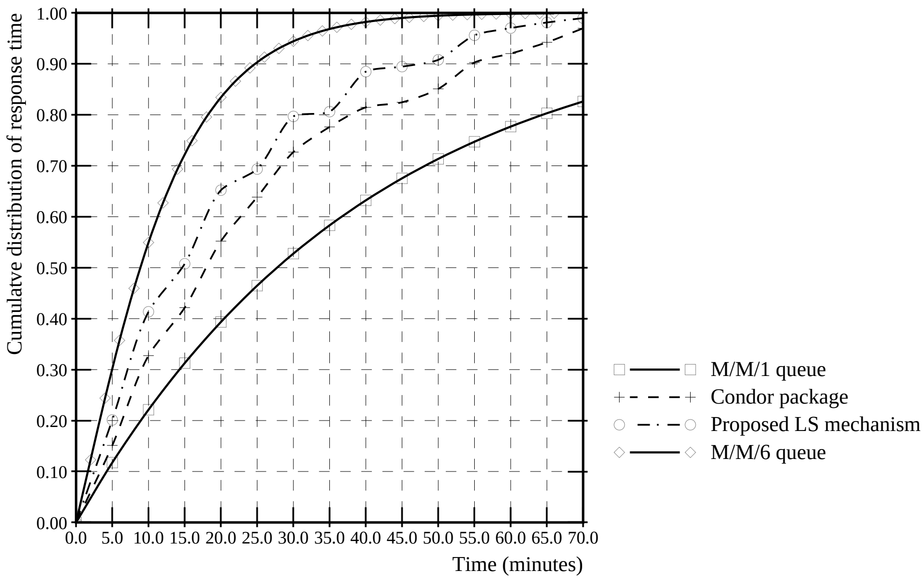


Figure 10: The response time distribution for $\lambda_i = 0.1$/mins, $\mu_i \sim 0.125$/mins, and $\frac{\lambda_i}{\mu_i} = 0.8, \forall\ i$.

ments are taken, including task response time distribution, the extent to which the LS mechanism distributes workload, and the frequency of task transfer.

The performance figures presented here are obtained from experiments conducted on 6 SUN SPARCstations connected via a 10Mbit Ethernet local area network (along with other workstations not used in this experiment). These 6 workstations were not used by other interactive users during the period of experimentation. Identical copies of single-process computational-intensive scientific computation tasks are randomly submitted to each workstation i with exponentially-distributed interarrival times with rate λ_i (1/seconds). The number of iteration runs is specified in each submitted computation task, is used to "control" the execution time of the computation task, and is "approximated" to be exponentially distributed with μ_i (1/runs). A single iteration run takes approximately 48 seconds. We instrumented the LS mechanism to keep track of local/remote process execution. First, the period between the time when a task was submitted and the time when the corresponding process exited was recorded. Second, when a process exited, the Total_Tasks counter was incremented, and the total time during which the process executed was added to the Total_CPUtime counter; if the exited process has been transferred from elsewhere, the Remote_Tasks counter was incremented, and its time was added to the Remote_CPUtime counter as well. The ratio of Remote_Tasks to Total_Tasks gives the task transfer-out ratio,[9] and the ratio of Remote_CPUtime toTotal_CPUtime gives

[9] It is actually the task transfer-in ratio, but this ratio probabilistically equals the task transfer-out ratio in homogeneous systems over the long run.

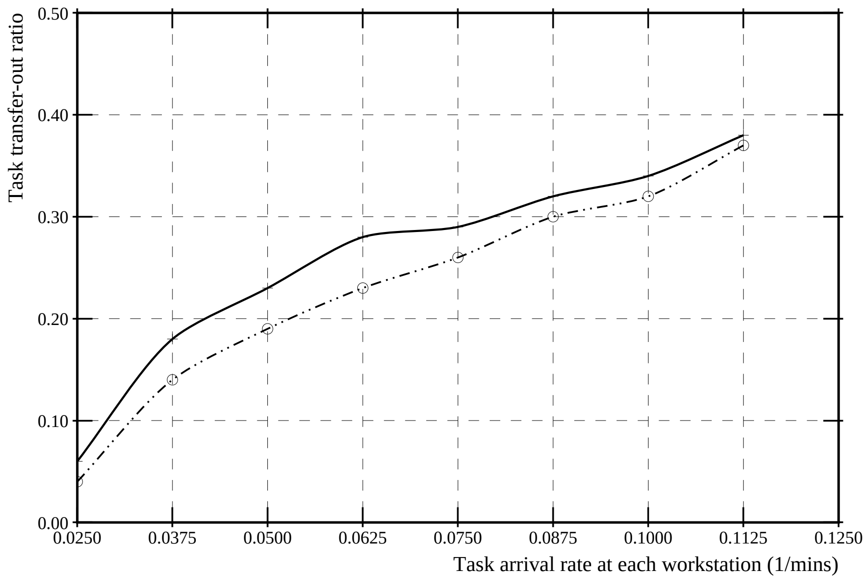


Figure 11: Transfer-out ratio with respect to different ρ_i's under homogeneous load distribution, where $\rho_i = \frac{\lambda_i}{\mu_i}$ and $\mu_i \sim 0.125$/mins, for all i.

the percentage of remote execution on a workstation.

Fig. 10 gives the response time distribution with $\lambda_i = 0.1$/mins and $\mu_i = 0.1$/runs $=$ 0.125/mins. Also shown in Fig. 10 are the two baseline curves corresponding to the $M/M/1$ queue (no LS) and the $M/M/6$ queue (perfect LS). The response time distribution under the LS mechanism approach unity much faster than that corresponding to no LS, justifying that the LS mechanism is effective to handle temporarily uneven task arrivals in distributed systems. Moreover, the proposed LS mechanism performs better than the Condor package as the system load increases (i.e., $\geq \lambda_i/\mu_i \geq 0.57$).

Table 1 gives numerical results on Total_CPUtime, Remote_CPUtime, and percentage of remote execution for uneven load distributions over a one-week period. As given in Table 1, remote processes accounted for 33.03% (43.76%) of all processing done in the case of $\overline{\lambda} = 0.5\mu$ ($\overline{\lambda} = 0.3\mu$) for the proposed LS mechanism, while they accounted for 39.37% (47.08%) in the case of $\overline{\lambda} = 0.5\mu$ ($\overline{\lambda} = 0.3\mu$) for the Condor package. We suspect the higher percentage of remote execution for the Condor package results from the fact that a task arrived at an idle workstation may be transferred to other idle workstations for execution for the Condor package. In addition, Total_CPUtime's are approximately the same over all workstations (although the local arrival rates λ_i's differ) for the proposed LS mechanism, while they vary in the Condor package. This demonstrates the advantage of using the

Workstation	Total CPU time		Remote CPU time		Percentage remote	
	Prop. LS	Condor	Prop. LS	Condor	Prop. LS	Condor
1	5,032	4,614	4,029	3,812	80.77%	82.61%
2	5,013	4,916	2,991	3,156	59.67%	64.20%
3	5,024	5,075	1,270	1,571	25.28%	30.95%
4	5,043	5,161	1,183	1,629	23.46%	31.56%
5	5,073	5,246	514	1,087	10.13%	20.72%
6	5,189	5,321	45	689	0.87%	12.95%
Total	30,374	30,333	10,032	10,944	33.03%	39.37%

(a) Load distribution: $\overline{\lambda} = 0.5\mu$, where $\lambda = 0.0125, 0.0375, 0.0625, 0.0625, 0.0875, 0.1125$/mins for workstation 1–6, respectively, and $\mu = 0.125$/mins)

Workstation	Total CPU time		Remote CPU time		Percentage remote	
	Prop. LS	Condor	Prop. LS	Condor	Prop. LS	Condor
1	3,073	2,743	2,019	1,762	65.70%	64.24%
2	3,015	2,610	2,027	1,598	67.23%	61.22%
3	2,987	2,845	2,110	1,681	70.64%	59.09%
4	3,037	3,276	798	1,487	26.27%	45.39%
5	3,098	3,238	1,060	1,402	34.23%	43.40%
6	3,143	3,532	17	659	0.03%	18.65%
Total	18,358	18,244	8,031	8,589	43.76%	47.08%

(b) Load distribution: $\overline{\lambda} = 0.3\mu$, where $\lambda = 0.0125, 0.0125, 0.0125, 0.0375, 0.0375, 0.1125$/mins for workstation 1–6, respectively, and $\mu = 0.125$/mins)

Table 1: Total CPU time, remote CPU time, and percentage of remote execution with respect to two different load distributions. Both total and remote CPU times are in minutes.

preferred lists to evenly distribute loads in the system.

Fig. 11 gives the transfer-out ratio with respect to λ_i's with μ_i fixed at 0.1/runs for homogeneous load distribution. More than 20% of the tasks are executed remotely for $\lambda \geq 0.0625$/mins even under homogeneous load distribution. That is, more than 20% of the tasks benefit from the LS facility. The transfer-out ratio for the Condor package is higher than that for the proposed LS mechanism, which may serve as another evidence of the possibility of Condor's transferring a task out of an idle workstation.

6 Related Work

Several load sharing mechanisms have been proposed/implemented, i.e., the V-system [15, 25], the Sprite OS [14], the Charlotte OS [16], and the Condor software package [17, 18]. We characterize/compare these mechanisms with respect to their transfer, location, and information policies.

V-System: The V-system [15, 25] uses a state-change driven information policy. The V-system's load measure is the CPU utilization at a node. To measure CPU utilization, a background process is constantly run at the lowest priority. Each node broadcasts its state whenever its state significantly changes. State information consists of expected CPU and memory utilization and architectural specification about the machine itself. The broadcast state information is cached by all the nodes. If the size of the system is large, each machine can cache state information only for the N most lightly-loaded nodes, e.g., those nodes with idle or underloaded CPU and memory utilization.

The V-system's transfer policy selects only newly-arrived tasks for transfer. Its location policy considers a node as a receiver if it is one of the N most lightly-loaded nodes in the system, and as a sender if it is not. When a task arrives at a machine, the (decentralized) location policy consults the local cache and finds the set of N machines which are most light-loaded and at the same time satisfy the system architecture/OS requirements of the task. If the local machine falls in the set of N machines, the task is scheduled locally. Otherwise, a machine is chosen *randomly* from the set and is *polled* to make sure that the data locally cached is not outdated. If the cached data is indeed updated, the polled machine is chosen to execute the task. Otherwise, the state of the polled machine is updated and the location policy is repeated. Note that the performance of the location policy depends heavily on fine tuning of the parameter N. Moreover, random selection of a target server machine cannot totally eliminate the possibility of multiple machines selecting the same remote machine for task transfer.

Sprite: The Sprite OS [14] uses a centralized information policy. Each workstation runs a background process called the *load-average* daemon to monitor the workload of that machine. When the workstation is measured to be idle, the load-average daemon notifies a *central migration server* that the machine is ready to accept migrated processes. The rule for deciding whether or not a workstation is idle is: when the workstation has had no keyboard or mouse input for at least 30 seconds and the number of active tasks is less than the number of processors at the workstation, it is considered idle.

The transfer policy used in Sprite takes place in three cases. In the first case, tasks may be chosen manually by users for remote execution. Tasks are submitted for process migration through two application programs, pmake and mig. The transfer policy is thus not completely automated. Another case occurs when remote tasks executing at a workstation must be evicted (and migrated back to their home workstations) to ensure that workstation owners have the workstation resources at their disposal. The third case arises to ensure fair resource allocation. If the centralized server cannot find an idle workstation for a remote execution request while a process is observed to have allocated more than its fair share of workstation resources, the server reclaims one of the workstations being used by that process and evicts that process. The freed workstation is then granted to the waiting remote execution request.

The location policy is centralized: to locate a receiver, a workstation contacts the central migration server (through a standard library procedure) which then randomly selects an idle workstation on behalf of the requesting workstation.

Charlotte: In Charlotte [16], the transfer and location policies are implemented in *Starter utility* processes. Starter utility processes use the statistical information provided by the kernels they control and on state information they exchange among themselves to make the LS decisions. Various transfer/location policies can be used by customizing the policy procedure in Starter utility processes. The policy procedure can choose either to request some source kernel to undertake process migration or to accept advice from privileged utilities (to allow manually-directed transfers and to disable automatic control). Starter-to-Starter negotiation begins when a process migration operation is invoked and may result in either a decision to migrate a process or a rejection.

The information policy is realized by a *statistician thread* which periodically samples, averages, and reports statistics to Starter. State information is then periodically exchanged among the Starter utility processes. The determination of a exchange period thus becomes crucial to the performance.

Condor: As discussed in Section 2, Condor is based on the central manager acting as a task dispatcher. The transfer policy is invoked either upon submission of a task or upon a schedule timeout. In both cases, the central manager finds idle workstations not only for the submitted task but also for tasks queued on all participating workstations. Condor uses a *periodic* information policy: each workstations reports to the central manager periodically the average workload, the keyboard/mouse idle time, and the task queue. A workstation is considered idle if the average workload is less than or equal to 0.3 and the owner has not been active for at least 15 minutes. For each queued task, the central manager negotiates with those workstations observed to be idle using the location policy described in Section 2. If the negotiation succeeds, the central manager notifies the workstation with the queued task of the target server workstation.

To ensure that workstation resources are available to the owner of the workstation, the Startd at each workstation checks for local activity from the owner periodically (e.g., 30 seconds) if a foreign task is being served by that workstation and uses the **Task_state** transition diagram in Fig. 3 to determine whether or not the foreign task should be evicted. A foreign task, if evicted, is returned to its home workstation, and may be transferred, upon the next schedule timeout, to an idle workstation if such a workstation is located by the central manager.

Proposed LS mechanism: As discussed in Section 3, our LS mechanism automatically invokes the transfer policy upon arrival of a task or upon schedule timeout, and hence, task transfer does not involve the transfer of process state. The information policy used in our LS mechanism is region-change broadcasts which, in contrast to a periodic information policy, cause messages to be exchanged only when the workload state of a workstation changes significantly. The state information kept at each workstation is very likely to be up-to-date. The proposed mechanism uses the preferred lists in the location policy to identify a server among several candidate workstations as well as to minimize the possibility of multiple workstations sending their tasks to the same idle workstation. Moreover, with the preferred lists, transferred tasks are, over the long run, evenly distributed among workstations.

7 Conclusion

We discussed the design and implementation of our decentralized LS mechanism based on the Condor software package. We removed the central manager in Condor, and incorporated the functionality of the central manager into every participating workstation. Each participating workstation collects state information on its own via region-change broadcasts, and makes LS decisions based on the state information collected. The probability of multiple machines sending their tasks to the same idle machine is minimized by using the concept of preferred list in the location policy. With such a functionality reconfiguration, Condor is more resilient to single workstation failure.

Special care has been taken to fuse our decentralized LS policies into the existing Condor software so as to require as little modification as possible. The remote system call and process checkpoint facilities in Condor are adopted to provide location transparency, to preserve the home workstation's execution environment, and to transfer the state of a process.

The current implementation based on Condor does not support applications that use IPCs, signals, and timers. We plan to reconfigure some of the low–level process and memory management functions into a *kernel server* that resides inside the OS kernel to handle IPC and signal facilities. We also plan to incorporate features we proposed in [20, 22, 21, 23, 24] into the LS mechanism, and equip the LS mechanism with the abilities to deal with large communication latencies, excessive task transfers and task collisions, and component failures.

Acknowledgement

The authors would like to thank the developers of the Condor software package for making their sources available via anonymous ftp from "shorty.cs.wisc.edu."

References

[1] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Proc. ACM Comput. Network Performance Symp.*, pp. 47–55, 1982.

[2] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. on Software Engineering*, vol. SE–12, no. 5, pp. 662–675, 1986.

[3] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, no. 12, pp. 33–44, 1992.

[4] C.-Y. H. Hsu and J. W.-S. Liu, "Dynamic load balancing algorithms in homogeneous distributed systems," *IEEE Proc. 6th International Conf. on Distributed Computing Systems*, pp. 216–223, 1986.

[5] J. A. Stankovic, K. Ramamritham, and S. Chang, "Evaluation of a flexible task scheduling algorithm for distributed hard real-systems," *IEEE Trans. on Computers*, vol. C–34, no. 12, pp. 1130–1141, December 1985.

[6] J. F. Kurose and R. Chipalkatti, "Load sharing in soft real-time distributed computer systems," *IEEE Trans. on Computers*, vol. C–36, no. 8, pp. 993–999, August 1987.

[7] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. on Computers*, vol. C–38, no. 8, pp. 1110–1123, August 1989.

[8] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state change broadcasts," *IEEE Trans. on Computers*, vol. C–38, no. 8, pp. 1124–1142, August 1989.

[9] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Adaptive load sharing in heterogeneous systems," *IEEE Proc. 9th International Conf. on Distributed Computing Systems*, pp. 298–306, 1989.

[10] T. P. Yum and H.-C. Lin, "Adaptive load balancing for parallel queues with traffic constraints," *IEEE Trans. on Communications*, vol. COM–32, no. 12, pp. 1339–1342, December 1984.

[11] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. on Computers*, vol. C–34, no. 3, pp. 204–217, March 1985.

[12] T. C. K. Chou and J. A. Abraham, "Distributed control of computer systems," *IEEE Trans. on Computers*, vol. C–35, no. 6, , June 1986.

[13] A. Weinrib and S. Shenker, "Greed is not enough: Adaptive load sharing in large heterogeneous systems," *IEEE INFOCOM'88–The Conference on Computer Communications Proceedings*, pp. 986–994, 1988.

[14] F. Douglis and J. Ousterhout, "Transparent process migration: design alternatives and the Sprite implementation," *Software – Practice and Experience*, vol. 21, no. 8, pp. 757–785, August 1991.

[15] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V–system," *Proc. of 10th Symp. on Operating System Principles*, December 1985.

[16] Y. Artsy and R. Finkel, "Designing a process migration facility: the Charlotte experience," *IEEE Computer*, vol. 22, no. 9, pp. 47–56, September 1989.

[17] M. Litzkow, M. Livny, and M. Mutka, "Condor — a hunter of idle workstations," *Proc. of 8th Int'l Conf. on Distributed Computing Systems*, June 1988.

[18] M. Litzkow and M. Livny, "Experience with the Condor distributed batch systems," *Proc. of IEEE Workshop on Experimental Distributed Systems*, October 1990.

[19] K. G. Shin and C.-J. Hou, "Analytic models of adaptive load sharing schemes in distributed real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 740–761, July 1993.

[20] K. G. Shin and C.-J. Hou, "Design and evaluation of effective load sharing in distributed real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 7, , July 1994.

[21] C.-J. Hou and K. G. Shin, "Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems," *IEEE Trans. on Computers*, vol. 43, no. 7, , July 1994.

[22] C.-J. Hou and K. G. Shin, "Incorporation of optimal timeouts into distributed real–time load sharing," *IEEE Trans. on Computers*, vol. 43, pp. 528–547, May 1993.

[23] K. G. Shin and C.-J. Hou, "Evaluation of load sharing in HARTS with consideration of its communication activities," *ACM 1993 Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 270–271, May 1993. Extended abstract.

[24] Y.-C. Chang and K. G. Shin, "Load sharing in hypercube multicomputers in the presence of node failure," *Proceedings of the 21th International Symposium on Fault-Tolerant Computing*, pp. 188–195, 1991.

[25] M. Stumm, "The design and implementation of a decentralized scheduling facility for a workstation cluster," *IEEE Proc. Secnod Conf. Computer Workstations*, pp. 12–22, 1988.