A Resource Management System for Network Computing using Java

Muthucumaru Maheswaran†, Hongtu Chen‡, Satyabrata Pradhan†, Patrick Pantel†, Lei Zheng†, Rui Min‡, and Timothy Groner†

†Dept. of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2
Canada

‡Dept. of Electrical and Computer Engineering
University of Manitoba
Winnipeg, MB R3T 2N2
Canada

1. Introduction

A *network computing* (NC) system is a virtual system formed by a networked set of machines that are willing to work together by sharing work load and data. The operation of a NC system is coordinated by the *resource management system* (RMS). One of the major functions of an RMS is to assign the application tasks to a target machine such that the execution of the application tasks are completed in a timely fashion. The RMS can be optimizing different metrics to obtain the task-machine assignments [MaA99]. To perform the task-machine assignment in an effective manner, several issues should be addressed while designing the RMS.

The RMS should be portable, i.e., it should be able to execute on different platforms. In this implementation, this is achieved partly by using Java for implementing the RMS. As the number of machines in the NC system is increased, scalability of the RMS and heterogeneity among the constituent machines become critical issues. Further, as the number of machines is increased, the NC system is very likely to have machines that belong to several different administrative domains and with different computational capabilities. Architectural enhancements that will enable the RMS to deal with the aforementioned issues are presented in this paper. Other issues considered in the paper include (a) learning task attributes (in particular execution times) from actual execution times obtained from previous runs of the application, and (b) remote access to input and output data files.

In Section 2, the architecture of the RMS is presented. The implementational details are discussed in Section 3. Section 4 investigates an enhancement that makes the basic architecture presented in Section 2 more scalable.

2. Architecture of the RMS

The architecture of the RMS developed in this study is presented in Figure 1. As can be noted from Figure 1, the RMS is made up of several modules (referred to as "servers") that support specific functionality.

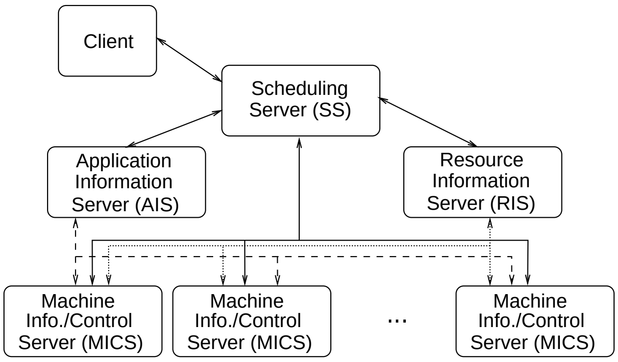

Figure 1: Architectural block diagram of the RMS.

The task assignment functionality is performed by the *scheduling server* (SS). To exploit the heterogeneity among the machines that constitute the NC system, it is necessary to know the expected execution times of the different applications on the different machines [SiA96]. This information is maintained by the *application information server* (AIS). The SS interacts with the AIS to obtain this information as needed. The scheduling algorithms used by the SS also needs the machine loading status. This information is collected and maintained by the *resource information server* (RIS). The target machines need to execute the tasks sent to them by the SS. This functionality is performed by the *machine information/control server* (MICS).

3. Implementation of the RMS

When a new target machine joins a NC system, the corresponding MICS server is connected to the RMS. Likewise, when a target machine leaves the NC system, the corresponding MICS server will be disconnected from the RMS. To facilitate this dynamic entry and exit of MICS servers, a simple communication middleware was developed. This middleware contains a message router that routes packets from a source server to a destination server.

The SS server is a multi-threaded server where some threads are pre-spawned and others are spawned on demand. The multi-threading capability of Java is extensively used in the implementation of this server.

The *input buffer thread* (IBT) is responsible for checking the incoming socket from the message router. This thread adds any packets coming from the message router to the input buffer queue. Similarly, the *output buffer thread* (OBT) continuously checks the output buffer queue and sends any packet in this queue to the message router.

The packets arriving at the SS from AIS, MICS, or RIS are received by the input buffer queue. The *manager thread* (MT) is responsible for handling the packets from the different sources in an appropriate manner.

The *scheduler thread* (ST) is responsible for scheduling the jobs that are arriving from the clients onto the target machines. The ST uses a dynamic scheduling algorithm for scheduling the jobs. The operation of the ST depends on the type of the scheduling algorithm [MaA99]. A detailed explanation of ST is given below. The *client handler thread* (CHT) authenticates the client and keeps listening to the particular client it is connected for requests.

The interaction of the different threads and data structures are shown in Figure 2. The direction of data flow between two data structures is represented by an arrow. The thread which transfers the data is noted beside the arrow joining the data structures.

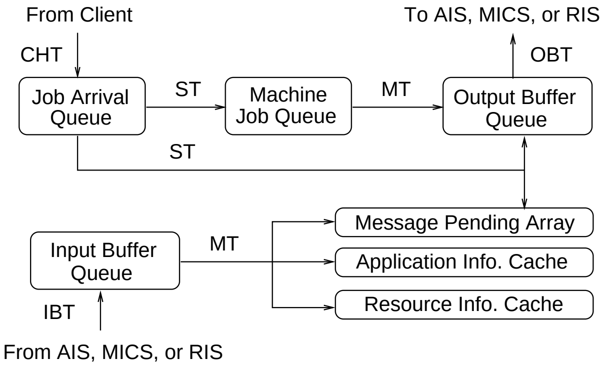

Figure 2: Data flow between different data structures in the scheduling server.

Because the application tasks are arriving in real time and the set of machines that constitute the NC system can change in real time a dynamic scheduling method is used for implementing the ST. The dynamic scheduling method chosen to implement the ST should be capable of exploiting the heterogeneity available among the different machines of the NC system. Existing dynamic scheduling algorithms can be categorized into two classes: on-line mode scheduling algorithm and batch mode scheduling algorithm [MaA99].

In the on-line mode, a task is scheduled onto a machine as soon as it arrives at the scheduler. In batch mode, tasks are not scheduled onto a machine as they arrive; instead they are collected into a set that is examined for scheduling at predefined times called *scheduling events*. The independent set of tasks that is considered for scheduling at a scheduling event is called a *meta-task*. A meta-task can include the newly arrived tasks (i.e., the ones arriving after the last scheduling event) and the tasks that were scheduled at previous scheduling events that are yet to begin their execution. While the on-line mode algorithms consider a task for scheduling only once, batch mode algorithms consider a task for scheduling at each scheduling event until the task begins its execution.

For certain arrival and completion rates, the machines may be ready to execute a task as soon as it arrives at the scheduler [MaA99]. For such situations, it may be beneficial to use the scheduler in the on-line mode so that a task need not wait until the next scheduling event to begin its execution. If there are sufficient number of tasks to keep the machines busy in between the scheduling events and while the scheduling is computed, then batch-mode scheduling will be more appropriate. A detailed discussion of the trade-offs between the two modes are given in [MaA99].

Figure 3 shows the pseudo-code for the ST when a batch mode scheduling algorithm is used. The ST has an infinite loop with two phases: scheduling and sleeping. In the scheduling phase, the ST collects all the jobs in the job arrival queue and the jobs in the machine job queues that are yet to begin their execution into a temporary queue. For all machines other than those busy executing a task, the machine available times are set equal to the current time. For those machines where tasks are currently executing, the machine available times are set to the expected completion times of the tasks executing on the machines. The ST also obtains the resource usage information of the applications and the machine status information from the AIS and RIS, respectively. Once the information from the AIS and RIS is received, a batch mode algorithm is to compute the task-machine assignments. The tasks are then inserted into the appropriate machine job queues. After the scheduling phase is completed, the ST sleeps for a predefined time period and reenters the scheduling phase.

Figure 4 shows the pseudo-code for the ST when an on-line mode scheduling algorithm is used. The ST checks the job arrival queue for any incoming job. If the number of jobs in the job arrival queue is zero, the ST will wait on a semaphore attached with the job arrival queue. Otherwise, the ST will extract the first job in the queue and schedule it. The ST will request the job's resource usage information and the machine status information from the AIS and RIS, respectively. Once the replies from AIS and RIS arrive, an on-line scheduling algorithm will be used to determine the target machine for the job. The job will be inserted into the machine job queue that corresponds to the target machine. Then the ST will reexamine the job arrival queue for more incoming jobs.

```
while (true) {
   // remove all jobs from the job arrival queue
   // add these jobs to the temporary queue -- jobsQ
   while (SchedulingServer.sizeofjobArrivalQ() > 0) {
      aJob = SchedulingServer.removeJobArrivalQ();
      jobsQ.add(aJob);
   }
   // set the machine available time for each machine to the
   // current time unless the machine has a job running on it.
   // if there is a job running, then set the machine available
   // time to the expected finishing time of the job

   // remove all jobs not yet stated on the machine job queue
   // add these jobs to the temporary queue
   // at this point, JobsQ contains all jobs to be scheduled
  // get the current machine load information in the scheduling
   // Server's theMachineLoadinfo member

   // retrieve the ETC array for this application

   // use a batch mode scheduling algorithm to schedule
   // the jobs in the JobsQ
   // Minmin is used in this case
   // once the target machines are determined,
   // insert the jobs in the appropriate queues in the
   // machine job queue

   // sleep for a while before scheduling again
}
```

Figure 3: Pseudo-code of the scheduler thread for batch scheduling.

The above discussion on the on-line mode and batch mode algorithms assumed that the machines contained within the NC system are operating in a dedicated fashion. If the environment is non-dedicated, as it is in most practical settings the following heuristics can be used in computing the expected completion times of the application task on the different machines.

The application resource usage information obtained from the AIS provides the expected execution time of the task when the machine is fully dedicated to this task, i.e., if this task is not running the machine would be 100% idle for the duration of the execution. From the current machine status, the loading on the different machines can be determined. The MICS executes the application on the target machines at a lower priority than the priority of normal tasks so that the NC system would be as less intrusive as possible. Therefore, if a task is scheduled onto a machine that is only 50% idle, then the task's ETC should be multiplied by a factor of two. If the SS does not have a job running on a machine and its loading is such that the idle time is less than 10%, then the machine is considered to be busy executing jobs that do not belong the SS, i.e., the machine is executing jobs that are not submitted via the NC system. Such machines are ignored by the SS for that particular scheduling event.

```
while(true) {
   // check the job arrival queue
   // if there is no jobs wait on a semaphore
   // once a job arrives the thread wakes up

   // after waking up, remove a job from the job arrival queue
   currJob = server.removeJobQueue();
   // schedule this job -- using MCT algorithm

   // get the application resource requirements from the
   // application information cache or the AIS
   // get the current machine status

   // use the MCT algorithm to find the machine that
   // gives the earliest completion time for this application
   // the algorithm is scheduled on that machine

   // insert the job in the appropriate queue in the
   // machine job queue
}
```

Figure 4: Pseudo-code of the scheduler thread for on-line scheduling.

Let `CompletionTime (i, j)` be the expected completion time of task i on machine j. Figure 5 shows the heuristic used to compute the completion times of a job on the different machines.

4. Interconnecting RMSs

In Figure 6, two RMSs are interconnected using a special client called the leasing client. The leasing client `LCa` connects `RMS A` to `RMS B`. Its connection to `RMS A` is similar to that of a client with few differences. The `LCa`'s connection to `RMS B` is similar to that of a MICS server with some differences. The `LCa` polls the SS server of `RMS A` for the average machine load information. If the average machine loading returned by `LCa` is below a given threshold, then `LCa` will be treated by the

```
for each machine j
   if ( a scheduled job is running on machine j )
      CompletionTime(i, j) = Available time of j + ETC(i, j)
   else if ( idle Time of machine j  >  10%)
      CompletionTime (i,j) = CurrentTime + ETC(i,j)
                        * (Idle Time of machine)
   else
      ignore machine j
   endif
endfor
```

Figure 5: Computing the completion time of a task i.

SS server of `RMS B` as a special MICS server. Unlike a MICS of `RMS B`, the job presented to `LCa` by `RMS B` is not controlled by the SS server of `RMS B`. The machine

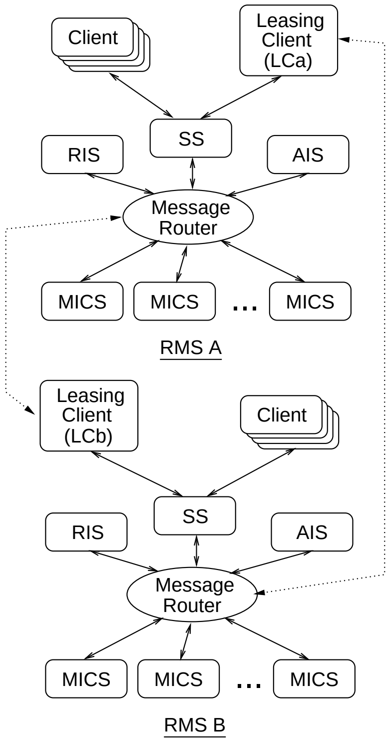

Figure 6: Networking the RMSs using the leasing client.

assignment of the job presented to `LCa` is controlled by the SS server of `RMS A`. Once the execution of the job is completed in a MICS that belongs to `RMS A` the result are returned to the client that is connected to `RMS B`.

The above technique provides a flexible way of interconnecting RMSs. The RMSs can be interconnected in an arbitrary topology. Several alternative strategies could be used for returning the loading information from `RMS A` to `RMS B`. Instead of returning only the average machine load information it is possible to return (a) the average machine load and the variance of the machine load, or (b) maximum, average, minimum machine load. Using these values, the SS server of `RMS B` should use some heuristics to determine whether it is beneficial to send a job over to `RMS A`.

5. Conclusions and Future Work

This paper describes issues involved in designing and implementing an RMS using Java for NC systems that contain heterogeneous machines from different administrative domains connected via high-speed networks. Several issues including portability, scalability, heterogeneous substrate, and variability of the execution time estimates are some issues addressed in this study.

Following are some directions identified for future work. Supporting differentiated services for the application tasks by taking into consideration the quality of service requirements is an important issue that needs to be studied in the future [Mah99]. In order to ensure adequate service, users may request in advance that time should be allocated for their applications. Present implementation of the RMS does not support advance reservations. Providing support for advance reservations will be examined in a future study.

References

[MaA99] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of meta-tasks on heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, accepted and scheduled to appear in 1999.

[Mah99] M. Maheswaran, "Quality of service driven resource management algorithms for network computing," *1999 International Conference on Parallel and Distributed Processing Technologies and Applications (PDPTA '99)*, June 1999, pp. 1090–1096.

[SiA96] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," in *Parallel and Distributed Computing Handbook*, A. Y. Zomaya, ed., McGraw-Hill, New York, NY, 1996, pp. 725–761.