REMOTE: A Tool for Automatic Remote Execution of CSIM Simulation Models

Kenneth J. Christensen
Department of Computer Science and Engineering
University of South Florida
christen@csee.usf.edu

Abstract

CSIM18 simulation models are often built as Microsoft Windows console-mode programs. A tool named REMOTE was developed to enable automatic, remote execution of CSIM18 models (and of other console mode programs) in a Windows environment. A master PC distributes the programs to remote PCs and collects the output files as the remote model executions complete. The status of remotely executing models can be viewed at the master. The REMOTE tool utilizes idle non-dedicated PC CPU cycles during nights and weekends to speed-up the model execution phase of a large modeling project. The REMOTE executable can be distributed via email attachment and does not need any special installation or configuration. REMOTE is currently being used in an ongoing investigation of new variable-length packet switch architectures. REMOTE is freely available from the author.

1. Introduction

The simulation modeling process can always benefit from reduced model execution time. Models that execute faster allow for greater accuracy in results and for a wider range of control variables to be evaluated. Faster model execution can also speed-up the model development phase. One way to achieve faster execution time is by parallelizing model execution. Very large models cannot satisfactorily execute on a single machine due to limited CPU and/or memory resources. Significant research continues in finding methods to “break-apart” and parallelize such large models to run on multiple machines such as on a cluster computer or Network Of Workstations (NOW). These efforts are focused on a space-parallelization of models. Space parallelization methods typically require new modeling languages and techniques. Another class of simulation models, such as real-time battlefield simulations, are by their nature geographically dispersed and thus inherently require parallelization. Less exciting, but still of importance to the simulation community, are smaller, single-machine models that require many runs to complete a single experiment. A single experiment could, for example, consist of a range of workload parameters. Experiment run-times can be reduced by being able to run a single model on many machines with different input parameters for each individual run.

Parallel Independent Replications (PIR) [5] of a simulation model enables time-based parallelization. In time-based parallelization, identical models are distributed to multiple machines where each instance independently simulates a different sample path. The results from the independent runs can be combined into an experiment mean with a confidence interval. Batch execution systems have long been used to distribute jobs to remote systems. Most existing remote execution systems require Unix workstations. This is because Unix is very “friendly” to remote computing (e.g., the r* commands). However, desktop computing is becoming pervasively Wintel PC based with TCP/IP connectivity. These pervasive Internet-connected desktop PCs are also of increased computing capability. What is needed is a simple means of exploiting unused CPU cycles in non-dedicated PCs for executing simulation models. This need is addressed with the development of the REMOTE tool for automatic, remote execution of simulation models.

The remainder of this paper is organized as follows. Section 2 briefly reviews existing batch and remote execution systems and also reviews CSIM18. Section 3 describes the requirements and user view of the REMOTE tool. Section 4 describes the design and implementation of the REMOTE tool. A description of the use of the REMOTE tool for modeling efforts for investigating new high-speed switch architectures is in Section 5. Section 6 is the summary and is followed by references and two appendices.
2. Review of Existing Methods

Distribution of programs to remote computers is handled by many batch processing and program migration schemes. These program distribution schemes are not always directly applicable to remote execution of simulation models.

2.1 Existing methods for remote execution

Unix workstations readily support remote program execution with the \texttt{r*} commands including \texttt{rsh} and \texttt{rexecd}. These commands allow for a locally stored program to be executed on a remote Unix host with results redirected back to the local host. These tools are intended for manual use (i.e., for remote execution of only a few instances of a program) but can be scripted to scale-up to close-to-automatic distribution and execution of programs. To support multiple machines and a list of programs to execute, a non-trivial script (e.g., written in Perl) needs to be developed. To use the \texttt{r*} commands, access rights must be given at all the remote hosts. Once these access rights have been granted, there are no further security mechanisms to prevent execution of malicious programs. Windows supports some of the \texttt{r*} commands (including \texttt{rsh}, \texttt{rexec}, and \texttt{rcp}), but allow only remote execution on a Unix host. Commercial products make it possible for a Windows server to offer these services to Windows clients. For example, MKS Software [7] offers the \texttt{rexecd} remote execution service for Windows and WINRSHD software [15] a \texttt{winrshd} remote shell. These services require the Windows server version, installation of software, and/or configuration of the remote PC to be accepted within a known Windows network domain. Development of a script is still needed for automatic distribution and execution of many jobs to many PCs.

Batch processing and process distribution tools have long existed in the Unix environment. Xdistribute [9] takes a list of hosts and a list of jobs and distributes the jobs to the hosts. It is not clear how input files are transferred to remote hosts, or output files transferred back to the “Xdistribute-ing” host. Condor [6] implements process migration between workstations. Condor was developed to use idle CPU cycles, and when a workstation becomes active again (e.g., when its human owner returns and begins typing on the keyboard), the currently running process is migrated to another idle workstation. Both Xdistribute and Condor are Unix-based and have dependencies on configuration and software (e.g., Perl) installed at remote hosts.

Grid computing [1] is a means of using distributed processing power to complete a single task. The Sun Grid Engine [13] is an open source (500,000+ lines of code) “distributed resource manager” evolving out of the Codine batch processing product. The Grid Engine enables unused Unix workstation CPU cycles to be used for distributable, compute intensive tasks. In the SETI@home project [12], home PC users are solicited to execute an analysis program that downloads and analyzes radio telescope data for signs of extra-terrestrial life. Processing of the data is done only when a PC is idle and results are uploaded to the main SETI@home site when the PC reconnects (e.g., dials-in) to the Internet.

There appears to be no existing system tailored to the Windows environment that does not require configuration of remote hosts and/or significant amounts of script development. Distribution and remote execution of simulation models requires a method of distributing not only executable files, but also associated input and output (results) files with each executable model file. Also, a means of monitoring the status of remotely executing simulation models is needed.

2.2 Review of CSIM18 simulation models

CSIM18 is a commercial (offered by Mesquite Software, Inc.) C/C++ function library used to build process-oriented, discrete-event simulation models [11]. CSIM18 is supported on all major platforms, operating systems, and compilers and supports both pure C and object-oriented C++ model development. A CSIM18 program models a system as a set of interacting processes. CSIM18 can be used for modeling network protocols (e.g., IEEE 802.3 Ethernet [2]) and devices (e.g., switch architectures [16]). The resulting models compile into console mode programs that can read input and write output to text files. Example CSIM18 simulation models of an M/M/1 single server queue, of an IEEE 802.3 Ethernet, and of high-speed switch architectures are available from the author. Key CSIM18 constructs are the \texttt{create()} calls that make a C function behave as a lightweight process and the \texttt{hold()} calls that make simulated time pass during the execution of a process. In an M/M/1 model, a \texttt{generate()} process generates exponentially distributed arrivals to a \texttt{queue1()} process. The \texttt{queue1()} process contains a \texttt{hold()} with an exponentially distributed service time. A server facility models the single server of the M/M/1 queue where only one customer can be “in service” between \texttt{reserve()} and \texttt{release()} statements. Statistics are maintained internally by CSIM18 and can be accessed via reporting functions. A real-time status reporting function can be added to a CSIM18 model. This function is described later in this paper.
3. Requirements and User View

This section describes the requirements for remote execution of simulation models and presents the “user view” of the REMOTE tool. The general scope of remote program execution is shown in Figure 1. A single master PC controls multiple network-attached remote PCs. The master PC distributes executable, stand-alone console mode programs and their associated input files to the remote PCs. The remote PCs execute the programs and then return the results (the output files) to the master PC. As much as possible, it should appear to the master PC that all programs are actually executed locally and there must be no required manual interaction between the master and remote PCs.

3.1 Requirements for remote model execution

The key requirement is to be able to harness unused CPU cycles in network-connected PCs to execute CSIM18 simulation models. This will result in a speed-up of model execution when a large number of models need to be executed and/or a single model needs be executed with many different input parameters for a given simulation experiment. Specific requirements for the master program and remote program are:

1) There must be no changes required to the simulation models to support remote execution.
2) Any network-attached Windows PC with an IP address must be able to support remote model execution. Special configuration, installation of a scripting language (e.g., Perl), and/or use of a “server version” of Windows must not be required.
3) Execution of the simulation models must occur automatically and not require any manual `telnet', `ftp', `rsh', or other interaction. All input and output files associated with a model must be automatically transferred to and from the master and remote PCs.
4) The remote program must not interfere with normal use (e.g., by the owner) of the remote PC.
5) The remote program must be a single file and be transferable via email. Execution of the remote program must be very simple.
6) It must be simple to describe the set of available remote PCs and the set of console-mode programs (models) to be run. This description is used by the master PC to distribute programs and collect results.
7) The status of remote PCs must be viewable at the master PC. This status must include the readiness of the remote PCs to accept models for execution and their current progress if running a model.
8) Status and error messages must be displayed at the master and remote PCs to allow for diagnosis of run-time failures of both the master and remote programs and of the remotely executing simulation model.
9) There must be a way to reset, from the master PC, the remote programs at all remote PC’s to a known state. This includes the ability to terminate any remotely executing models. This reset must not affect normal operation of the PC.
10) The remote program must be secure and not allow unknown hosts to maliciously or accidentally destroy files, execute programs, or access data. The remote programs must only respond to the known master PC.
3.2 User view of the REMOTE tool

The user view of the REMOTE tool is at the Windows console or “DOS prompt”. At the master PC, multiple instances of master.exe can be run. At each remote PC, one instance of remote.exe is executed. At each remote PC, remote RUN master_host_name is executed in a console window (which can be minimized to the task bar). The argument master_host_name is the full hostname of the master PC. At the master PC, the following master functions can be run:

1) Display a help screen (HELP command - see Appendix B)
2) Transfer a file (XFER command)
3) Run a list of simulation models (RUN command)
4) Reset remote PCs (RESET command)
5) Display real-time status (STATUS command)

The master RUN command assumes a list of programs (or jobs) in a default joblist.txt file and a list of available PCs in a default hostlist.txt file. Figure 2 shows example job and host list files.

Figure 2. Example (a) joblist.txt and (b) hostlist.txt

The programs in the job list are executed on the remote PCs via a master RUN command. Figure 3 shows a sample master console window. In Figure 3, remote executions are taking place with one output file already having been transferred back to the master. Figure 4 shows a master STATUS command. Each remote PC is reporting run time status for its execution. It can be seen that host giga4.csee.usf.edu is 60% complete in its execution of mm1.exe and giga5 and giga6 are 10% and 30% complete, respectively. Figure 5 shows a representative remote console window at the completion of execution. It can be seen that three models have completed execution and currently no model is executing.
4. Design and Implementation of REMOTE

The REMOTE tool uses communications between functions and Windows threads to transfer input, output, and executable files and to report remote status. In the remote PC, a process is spawned for the simulation model to be executed. Threads communicate between master and remote PCs using the TCP/IP and UDP/IP protocols. The REMOTE tool is implemented in “C” using the Winsock interface [14] for all communications. Figure 6 shows the structure of the REMOTE tool. Each module contains a single function and possibly a single thread to handle the command associated with the module name (e.g., rrun.c contains rrun_thread() to listen for connections from the master and to spawn a process for remote program execution). A common build-time file exists that contains all Winsock-related functions. The arrows in Figure 6 show the communications between functions and threads. For example, the mxfer.c function communicates with the rxfer_thread() thread. The rxfer_thread() waits to receive files from the master. Each arrow corresponds to a unique IP port. There are separate port numbers assigned to the XFER, RUN, STATUS, and RESET commands.

The “C” main() for the master and remote programs parse command line inputs and then call the appropriate function. The common.c file contains common Winsock functions used for communications. These functions are:

- get_addr() – Gets the IP address of a specified hostname
- get_name() – Gets the hostname of a specified IP address
- recv_file() – Receives a file from hostname at a specified port
- send_file() – Sends a specified file to hostname at a specified port
- recv_command() – Receives an ASCII command string at a specified port
- send_command() – Sends a specified ASCII command string to a specified port
- get_time() – Gets the date and time from the internal system clock

The remote status thread uses UDP/IP to send a short (typically one line) status.txt file as a datagram to the master PC. These datagrams are viewable with a master STATUS command. A datagram does not have assured delivery (i.e., it may be lost due to network congestion). All other communications use TCP/IP for assured delivery of files and commands.

![Figure 6. Structure of REMOTE implementation](image-url)
4.1 Execution of the remote simulation program

In the master PC, the master run thread checks the idle/busy status of all hosts in the host list once every second. This check is always made from the top of the host list, making it possible to control which remote PCs are selected first when simulation models from the job list are distributed. When a remote PC becomes idle, the next program in the job list is transferred to the remote PC. The master PC transfers the executable, input, and output files associated with each simulation mode to the remote PC. At the remote PC within the remote run thread, a separate process is created to run the executable file (shown in Figure 6). The remote run thread waits for the process to complete and then transfers the output file to the master. When the output file is transferred from remote to master, the master identifies the remote PC as being idle and ready for another program to be transferred to it. Windows CreateProcess() is used to execute the remote program and WaitForSingleObject() to wait for its completion.

4.2 Status reporting by a CSIM18 model

The remote program can determine if it is busy running a remote execution, or idle and waiting to receive a program to execute. This idle/busy information is sent to the master via status datagrams. The remote program creates and periodically returns a file named status.txt to the master. If the executing simulation model can update the default status.txt file to contain real-time status (e.g., percentage of completion) then this status information will be displayed at the master (when a STATUS command is invoked) as shown in Figure 5. In Appendix A, the CSIM18 run_sim() function holds for a simulation time in ten increments. At each increment, the percentage of completion (based on elapsed simulation time) is written to the status.txt file. It is not required that a program write to the status.txt file for the REMOTE tool to remotely execute the program.

4.3 Implementing security

Security features are implemented in the master and remote programs. At the remote PCs, only the known master PC is allowed to access files and run programs. At the master PC, only known (in the hostlist.txt file) remote PCs are allowed to upload files. When a connection is made (either to a remote PC or to the master PC), the IP address of the connecting host is checked. If this IP address is not a known address, then the connection is immediately dropped. In the remote PC xfer_thread(), a check is made that any transferred file is destined only for local directory (and thus not overwriting an existing system file). This check is made by scanning the filename for slashes or colons indicating a change of directory.

4.4 Known problems and issues

Known problems with the REMOTE tool include difficulties in using PCs that use temporary IP addresses, no validation of the entries in the host or job list, and no ability to detect and recover from remote PCs that fail. A PC can be configured to have a permanent IP address, or to use Dynamic Host Control Protocol (DHCP) [3] to acquire an address when booting-up. Currently, the REMOTE tool will work with remote PCs with DHCP assigned addresses, but the master PC must have a permanent IP address.

Validating the entries in the host list (i.e., that all entered host names resolve into IP addresses and that a REMOTE is running on these PCs) has not yet been implemented. Also not implemented is a means of validating that all programs listed in the job list are present in the current directory. Both of these tasks could be performed by a separate program executed before master RUN is executed. Failed remote PCs could be detected via a failure to receive a periodic status message. When a remote PC is detected as “not there”, the job currently being run by this remote PC should be distributed to another remote PC in the host list and the host list updated to remove the failed PC host name.

5. Use of REMOTE for Research

The REMOTE tool is currently being used to execute CSIM18 simulation models of new packet switch architectures. Link speeds in modern and near future networks are in the 10 to 100’s of gigabits per second. At these link speeds, the speed of memory becomes the constraint for scaling-up switch designs. In a typical Output Queued (OQ) switch architecture, the memory speed must equal N times link speed for N input links (for the case of all N input ports simultaneously forwarding a packet to a single output port). Such OQ switch designs are not scalable to a large number of ports for high-speed links. Input Queued (IQ) switches do not have this same scaling problem (in an IQ switch, memory speed must only equal link speed, which is independent of the number of ports). However, in IQ switches scheduling algorithms must be employed to match input and output ports to forward packets. These scheduling algorithms can exhibit instability and other behaviors that need to be fully understood.
Figure 8 shows the architecture of a Combined Input and Crossbar Queued (CICQ) switch architecture that has been the subject of study in [8], [4], [10], and [17]. Cells buffered in the input ports are scheduled to be forwarded from the input queues to the crossbar queues and then from the crossbar queues to the output link. In an input port $i$, there are Virtual Output Queues (VOQ) where cells destined for an output $j$ are queued. The classifier puts arriving packets into the appropriate VOQ.

One method of scheduling is to use Round Robin (RR) polling for both the VOQs in the input ports and the Cross Point (CP) buffers. Instability occurs when a switch cannot carry an offered load that is less than, or equal to, 1.0 (or 100% utilization). Instability will always occur if the offered load exceeds 1.0 for any output port. In [5], a region of instability for RR/RR CICQ switches is demonstrated for a two-port switch with unbalanced traffic load and offered load less than 1.0. All offered load is for packets of fixed lengths (called cells) and with a Bernoulli distribution (i.e., a given cell slot has a cell in it with independent probability $\lambda$ and the slot is empty with probability $1 - \lambda$). The traffic load is $\lambda_{ij}$ for $i, j = 1, \ldots, N$ where $i$ is the input port and $j$ the output port. For the case of $\lambda_{1,1} + \lambda_{1,2} = \text{offered load}$, $\lambda_{2,1} = \lambda_{1,2}$, and $\lambda_{2,2} = 0$ within a region of $\lambda_{1,1} > 0.5$ and high offered load, instability occurs.

Figure 9 shows the region of instability for the two-port RR/RR CICQ switch of Figure 8. This figure shows results similar to those presented in [4]. To find the edge of the instability region, a simulation model of the two-port RR/RR CICQ switch was run for $\lambda_{1,1}$ values ranging from 0.50 to 1.00. For each $\lambda_{1,1}$ value, the simulation starts with $\lambda_{1,2} = 1 - \lambda_{1,1}$ and then decreases $\lambda_{1,2}$ in steps of 0.01 until the simulation runs for 10 million cell times without VOQ$_{1,1}$ exceeding an arbitrary 5000 cells in queue length. An instantaneous VOQ queue length of 5000 is considered to be a signature of instability. While not strictly unstable, such a large queue length would be infeasible in a real switch implementation. The results of Figure 9 required about 24 hours of run time using six idle PC’s (three at 733-Mhz and three at 800-Mhz). On a single 733-Mhz PC, the run time to generate this graph would have been close to one week.

Figure 7. RR/RR CICQ switch architecture

The REMOTE tool is currently being used to evaluate the stability of alternative scheduling mechanisms. For example, scheduling schemes that apply thresholding and bursting to RR polling offer a promise of stability [17]. The REMOTE tool will be used in the future to evaluate the CICQ switch architecture for natively switching variable length packets, such as IP packets.

6. Summary and Future Work

The REMOTE tool was developed to distribute and execute CSIM18 simulation models on non-dedicated, idle Windows PCs. The REMOTE tool requires only that a single program (remote.exe) be executed on a remote PC to enable it to receive programs with input files for execution and to return results in the form of an output file. The implementation of the REMOTE tool is in “C” and uses threads, processes, and the Winsock interface. The size of the REMOTE tool source code is less than 1100 lines of C code (and less than 1700 lines including all comment and blank lines) making the tool reasonably easy to maintain and/or modify. The REMOTE tool, as well as the CSIM18 switch simulation
models, are available from the author as open source with no restrictions on use. Future work will fix some of the shortcomings described in Section 4.4 of this paper.

Acknowledgements

The author thanks the anonymous referees for their helpful comments.

References


Appendix A – CSIM18 Status Reporting Function for use with REMOTE

This appendix contains a source listing of a CSIM18 status reporting function. The run_sim() function demonstrates how real-time status can be updated in a status.txt file (that can be sent to the master PC).

```c
void run_sim(double run_time) {
    FILE *fp_status; // Pointer to status output file
    int num_inc = 10; // Number of increments
    int i; // Loop counter

    // Open status.txt for writing status updates
    fp_status = fopen("status.txt", "a+");

    // Loop to output status in increments
    fprintf(fp_status, "0% ");
    fclose(fp_status); fp_status = fopen("status.txt", "a");
    for (i=1; i<=num_inc; i++) {
        hold(run_time / num_inc);
        fprintf(fp_status, "%d ", (i * num_inc));
        fclose(fp_status); fp_status = fopen("status.txt", "a");
    }
}
```
Appendix B – Help Screen for REMOTE

This appendix shows the help screens for the REMOTE tool. Figure 1B shows the help screen for the master, and Figure 2B the help screen for the remote. The master and remote programs consist of single executable files (master.exe and remote.exe). Help is invoked, as shown in the figures, by running master HELP and remote HELP on a command line.

Figure 1B. Help screen for the master program

Figure 2B. Help screen for the remote program