Introduction to Course Projects
The RedNet Distributed System Programming Environment
COMP 420, COMP 532

1 Introduction

The course projects in COMP 420/COMP 532 this semester will run on the (simulated) RedNet distributed system at Rice. The programming environment for the RedNet distributed system allows you to easily run programs on different computers connected over the network within the distributed system, with those programs able to easily send and receive messages between each other. The environment allows you not only to program the user program (or programs) to be run as processes on the system, but also to program the relevant parts of the (simulated) operating system kernel that controls the message-passing communication between these processes.

This document provides an introduction to the RedNet distributed system programming environment and how to use it. Future documents (very soon!) will define the specific programming projects we will do this semester using the RedNet distributed system.

This document provides a specification of the RedNet distributed system programming environment. This new version of the specification expands in a number of ways beyond the initial version distributed earlier in class. The version number of the document will always be specified in the lower right corner of page 1. This new version of the specification, which is complete for what is needed in Lab 1 and Lab 2, is version 1.1. If any additional changes in this specification become necessary, they will be announced in class, on the course website, and on Piazza.

2 The RedNet System Model

In the RedNet distributed system, a separate copy of the same operating system kernel runs on each computer within the distributed system. Among other responsibilities, the operating system kernel supports and controls the execution of any processes running on that computer. The RedNet programming environment is designed so that you do not need to be concerned with how the entire operating system kernel works. Primarily, this means you need only write a small amount of kernel code to control the transmission and delivery of messages between user-level processes; the actual implementation of this message passing by the operating system kernel is supplied to you and you need not worry about it here.

Each user process executing on the system may run a different user-level program and/or may run a copy of the same user-level program as is being run by other processes. Each process executes independently, with its own separate, private memory address space, not shared by any other processes running on the system.

When a process is executing in user mode, it sequentially executes the instructions that make up the user-level program being executed in that process. At times, the execution of that process may change into kernel mode in order to execute some action required of the operating system kernel. For example, when a user process sends a message, the kernel is responsible for actually transmitting that message over the

---

1 The name RedNet is intended to have a humorous meaning. One of the common themes within distributed systems is replication, for example to improve performance and availability and to increase fault tolerance. The Houston area is home to many Spanish-speaking people, and in the Spanish language, the word “red” means “network,” including as in a computer communication network such as an Ethernet or the Internet. Thus, the name RedNet means, in effect, “network network,” making the name of the distributed system itself replicated.
network of the distributed system to the intended destination process; and when a message arrives over the network at some computer, the operating system kernel is responsible for delivering that message to the correct local destination user-level process.

Although, conceptually, any number of individual processes could run on each computer within the distributed system, for the projects in this course, each new user-level process created will run on a new (simulated) computer connected to the distributed system. When a new user-level process is created, the operating system kernel automatically finds and selects an available (simulated) computer on which to create that new process.

At system startup time, one (simulated) computer of the RedNet distributed system is initialized to be executing the operating system kernel and one user-level process. Specifically, when you run the system, you may specify any operating system kernel executable file you would like to use, and that kernel will begin execution on this initial computer being used. As new computers are utilized to execute additional processes on them as well, a copy of \textit{this same} operating system kernel will automatically be started on each of these computers, as well. Similarly, when you run the system, you may specify any user-level executable program file you would like to execute as the initial user-level process. Unlike with the operating system kernel, though, for which a copy of \textit{the same} kernel is used on all computers, when you create a new process, you may (and indeed must) specify a specific user-level program executable file to be executed in this new process; this may be the same program or a different program as is being executed in any other user-level process, but you must specify what program you want to execute.

While executing a procedure within the operating system kernel, your code is \textit{not interruptable} and executes \textit{single threaded}. Thus, you do not need to deal with any complexities of synchronization or concurrency control within the operating system kernel.

In order to be able to be executed as the initial user-level program loaded at system startup time (or likewise to be able to be executed when creating a new user-level process later), the program \textit{must} have been compiled and linked specifically for the RedNet environment; you \textit{may not} execute arbitrary (i.e., non-RedNet) programs this way. Likewise, your operating system kernel \textit{must} have been compiled and linked specifically for the RedNet environment.

Each user-level process executing in the system has a unique integer process ID. As detailed below (Section 3.1), when a new process is created, the process creation operation \texttt{CreateProcess} returns the process ID of the newly created process. Any process can at any time determine its own process ID, if needed, by calling the \texttt{GetPid} operation, also detailed below (Section 3.1). All process IDs are unique across the entire RedNet system, regardless of on which computer within the system the process is executing. No process will ever have a process ID of 0; all process IDs are positive integers.

User-level programs and kernel code for the RedNet distributed system may be written in either the C or C++ programming languages. No other programming languages are supported.

## 3 Defined RedNet Programming Interfaces

The subsections below define the programming interfaces that are a part of the RedNet distributed system environment. Function prototype and constant definitions that are a part of this interface are defined within the include file rednet.h, located in

```
/clear/courses/comp420/pub/include/rednet.h
```

Although you can (and should) read through this file using the above absolute pathname, to actually make use of this include file in your source code, you should simply include it using the source code line

```
#include <rednet.h>
```

\textit{You should not make a copy of this include file in your own local directory.}
3.1 Available User-Level Operations

The RedNet distributed system programming environment makes available a number of different operations that may be invoked by user-level processes in order to request various information or services. This section summarizes each of these available user-level operations in RedNet:

- **int SendMessage(int dest, const void *msg, int len)**
  
The `SendMessage` call may be used to send a message from one user-level process to another. The argument `dest` is the process ID of the destination process to which the message should be sent, the argument `msg` is a pointer to the location in memory where the message to be sent is located, and the argument `len` is the length (in bytes) of the message. The `SendMessage` call does not modify in any way the contents of this memory at location `msg`. The call returns immediately, queuing the message for eventual receipt by the destination process when that process calls `ReceiveMessage` as defined below. The return value from the call is 0, if no errors occurred, or some negative value indicating a specific error condition encountered in attempting to send the message. If an error is returned, no message is actually sent. The maximum length `len` value that may be passed to `SendMessage` is given by the constant `MSG_USR_MAXLEN`, specified in the include file `rednet.h`.

  As a special case, on a call to `SendMessage`, if the destination is specified as the value 0, then this message is destined specifically (and only) to the operating system kernel program executing on the same computer on which the calling user program is executing. The operating system kernel program is not expected to deliver the message itself to any user process or to transmit the message itself over the network. This special case of a `SendMessage` call sending a message only to the operating system kernel program executing on this same computer is intended to allow arbitrary communication (e.g., requests) to be passed from the user program to the kernel on this computer.

- **int ReceiveMessage(int *src, void *msg, int len)**
  
The `ReceiveMessage` call may be used to attempt to actually receive a message that has been sent to the requesting user-level process. The argument `msg` is a pointer to the location in memory where the received message should be placed on return, and the argument `len` is the length (in bytes) of the buffer in memory available there to store the received message. The argument `src` should be the address of an `int` into is the process ID of the sending process process of the received message is to be written on return from this call.

  On success, the length of the received message is returned as the return value of the `ReceiveMessage` call. If any error occurs in processing the `ReceiveMessage` call, a negative value is returned indicating the specific error condition encountered in attempting to receive the message. If an error is returned, no message is actually received, and any existing messages queued pending to be received by this process are still available (unmodified) for future `ReceiveMessage` attempts.

  If the actual length of the received message is less than the length `len`, then the remaining bytes of the buffer beginning at address `msg` are unmodified on return. However, if the actual length of the message is greater than the length `len`, then an error will be returned (as described above) and no message will be received (and the queue of pending messages able to be received by future calls is unmodified). The maximum length `len` value that may be passed to `ReceiveMessage` is given by the constant `MSG_USR_MAXLEN`, specified in the include file `rednet.h`.

3
If no messages are currently queued pending receipt by this user-level process, then the calling user-level process is blocked until a message is available to be received.

- **int CreateProcess(const char *filename, const char **argvec)**
  The `CreateProcess` call creates a new user-level process executing on a new (simulated) computer within the RedNet distributed system. This call returns immediately (i.e., it does not wait for completion of the execution of the new user-level process). On success, the call returns the process ID of the newly created process; on any error, the call returns a negative value indicating the specific error condition.

  The argument `filename` points to the null-terminated string file name of the executable program file to be executed in this new process. The argument `argvec` points to an array of null-terminated strings, giving the arguments to be passed to the `main` procedure of the program when the new process begins execution. This array is in exactly the same format as the standard `argv` argument passed to `main` in any normal Unix/Linux program. In particular, by convention, `argvec[0]` should be the name of the program itself, typically the same as `filename`, and the `argvec` array must be terminated by a `NULL` pointer value in the last entry in the array. The `argc` value that will be passed to the newly executing `main` will be the count of non-NULL string pointers before this last `NULL` entry in the `argvec` array.

  In order to be able to be executed by a call to `CreateProcess` (or to be able to be used as the program executed by the initial user-level process created at system startup time), the program must have been compiled and linked specifically for the RedNet environment; you may not execute arbitrary (i.e., non-RedNet) programs this way.

- **int GetPid(void)**
  The `GetPid` call returns the process ID of the calling user-level process. This call takes no arguments.

- **int GetPPid(void)**
  The `GetPPid` call returns the process ID of the parent process of the calling user-level process. The parent process of a given user-level process is defined to be the process that executed the `CreateProcess` call that created this user-level process. For the initial user-level process created at system startup time, the parent process ID for that process is defined always to be 0. This call takes no arguments.

- **void MilliSleep(int msec)**
  The `MilliSleep` call causes the calling user process to sleep (i.e., pause) for a period of `msec` milliseconds. Thus, for example, the call `MilliSleep(1000)` will cause the user process to sleep for a period of 1 second. This call may be very useful in creating user test programs since it allows you to easily extend the running time of the program without needing to consume a large amount of CPU time such as would occur in a long-running compute-bound application.

### 3.2 Operating System Kernel Invocation Interface

When some user-level process makes some call that requires the support of the operating system kernel, execution automatically begins within the kernel in a procedure that can handle that specific type of user-
level call. When the kernel completes handling this invocation, the kernel returns and execution in the user-level program resumes at the point at which the original call had been made.

As noted above in Section 2, when executing a procedure within the operating system kernel, your code is not interruptable and executes single threaded. Thus, you do not need to deal with any complexities of synchronization or concurrency control within the operating system kernel.

For this class, most of the functionality of the operating system kernel is abstracted away and provided for you, avoiding the need for you to write the entire operating system kernel (or even any large part of the entire operating system kernel).

Specifically, for the RedNet environment defined within this document, you need write only a single procedure of the operating system kernel. However, note that you must write this procedure yourself, with the procedure name and interface defined below (this procedure is not provided to you):

- void HandleMessage(int src, int dest, const void *msg, int len)

The HandleMessage procedure within the operating system kernel on a computer within the distributed system is automatically invoked to handle any message attempting to be sent by a user-level process executing on this same computer, or to handle any message sent by a user-level process executing on a different computer destined to a user-level process that is executing on this computer. The operating system kernel, through the HandleMessage procedure within the kernel, is thus able to inspect, modify, and/or add to the contents of messages being sent from or to user-level processes on this computer, and is able to mediate, as needed, whether or when such messages should be able to be actually sent or received. The HandleMessage procedure is able to do this by utilizing the TransmitMessage and DeliverMessage calls as defined below in Section 3.3.

For the HandleMessage procedure, the argument src is the process ID of the sending process, and the argument dest is the process ID of the destination process to which the message should be sent. The argument msg is a pointer to a location in memory where the message is currently stored, and the argument len is the length (in bytes) of the message at that location. The HandleMessage procedure should not modify in any way the contents of this memory at location msg.

In addition to these calls to HandleMessage for each such message, the HandleMessage procedure within the operating system kernel on a computer may also be invoked periodically, if you enable this feature using the -P period option on the command line when you run the RedNet environment, as described below in Section 4.2. If this command-line option is used, HandleMessage will be called periodically every period seconds; otherwise, no periodic HandleMessage calls will be made. A periodic call to HandleMessage can be distinguished from one to handle some message, since for a periodic call, the arguments src, dest, and len will all have the value 0, and the argument msg will have the value NULL.

There is no return value from the HandleMessage procedure. This procedure, when automatically called within the operating system kernel, should handle the message as appropriate and then simply return, or for a periodic call, should handle whatever periodic activities (or timer-based activities) you need within your kernel. There are no specific error conditions in terms of any of the arguments to this procedure that you need to check for or deal with. For example, other parts of the operating system kernel handle checking for conditions such as whether or not the destination process ID for handling a message is valid.

As described above, the HandleMessage kernel procedure should utilize the TransmitMessage or DeliverMessage call to actually cause, respectively, the transmission over the network or local...
user-level process delivery of a message; the TransmitMessage and DeliverMessage kernel operations are defined more fully below in Section 3.3. In general, though, if the message passed to HandleMessage is destined to the user-level process executing on the current computer (at which the HandleMessage procedure was invoked), then HandleMessage should use DeliverMessage if (and when) it wants to actually deliver the message to the locally executing destination process. If the message passed to HandleMessage is destined to a user-level process executing on some other computer, then HandleMessage should use TransmitMessage if (and when) it wants to actually transmit the message over the network to that remotely executing destination process. The HandleMessage procedure can distinguish between these two cases by checking whether or not the destination process ID is equal to the process ID of the local user-level process (i.e., as reported by GetPid()).

The separation in message handling between, at the user level, SendMessage and ReceiveMessage, and, in the kernel, TransmitMessage and DeliverMessage, allows your kernel a great deal of flexibility in how you handle messages in your own kernel HandleMessage procedure. For example, you may add additional information to a message beyond the message contents from the SendMessage call, such as adding a new kernel-level header to the message, when transmitting the message over the network with TransmitMessage, and may remove such additional information from the message as it arrives over the network, before delivering the message with DeliverMessage. You may similarly queue, within your own HandleMessage kernel procedure, any user-level message from SendMessage, before later transmitting that message over the network, or as a message arrives over the network, queue the message before later delivering it locally with DeliverMessage. You may also use TransmitMessage to transmit messages to be handled entirely by the operating system kernel on another computer, without delivering such messages from that operating system kernel to any user-level process.

It is also possible for the kernel HandleMessage procedure, together with use of TransmitMessage, to send and receive (and completely process) messages directly between the operating system kernel executing on one computer and the kernel executing on another computer. In particular, while executing inside HandleMessage, it is possible to also create new messages (that did not originate with a SendMessage call from some user-level process) and to use TransmitMessage to transmit them over the network other other computers. When a message arrives from the network, HandleMessage in the kernel on the destination computer is invoked. It is possible for HandleMessage at that destination computer to completely deal with such a message, without delivering it to any user-level process. Such messages would effectively be kernel-to-kernel messages, rather than the more typical messages that are actually between two user-level processes. For example, you could add a new kernel-level header to the message before transmitting it, as suggested above, to indicate to the receiving kernel HandleMessage procedure how to handle the message.

Again, note that you must write a procedure named HandleMessage, as defined above. If you do not implement a procedure of this name (with this interface), your operating system kernel will not link correctly and instead will produce an error from the linker. You must write your own HandleMessage procedure; this procedure is not provided to you.

### 3.3 Available System-Level Operations within the Kernel

As with the user-level operations defined above in Section 3.1, the RedNet distributed system programming environment also makes available a number of different operations that may be invoked within the operating system kernel when executing in the kernel on behalf of some particular user-level processes (i.e., when executing within a procedure called for an operating system kernel invocation, as defined above in Section 3.2). This section summarizes each of these available system-level kernel operations in RedNet:
When a user-level process attempts to send a message using the user-level SendMessage call (Section 3.1), the message is not actually sent immediately. Rather, the SendMessage user-level call cause the kernel’s HandleMessage procedure, as defined above in Section 3.2, to be invoked to mediate the transmission of this message. Through the HandleMessage procedure, the kernel is able to, for example, inspect, modify, and/or add to the contents of messages being sent, and to determine, as needed, whether or when the messages should actually be able to be transmitted over the network to the destination process.

If/when the kernel HandleMessage procedure wants to actually transmit the message to the destination process over the network, it should call TransmitMessage. The argument src to TransmitMessage is the process ID of the sending process, and the argument dest is the process ID of the destination process to which the message should be transmitted. The argument msg is a pointer to a location in memory where the message is currently stored, and the argument len is the length (in bytes) of the message at that location. The TransmitMessage call does not modify in any way the contents of this memory at location msg. The maximum length len value that may be passed to TransmitMessage is given by the constant MSG_SYS_MAXLEN, specified in the include file rednet.h.

As noted above in Section 3.2, the separation in message handling between, at the user level, SendMessage and ReceiveMessage, and, in the kernel, TransmitMessage and DeliverMessage, allows your kernel a great deal of flexibility in how you handle messages in your own kernel HandleMessage procedure. For example, you may add additional information to a message beyond the message contents from the SendMessage call, such as adding a new kernel-level header to the message, when transmitting the message over the network with TransmitMessage, and may remove such additional information from the message as it arrives over the network, before delivering the message with DeliverMessage. You may similarly queue, within your own HandleMessage kernel procedure, any user-level message from SendMessage, before later transmitting that message over the network, or as a message arrives over the network, queue the message before later delivering it locally with DeliverMessage. You may also use TransmitMessage to transmit messages to be handled entirely by the operating system kernel on another computer, without delivering such messages from that operating system kernel to any user-level process.

As a special case, if enabled by use of the -N option on the command line, as described in Section 4.2, to generate a complete network of computers and simulated “network topology” in which each of these computers is directly connected to only a small number of other computers in the system, it is possible to specify in a call to TransmitMessage that the message is to be “broadcast” to all other computers to which this computer is directly connected. Such a broadcast TransmitMessage call can be specified by giving the special value -1 as the destination address dest. If the -N option is not enabled, any such call to TransmitMessage with a dest value of -1 will return an error.

When a message sent by a user-level process on another computer within the distributed system arrives over the network at the computer on which the destination process is executing, the message is not actually delivered to that user-level process immediately. Rather, the arrival of the message from the network cause the kernel’s HandleMessage procedure, as defined above in Section 3.2, to be invoked to mediate the delivery of this message. Through the HandleMessage procedure, the
kernel is able to, for example, inspect, modify, and/or add to the contents of messages being delivered, and to determine, as needed, whether or when the messages should actually be able to be delivered to the local destination process.

If/when the kernel HandleMessage procedure wants to actually deliver the message to the local user-level process, it should call DeliverMessage. The argument src to DeliverMessage is the process ID of the sending process, and the argument dest is the process ID of the local destination process to which the message should be delivered. The argument msg is a pointer to a location in memory where the message is currently stored, and the argument len is the length (in bytes) of the message at that location. The DeliverMessage call does not modify in any way the contents of this memory at location msg. The maximum length len value that may be passed to DeliverMessage is given by the constant MSG_SYS_MAXLEN, specified in the include file rednet.h.

As noted above in Section 3.2, the separation in message handling between, at the user level, SendMessage and ReceiveMessage, and, in the kernel, TransmitMessage and DeliverMessage, allows your kernel a great deal of flexibility in how you handle messages in your own kernel HandleMessage procedure. For example, you may add additional information to a message beyond the message contents from the SendMessage call, such as adding a new kernel-level header to the message, when transmitting the message over the network with TransmitMessage, and may remove such additional information from the message as it arrives over the network, before delivering the message with DeliverMessage. You may similarly queue, within your own HandleMessage kernel procedure, any user-level message from SendMessage, before later transmitting that message over the network, or as a message arrives over the network, queue the message before later delivering it locally with DeliverMessage. You may also use TransmitMessage to transmit messages to be handled entirely by the operating system kernel on another computer, without delivering such messages from that operating system kernel to any user-level process.

- int GetPid(void)

The GetPid call within the kernel returns the process ID of the user-level process on behalf of which the kernel is currently executing. This call takes no arguments. This call is directly analogous to the GetPid user-level call defined above in Section 3.1.

- int GetPPid(void)

The GetPPid call returns the process ID of the parent process of the user-level process on behalf of which the kernel is currently executing. This call is directly analogous to the GetPPid user-level call defined above in Section 3.1.

- int CreateCheckpoint(void) The CreateCheckpoint call creates a new (simulated) tentative checkpoint of the user process running on the computer from which this system kernel program invokes the call. Each checkpoint created is assigned a “checkpoint handle,” and the checkpoint handle of the new tentative checkpoint is returned as the result of the call. If a tentative checkpoint of this process already exists, it is deleted first.

- int CommitCheckpoint(int ckph) The CommitCheckpoint call commits the tentative checkpoint of this process specified by the checkpoint handle ckph, making it become a permanent checkpoint. If a permanent checkpoint of this process already exists, it is deleted first. If the specified
ckph does not define an existing tentative checkpoint of this process, an error code (less than 0) is returned; otherwise, on success, the value 0 is returned.

- `int DiscardCheckpoint(int ckph)` The `DiscardCheckpoint` call discards the tentative checkpoint of this process specified by the checkpoint handle `ckph`. If a permanent checkpoint of this process exists, it is not affected by this call. If the specified `ckph` does not define an existing tentative checkpoint of this process, an error code (less than 0) is returned; otherwise, on success, the value 0 is returned.

## 4 Compiling, Running, and Using RedNet

### 4.1 Compiling Your System and User Programs

Your system (operating system) kernel and user test programs may be implemented in either the C or C++ programming languages; no other programming languages are supported. All programs must be compiled and executed on the Rice CLEAR Linux facility.

Template Makefiles that you can use for compiling your system and user programs are available in the directory

```
/clear/courses/comp420/pub/samples
```

In particular, the following three Makefile templates are available there:

- `Makefile` is a top-level Makefile that can be used as-is to build your system kernel test program as well as your user-level test programs.

- `Makefile.sys` is a template for a Makefile that can be used to build your system kernel test program.

- `Makefile.user` is a template for a Makefile that can be used to build your user-level test programs.

Please copy these three files to your own directory in order to use them. Also, please read the comments at the top of each of these three files for more information on how to use them. In particular, you must modify your copy of each of `Makefile.sys` and `Makefile.user`, as described in the comments at the top of each file.

In addition to these three Makefile-type files, this directory also contains the source code to a simple example user-level test program. This program, in the source file `pingpong.c`, sends a sequence of “ping-pong” messages back and forth between two RedNet processes. Please see comments at the top of this source file, and the source code itself, for more information. This sample program is provided in order to show examples of programming for RedNet and to get you started in thinking about writing your own user-level test programs.

To help with debugging your programs the RedNet environment provides a “tracing” output procedure, similar to `printf`, that you can call to output information as your code executes. Specifically, to generate trace output from anywhere in your code, you may call

```
TracePrintf(int level, char *fmt, args...)
```

where `level` is an integer tracing level, `fmt` is a format specification in the style of `printf`, and `args...` are the arguments needed by `fmt`. Generally, `level` should be greater than or equal to 0.

When you run the RedNet environment, as described below in Section 4.2, you may specify a trace level to enable separately for your operating system kernel program and for your test user program. You should
generally not need to, but you may also enable tracing for the internal RedNet control mechanisms (my simulation support code), which may at times help with debugging strange problems if you encounter any.

In particular, for each of these three different parts of the RedNet environment (operating system kernel program, user test program, and internal control mechanisms), you can set the enabled “tracing” level set to any integer. For any call to TracePrintf, if the enabled tracing level (for the part of the RedNet environment from which the call is made) is greater than or equal to the level argument to TracePrintf, the output generated by fmt and args... will be added to the trace. Otherwise, no trace output is generated from this call to TracePrintf.

This TracePrintf tracing feature is designed to aid in your debugging, and it is strongly recommended you make use of it. Specifically, TracePrintf is similar to normal printf but is implemented specially to provide much more capability than normal printf. In particular, with TracePrintf, you can, for example, on the command line dynamically turn up or down (or on or off) different types of debugging tracing each time you run the system. The TracePrintf facility also guarantees all tracing output is written atomically to the trace file, appending each new line of tracing output exactly at the current end of the trace file, thus recording a total order trace across all computers in the (simulated) system (possible only because it is a simulated distributed system).

4.2 Running RedNet

Your operating system kernel program and user-level test programs must be compiled using the Makefiles described above in Section 4.1 and must be executed on the Rice CLEAR Linux facility.

As described above in Section 2, in the RedNet distributed system, a separate copy of the same operating system kernel runs on each computer within the distributed system. Each user process executing on the RedNet distributed system may run a different user-level program and/or may run a copy of the same user-level program as is being run by other processes. Each user process executes independently on a separate computer in the distributed system, with its own separate, private memory address space, not shared by any other processes running on the system. An independent copy of the same operating system kernel program executes on each of these computers in the system.

To run the RedNet distributed system (simulator), you should run the program

```
/clear/courses/comp420/pub/bin/rednet
```

This program should be run from the Unix command line while you are logged in to any of the CLEAR compute servers. You may run this program in any of several different ways, whichever you find more convenient:

- You may run it by using the full absolute pathname above.

- You may alternatively put the directory name of the rednet program, `/clear/courses/comp420/pub/bin` on your shell’s command search path. Once you have added this directory to your shell’s search path, you can then run the program simply using the command name rednet.

- You may alternatively create a symbolic link to that absolute pathname in your current directory (e.g., the directory where you work on the project) by using the shell command

  ```
  ln -s /clear/courses/comp420/pub/bin/rednet 
  ```

  Once you have created this symbolic link, you can then run the program simply using the command `./rednet` while you are in that directory.
You may not make a copy of the rednet program itself.

When you run the rednet program, you must supply on the command line the name of the operating system kernel program and test user program, and you may also put a number of Unix-style switches on the command line to control some aspects of execution.

In particular, the full command line to run rednet (e.g., using the third method of running the program, using a symbolic link, described above), should look like

```
./rednet control_args system_program -- user_program user_args...
```

The use of control_args on the command line is optional but if present may include any of the following arguments that affect how the RedNet simulation behaves:

- `-ls level`: Set the tracing level to be applied to TracePrintf calls made during this run by your operating system kernel program.

- `-lu level`: Set the tracing level to be applied to TracePrintf calls made during this run by your user-level test program.

- `-lc level`: Set the tracing level to be applied to TracePrintf calls made during this run by the internal RedNet control mechanisms (my support code).

  You should generally not need to enable this kind of tracing, but it may be helpful in debugging some strange problems if you encounter any.

- `-t tracefile`: This option can be used to specify the name of the file to which the tracing output should be written. If this option is not specified, the default tracefile name is the file “TRACE” in the current directory.

- `-s`: Send any tracing output to the Linux stderr file (this is usually your screen) in addition to sending it to the tracefile.

- `-P period`: Enable periodic calls to the HandleMessage procedure in your operating system kernel, as described above in Section 3.2. The value period specifies the periodic interval (time between successive such HandleMessage calls) in seconds. These periodic HandleMessage calls are not related to any message handling activities indicated by other HandleMessage calls. If the `-P period` option is given on the command line, the value period must be greater than or equal to 1; if the `-P period` option is not given on the command line, periodic calls to HandleMessage are not made.

- `-N`: Generate a complete network of 32 separate computers and a simulated “network topology” in which each of these computers is directly connected to only a small number of other computers in the system. Transmitting a “broadcast” message from your operating system kernel program, as described above in Section 3.3, will be delivered only to those other computers to which the transmitting computer is directly connected in this network topology. All 32 computers in this complete network topology begin executing the same user program as well as the same operating system kernel program. The simulated network topology is guaranteed not to be partitioned, in that all computers are guaranteed to be transitively (possibly multi-hop) connected to each other, even though only a limited number of computers are directly connected.

The control_args command line arguments described above, if present, are automatically parsed and interpreted for you before your operating system kernel or user test program begins execution. The user of any of the control_args arguments is optional.
Following the `control_args` on the command line, you must specify the file name of the operating system kernel program you want to execute on each computer in the distributed system.

Following that on the command line must be the special separator argument “--” followed by the file name of the user test program you want to execute as the initial program, plus any `argv` arguments you want to pass to that user program’s `main` procedure. The separator argument “--” must be present on the command line, as a separate command-line argument, between the file name of the operating system kernel program you want to execute on each computer and the file name of the user test program you want to execute as the initial program.

For example, the following Unix command

```
./rednet -ls 10 -lu 5 testsys -- testusr arg1 arg2
```

runs the RedNet distributed system environment (simulator), running the operating system kernel program `testsys` and the user test program `testusr`. The arguments `arg1` and `arg2` are visible to the user program in `argv[1]` and `argv[2]`, respectively, as passed to the user program’s `main` procedure. Any `TracePrintf` calls from the operating system kernel program with a tracing level less than or equal to 10 are recorded in the trace file, and any `TracePrintf` calls from the user program with a tracing level less than or equal to 5 are likewise recorded in the trace file. All such tracing is recorded in the trace file with the default name of “TRACE” in the current directory. If the trace file already exists when this run of `rednet` begins, the existing file is deleted and a new trace file of this same name is begun.