SPEDE

A Simple Programming Environment
for
Distributed Execution
Rev 4.0

Jim Gochee, © 1992

Senior Thesis
Dartmouth College
1992

The Author would like to thank the following people for their ideas and support:

David Kotz - Thesis Advisor

Mark Franklin - Thayer School of Engineering

Nicholas Wilt - Fellow '92

Table of Contents

SPEDE Overview

Traditional single processor computers are quickly reaching their full computational potentials. The quest for faster and faster chips have brought technology to the point where the laws of physics are hampering future gains. Significant gains in speed must therefore come from using multiple processors instead of a single processor. This technology usually represents itself in the form of a parallel computer, such as the Connection Machine Model 5. Recently however, much interest has been focused on software that organizes single processor computers to behave like a parallel computer. This is desirable for sites which have large installations of workstations, since the cost of new parallel systems are prohibitive. SPEDE, a Simple Programming Environment for Distributed Execution, was designed for this purpose. It allows UNIX based machines of varying hardware types to be organized and utilized by a programmer of parallel applications. SPEDE is a user level system in that it requires no special privileges to run. Every user keeps a separate copy of the system so that security issues are covered by the normal UNIX operating environment. SPEDE is characterized as a large grained distributed environment. This means that applications which have a large processing to I/O ratio will be much more effective than those with a small ratio.

SPEDE allows users to coordinate the use of many computers through a straightforward interface. Machines are organized by classes, which are terms that can be used to label and group them into more manageable units. For example, users might want to create a class based on the byte ordering of machines, or by their location. Users can then specify more completely which machines they want to use for a particular session. Sessions are essentially the interaction between *objects* in the SPEDE environment. A user creates an object to perform a certain task, such as constructing part of a fractal image. Objects can send and receive messages from other objects using a simple interface provided with SPEDE. Objects are machine independent, which means that the same object can be run simultaneously on different platforms. This is achieved by translating all messages into standard network byte ordering. However, if user data is being passed between objects, it is the user's responsibility to make sure byte ordering is correct.

The SPEDE system involves several major components. These components help control and manage object interactions. Figure 1 shows a running session running with three machines (each surrounded by an oval rectangle). There are also three objects running, two named MandComp and one named Mand. Each object is on a different machine, although it is possible to have multiple objects on a single machine. In the figure, the lines connecting the various entities represent socket connections. UNIX sockets are the transport mechanism used in SPEDE, although one could implement a lower level protocol for more efficient communication. Sockets can also be a problem because some machines have strict limits on the number of connections a user can have open at any given time.

Warning:

SPEDE should not be run by a priveledged user, nor should priveledged objects be compiled. Components in the SPEDE system assume a trusted environment, so no authentication is performed. It is possible for a mal-intentioned user to command someone else’s run-time daemon to launch objects under the other user’s name. Only objects in the special run time directory can be accessed.

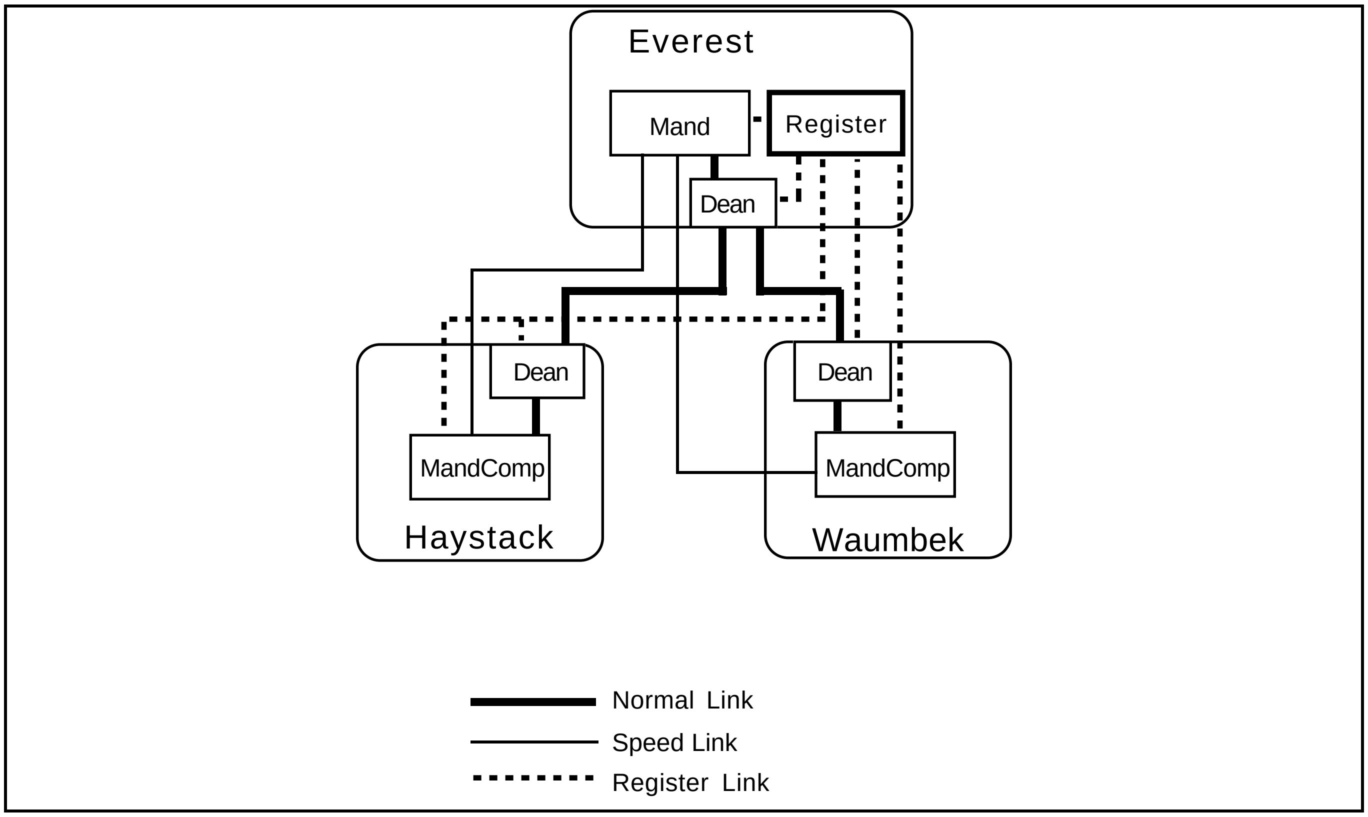

Figure 1

Along with the objects, each machine also has a process called the *dean*. Objects and deans are linked together with sockets. Every process also has a connection with a process called *Register*. Register keeps a table of all running entities such as deans and objects, with the UNIX port number that each can be reached at. Register is also used to start and stop deans.

The dean process (which runs as a daemon) is the backbone of the SPEDE system. It is the means by which objects create other objects and the channels through which all object communication passes. Deans can either be started by the user from Register, or automatically when an object needs to make a connection to a remote machine (for which a dean must be present). It is more efficient to manually start the deans before a session begins, since the overhead of automatically creating a dean is quite high. Once the deans are running, they remain in a passive state waiting for commands from objects. If a dean is idle for more than 10 minutes, it shuts down. This is desirable in most circumstances because it prevents deans from reaching a state where they run in forever. If the user does not want the deans to terminate, it is possible to tell Register to keep the deans alive. Register can also be used to kill running deans.

SPEDE Installation

SPEDE comes as a tar file called `spede.tar`. Copy this file into the directory where you want the SPEDE source to be installed. When this is done, enter the following command to untar the file:

```
tar -xf spede.tar
```

SPEDE creates one directory at the level of the tar file named `spede`. This directory contains the SPEDE system programs and some sample objects. The only other directory you will need is a binaries directory. A binaries directory is used to keep the SPEDE executables in a well known place. If a shared file system is used with machines of differing types, then each type of machine should have its own binaries directory. One common way to do this is to have a standard root binary directory, such as `~/bin/spede`. From there, each type of machine, such as 'sun3' or 'ibm_rs6000' can have its own sub directory. In order for SPEDE to tell which binaries directory a machine is using, an environment variable called `SPEDE_BIN` must be set.

To set `SPEDE_BIN` when using a shared file system, your system must have some way of determining its architecture type. The UNIX command "`uname -m`" is one common way to get this information, although some systems have different commands. Contact your system operator if you do not know how to get this information. Note that you must also create the directories described by `$SPEDE_BIN` .

Example Environment Settings

```
setenv MACHINE `uname -m`
setenv SPEDE_BIN ~/bin/spede/$MACHINE
```

Compilation

Once the SPEDE binaries directory has been set up, you are ready to compile. To do this:

1) Change into the SPEDE directory

```
cd spede
```

2) Make sure no old object files or binaries are lying around

```
make clean
```

3) Compile the system executables and sample objects using the "`make`" command. NOTE: some of the sample objects may not compile due to limitations of you local machine.

```
make install
```

When you run "`make`" from the `spede` directory, binaries for the dean, register, and objects will be automatically stored in `$SPEDE_BIN`. This is why it is necessary to set `SPEDE_BIN` before running "`make`".

Data Structures

The interface to SPEDE is provided to the user through C function calls. These calls are implemented in a special library that is linked in with every object. The following data structures are used throughout the SPEDE system.

Object Descriptors

Most routines that manipulate objects will take a parameter of type **object_instance ***. This type describes how to communicate with the object and contains other pertinent information.

```
typedef struct{
  char name[40];                        /* Name of remote machine         */
  long addr;                            /* Unix address of remote machine */
} node_des;

typedef struct {
  node_des node;                        /* Remote machine descriptor      */
  long obj_ref;                         /* Ref num to use with remote     */
} link_des;

typedef struct {
  link_des creator;                     /* Points to who created the obj  */
  link_des storesite;                   /* Points where obj is backed up  */
  link_des location;                    /* Points to where obj is running */
  char type[20];                        /* Ascii name of the object       */
  long  id;                             /* The object's ID number         */
  boolean is_fault_tolerant;            /* Is the object fault tolerant   */
  boolean debug;                        /* Is the object being debugged   */
  boolean state;                        /* The state of the object        */
  void *data;                           /* Used internally                */
  long dean_ref;                        /* Ref num to use with dean       */
} object_instance;
```

The type **node_des** is used to describe a remote machine as a combination of the machine name and the machine's UNIX address. The type **link_des** is used to describe an *object* that has already been created by a remote dean. The location of the object is given by the node and the reference number of the object is given by **obj_ref**. The user need not access the fields of this structure directly since there are library calls to manipulate the information. The most widely used data type is the **object_instance**. This type is returned by the object creation routine and is used in all communications with the object. None of the fields in this structure should be altered by the user because the SPEDE system stores critical information here (particularly in the data field).

Object Communication

Communication between objects is done through commands. The underlying SPEDE system uses many unique commands to control object interactions. The user however, is restricted to using only the **invoke function** command. This command tells a remote object to execute a specific user defined function given a set of parameters passed with the command. The user accesses the **obj_message** command through C function calls described later on. When a local object sends a message to a remote object, the remote object returns a result and optionally some parameters.

```
#define MAX_PARAMS 20

typedef struct {
  void *data;                             /* A pointer to malloced memory */
  long len;                               /* The length of the data       */
} parameter;
```

```
typedef struct {
  short num_params;                    /* How many params are there */
  parameter param[MAX_PARAMS];         /* An array of parameters    */
} parameter_block, pblock;

typedef struct {
  long status;                         /* Has the message completed yet */
  long result;                         /* What is the return value      */
  pblock *r_params;                    /* Return parameters (or NULL)   */
} return_info;
```

The **parameter_block** structure has a set of library routines that are used to manipulate it. The user may access these fields directly, although this is discouraged. The **return_info** structure is sent back from the remote object when the command has finished processing. If no return parameters are used, the **r_params** field will be null.

Running Objects

Every object that runs from the UNIX command line has an **object_descriptor** set up for it. This descriptor is an interface between the user and the object. Objects will often want to create other objects, and the descriptor can be used to see where the user wants those objects to run. For example, if the role of an object called **mand** is create worker objects to complete a task, then **mand** would like to know how many objects to create and on what machines. This could be compiled into the object itself although that would require recompiling the program every time this information changed. With the object_descriptor, **mand** has direct access to the user's command line specifications.

```
typedef struct {
  int num_objects;                  /* Default to 0 */
  char classes[100];                /* Default to empty string */
  int num_machines;                 /* Default to 0 */
  char **machine;                   /* Default to 0 pointer */
} object_descriptor;
```

The fields of this structure are constructed by SPEDE when the user launches an object with certain command line parameters. These parameters are described in a section below.

Programming an Object

A SPEDE object is defined as a set of user functions (and data). The object is a C file compiled into a standalone program, whose interactions with other objects is controlled by the compiled-in SPEDE library. The user's functions are accessed through a callback function with a message type. The message type will indicate to the object which function should be performed. Every function that an object performs will have a different message number, which must be specified by the user. Message numbers 0-10 are reserved by SPEDE for sending special messages to an object. These messages include CREATE and DELETE, which the object can ignore if no special processing is required. The general form of an object file is:

Header Information
<User Functions>
Message Callback Function
Main
Free Launch
Normal Launch

Each of these sections is described below.

Header Information

The following header files must be included for the object to compile properly. Also, a string called **object_name** must be defined with the name of the object.

```
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>

/* These include files are necessary for objects */
#include "sockets.h"
#include "spede.h"
#include "spede_io.h"
#include "objects.h"
#include "objlib.h"


char *object_name = "template_obj";
```

User Functions

User functions are the functions written by the user to handle messages. Here is where the actual computations occur.

Message Callback Function

The message callback function is the glue between SPEDE and an object. Object command messages received by SPEDE are sent to an object via the callback. The callback routine is chosen by the user and passed as a parameter when the object is initialized. The callback routine should have the following format:

```
static long MyCallback(short message, pblock *params, block **rparams);
```

The function to be performed is indicated by **message**. Parameters to the function are sent in **params**. If there are any return parameters, **rparams** is used. Either **params** or **rparams** can be NULL, indicating no parameters. A typical callback looks something like this:

```
static long callback(short message, pblock *params, pblock **r_params)
{
  long result;

  r_params = NULL;                        /* Null unless we set elsewhere */
  switch(message) {                       /* Which message were we sent?  */
  case CREATE:                            /* Allows object to alloc mem   */
    break;
  case DELETE:                            /* Tell object to shut down     */
    break;
  case user_message1:
    result = my_function(params);         /* Don't need to use r_params   */
    break;
  case user_message2:
    result = my_function2(params, r_params);   /* This uses r_params      */
    break;
  default:
    result = 0 ;
    break;
  }
  return(result);
}
```

Main

The **main** function of the object file is shown below. The user should not put any other code here. When an object starts up, the SPEDE library will call one of two user functions which should be used by the user instead of **main**. These two functions are described in the next section. The third parameter to the function **init_objects** is the address of the callback function from the previous section.

```
main(int argc, char **argv)
{
  if (init_objects(argc, argv, callback)) {
    perror("Error trying to initialize the object library");
    exit(1);
  }
  register_dean();                      /* Connect into the SPEDE network */
}
```

Free_Launch

When an object is started up, the SPEDE library will call one of two routines in the object file. If the object was started from the command line (known as a free launch), the function '**free_launch**' is called. If the object was started by a *dean* in response to a creation command, then '**normal_launch**' is called. These routines give the user a chance to set up configuration info, which would have normally been done in **main**. The format of the free_launch command is:

```
static void free_launch(int argc, char **argv);
```

Normal_Launch

When an object is started by a dean (in response to a creation command from another object), the routine **normal_launch** is called. Like the function **free-launch**, **normal_launch** gives the user a chance to set up an configuration information.

```
static void normal_launch(void);
```

Sample

A sample object, which does nothing but show the framework of an object, is given below:

```
/* Sample Object */
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>

#include "sockets.h"            /* These include files are necessary for objects */
#include "spede.h"
#include "spede_io.h"
#include "objects.h"
#include "objlib.h"

char *object_name = "MyObject";

void initialize()                 /* Allocate memory */
{
}

void deallocate_mem()             /* Deallocate memory */
{
}

long do_work1(pblock *params)     /* Do some work */
{
   return(0);
}

static long callback(short message, pblock *params, pblock **r_params)
{
  long result = 0;

  r_params = 0L;                          /* I never have r_params */
  switch(message) {
  case CREATE:                            /* Set up some memory    */
    initialize();
    break;
  case DELETE:                            /* Free the memory       */
    deallocate_mem();
    break;
  case MESSAGE1:
    result = do_work1(params);            /* Do some work          */
    break;
  }
  return(result);
}

main(int argc, char **argv)
{
  if (init_objects(argc, argv, callback)) {
    perror("Error trying to initialize the object library");
    exit(1);
  }
  register_dean();                     /* Connect into the SPEDE network */
}

void free_launch(int argc, char **argv)
{
  printf("ERROR!!! You cannot start this object from the command line\n");
  exit(0);
}

void normal_launch()
{
  /* We allocate memory when the CREATE message is received, but we could
   * do it here instead.
   */
}
```

Compilation and Makefiles

For an object to work in the SPEDE environment, it must be linked to the correct library file. Shown below is a sample Makefile that will correctly build an object.

```
# Makefile to compile an object, SPEDE_BIN should be an environment variable

LIB = $(HOME)/spede/lib
INCLUDE = $(HOME)/spede/include
CFLAGS = -I$(INCLUDE)
MY_OBJS = myobject.o $(LIB)/objlib           # Links the correct library
```

```
myobject: $(MY_OBJS)
        $(CC) $(CFLAGS) -o myobject $(MY_OBJS)

install:
        cp mand $(SPEDE_BIN)                  # A well known location for objects
```

Command Line Parameters

The following options can be used on the command line when launching an object:

```
- h <hostname>       - Specifies the hostname of the Register process
- p <port number> - Specifies the port number of the Register process
- n <number objects>     - Specifies the number of objects to create
- c <class list>  - Specifies which machines to create objects on (by class)
- m <machine list>       - Explicitly specifies which machines to create objects on
- f <file name>   - Lists a file that has the object information.
```

Every object that runs must connect with a Register process. There are two ways this can be accomplished. One is by the user specifying the location of a Register process from the command line. This is done with the -h and -p options. If these options are not present, then the object attempts to read the file '~/.reginfo'. This file should contain a hostname and port for the last running Register. If that Register is still running, then the object connects. If not, then the object aborts and the user has to start a new Register.

The other command line options give an object hints about what other machines to use. These options will set the **run_info** global variable, whose structure was defined earlier. The object can choose to use these user hints, or ignore them and use its own information. The format for the file used in the -f option is:

```
# This is an object_info file
# Number of objects to create
5
# Class descriptors
rs6000#dec
# Machine descriptors
polaris
```

If the user specifies both machines and classes, SPEDE will combine the resulting machine list into one list. The ordering of the list gives the specified machines priority over the class chosen machines. If the user does not specify the number of objects to use, SPEDE sets that to be the number of machines that were specified (by the -m and -c parameters).

Example

```
mand -h polaris -p 4055 -n 10 -c rs6000
mand -c rs6000
```

The first example starts the object **mand** and tells it to use the Register process located on the machine 'polaris' at port number 4055. Ten objects are recommended to **mand**. These objects should run on machines of the class **rs6000**. The second example assumes that the file '~/.reginfo' exists, so no Register information is given. Machines of the class **rs6000** are to be used, and since there are no specified number of objects to use, one object will be used for each machine.

Object Functions

Objects are able to function and communicate with other entities in the SPEDE system by a special library that is linked in with each object. This object library offers an array of function calls to allow an object to interact with other objects. The types of calls are broken down into the following categories:

- Initialization
- Creating/Deleting objects
- Parameter Blocks
- Sending messages to objects
- Miscellaneous calls

Initialization

The **main** function of the object file (which has a fixed format) uses two functions to connect with the SPEDE library. The format for these are:

```
int init_objects(int argc, char **argv, void *function_callback);
void register_dean(void);
```

Global Variables and Defines

```
#define OBJECT_OK        0
#define CREATING_OBJECT  1

#define INVALID_REMOTE   -100    /* SPEDE error messages */
#define INVALID_OBJECT   -101
#define NETWORK_ERROR    -102
#define CREATION_TIMEOUT -103

int SpedeError;                   /* Set after each SPEDE library call */
```

Creating and Delete Objects

There are two ways to create an object, synchronously or asynchronously. If the object is created synchronously, the call blocks until the remote object has been created. This can be a long wait, especially if the remote machine does not have a dean running on it. A more efficient approach is to create the object asynchronously. This requires one more function call to wait until all asynchronous objects have been created.

```
object_instance *create_object(char *machine_name, char *type, boolean debug,
                                       boolean is_fault_tolerant, boolean async);

int sync_created_objects();

int get_object_status(object_instance *object);

int delete_object(object_instance *object);

object_instance *link_object(link_des *remote_link);

object_instance *speed_link(link_des *remote_link);
```

Create_object will communicate with the SPEDE network to create the specified object on the given remote machine. If **async** is true (non zero), then the call will return immediately even though the returned **object_instance** will not be valid until the object has been created and verified.
Sync_created_objects should be called before accessing any asynchronously created object. If an error occurred while creating a synchronous object, the return value will be NULL and **SpedeError** will

contain the error number. If an error occurs when an object is created asynchronously, **sync_created_objects** will return false (0). The user must then check the status of each object to see which one(s) could not be created. This is done with the **get_object_status** routine. **Get_object_status** will return either OBJECT_OK, CREATING_OBJECT, or one of the SPEDE error messages.

Call **delete_object** to terminate an object. The **object_instance** pointer is freed and a delete message is sent to the remote object. This call blocks until the object has been deleted, or an error occurs. The return value will be OBJECT_OK, or one of the SPEDE error messages.

Link_object and **speed_link** allow access to an object from an object's link descriptor (link_des). This descriptor gives the host information and object reference number for a given object, which can be gotten from the **location** field of an **object_instance**. These routines are provided to allow child objects to communicate amongst themselves. For instance, object A creates two child objects, B and C. A then wants B and C to send messages back and forth. For this to happen, B and C must be told of each others existence. This is done by passing B and C a link descriptor for each other, at which point they can form a link using **link_object**. They can then communicate over the link as if they had created the object themselves. A speed link is a variation on **link_object**. It is used to bypass the remote object's dean. This will cut down on network overhead, although the in future, the object will lose the ability to be debugged and kept fault tolerant (to be implemented).

Example

```
/* Create two objects. Return 0 if call completed, or error code
 * if call failed.
 */
int my_create_objects(object_instance **object1, object_instance **object2)
{
  *object1 = create_object("everest", "MandComp", FALSE, FALSE, TRUE);
  *object2 = create_object("haystack", "MandComp", FALSE, FALSE, TRUE);
  return(sync_created_objects());
}
```

Parameter Blocks

Parameters are used when objects have data to send to one another. Parameter blocks can be manipulated with several SPEDE routines.

```
pblock *new_params(int num_parms, long param1 size,...);

void set_params(pblock *p, void *p1data, ...);

void get_params(pblock *p, void **p1data, ...);

void free_params(pblock *p);
```

To create a new **pblock** structure, use **new_params**. The first parameter to **new_params** is the number of parameters to reserve in the **pblock**. Each parameter after that will describe the size in bytes for every parameter in the **pblock**. To assign values into a **pblock** structure, use **set_params**. **Set_params** takes as its first argument the **pblock** to work with. The next arguments are pointers the data of the parameters, which **set_params** copies into the parameter block. It knows the size of each parameter from the **new_params** call. **Get_params** works in the other direction. It assigns user variables to point to its internal data for each parameter. Use **free_params** to deallocate memory associated with a parameter block.

Sending Messages

Objects communicate amongst themselves by sending each other 'messages'. These messages are integer values representing a certain function to be done. When an object sends a message to another object, a return message is sent unless the user instructs SPEDE to ignore the return.

```
return_info *send_obj_message(object_instance *object, short message,
          pblock *params, boolean async, boolean send_return);

int send_obj_mess_callback(object_instance *object, short message,
          parameter_block *params, void *callback, void *data);
```

The simplest call to use when communicating with another object is **send_obj_message**. It takes the destination object as the first parameter, followed by a message number, some parameters, and then **async** and **send_return**. **Async**, if true, will force the call to return immediately. The user must then poll the status filed of the **return_info** structure. The status field will be set to 0 if the call is completed, 1 if the call is still in progress, or a negative number if an error occurred. If **send_return** is set to false, in which case the user expects no return value, the routine returns immediately with NULL for the **return_info**. There is no way to know when the remote call completes **send_return** is false.

Spede Idle

In order for SPEDE to get processing time to check for object messages, the user must periodically call **idle_objects**. This is actually only necessary if the user is expecting data. The interface to idle_objects allows the user to specify a MAXIMUM time given to SPEDE for processing object commands. The routine will return earlier than the specified time however after processing a single command. This allows the user to call idle_objects, then check a return status value to see if an outstanding call has completed.

```
void idle_objects(int secs, int usecs);
```

The two parameters specify how long idle_objects should wait if no incoming messages exist.

Example

```
r_info = send_object_message(obj, DO_WORK, params, TRUE, TRUE);

while(r_info->status == 1)               /* While call is outstanding   */
  idle_objects(1000, 0);                 /* Wait for return message     */
```

Miscellaneous

The miscellaneous routines are as follows:

```
void reset_times(object_instance *object);

void get_object_times(object_instance *object, double *idle, double *cpu,
                                double *writing);
```

These two routines allow the user to reset the internal timing info of an object, and get the timing info at a later time. Time values are in seconds. Idle time expresses the seconds spent idle (i.e., waiting for something to do). The cpu time is the time spent within an object_message call, which means the object is doing work. The writing time measures how many seconds the object spent writing return data.

Sample Objects

To see some sample objects, refer to the Appendix of this paper.

Register

The Register application must be running at a known location before any other of the SPEDE components can function. The reason for this is that only Register will know how to contact remote machines. Without Register, a user would have to manually run deans on every remote machine and keep track of their port numbers. If the Register process dies for any reason, all running deans will loop trying to connect. If a Register process is then re-started, the remote deans will automatically connect.

SPEDE components, including the dean and all objects, get the location of Register by either reading a configuration file in the user's home directory, or by getting the information from the command line. The second option is always used when Register itself launches a remote object or dean. If the user launches an object from the command line, SPEDE first looks to see if the user provided the Register information. If not, SPEDE attempts to find a Register process by reading a file named '~/.reginfo'. This file is written by a Register process each time it is started. The file has information about where and on what port the Register process is running. The format for the file is (although the user need not alter this file):

```
#This is .reginfo
everest.cs.dartmouth.edu        # Machine name
4056                            # Port number
```

To run the Register, simply cd to the correct binaries directory for the machine you are on (or have that binaries directory in your PATH). The simply type 'reg' on the command line. Register then starts up and reads the '~/.reginfo' file. It attempts to open a listening socket on the specified port. If it cannot open a socket on that port, it asks the user if it should use a new port number. The reason you might want to keep using the last port that Register used is because there may still be deans running. The deans keep trying to connect to Register, but they only try the old port number. If Register runs on a new port, any currently running deans will not be able to find it. The deans eventually give up after 10 minutes if they cannot reconnect to Register.

Register Commands

The following user commands are implemented in Register:

- show
- machines
- start
- stop
- tickle
- quit
- help (?)

show - Use this command to print all currently connected deans and objects.

machines - Use this command to print all machines known to Reg.

start dean <machinelist | classlist> - Use this to start deans on remote machines. The machines on which the deans will be started can be specified by a machinelist descriptor or a classlist descriptor. A machinelist descriptor is one or more machine names separated by the '#' character. Example, *start dean everest#haystack.* A classlist is similar to a machinelist except that machine classes are specified. Example, *start dean dec#b105***.** This would launch deans on all machines which are in class **dec** and in class **b105**. Note that a wild card '*' can also be used to name all machines or all classes.

stop dean <machinelist | classlist> - Use this to stop deans. This is usually done after a session is complete. The machinelist and classlist specifiers are the same as for **start dean**, and all deans can be stopped with 'stop dean *'.

tickle <on | off> - deans can be tickled every minute to keep them alive, if the user sets this to 'on'. If deans are not tickled and they do not receive input for 10 minutes, they will exit. This keeps them running while the user is doing other things like editing code or compiling. Tickle is off by default.

quit - Closes all connections and exits. Note that due to limitations of UNIX sockets, some sockets may remain open for a while after Register quits. This may inhibit Register from running again soon after quitting. The connected deans are not shut down, rather they are left running until Register starts up again, or until the 10 minute time-out is reached.

help - Gives these command descriptions on-line.

Reg machine file

When Reg launches, it looks for a special file in the user's home directory called '.reg.machines'. This file contains information about all machines that are going to interact in the SPEDE system. The format for this file is shown below. Lines with '#' as the first character are ignored.

```
#This is ~/.reg.machines
%classes                        # We are defining some classes
rs6000                          # Class of IBM machines by model type
ibm_rt
dec                             # Class of machine by manufacturer
b105                            # Class of machine by the room they live in
#
%machines                       # Valid machines
#
# machine information is listed as:
# <machine_name>, classes, base directory for deans and objects
everest, dec|b105, prog
waumbek, dec|b105, prog
#
turing, rs6000, rs6000/prog
babbage, rs6000, rs6000/prog
northstar65, ibm_rt, rt/prog
```

Problems with Reg

Reg is able to launch remote deans through the UNIX 'rsh' program. Sometimes this program hangs without launching the remote dean. The side effect of this is that sockets will remain open and the Reg will not be able to run again. To fix this, the user must do a 'ps -ux' to find out if any 'rsh' commands are outstanding. If they are, the user should kill the processes using 'kill -9 <proc id>'. This is a bug in rsh.

Appendix: Sample Objects

The following program is a computational object for producing a Mandelbrot image.

```
/* MAND_COMP
 * Jim Gochee
 * 5/10/92
 *
 * This object is a mandelbrot computation object. It is accessed from a 'mand'
 * object. Mand_comp will allow 'mand' to assign it part of an image to
 * compute.
 */
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>

/* Include files necessary for objects */
#include "sockets.h"
#include "spede.h"
#include "spede_io.h"
#include "objects.h"
#include "objlib.h"

#define TRUE 1
#define FALSE  0
#define SET_JOB     10
#define START_TASK  11

/**
 ** Mandelbrot types
 **/
struct mandel_task {
  int width, height;
  double left, right, top, bottom;
  double cr, ci;
};

struct mandel_job {
  int left, right, top, bottom;
  int width, height;
  unsigned char *pixels;
  int x,y;
};


/**
 ** Globals
 **/
char *object_name = "mand_comp";          /* What is the name of the object */
struct mandel_job *job;
struct mandel_task our_task;
int obj_num;                              /* Our object number */



/**
 ** Mandelbrot computation functions
 **/

int mandel_do_pix(task, job)
     struct mandel_task *task;
     struct mandel_job *job;
{
  double rx, ry;

  rx  = task->left + ((task->right - task->left) * (double) job->x) / (double) t
ask->width;
  ry = task->top + ((task->bottom - task->top) * (double) job->y) / (double) tas
k->height;

  job->pixels[(job->y - job->top)*job->width + job->x - job->left] = mandel(rx,
ry, task->cr, task->ci, 256, 1);
  if (++job->x == job->right) {
    job->x = job->left;
    job->y++;
    if (job->y > job->bottom)
      return 1;
  }
  return 0;
}

int
mandel_do_job(task, job)
     struct mandel_task *task;
     struct mandel_job *job;
{
  do
    if (mandel_do_pix(task, job))
      return 1;
  while (TRUE);
  return 0;
}

/* mandel.c: mandelbrot routine
 * From Fractint public domain fractal-manipulation program.
 */
int
mandel(initreal,initimag,parm1,parm2,maxit,inside)
double initreal,initimag,parm1,parm2;
int maxit,inside;
{
   double oldreal, oldimag, newreal, newimag, magnitude;
   int color;

   oldreal=parm1;
   oldimag=parm2;
   magnitude = 0.0;
   color = 0;
   while ((magnitude < 4.0) && (color < maxit)) {
      newreal = oldreal * oldreal - oldimag * oldimag + initreal;
      newimag = 2 * oldreal * oldimag + initimag;
      color++;
      oldreal = newreal;
      oldimag = newimag;
      magnitude = newreal * newreal + newimag * newimag;
   }
   if (color >= maxit) color = inside;
   return(color);
}



/* SET_JOB
 *
 * This routine is called when a 'set_job' message is received.
 */
void set_job(params)
     pblock *params;
{
  char *task;
  int *start, *stop;
  struct mandel_task *t = &our_task;
  int *t_obj_num;

  get_params(params, &t_obj_num, &task, &start, &stop);
  *start = ntohl(*start);
  *stop = ntohl(*stop);
  obj_num = *t_obj_num;

  /* Get the task description out of the string */
  sscanf(task, "%d %d %lg %lg %lg %lg %lg %lg", &t->width,
         &t->height, &t->left, &t->right, &t->top,
         &t->bottom, &t->cr, &t->ci);

  job->left = 0;
  job->right = job->width = t->width;
  job->top = *start;
  job->bottom = *stop;
  job->height = (*stop - *start + 1);
  job->x = 0;
  job->y = *start;
  job->pixels = (unsigned char *) malloc(job->width * job->height * sizeof(unsig
ned char));

  printf("(%s %d) Will compute from %d to %d\n", object_name, obj_num, *start, *
stop);
}


/* DO_WORK
 *
 * This routine is called when the a 'start_task' message has been received.
 */
void do_work(params, r_params)
     pblock *params, **r_params;
{
  pblock *results;
  long k;

  mandel_do_job(&our_task, job);               /* Compute the image section */

  /* Allocate return parameters */
  results = new_params(2, sizeof(long), job->width * job->height);
  k = htonl(job->top);
  set_params(results, &k, job->pixels);
  *r_params = results;
}


static long callback(message, params, r_params)
```

```
     int message;
     parameter_block *params, **r_params;
{
  long result;
  parameter_block param;
  char rw_des[256];

  param.num_params = 0;
  if (r_params != 0)
    *r_params = &param;

  switch(message) {
  case CREATE:
    job = (struct mandel_job *) malloc(sizeof(struct mandel_job));
    break;
  case DELETE:
    break;
  case SET_JOB:
    set_job(params);
    sprintf(rw_des, "%s%d", object_name, obj_num);
    break;
  case START_TASK:
    printf("Start computation.\n");
    do_work(params, r_params);
    printf("Done with computation.\n");
    break;
  default:
    printf("Invalid message.\n");
    return(0);
  }
}


main(argc, argv)
     int argc;
     char **argv;
{
  if(init_objects(argc, argv, callback)) {
    perror("Error in init_objects");
    exit(1);
  }

  register_dean();                      /* Tell local dean we exist */
}


void free_launch(argc, argv)
     int argc;
     char **argv;
{
  /* Free launch comes here */
  printf("ERROR!!! This object cannot be free-launched\n");
  exit(0);
}

void normal_launch()
{
  /* Nothing to do */
}
```

This sample is the mandelbrot organizer. It creates worker objects and reconstructs the return image fragments.

```
/* MAND
 * Jim Gochee
 * 5/10/92
 *
 * This object, when used with the PDPE (Personal Distributed Programming
 * Environment) will draw a mandelbrot image utilizing multiple cpus.
 */
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>

/* Include files necessary for objects */
#include "sockets.h"
#include "spede.h"
#include "spede_io.h"
#include "objects.h"
#include "objlib.h"

/* Non-necessary include files */
#include "misc.h"

#define TRUE 1
#define FALSE  0
#define MAX_OBJECTS 100
#define ZOOM_FACTOR 2.5             /* The closer to 2, the bigger the zoom */

#define LINE_INC 10
```

```
#define SET_JOB     10              /* Object messages */
#define START_TASK  11

/**
 ** Some typedefs describing the image and what a 'job' is.
 **/
struct mandel_task {
  int width, height;
  double left, right, top, bottom;
  double cr, ci;
};

struct mandel_job {
  int top, bottom;
};


char *object_name = "mand";                /* This name is the object name */
object_instance *task_obj[MAX_OBJECTS];    /* An array of object pointers  */
unsigned char *final_pix;                  /* Pointer to the image in memory */
struct mandel_task g_task;                 /* The mandelbrot descriptor    */
int end_line, current_line, num_working;   /* Used to monitor image lines  */
pblock *params;                            /* Params for work objects      */
int num_machines;                          /* The num of available nodes   */
int num_objects;                           /* The num of work objects in use */
char class_name[256];                      /* Which classes to use         */



/* CALLBACK
 *
 * This routine is the callback for when an object receives a OBJ_MESSAGE
 * command. This object will never receive one of those, so this routine is
 * not used.
 */
static long callback(message, params, r_params)
     int message;
     parameter_block *params, **r_params;
{
  long result;
  parameter_block param;

  param.num_params = 0;
  if (r_params != 0)
    *r_params = &param;

  switch(message) {
  case CREATE:
    break;
  case DELETE:
    break;
  default:
    printf("Invalid message.\n");
    return(0);
  }
}


/* MAIN
 *
 * This is the same for every object.
 */
main(argc, argv)
     int argc;
     char **argv;
{
  if(init_objects(argc, argv, callback)) {
    perror("Error in init_objects");
    exit(1);
  }

  register_dean();                      /* Tell local dean we exist */
}



/* INIT_TASK
 *
 * This routine sets up the mandelbrot image to be computed.
 */
void init_task(task)
     struct mandel_task *task;
{
  long size;

  task->width = 1024;
  task->height = 768;
/*  task->left = -2.5;
  task->right = 1.5;
  task->top = -1.5;
  task->bottom = 1.5;
```

```
*/
  task->left = -0.6633725;
  task->right = -0.6533721;
  task->top = -0.45337413;
  task->bottom = -0.4458738;

  task->cr = 0;
  task->ci = 0;

  size = task->width * task->height * sizeof(unsigned char);
  final_pix = (unsigned char *) malloc(size);
}


/* CREATE_OBJECTS
 *
 * This routine is responsible for creating the work objects to compute the
 * actual image. The machines that will be used are gotten from the command
 * line in the form of a class name. The 'reg' daemon is queried to find out
 * which machines are available in the given class. Note that machine names
 * could also have been entered on the command line with or in place of a
 * class name. Using a class name only is a choice made by the object and not
 * the restriction of the underlying mechanism.
 */
void create_objects()
{
  char *machine;
  int k, m, err;
  object_instance *obj;
  link_des link;
  char **the_machines;
  extern object_descriptor run_info;

  printf("Creating the work objects\n");

  /* Query the 'reg' daemon for machines of the given class */
  num_machines = run_info.num_machines;

  printf("Create objects on %d machines\n", num_machines, class_name);
  num_objects = (run_info.num_objects > MAX_OBJECTS) ? MAX_OBJECTS :
    run_info.num_objects;

  printf("Attempting to creating %d objects\n", num_objects);
  m = 0;
  for(k=0; k < num_objects;) {
    if (num_machines < 1) {
      printf("There are no machines specified!!\n");
      exit(0);
    }
    machine = run_info.machine[m % num_machines];
    printf("Asynchronously creating mand_comp on machine '%s'\n", machine);
    task_obj[k] = create_object(machine, "mand_comp", FALSE, FALSE, TRUE);
    if (task_obj[k] != 0) {
      k++;
    }
    else {
      run_info.machine[m] = run_info.machine[num_machines-1];
      num_machines--;
    }
    m++;
  }
  num_objects = k;
  if (num_objects < 1) {
    printf("There are no objects\n");
    exit(0);
  }

  err = sync_created_objects();                /* Wait until all are created */
  if (err) {
    printf("There was an error creating an object!!!\n");
  }

  printf("%d objects were actually created\n", num_objects);

  for(k=0; k < num_objects; k++) {
    /* These next lines form a speed link, which bypasses the dean */
    link = task_obj[k]->location;             /* Link to the object location */
    task_obj[k] = speed_link(&link);          /* Form the link               */
  }
}


/* OBJECT_DONE
 *
 * This routine is called by the object library when a return message has
 * arrived.
 */
void object_done(object, rinfo, obj_num)
     object_instance *object;
     return_info *rinfo;
     long obj_num;
{
  int *p_start, start, stop, pix_start;
  struct mandel_task *task = &g_task;
  void *data;
  long len;

  get_params(rinfo->r_params, &p_start, &data);
  get_param_len(rinfo->r_params, 0, &len);

  pix_start = start = ntohl(*p_start);
  memcpy(final_pix + start*task->width, data, len);

/*  printf("Object %d is done with %d to %d\n", obj_num, start,
         start+len/task->width); */

  /* See if there is more work to do */
  if (current_line < end_line) {
    stop = current_line + LINE_INC;
    if (stop > end_line)
      stop = end_line;
    start = current_line;
    current_line = stop;
    object_work(obj_num, object, params, start, stop);
  }
  else {
    num_working--;                  /* One less object is working */
  }
  draw_pixmap(final_pix + pix_start*task->width, 0, pix_start, g_task.width, len
/task->width);

}



/* OBJECT_WORK
 *
 * This routine is called when some work is being sent to an object.
 */
object_work(k, object, params, start, stop)
     int k;
     object_instance *object;
     pblock *params;
     int start, stop;
{
  int *start_p, *stop_p, *obj_num;

  /* Get pointers to the param data */
  get_params(params, &obj_num, 0, &start_p, &stop_p);

  *obj_num = k;                           /* Object number */
  *start_p = htonl(start);                /* Starting line */
  *stop_p = htonl(stop);                  /* Ending line */

/*  printf("Assigning object %d image line %d to %d\n", k, start, stop);  */

  /* Send this message with no return expected */
  send_obj_message(object, SET_JOB, params, TRUE, FALSE);

  /* Send the command to start work, with a callback when the return */
  /* message arrives                                                 */
  send_obj_mess_callback(object, START_TASK, params, object_done, (void *) k);
}


/* MANDEL_DO_TASK
 *
 * This routine is called to start the computation of the image.
 */
void mandel_do_task(task)
     struct mandel_task *task;
{
  char task_string[256];
  struct mandel_job job;
  int k, start, stop, *start_p, *stop_p, *obj_num, num_completed;
  double start_time;
  double idle, cpu, writing;

  start_time = get_seconds();            /* Start timing the draw */

  /* Build the task descriptor, which happens to be an ascii string */
  sprintf(task_string, "%d %d %.18g %.18g %.18g %.18g %.18g %.18g",
          task->width, task->height, task->left, task->right, task->top,
          task->bottom, task->cr, task->ci);

  end_line = task->height-1;

  params = new_params(4, sizeof(int), strlen(task_string)+1, sizeof(int), sizeof
(int));
  set_params(params, 0, task_string, 0, 0);

  current_line = 0;                      /* Which image line are we on */
  num_working=0;                         /* The number of objects working */
  for(k=0; k < num_objects; k++) {
    if (current_line+LINE_INC > (end_line+1)) {
      printf("WARNING...not all objects have a section to work on\n");
      break;
    }

    current_line += LINE_INC;
    reset_times(task_obj[k]);            /* Reset internal object timer */
    object_work(k, task_obj[k], params, current_line - LINE_INC, current_line);
    num_working++;
  }

  while(num_working) {                   /* Work until all objects done */
    idle_objects(20, 0);
  }

  printf("Time taken to draw pict was %.5g seconds\n", get_seconds() - start_tim
e);


  for(k=0; k < num_objects; k++) {
    get_object_times(task_obj[k], &idle, &cpu, &writing);  /* Get internal timin
g info */
    printf("Object %d: %.4g secs idle, %.4g secs writing, and %.4g secs computin
g\n", k, idle, writing, cpu);
  }


  /* Don't actually delete objects, since another image might be generated */

/*

  printf("All objects have completed.\n");
  printf("Deleting all objects\n");

  for(k=0; k < num_objects; k++)
    delete_object(task_obj[k]);
*/
}


/* ZOOM_TASK
 *
 * This routine zooms in on a portion of the image.
 */
void zoom_task(x, y, task, direction)
     int x, y;
     struct mandel_task *task;
     int direction;
{
  float width, height;
  float x_rat, y_rat, difx, dify;

  width = task->right - task->left;
  height = task->bottom - task->top;

  x_rat = width / (float) task->width;
  y_rat = height / (float) task->height;

  difx = (x - (task->width/2)) * x_rat;
  dify = (y - (task->height/2)) * y_rat;

  task->left += difx + (width/ZOOM_FACTOR)*direction;
  task->right += difx - (width/ZOOM_FACTOR)*direction;
  task->top += dify + (height/ZOOM_FACTOR)*direction;
  task->bottom += dify - (height/ZOOM_FACTOR)*direction;
}


/* FREE_LAUNCH
 *
 * This routine is called if the image was launched from the command line.
 */
void free_launch(argc, argv)
     int argc;
     char **argv;
{

  FILE *fp;
  int red[256], green[256], blue[256];
  int i, x, y, button;
  char line[256];
  extern object_descriptor run_info;

  fp = fopen("mapfile", "r");
  if (fp == 0) {
    perror("Could not find mapfile...\n");
    exit(1);
  }

  for(i=0; i < 256; i++) {
    if (!fgets(line, 255, fp)) {
      fclose(fp);
      printf("err");
      exit(1);
    }
    sscanf(line, "%d %d %d", red+i, green+i, blue+i);
  }
  fclose(fp);

  init_task(&g_task);
  setup_window(argc, argv, red, green, blue, g_task.width, g_task.height);

  create_objects();
  mandel_do_task(&g_task);

  while(1) {
    go_picture(final_pix, g_task.width, g_task.height, &x, &y, &button);
    zoom_task(x,y, &g_task, button);
    mandel_do_task(&g_task);
  }
}


/* NORMAL_LAUNCH
 *
 * This routine is entered when an object is created from another object.
 * Mand does not allow this, so exit if it happens.
 */
void normal_launch()
{
  printf("ERROR!!! This object must be launched from the command line\n");
  exit(0);
}
```