The SunNET 3.0 Manual

by Jason R. Mills (aka SunWiz, or SunRay)

May 29, 1998

SunNET Theory

SunNET is a hubless network for the LambdaMOO server. It is known to Work with the latest LambdaCore and SenseMedia Core. SunNET is designed to be simple to use for casual programmer, yet still able to provide a great deal of flexibility for more advanced programmer.

SunNET makes connections to other sites using special players. This seems to be the most stable way of transmitting data to and from MOOs. MultiPort listening has the problem of dropping the connection if it becomes inactive for a period of time (which can be changed when the server is compiled). Each network player's name is of the form, **SunNET_*SiteName***, where ***SiteName*** is the name of the remote site which the link represents. These link objects comprise the physical connections on the SunNET.

Logical connections ride on top of the physical connections. Currently these are represented in special properties on the *`$sunnet`* object.

SunNET's Language

SunNET has only a three commands of note.

DATA - transmits normal data through SunNET
SECURE - though currently not implemented, this command will send secured data.
CLOSE - Close down the connection this command was sent on. This can be used when a site is about to go offline for some reason.

The DATA command

The **DATA** is of the following form:

DATA {ReturnACK, TimeStamp, PacketID, Path, Destination, Protocol, Message}

Each **DATA** command has the following seven elements:

ReturnACK	This is somewhat obsolete in 2.1. SunNET 2.0 used it to determine if an **ACK** packet should be returned to the calling site. In SunNET 2.1, **ACK** is always sent except when **ACK** itself is received, or if the Destination is **GENERAL**. I keep this because I may use it in the future for another purpose, at which point **ReturnACK** will disappear entirely and replaced with something new.
TimeStamp	This is the time when the packet was sent from the source site.
PacketID	This is the number of the packet. It is a unique number generated on the source site. Usually, two consecutive packets are consecutively numbered.
Path	This is the path this packet followed from the source to the destination. It is a list of strings containing the names of the sites which helped deliver the packet. **Path[1]** is the source site, while **Path[$]** is the last site to handle this packet. Each site simply adds their name to the end of this list.
Destination	This is the name of the site that should eventually handle this packet. A special Destination of **GENERAL** simply says that the packet should be sent to all SunNET capable sites.
Protocol	This is the name of the protocol for this message. If there is a protocol handler for this protocol, it is called with a set of arguments including the **Message** field.
Message	The **Message** field is any valid MOO value that will be passed to the protocol handler on the **Destination** Site.

The SECURE command

The **Secure** command has not yet been implemented entirely, but is expected to have the same format as the data command.

The CLOSE command

The **close** command takes no arguments. It causes the link to shutdown and turn itself off (ie, not attempt to regenerate).

SunNET Composition

Currently there are nine objects which make up SunNET. They are:

1. *`$sunnet`* - Contains the base driver for dealing with connections as well as sending and interpreting information.

2. *$sunnet_utils* - Contains the programmer interface for SunNET as well as the packet creation routines.
3. *$sunnet_link* - This is the parent to the actual io ports on SunNET. All communication to or from a MOO must go through a child of this object.
4. *$sunnet_protocols* - Contains the base level protocols for SunNET including **RCALL**, **RCALLRESULT**, **ACK**, **ALIAS**, **IKNOW**, and **QUERY**. This also contains the unimplemented **SHUTDOWN** protocol.
5. *$sunnet_db* - Holds a list of key/data pairs for use on SunNET. Most notably, it is used with *$sunnet_fo*.
6. *$sunnet_scheduler* - This is the timing mechanism for SunNET. In addition, it is a general use scheduler for programmers to queue tasks on.
7. *$sunnet_pc* - This is the normal users interface to SunNET. By default it contains a modified page and an *@rwho* verb. This player class still needs some fleshing out.
8. *$sunnet_fo* - *$sunnet_fo* contains the remote (and local) login watcher. It has very little use outside of that and is an optional part of the SunNET.

Database Assumptions

The following verbs or properties and the corresponding objects

`$network.moo_name`	Assumed to contain the one-word name of the moo.
`#0.maxint`	Assumed to contain the maximum integer value.
`#0.generic_db`	Points to an object which at least simulates the Generic Database from the LambdaCore.
`#0.player`	Base object for all players.
`#0.generic_utils`	A placeholder object for utility objects (used in update scripter).
`#0.feature`	Parent for feature packages (used in update scripter).
`.description`	This property is used to describe an object’s meaning, use, or as tinyscenery.
`.aliases`	This property is used to help match objects in the database.
`Eval`	This verb is required for the installation script spooler to execute properly.
`$login:register`	Called when a link attempts to register on a remote site.
`$player:confunc` `$player:disfunc`	Called when a player connects and disconnects. Needed to help the links establish themselves.

There may be other assumptions that have been forgotten. Please let me know if you find any.

Installing SunNET

Installing sunnet consists of:

1. Uploading a SunNET Script. This can be done for you with a call to *`$sunnet:spool_script`* on a site which already has SunNET installed. You provide the username, password and site information for the MOO you wish to install the program on. It is important that the MOO you are installing SunNET onto has byte quota enabled, or that you set the object quota of the player represented by the username to some reasonable number of objects. A sample call to *`:spool_script`* would be *`;$sunnet:spool_script("mysite.com", 8888, "WhoIAm", "MyPassword");`*
2. Running the *`init_for_core`* verbs on *`$sunnet_protocols`* and *`$sunnet_fo`* so that the built-in protocols can be registered.
3. In order to support the **RLOGIN** protocol, you must add the line *`$sunnet_fo:(verb)(user)`* to the *`#0:user_created`*, *`#0:user_connected`*, *`#0:user_disconnected`*, *`#0:user_client_disconnected`*, and *`#0:user_reconnected`*, possibly wrapped in a *`fork(0)…endfork`* statement.
4. Once started, SunNET should continue operating between server restarts. However, you may wish to add the line *`$sunnet:bootstrap()`* to *`#0:server_started`* to cause a quicker restart.
5. Create SunNET Links (see below).
6. Calling *`$sunnet:bootstrap()`*

SunNET configuration

The following properties can be customized to change SunNET's behavior:

`$sunnet.regenerate_time`	How long a link should wait before trying to reconnect.
`$sunnet.debug_level`	A mostly useless option. Debug level 0 shows nothing. Debug level 1 shows outbound packet destinations, or failure of a packet. Debug level 2 shows packet content, and possible path nodes. Debug level 3 shows path nodes as excluded from route consideration. (This was used to help debug the routing algorithms and is probably no longer needed.)
`$sunnet.packet_time`	Used in routing to determine how long a packet should wait before a node push is considered as failure (and the packet is pushed out on the next node, or the failure callback is executed)
`$sunnet.IDKeep`	Number of packet IDs to hold before discarding old IDs.
`$sunnet.ansi_compatible`	If `$ansi_utils` (written by Dark_Owl@Lambda) is installed on your MOO, set this flag to 1 to remove ansi tags, otherwise set to 0.

`$sunnet.notify_interval`	Number of seconds between broadcasts of a site's IKNOW, ALIAS, and PROTOS information.
`$sunnet_utils.max_tries`	Timeouts with the RCALL mechanisms occur in `(5*$sunnet_utils.max_tries)` seconds.

Creating a link between two sites on the SunNET:

1. Choose a password for the link (say PassWord). At this time, passwords are not case sensitive, but this may change in future releases of SunNET.
2. Decide if SiteA or SiteB will initiate the link (for this walk-through, SiteA will open to SiteB).
3. On SiteA, eval: ***`$sunnet:create_connection("OUT", "SiteB", "Address.to.SiteB.org", PortNum8888, "PassWord");`***
4. On SiteB, eval: ***`$sunnet:create_connection("IN", "SiteA", "PassWord");`***
5. Open the link by evaling on SiteA: ***`$sunnet:open_connection("SiteB");`***

Removing a link between two sites from SunNET:

1. Find the password for the link. Look in *`$sunnet.incoming_connection_info`* or *`$sunnet.outbound_connection_info`* if you need a refresher.
2. On SiteA, eval: ***`$sunnet:remove_connection("SiteB", "PassWord");`***
3. On SiteB, eval: ***`$sunnet:remove_connection("SiteA", "PassWord");`***

Changing a link's password.

Just follow the steps for creating a link.

The Built-in Protocols

The following is a list of builtin protocols (in decreasing importance) for the SunNET program.

`IKNOW`	A list of sites the sender knows directly.	This protocol is responsible for maintaining SunNET routing information. SunNET can still run without it, but has to assume everything.
`ACK`	A list of the form: {Packet ID of original packet, the path original packet took, the time of the original packet, our current time}	Responsible for clearing the outbound message queue. Eventually this will be used to obtain timing information about links. Without this protocol, Packet buildup can occur.
`RCALL`	A list of the form {task ID of task on sending moo or 0 if no result needs to be returned, object as a string to be matched or as an object number, verb to call, list of arguments} Note that if the object results to #-1, then a builtin function with the name in the verb to call slot is called. This is how remote eval is accomplished.	Used to remotely evaluate bits of MOO code. It is tied heavily with **`RCALLRESULT`** for returning the information to the calling MOO.
`RCALLRESULT`	A list of the form {message sent to **`RCALL`**, result of the evaluation}	Used with **`RCALL`** to return information to the calling task on the calling MOO.
`ALIAS`	A list of aliases for the sending site.	This is not necessary for SunNET, but does provide the capability to abbreviate longer site names.
`PROTOS`	A list of protocols the sending site knows about.	This is not used internally to SunNET. It is provided to SunNET application writers.
`RLOGIN`	A list of the form: {PlayerName, PlayerNum, Action (connected, disconnected, etc), total number of players, Location object number, Location Name, Site information}	Used for the remote login watcher.
`RWHO`	A list of connected players on the sending site	Used to ease the traffic generated by @rwho and @rwho cache updates.

Other protocols not listed here are not part of the SunNET program itself.

General SunNET programming

The more versatile of the two programming models is the remote evaluation/verb calling. Many applications which involve only a few sites are better programmed using this model. One example of this is the *$vravatar* program developed by the WizTraveller on the SenseMedia MOOs. Here, I will attempt to show how two rooms can be linked together.

Planning

What should be the goal of the project? Obviously textual activity between the two sites should be transmitted. This involves hooking into *:announce*, *:announce_all*, and *:announce_all_but*. We also want to keep the appearance of a rooms contents consistent between the two sites. This involves hooking into *:enterfunc* and *:exitfunc* on the room. A property will also need to be created to hold remote object information, and *:tell_contents* will need to be overridden to display objects on the other site. Each of the two objects will be a mirror image of the other site. Note that unless otherwise specified, these verbs are all +x with the args this, none, this.

Creating

To start off, we need to create a *$room* on each of the two sites with whatever name you wish. Mark down the number, #A for Site1 and #B for Site 2. Create a property named *.remote_contents* on each site set to {}. The property will hold the names of objects on the opposite site. Create the properties *.remote_site* and *.remote_room*. For Site 1, the properties will hold "Site 2" and #B respectively and for Site 2, they will hold "Site 1" and #A. This is to prevent hard-coding site information in the objects.

Programming to Receive Actions

First write a verb for receiving information from remote *:announce** calls. We will call this verb *:remote_announce*. We must be careful here. We can not call *:announce** without having *:announce** do some special checking to see if *:remote_announce* is the calling verb (which will be the route taken for this example). Note that there are some reasons for not having a *:remote_announce_all* and *:remote_announce_all_but*, so feel free to ponder this question or ask me about it.

Security checks on *:remote_announce* will need to ensure that the *caller_perms()* control the object or that the caller is SunNET. The following code snippet performs this check and raises an error if the conditions are not met. *$sunnet_protocols* is used here because the remote functions are driven on **RCALL**. *$sunnet_protocols* is responsible for dispatching **RCALL** events.

```
if (caller != $sunnet_protocols &&
     !$perm_utils:controls(caller_perms(), this))
  raise(E_PERM);
endif
```

All that's needed from here is to simply call *`:tell`* on all the real contents with the argument(s) provided. The following lines do this for you:

```
for x in (this:contents())
  x:tell(@args);
endfor
```

Combine the two to program *`:remote_announce`*

Programming to Send Actions

Of the three forms of *`:announce`*, *`:announce_all_but`* has a special argument for an exemption list. Other than that, all three forms will be programmed identically. The announce verbs usually have no security checks, so they are omitted here as well.

`:announce` and *`:announce_all`* can be combined into one verb named *`"announce announce_all"`* and programmed as follows:

```
pass(@args);
$sunnet_utils:verb_call(this.remote_site,
     this.remote_room, "remote_announce", @args);
```

`:announce_all_but` can be programmed as follows:

```
pass(@args);
$sunnet_utils:send(this.remote_site, this.remote_room,
     "remote_announce", @listdelete(args, 1));
```

Note that had we wanted the result from the call, we would replace *`:send`* with *`:verb_call`* above. Using *`:send`* can save SunNET bandwidth when you do not really need the result of the verb call.

Programming to Receive Contents

In MOO, the *`move()`* function calls exitfunc on an objects old location and enterfunc on the object's new location. As such these two verbs will be used below to send object names to the opposite moo to the verbs in this section. *`.remote_contents`* stores this information and is just a list of object names in the room on the remote site. Rather than send the entire list whenever the contents change, we will just send additions and subtractions to the room's contents to the remote site. Therefore we will need two verbs, one to add an object name and one to remove an object name from the *`.remote_contents`* property. The verbs will be called *`:remote_add`* and *`:remote_remove`*. Both use the same security check as described in the action receiving verb. They both take a single argument, which is the name of the object to add or remove respectively.

To add an object, the following line does the trick in *:remote_add*:

```
this.remote_contents={@this.remote_contents, args[1]};
```

To remove an object, use the following line in *:remote_remove*:

```
this.remote_contents =
     `listdelete(this.remote_contents, args[1] in
     this.remote_contents) ! ANY =>
     this.remote_contents’;
```

Combine the security check above with these lines to create the respective verbs.

Programming to Send Contents

This involves overriding the enterfunc and exitfunc verbs to send the information. These verbs receive the object number of the object passing in or out of the location.

:enterfunc can be programmed as follows:

```
$sunnet_utils:send(this.remote_site, this.remote_room,
     “remote_add”, args[1].name);
pass(@args);
```

And *:exitfunc* can be programmed as:

```
$sunnet_utils:send(this.remote_site, this.remote_room,
     “remote_add”, args[1].name);
pass(@args);
```

No security checks are used in *$room:enterfunc* and *:exitfunc*, so should be unnecessary here.

Programming to Display Contents

A room is displayed with the *:look_self* verb with tells information to the player. *:look_self*, in turn, calls upon utility verbs to display various aspects of the room, such as the contents and exits. For this section, we will override the *:tell_contents* verb to incorporate the remote object names. By default, *#3:tell_contents* recognizes four different object list layouts. For the purpose of this tutorial we will simply print out a list of the objects. This is simple enough to accomplish using the following code, again without the need for security checks.

```
lst=this.remote_contents;
for x in (this:contents())
  lst={@lst, x == player ? “you” | x.name};
endfor
player:tell(“  “, $string_utils:english_list(lst));
```

That’s all you need.

Done

This concludes the example. This code is untested, so if anyone wishes to implement it, please let me know if it works or not. For further exploration, you may wish to look into how to prevent an objects departure or arrival from being spoofed on the opposite site, how to fake stage talk into thinking the players are `real’ and how to privately whisper remotely in the same room.

Writing a SunNET protocol

The protocols of SunNET is what everything else rides on top of. Protocols are used when you wish to broadcast information to more than one site at the same time. If not for the protocol layer, SunNET would have a difficult time operating properly. This sample application is similar to the one above.

Planning

Instead of linking two rooms, this application will link rooms on several different sites. We will keep the *`.remote_contents`* property and the *`:tell_contents`* verb written above. To save bandwidth we will use only one protocol for addition, removal, and broadcasting of actions. We will need to decide on an appropriate protocol name. It may be wise to check *`$sunnet_utils:protocols`* with the argument of every MOO on the network to ensure that the protocol is free to use. For the purposes of this example, we will use **ROOM** as the protocol name. The message for this protocol will be a list with two elements. The first element is a switch that tells the handler what to do with the second part, whether it be an addition, removal, or action broadcast.

Creating

A room must be created on each MOO for which the protocol will be received. Create a property with the name *`.remote_contents`*, setting it to *`{}`*, and copy the *`:description`* verb from the General Programming section above. Since we are going to run across several MOOs, the *`.remote_site`* and *`.remote_room`* properties are unnecessary. We will need to override the same verbs as above, but they will be written slightly differently. Security checks, if needed, are identical to the one outlined in the section above, replacing *`$sunnet_protocols`* with just *`$sunnet`*.

Sending Actions

To send actions, we need to override the three *`:announce`* verbs. *`:announce`* and *`:announce_all`* can be combined into one verb, while it is easier to put *`:announce_all_but`* into a verb to itself. Once we have the message built, we use *`$sunnet_utils:broadcast`* to send the message. At a minimum, *`:broadcast`* takes three arguments: the destination, which is **GENERAL** in this case so that the packets will go to all connected sites; the protocol, which is **ROOM**; and the third element is the message. We will symbolically represent the switch parameter of the message as a string value. For sending actions, this will be *“ACTION”*.

Keeping that in mind (as well as the fact that the *:announce* suite has no security checks), program *“announce announce_all”* as:

```
     $sunnet_utils:broadcast(“GENERAL”, “ROOM”, {“ACTION”,
          @args});
```

and *:announce_all_but* as:

```
     $sunnet_utils:broadcast(“GENERAL”, “ROOM”, {“ACTION”,
          @listdelete(args, 1)});
```

Sending Contents

To send contents, we must override the *:enterfunc* and *:exitfunc* verbs. The second argument in both cases will be the name of the object to add or remove. For adding an element to the contents, the switch will be *“ADD”* and for removal, it will be *“REMOVE”*. Again, no security checks are needed. Program *:enterfunc* as:

```
     pass(@args);
     $sunnet_utils:broadcast(“GENERAL”, “ROOM”, {“ADD”,
          args[1].name});
```

and program *:exitfunc* as:

```
     pass(@args);
     $sunnet_utils:broadcast(“GENERAL”, “ROOM”, {“REMOVE”,
          args[1].name});
```

Programming the Handler

The protocol handler will be the most complicated verb discussed so far in these tutorials. Even so, the verb is still fairly simple. You will need the security check from the previous section (changing *$sunnet_protocols* to just *$sunnet*) in the top of the verb and the following line to ensure the arguments are properly handled. We will call this verb, *:room_handler*.

```
     {from, protocol, message, path, packetid, packettime,
          @rest} = args;
```

Here’s a description of the arguments from left to right.

from is the MOO on which the packet originated.
protocol is the protocol this handler should use. Note that this opens up the possibility that one handler can handle more than one protocol.
message is the information that the handler should act upon.

path is the path the packet took in between the originating site and the current site.
packetid is the ID stamp for the packet.
packettime is the time stamp for the packet (based on the originating MOO's *`time()`* function).
@rest takes the remainder of the arguments. This use is encouraged to prevent handlers from breaking as new information is passed to the handler.

For simplicity, the line above can be replaced with:

```
{from, protocol, message, @rest} = args;
```

where the arguments are as described above.

There are three parts that must be addressed. The broadcast of actions can be handled with the following lines:

```
for x in (this:contents())
  x:tell(@listdelete(message, 1));
endfor
```

Notice that we used *`listdelete()`* to remove the switch part of the message. The actual use of the switch will be demonstrated later.

To add contents, we use the following line:

```
this.remote_contents = {@this.remote_contents,
     message[2]};
```

To remove contents, we use the following line:

```
this.remote_contents =
     `listdelete(this.remote_contents, message[2] in
     this.remote_contents) ! ANY =>
     this.remote_contents;
```

Now we use a series of *`if`* and *`elseif`*, to determine which part to execute based on the switch embedded in the message. The skeleton of this structure looks like:

```
if (message[1]==”ACTION”)
  … code to tell all contents the action …
elseif (message[1] == “ADD”)
  … code to add an item to the remote contents list …
elseif (message[2] == “REMOVE”)
  … code to remove an item from the remote contents
     list.
endif
```

Putting this all together, the verb code would be:

```
if (caller != $sunnet &&
     !$perm_utils:controls(caller_perms(), this))
  raise(E_PERM);
endif
{from, protocol, message, @rest} = args;
if (message[1]==”ACTION”)
  for x in (this:contents())
    x:tell(@listdelete(message, 1));
  endfor
elseif (message[1] == “ADD”)
  this.remote_contents = {@this.remote_contents,
     message[2]};
elseif (message[2] == “REMOVE”)
  this.remote_contents =
     `listdelete(this.remote_contents, message[2] in
     this.remote_contents) ! ANY =>
     this.remote_contents;
endif
```

Displaying Remote Contents

Just use the *`:tell_contents`* from the above section to display the remote contents.

Registering the Protocol

The next step is to let SunNET know that it can now handle the protocol. To do this, just eval the following line:

```
;$sunnet:add_protocol(“ROOM”, #OfTheRoom,
     “room_handler”);
```

From here on out, SunNET will call *#OfTheRoom:room_handler* whenever it receives a packet containing information for the *“ROOM”* protocol.

Done

Since this document is for programming on the SunNET, it may not be a good idea to leave the protocol created in this document registered. In order to remove the protocol, use the following eval:

```
;$sunnet:remove_protocol(“ROOM”);
```

Further enhancements for this tutorial are identical to the further enhancements for the general programming section. Please use protocols sparingly and research to ensure that a protocol or one similar is not already being used before implementing it.

Once again, like the previous section, this is untested and may have some unforeseen bugs.

Historical Notes

ChatLink

ChatLink was designed to connect Generic Group Interface Players (GGIP) from different sites together. The basic transport was a constant connection through the GGIP itself. It listened to calls on its ***:broadcast*** verb and sent the text across the link to a centralized HUB which would distribute the text to all links except the broadcasting link.

MOOLink

MOOLink was an extended version of the ChatLink which could perform more functions. The basic connection was still a GGIP character, but the network could be extended to other GGIPs and other functions. Interpage, which was a special player, was the interface to page someone on other sites. MOOLink could also perform remote @who-like requests on the GGIPs as well as the remote MOO, however, specific players could not be specified. MOOLink was also a HUB-oriented networking scheme.

SunNET 1

SunNET 1.0 was the first network program in SunNET's history that moved away from the HUB connections. It was designed to be able to execute remote snippets of MOO code and return the results to the caller. Like the current SunNET version, it had two ways of of using it: Remote evaluation and Message Broadcasting. It contained somewhat primitive routing code. The basic scheme was to pass on all messages to all sites with the exeptions, a) the packet had already been seen by the receiving site, or b) the non-**GENERAL** destination of the packet was the site itself. This scheme became problematic as more sites were added to the network as it caused exponential growth in the number of packets received by a site at any given time and was the reason SunNET 2.0 was written.

The link strategy was to have connection player objects, and have the moo listen on special ports (Multi-Port listening) and connect to special ports. MPL was rarely used as it tended to timeout after five minutes of no activity. Almost all functionality resided in the ***$sunnet***, as ***$sunnet_utils*** did not exist. The transport commands had to be parsed by the site in MOO code, causing more lag in times of high activity

SunNET 2

SunNET 2.0 changed the transport commands, making it completely incompatible with SunNET 1. The packets contained the **DATA** transport command followed by a ***argstr*** which could be passed to the ***eval()*** builtin to obtain the parameters.

Routing in SunNET 2 was determined by a set of path tables. The table for any given site was sorted in increasing order by the number of sites the packet needed to go through to get to the destination. Each one was tried until a success packet (**ACK**) was returned from the destination. **GENERAL** destination packets do not return **ACK**, instead they are broadcast to all directly connected links and assumed to succeed.

Version 2.1 was the first version to be dependent on MOO server 1.8.0 or better for the speedups involved with *`suspend()`* and *`resume()`*, as well as using the scattering assignment for faster interpretation of the message values..

SunNET 3

SunNET 3.0 remained compatible with SunNET 2 in its transport format and can communicate with SunNET 2.0. Some protocols that 2 relied on have been deprecated in 3, such as **QUERY**, which asked a site if it was on the network. **QUERY** had a tendency to cause many response packets to be generated unneccessarily by all sites receiving it.

SunNET 3 also adds functions to allow users to easily install and update the source code on various MOOs and is outlined in the installation section of this manual. It allows a site to determine if it has *`$ansi_utils`* installed.

SunNET 3.0a2

Routing has been changed to be based on time, rather than path length. This means that it now takes a few minutes before SunNET figures the best path it can find and uses it. **GENERAL** destination and **ACK** packets no longer enter the outbound message queue of SunNET. Each message is also handled in its own task making the time overhead for prcessing messages smaller. This has the added advantage on MOOs where task quota is enforced by preventing SunNET packets from growing out of control. A new protocol, **RWHO**, was added in this release, which will make @rwho friendlier to the system in the next release (this tries to make sure all nodes are running the **RWHO** protocol before it is used).

SunNET 3 is still evolving and is considered alpha software as the features are not all implemented. The following list of changes are expected for SunNET 3:

1. Routing tables will be based on time rather than path length. Ö
2. Logical connection objects (ports) will be created and destroyed as needed, but will be able to be made persistent (remaining across connections) to permit special filtering, such as removing ansi codes for a message destined for a non-ansi-compatible site, and possibly allowing some blocking functions. All input and output to SunNET will pass through these objects.
3. Use of the **RWHO** protocol. This is to be done one release later so that all nodes are using the updated SunNET version.

Troubleshooting

The following is a list of questions that are being asked about administrating SunNET. It is a growing list and your questions will help flesh it out over time.

Q1. Why does a newly created link connect then immediately disconnect? Why does a newly created link not connect at all?

A1. `$sunnet_utils.log` contains a list of failed **REGISTER** commands. This property is planned to be used for other features at some point (such as link creation/destruction). If a link does not stay connected look here to see if there is an error message. If there is a message, check the password for the link (you can follow the procedure for creating a link above to also change the password. If there are no error messages, ensure that the `$login:REGISTER` is callable (+x) on both sides of the link. If this is not the problem, check `$sunnet.ansi_compatible` and be sure it is set correctly (see customization above).

Q2. Why does a link connect and then disconnect a few minutes later?

A2. This could happen if an alternate copy of a MOO's database is brought up. Changing the password should cure this, or deleting the link on the alternate copy of the database. This generally happens with outbound links.
A link may also be timeing out on the remote side. See Q1 for a possible solution.

Q3. Why does the number of outbound messages increase to very high numbers?

A3. I don't know. Speculation is that one site has a slow outgoing net, creating a sort of packet trap. Another possibility is that `$sunnet.ansi_compatible` has been changed to 1 when `$ansi_utils` does not exist. If the number of messages does not decrease, set `$sunnet.outbound_messages` to `{}` and see if that stops the messages from building up.

Q4. Why does `$sunnet_scheduler` spawn tasks continually, and what to do about it?

A4. At least two things can be going wrong here. The first is that your site is so lagged that the scheduler tasks can not be executed in a timely manner. You could kill the tasks which are waiting to run, but this may cause some unexpected results, such as causing outgoing packets to not time out and keep them from being removed from the outbound messages queue. The second is that a task has been entered into the scheduler with `:schedule_every` with a small number of seconds between calls. If you look at the scheduler, you should see an asterisk (*) beside tasks which repeat forever. Killing these may stop the task from respawning indefinitely.

Q5. When would I want to turn off SunNET and how?

A5. There are times when lag on the system rises due to activities by SunNET, other moo programs, and possibly tasks on the underlying operating system of the MOO. In times like this, it may be beneficial to turn SunNET off until the system cools down some. To do this, call `$sunnet:shutdown()`. This takes care of closing all connections and refusing to accept new connections. Sometimes, however, this is not sufficient to shut things down, especially in times of very high lag. A symptom of failure will be an out of seconds/ticks traceback resulting from the call. To shutdown SunNET in this case, set *`$sunnet.outbound_messages`* to *`{}`* and try again. If that fails, set *`$sunnet.active`* to *`0`* and manually boot all players whose name is of the form **SunNET_XXXX**.

Q6. What if I call *`$sunnet:init_for_core`* by accident? Why will a node not respond to other nodes (but the links do connect)?

A6. This resets *`$sunnet`* to installation state. This means that all options will need to be modified as appropriate for you needs. Also, all protocols have been wiped out. Call *`$sunnet_protocols:init_for_core`* and *`$sunnet_fo:init_for_core`* to reset the base protocols. Non-standard protocols will also need to be readded using *`$sunnet:add_protocol`*. Once the protocols are setup, the connection information tables need to be rebuilt. This is done by using *`$sunnet:create_connection`* with the original link mode (**IN** or **OUT**) and parameters. The password can be retrieved by using *`$sunnet_link.net_password`*.

Q7. How can I ask questions or contribute to discussions about SunNET?

A7. Subscribe to the SunNET listserve by sending an email message with the word **SUBSCRIBE** in the body of the message to *`sunnet-request@rupert.memphis.edu`* and then sending subsequent messages to *`sunnet@rupert.memphis.edu`*.

Q8. How do I find out what the latest updates to SunNET are?

A8. Read *sunnet on Rupert for the latest changes to SunNET.