TRAFFIC CONTROLLER
User's Guide

Part 1
UCO/Lick Technical Report 55

R. J. Stover

April 6, 1990

1 Introduction

Traffic is a simple program developed for the Lick and Keck optical instrument data acquisition systems. *Traffic* is the central inter-process communications hub of the **MUSIC** system, as described in the **MUSIC** System Coordination Overview (Part I of UCO/Lick Technical Report 54). *Traffic* provides a centralized process to coordinate the routing and distribution of messages between the various **MUSIC** data acquisition processes. Typically each data acquisition process opens up a single channel of communication (a Unix socket) to *traffic* and this single channel is used for all communications with other processes.

In addition to simple message routing, *traffic* provides message broadcasting in which a single message is replicated and sent to a number of clients. This is an important function in the **MUSIC** system in which multiple user interfaces may all wish to receive the same update information.

Messages are sent between processes in the standard **MUSIC** message format. The descriptions given in this document assume the reader is familiar with this format and the typical means for sending and receiving such messages. See the **MUSIC** System Messages document (Part II of UCO/Lick

Technical Report 54) for a complete description. To review briefly, the simplest message can be sent with the following two function calls.

```
mstart(to,from,msgnumber);
msend(socket);
```

The first function, `mstart`, builds a simple message and the second function, `msend`, transmits the message. The `to`, and `from` parameters are message addresses, `msgnumber` is a message number, and `socket` is the socket to *traffic*.

The remainder of this document describes the use of *traffic*. Processes which use *traffic* will be referred to as clients. Section 2 describes communications with *traffic* itself. Section 3 briefly describes some C routines developed for handling the communications with *traffic*. Section 4 describes some details of sending messages through *traffic* to other processes.

2 Communicating with *Traffic*

2.1 Setting Up the *Traffic* Socket

When *traffic* starts up it uses the C routine `read_remote` to look in a configuration file (*dtakeservice* on the Lick system) for the name 'trafficport' from which it obtains the TCP/IP Internet port number. The value of 'trafficport' must be consistent among the dtakeservice files for all of the machines in the **MUSIC** system. *Traffic* then creates a Unix TCP/IP socket and listens for connections at the specified port number. *Traffic* clients can use `read_remote` to obtain the port number, can create a Unix socket, and can connect to the *traffic* port. The routine `startproc`, described in Section 3.1, performs these functions for *traffic* clients. The routine `read_remote` was discussed in the **MUSIC** System Coordination Overview, Part 1 of UCO/Lick Technical Report 54. It is also listed in Section 3.3 of this document.

Clients can send messages through *traffic* to other clients, or they can send messages directly to *traffic*. Most clients will do both. Routines which communicate with *traffic* will need to include the C header file `music/traffic.h` which symbolically defines all of the message numbers for the messages sent by clients to *traffic* or sent by *traffic* to clients. Table 1 lists all of the *traffic* messages involving direct communications with *traffic*.

Table 1: Symbolic Names for *Traffic* Messages

Client Sends	*Traffic Responds*
SOCK_CONNECT	R_SOCK_CONNECT
CONNECT_PROC	R_CONNECT_PROC
DESIRED_MSGS	R_DESIRED_MSGS
TURN_OFF_MSGS	R_TURN_OFF_MSGS
GOODBYE	none
anything else	INEXPLICABLE

The file `music/traffic.h` also defines the symbol `TRAFFIC_CONTROLLER` which is the message address to use when sending messages to the traffic controller. This goes in the `to` parameter of the `mstart` function call. The body of all *traffic* messages is briefly described in `music/traffic.h`, and we will repeat those descriptions here. Elements of the body are either 32-bit integers or character arrays. In these descriptions, an integer called `X` is declared as `int X` and a character array called `C` is declared as `char C[]`. The elements are placed in the body of the message in the order they are listed in the description. We begin with the description of `SOCK_CONNECT`.

```
msg number = SOCK_CONNECT (client sends this to traffic to identify
                           itself)
msg body =      char name[]     = Client's process name
```

Once a client has established the socket connection to the traffic controller, the client needs to identify itself to *traffic* using the `SOCK_CONNECT` message. This message provides to *traffic* a name which traffic saves in an internal table along with the client's message address which is assigned by *traffic*. Later, other clients can ask traffic for the message address corresponding to that name. The name should be a unique string. On the Lick system this name is usually constructed from the name of the program, the computer host name, and the Unix process ID number. For processes started by requests to *runner*, *runner* generates this unique name and supplies it both to the process being run and to the requesting process. Other processes, such as user interfaces which are not started by *runner*, need to build this unique name themselves. See the **Runner User's Guide**, Part 2 of this Technical Report, for details on *runner*.

Traffic responds to `SOCK_CONNECT` by returning the `R_SOCK_CONNECT` message.

```
msg number = R_SOCK_CONNECT (traffic sends this in response to SOCK_CONNECT)

msg body =      int msgaddr     = The clients message address if positive
                                  or -1 if some error occurs.
```

The returned message address should be used in all further messages. This is the number to put in the `from` parameter of the `mstart` function call. If a -1 is returned by *traffic* then some problem was encountered by *traffic* and an explicit error message will be logged by *traffic* into the error file. (In the Lick system the error file is `/u/ccd/trafficlog`. In the Keck system the error file name is TBD.)

2.2 Getting Message Numbers

Once a client has successfully completed the `SOCK_CONNECT` transaction it has its own message address. To send messages to another client it must obtain the message address of that client. It can do this by sending a `CONNECT_PROC` message to *traffic*.

```
msg number = CONNECT_PROC (client sends this to traffic to request
                           another process's message address)
msg body =      char name[]     = Name of the process for which the
                                  message address is being requested
```

The body of the message must contain the (null terminated) process name of the process for which the message address is requested. Obviously, to use `CONNECT_PROC` a client must know the process names of other clients. Exactly how the names become known to other processes is up to the programmer, but one way is described in the document **MUSIC** System Coordination Overview (Part I of UCO/Lick Technical Report 54) and uses the *runner* process. The *runner* process is described in the **Runner User's Guide**, Part 2 of this Technical Report. Another less general method is available for obtaining message addresses which does not require the knowledge of a process's name but instead depends on the *traffic* broadcast mechanism. See

Section 4.4 of the *Infoman* User's Guide (UCO/Lick Technical Report 56) for a detailed example.

When *traffic* receives the `CONNECT_PROC` message it searches its table of names supplied by previous `SOCK_CONNECT` messages and, if it finds a match, it returns the corresponding message address in an `R_CONNECT_PROC` message.

```
msg number = R_CONNECT_PROC (traffic sends this in response to CONNECT_PROC)

msg body =      char name[]     = Process name given in CONNECT_PROC
                int msgaddr     = The requested message address or -1
                                  if an error occurs.
```

If the returned message address is -1 then *traffic* failed to find a matching name, and a message will have been logged in the *traffic* error log file. The C routine `connectproc`, described in Section 3.2, has been developed to handle all of the message I/O needed to complete a series of `CONNECT_PROC` transactions for a list of process names.

2.3 Broadcast Messages

When *traffic* receives a message it examines the destination address in the header of the message and uses this address to look up the Unix socket onto which it is to forward the message. However, when the destination address is -1 (also symbolically defined as `BROADCAST` in `music/traffic.h`) *traffic* treats the message as a broadcast type message. It then uses the message number to look up a table of Unix socket numbers, and it writes a copy of the message to each of those sockets. Using this mechanism a client can have a single message broadcast to a group of clients, and the sending client does not even have to know which clients receive the message. Clients can put themselves on the lists to receive particular broadcast messages by sending the `DESIRED_MSGS` message to *traffic*.

```
msg number = DESIRED_MSGS (client sends this to traffic to request
                          certain broadcast messages)
msg body =      int msgnum1     = First message number
                int msgnum2     = Next message number
                ...
                int end         = Any negative value (-1 is nice) to
                                  mark the end of the list.
```

The body of the message is a list of message numbers for which the client would like to receive any messages sent in broadcast mode. For each message number in the list, *traffic* creates an entry in that message number's broadcast table to hold the client's socket number. To make the broadcast mechanism as fast as possible *traffic* defines another fixed size pointer-table which is indexed by message number and whose elements point to the associated tables of broadcast socket numbers. Since this pointer-table is of fixed size, message numbers used in broadcast mode can be no larger than the size of this table and must be between 0 an 899.

After processing the list of `DESIRED_MSGS` message numbers, *traffic* will return an R_DESIRED_MSGS message.

```
msg number = R_DESIRED_MSGS (traffic sends this in response to DESIRED_MSGS)

msg body =      int retcode     = 0 means OK; -1 means some error.
```

The `retcode` element will be 0 unless one of the requested message numbers is outside the legal range. Check the *traffic* error log file for a specific error message.

Having sent a `DESIRED_MSGS` message, a client may later wish to remove itself from the distribution lists of some or all broadcast message numbers. To do so it can send the `TURN_OFF_MSGS` message.

```
msg number = TURN_OFF_MSGS (client sends this to traffic to cancel
                           certain broadcast messages)
msg body =      int msgnum1     = First message number
                int msgnum2     = Next message number
                ...
                int end         = Any negative value (-1 is nice) to
                                  mark the end of the list.
```

Like the `DESIRED_MSGS` message, the body contains a list of message numbers for which broadcast type messages are no longer desired. The list of message numbers does not have to match exactly the list of a previous `DESIRED_MSGS` message, but it should include only message numbers for which broadcast messages have been requested. In response to the `TURN_OFF_MSGS` message, *traffic* will return the `R_TURN_OFF_MSGS` message.

```
msg number = R_TURN_OFF_MSGS (traffic sends this in response to
               TURN_OFF_MSGS)

msg body =      int retcode     = 0 means OK; -1 means some error.
```

The element `retcode` will be -1, indicating an error, if any of the `TURN_OFF_MSGS` message numbers are outside the legal range for broadcast messages (0 to 899).

2.4 Terminating the Socket Connection

When a client has finished its work it can send a final message, `GOODBYE`, to *traffic*. When *traffic* receives this message it removes the client from the process name table and it removes the client's socket number from all broadcast tables. It then closes the client socket; no response message is sent.

```
msg number = GOODBYE (client sends this to traffic to terminate connection)
msg body =      no body
```

If a client process terminates without sending the `GOODBYE` message, *traffic* will immediately detect the closed socket and it will perform the same clean-up operations as if it had received the `GOODBYE` message.

2.5 Unrecognized Messages

If *traffic* receives any message it does not recognize (i.e. not in Table 1) then *traffic* will return the `INEXPLICABLE` message.

```
msg number = INEXPLICABLE (traffic sends this in response to an
                unrecognized message)
msg body =      struct msg_head head = Header of unrecognized message
```

The body of the message contains the header of the unrecognized message.

3 C Routines

The routines described here have been developed for the Lick **MUSIC** system. These routines, or modifications of them, may be applicable to the Keck system as well. These routines are available from UCO/Lick on an as-is basis.

3.1 Routine `startproc`

A client calls `startproc` to make the network connection to *traffic*. In fact, if a connection to *traffic* on the local host is being made, `startproc` will start the *traffic* process if it is not already running. In the Lick system we run the *traffic* process like a standard Unix daemon, so it is always up and running and the automatic startup feature of `startproc` is not used. The following shows the function calling sequence and briefly describes the parameters to the call.

```
startproc(procname,hostname,sock,msgaddr)

input  char *procname = Our process name
input  char *hostname = Host on which traffic process should be found
output int *sock      = Socket to traffic process
output int *msgaddr   = Our message address
function return       = 0 if successfull and <0 otherwise
```

The `startproc` function performs the operations discussed in section 2.1. It sets up the TCP/IP socket to *traffic* on the host given by the function parameter `hostname`, it sends the `SOCK_CONNECT` message using `procname` as the process name, and it waits for the `R_SOCK_CONNECT` reply. If any of these steps fail the function value will be a negative value and a specific error message will be logged with a call to the function `flogerror`. If all steps succeed the socket to *traffic* is stored at the address given by `sock` and the client's message address is stored at the address given by `msgaddr`.

3.2 Routine `connectproc`

A *traffic* client makes a call to `connectproc` to obtain the message address of other *traffic* clients. The following shows the function calling sequence and briefly describes the parameters to the call.

```
connectproc(sock,msgaddr,pnum,proclist,adlist,waitfunc)

input  int sock     =    Socket to traffic process
input  int msgaddr  =    Our message address
input  int pnum     =    Number of strings in proclist
input char *proclist[] = List of process names
output int adlist[] =    List of message address/status
```

```
input int (*waitfunc)()= Address of routine to call with
                         extraneous messages
```

As described in Section 2.2 of this document, for each of the processes listed in `proclist` this function sends a `CONNECT_PROC` message to *traffic* and waits for an `R_CONNECT_PROC` message. The message address returned in each `R_CONNECT_PROC` message is stored in the corresponding element of the array `adlist`.

Since there is a single connection to *traffic* through which all messages pass, it is possible that *traffic* may relay additional messages from other clients while `connectproc` is waiting for an `R_CONNECT_PROC` message. If such an extraneous (to `connectproc`) message arrives the function `waitfunc` is called with the message number as the single integer function parameter. This function is expected to deal in some way with the extraneous message. Perhaps the simplest function to use here is `add_msgque` which was described in the **MUSIC** System Message document, Part 2 of UCO/Lick Technical Report 54. `add_msgque` simply saves the message on a queue, from which it can be retrieved and dealt with later.

3.3 Other Routines

We list here, without much detail, a few additional C routines which are used internally by *traffic* or the routines described above.

3.3.1 Routine `read_remote`

The routine `read_remote` is called by *traffic*, its clients, and most other processes in the **MUSIC** system to read basic network configuration data from a configuration file. The routine `read_remote` was discussed in the **MUSIC** System Coordination Overview, Part 1 of UCO/Lick Technical Report 54.

3.3.2 Routine `waitfor`

The routine `waitfor` is used in `connectproc` and in many other **MUSIC** system routines. It permits a process to wait for a particular message from another process while still handling messages which may arrive in advance. In addition, a timeout period can be specified so that the wait for a particular message can be terminated if the expected message does not arrive.

3.3.3 Routine `flogerror`

The routine `flogerror` is just like the C `printf` function in that it takes a variable number of parameters, the first of which is a format string. A message is constructed using the format string and the resulting message is written to an error log file. the default name of the log file is `/u/ccd/errorlog`, but this can be changed by the client process. This is the standard error logging routine for the Lick **MUSIC** system. For the Keck system it may be desirable to enhance `flogerror` to have it transmit the errror message to some central network error logging process.

4 Sending Messages Through *Traffic*

4.1 Broadcast Messages

Messages which have a destination message address not equal to `TRAFFIC_CONTROLLER` are messages which are to be relayed by *traffic* from one client to another. If the destination address is equal to `BROADCAST` then the message number is used to determine the client sockets onto which the message is to be copied. If *traffic* determines that no clients wish to receive the broadcast message, then it will return a message to the original sender.

```
msg number = NO_INTEREST (traffic sends this in response to a
                broadcast message no one wants)
msg body =      struct msg_head head = Header of unwanted message
```

The body of the `NO_INTEREST` message contains the header of the original broadcast message. Any process which transmits broadcast messages should be prepared to receive this message since the reception of broadcast type messages is under the control of the receivers of the message, not the sender. What is done with the `NO_INTEREST` message once it is received is up to the client process. It could be ignored or it could be used as the trigger to stop further broadcast messages of the type returned.

4.2 Regular Messages

If the destination address is neither `TRAFFIC_CONTROLLER` nor `BROADCAST` then the message is a simple message directed from one client to another. In this case the destination address specifies the socket onto which the message is to be relayed. *Traffic* checks to make sure the destination address references a valid socket. If it does not then *traffic* can not relay the message and instead it returns an error message to the original sender.

```
msg number = NO_DELIVERY (traffic sends this if a message can't be
                forwarded)
msg body =      struct msg_head head = Header of original message
```

All clients should be prepared to receive the `NO_DELIVERY` message. This message can occur for two reasons. The first is simple programming error resulting in the use of an incorrect address. The second reason is that the intended receiving client has closed its socket to *traffic*.

5 Things To Do

Except for the `NO_DELIVERY` message described in the previous section, there is no way for one client to learn that another client has closed its socket to *traffic*. This means that the sender of a message has no way to know of the intended recepient's termination until after the message is sent. There needs to be an additional message which a client can send to *traffic* which tells *traffic* to send a termination message as soon as a particular client terminates either by closing its socket or by sending the `GOODBYE` message.

It might be nice if more explicit error codes were returned by *traffic*.

RUNNER
User's Guide

Part 2
UCO/Lick Technical Report 55

R. J. Stover

April 6, 1990

1 Introduction

Runner is one of the system coordination processes developed for the Lick and Keck optical instrument data acquisition systems. The *runner* process performs two primary functions, process start-up and process shutdown. It also provides an important secondary function by generating unique process names which are used later to make connections through the *traffic* controller (see Section 2 of the Traffic Controller User's Guide, Part 1 of this Technical Report).

When an observer starts up a user interface it establishes a TCP/IP connection to the *runner* process on whatever machines it wishes to start processes. It then sends requests to the *runner* process(es) to start up the other parts of the **MUSIC** system as are appropriate. For each request to start a process, the user interface will receive a response indicating whether or not *runner* could successfully start the requested process. In the case of a success response *runner* also sends a unique name to be used in making connections through the *traffic* controller, and it adds the new process to its internal list of currently running processes. The unique name is also passed as one of the program arguments to the process being started so it can identify itself to the *traffic* controller with the appropriate name.

If there were only one user interface the actions performed by *runner* could more simply be carried out directly by the user interface. The real advantages of the *runner* process are realized when multiple user interfaces are in use. When the second and subsequent user interfaces are started up they too make a connection to the *runner* process and they make the same requests as did the first user interface. But this time *runner* finds the requested processes in its list of currently running processes. So instead of starting up another copy it just returns a success message to the user interface for each requested process, and it records that another user interface has requested that process. As a result any number of user interfaces can be started in any order and as far as the user interfaces are concerned, they have all started up the **MUSIC** system in the same way.

As an observer closes down their own user interface the *runner* process detects the lost TCP/IP connection which the interface had originally established. In response to this lost connection *runner* will delete the record of that user interface from the list maintained for each requested process. When the last user interface is deleted from the list for a particular process, *runner* will send that process a Unix SIGTERM signal to terminate the process. Because of the way *runner* starts and stops processes, the entire **MUSIC** system is brought up when the first user interface is run and it remains up until the last user interface terminates.

The remainder of this document describes the use of *runner*. Section 2 describes communication with *runner*. Section 3 describes how *runner* starts new processes. Section 4 describes a C routine for running processes via *runner*.

2 Communicating with *Runner*

2.1 Setting Up the *Runner* Socket

Runner is typically started as a Unix daemon, and is therefore always up and running, waiting for new TCP/IP network connections. When *runner* starts up it uses the C routine `read_remote` to consult the dtakeservice file for the name 'runnerport' from which it obtains its TCP/IP port number. The value of 'runnerport' must be consistent between all of the dtakeservice files on all of the machines in the **MUSIC** system. (See the **MUSIC** System Coordination Overview document, Part 1 of UCO/Lick Technical Report 54, for a further description of `read_remote` and the file dtakeservice.) *Runner* then creates a Unix TCP/IP socket and listens for new connections at the specified port number. *Runner* clients can use `read_remote` to obtain the port number, can create a Unix socket, and can connect to the *runner* port. These functions are performed for the client using the `runproc` routine described in Section 4.

Runner uses the standard **MUSIC** message format to both send and receive messages from client processes. This document assumes the reader is familiar with this format and the various routines available for sending and receiving such messages. Read the **MUSIC** System Message document, Part 2 of UCO/Lick Technical Report 54, for a complete description. Since messages to or from *runner* do not go through the *traffic* controller the message address portion of the message header is unused in this application.

In the Lick system, *runner* uses the C routine `flogerror` to log errors and other activity in the log file `/u/ccd/runnerlog`.

2.2 Starting a Process

Once the socket to *runner* is established the client can send *runner* requests to start up processes. To start a process it sends the `START_PROC` message. This message number is symbolically defined in the C header file `runner.h`. The body of all *runner* messages is briefly described in `runner.h`, and we will repeat those descriptions here. Elements of the message body are either 32-bit integers or character arrays. In these descriptions, an integer called `X` is declared as `int X` and a character array called `C` is declared as `char C[]`. The elements are placed in the body of the message in the order they are listed in the description.

```
msg number = START_PROC (a client sends this to runner)

msg body =      char progname[] = The name of a program for runner
                                  to execute.
```

The body of the message contains the null-terminated name of the program to run. If this name begins with the virgule (`/`) then *runner* assumes the name is an actual pathname of the executable file containing the program. If the name does not begin with the virgule then *runner* assumes the name is a logical name for the desired program and it uses the logical name to look up the full pathname in the dtakeservice file. If the resulting file name appears to be executable and *runner* is not already executing the program then *runner* performs a Unix `fork` and the forked copy performs a Unix `execl` to run the requested program. If it is already running, *runner* simply returns a success status with the unique process name for the already-executing process.

For every `START_PROC` message received, *runner* sends back an `R_START_PROC` message which provides a completion status code and text message.

```
msg number = R_START_PROC (runner sends this in response to START_PROC)

msg body =      int replycode   = A code number to indicate the result
                                  of the START_PROC command.
                int textlen     = The length of the following message.
                char textmsg[]  = A text message describing the result
                                  of the START_PROC command.
```

The element `replycode` is one of the codes defined in `runner.h` and listed in Table 1. The `textmsg` portion of the body is similar to the text given under the 'Meaning' column of Table 1 in all of the failure cases. If the `replycode` is `START_PROC_SUCCESS` then the `textmsg` element contains the unique process name of the executed process. The client should use this name as the process name in the `CONNECT_PROC` message to *traffic*. (See the Traffic Controller User's Guide, Part 1 of this Technical Report).

Table 1: Symbolic Names for Runner Status Codes

Symbolic name	*Meaning*
`START_PROC_SUCCESS`	Success (Requested process execl'ed)
`START_PROC_FAIL1`	Logical translation failed
`START_PROC_FAIL2`	Can't determine if file is executable
`START_PROC_FAIL3`	File is not executable

The `...FAIL1` status is returned if *runner* tries to look up the program path name in the dtakeservice file but does not find it. *Runner* uses the Unix `stat` function to determine whether or not the requested program is executable. If the `stat` function fails (usually because the requested program does not exist) then *runner* returns the `...FAIL2` status. If the `stat` function succeeds but the data returned by `stat` indicates that the file is not executable then *runner* returns the `...FAIL3` status.

3 How Processes are Run

The processes started by *runner* execute on the *runner* host machine. However, the client and *runner* processes do not have to be executing on the same computer. It is therefore possible for a single client to start up processes on several different computers. There simply has to be *runner* processes on the appropriate machines. This is, in fact, the situation in the Lick **MUSIC** system in which there is an observer's computer and a CCD controller computer.

When *runner* executes a program it passes two arguments on the processes `argv[]` list. The first is the last component of the path name of the program's executable file. This is standard Unix practice. The second argument is the same unique process name returned to the requesting client in the `R_START_PROC` message. The executed process should use this name as its process name when it sends the `SOCK_CONNECT` message to *traffic*.

Before *runner* calls the `execl` function to run the new program it closes all open Unix file descriptors.

4 The `runproc` Routine

The C routine `runproc` has been developed for the Lick **MUSIC** system. It is used to request a *runner* to run a specified process for the client. A table of already-connected *runner* daemons is maintained by `runproc` so that a single network connection is made to any one *runner*.

```
int runproc(section,host,process,tname,tnamesize)

input  char *section  = Dtakeservice file section name
input  char *host     = Name of host to run the process on
input  char *process  = Name of process to run
output char tname[]   = Array containing unique process name
                        (or error message if return < 0)
input int tnamesize   = Size of the tname array
```

The return function value will be one of the values given in Table 2. If the function return value is positive then the function call was successful and the `tname` character array contains the unique process name of the newly run process. If `runproc` makes a new socket connection to *runner* then the positive return value will be the Unix descriptor for that socket. If `runproc` already finds it has a socket connection open to the necessary *runner* the positive return value will have 1000 added to the socket number.

Table 2: Function Return Values For `runproc`

Value	*Meaning*
Positive value:	Encoded such that:
< 1000	Socket number to a 'new' runner
≥ 1000	Socket number + 1000 to an 'old' runner
-1	Could not allocate memory for internal tables
-2	Could not make network connection with runner
-3	Too many hosts connected (50 max)
-4	Got no response from runner
-5	Error writing on runner socket
-6	Runner could not run the requested process

If the function return value is negative then the global character array `runproc_err` will contain a descriptive text message. And in the case of error -6 the message input buffer will contain the *runner* `R_START_PROC` message which can provide additional details on the error.

Given the standard configuration of the dtakeservice file the `section` parameter would normally be "network". As an example, assume that we wish to run a program called **dtake** on a host called sun1. Then our call to `runproc` might look like:

```
ret = runproc("network","sun1","dtake",tname,sizeof(tname));
```

`runproc` uses the "network" parameter and the name "runnerport" to look up the TCP/IP port number for *runner*. Using this number it establishes a connection to *runner* on host "sun1" and sends a `START_PROC` message with the name "dtake". Since "dtake" is not a full path name *runner* will look this name up in its local copy of the file dtakeservice to find the path name on sun1. The requested program is then run and the unique process name will be returned in the character array `tname`.

5 Things To Do

There needs to be a way for a client to tell *runner* that it would like to be notified whenever one of its requested processes terminates. The client would send *runner* an additional message requesting this service. Whenever *runner* detects that one of its executed processes has terminated it would then send a message to each client which both requested that process and requested the termination notice. Currently, clients keep their TCP/IP connection to *runner* open after the initial `START_PROC/R_START_PROC` transactions, but do not expect any further messages to arrive. When the termination notice feature is provided clients will need to be prepared to look for, and accept, additional messages from *runner*.