Tutorial on UVA SAFENET Lightweight Communications Architecture

Bert J. Dempsey, Jeffrey R. Michel and Alfred Weaver
Computer Networks Laboratory
University of Virginia

This document describes how to compile and run programs using the UVA SAFENET lightweight communications architecture. It covers the required hardware and software environment, program compilation issues and system administration. In the latter sections it presents a brief overview of important data structures and then describes example programs illustrating the communication paradigms supported by the SAFENET lightweight communications architecture. Finally this document describes how to run the demonstration software provided with the system.

1. Environment

Hardware

The University of Virginia SAFENET Lightweight System was developed for DTC-2 systems running SunOS 4.1. The SunOS kernel is modified to include a special character device, `/dev/xtpmo`. The SAFENET Lightweight Applications Interface is implemented in Ada with support from a library written in C. The C library makes calls to the character device driver `/dev/xtpmo` and this driver, in turn, communicates across the VME bus with an off-host processor located on a Motorola VME167 board, on which the XTP code runs. (The XTP code is stored in a PROM chip on the VME167 board.) The XTP code communicates with another board on the VME bus, the Network Peripherals FDDI card.

Software

In order to compile programs that use the UVA SAFENET binding, one must have an Ada compiler. While other Ada compilers may be adequate, the system was developed using the TeleGen2 Ada compiler. It should be the case that any Ada compiler that allows linking with object library code may be used with the UVA SAFENET binding.

Addressing information for this version of the UVA SAFENET code is handled statically from a text file named `/etc/fddi` on each SAFENET node. This system file must be present and consistent for all nodes in the network in order for communication to take place. Address formats and more addressing details are described in the following section.

2. Basic Issues in Constructing and Compiling Examples

Compiler Issues

Figure 1 shows a typical makefile for use with the SunOS `make` command. This makefile uses compiler flags that are specific to the TeleGen2 Ada compiler. Similar flags should exist for other compilers. Note the definition of the variable `SLAHOME`. It must be set to the full pathname of the directory containing the SAFENET Lightweight Application Services system. In this case that pathname is `/labhome/labjrm2k/sla`. The `BINARIES` variable may be set to the names of the binary files to be made, assuming their sources have the similar names, ending in "`.ada`" and contain compilation units having the name of the binary file. For example, the `hello_world` binary file will be made from the Ada source file `hello_world.ada`, which contains the compilation unit `HELLO_WORLD`.

```
#
# Makes Ada SLA applications
#

# Definitions:
SLAHOME= /labhome/labjrm2k/sla
BINARIES= hello_world
LIBDIR= $(SLAHOME)/C/lib
LDFLAGS= -L$(LIBDIR)
LDLIBS= -lsla


# Rules:
all: $(BINARIES)

%: %.ada
      ada -m $@ -P '$(LDFLAGS)' -p '$(LDLIBS)' $<
```

Figure 1

The `ada` command invokes the TeleGen2 Ada compiler. The TeleGen2-specific flags on the `ada` command have the following purposes. The `-m` flag gives its argument, the `$@`, as the name of the compilation unit name to be linked into the binary file. The `$@` is a special `make` symbol which stands for the name of the current target, i.e., the name of the binary file being made. The `-P` flag supplies the linker flags in the variable `$(LDFLAGS)` to the system linker. Similarly, the `-p` flag supplies the object libraries, `$(LDLIBS)` to the system linker. The `$<` is a special make symbol which stands for the name of the current dependency, i.e., the Ada source program.

In addition to the proper compiler options on the Ada command, the TeleGen2 Ada compiler requires a file called `liblst.alb` to be in the directory in which compilation takes place. This file gives the pathnames of the working Ada sublibrary and additional Ada sublibraries. One of the purposes of these sublibraries is that they contain the compilation units named in Ada context clauses (with's and use's). Figure 2 shows a sample `liblst.alb` file.

```
name: examples.sub
name: /labhome/labjrm2k/sla/ada/lib/sla.sub
name: $TELEGEN2/lib/h_unix.sub
name: $TELEGEN2/lib/h_rt.sub
```

Figure 2

The first line names the path to the working sublibrary `example.sub`. This sublibrary was created with the TeleGen2-specific `acr` command and resides in the same directory as the `liblst.alb` file. The working sublibrary contains all compilation units compiled in this directory. The next line names the path to the sublibrary containing the Ada binding itself. In this case the pathname is `/labhome/labjrm2k/sla/ada/lib/sla.sub`. The next line names the path to the TeleGen2-specific UNIX sublibrary. This sublibrary is used by some support packages that facilitate more convenient use of the Ada binding. It is not required for the use of the packages of the Ada binding, however. The last line names the path to the standard Ada sublibrary which contains all standard Ada compilation units, for example, `TEXT_IO`.

For more information about the `liblst.alb` file or the TeleGen2 Ada compiler options, see reference [1].

Addresses

Addressing is handled locally in this implementation of the SAFENET Lightweight stack. The communications code assumes the existence of a file, `/etc/fddi`, in which is found the host-specific addressing information used by the Lightweight stack communications code. This information consists of 48-bit FDDI addresses (the MAC) and ISO network layer addresses (Network Layer). In addition, each host has a unique string name associated with it.

The format of the file `/etc/fddi` is as follows. Each line contains the addressing information specific to a single host in the addressing domain. Each line must have 3 fields per line with each field separated by 1 or more whitespace characters. No whitespace characters can appear inside a field. Below is an example of a `/etc/fddi` file with addresses for 3 network nodes (`larry`, `curly`, and `moe`) and a multicast address (`stooges`):

```
00:80:d8:30:00:5A larry 0101:0101:0101:0101:0101:0101:0101:0101:0101:0147
00:80:d8:30:00:51 moe 0101:0101:0101:0101:0101:0101:0101:0101:0101:0145
00:80:d8:30:00:65 curly 0101:0101:0101:0101:0101:0101:0101:0101:0101:0146
f0:4D:75:6C:74:80 stooges 0101:0101:0101:0101:0101:0101:0101:0101:ffff:ffff
```

The first field contains a 48-bit FDDI MAC address expressed in hexadecimal form with a colon separating each byte. In the case of the `stooges` group address above, this address is an FDDI group address, as evidenced by the leading `f`.

In the unicast case, the second field is the string name of the host as returned by the UNIX (SunOS) system call, `gethostname().` In the multicast case, this string does not correspond to any UNIX naming, but is a system-wide identifier for the multicast group.

The third field is the ISO Network Layer Address for the host. This address is 20 bytes long, expressed in hexadecimal form, and has a colon on each 2 byte boundary, e.g. `0101:0101:0101:0101:0101:0101:0101:0101:0101:0146`. This third field must be followed by a newline character in order for the address file to be correctly parsed. As definitions for ISO network multicast addressing become standardized, the values used for multicast addresses here should conform to those standards (none exist at the time of this implementation). In this implementation, we adopt the convention of forcing the last 4 bytes of the ISO network layer address to be all `f`'s in the multicast case.

It is assumed that, if the `/etc/fddi` file is not physically shared (e.g., at a file server) by all hosts, then a (human) system administrator is responsible for ensuring that all the local copies of this file are consistent.

3. Important Package Specifications and Data Structures

Here we discuss important data structures used in the SLA interface. All SLA interface calls and the associated data types are presented in reference [2]. Here we simply remind the programmer of the most important and most often used data structures.

`ACTIVITY_BLOCK` (`LW_COMMUNICATIONS_SUPPORT`)

> The `ACTIVITY_BLOCK` primarily indicates scheduling parameters for the execution of a primitive. This structure is fundamental in that its `MODE` field determines whether a primitive executes synchronously (i.e., blocks until completion) or asynchronously (i.e., returns immediately, and the status of the event is later checked using either of the two functions: `GET_ACTIVITY_STATE` or `WAIT_ON_ACTIVITY`). The values for `mode` are `SYNCHRONOUS` and `ASYNCHRONOUS`.

`ACTIVITY_INDEX` (`LW_COMMUNICATIONS_SUPPORT`)

> This is a handle returned by SLA primitives which may be called asynchronously. It is used to refer to asynchronous activities. The query function `GET_ACTIVITY_STATE`, for example, takes an `ACTIVITY_INDEX` as its argument. For a synchronous activity, the index returned is 0.

`ADDRESS_ID` (`LW_ADDRESS_MANAGEMENT`)

> This is a handle returned from SLA interface by the `BIND_ADDRESS` function. It is used to refer to a transport endpoint on a particular machine. That is, an `ADDRESS_ID` is a handle for a (transport address, network address) pair.

`CONNECTION_PARAMETERS` (`LW_PROTOCOL_MANAGEMENT`) and `MESSAGE_OPTIONS` (`LW_PROTOCOL_MANAGEMENT`)

> These data structures are used to parameterize connection-oriented connection management and data transfer. The default values for these structures are obtained by calling the `RETURN_PROTOCOL_PARAMETERS` procedure in the package `LW_PROTOCOL_MANAGEMENT`. `CONNECTION_PARAMETERS` are set only at connection set-up time and thereafter are fixed for the lifetime of the connection while fields in `MESSAGE_OPTIONS` can be changed on a message-by-message basis, i.e., at arbitrary points in the lifetime of a connection.

UNITDATA_OPTIONS (LW_PROTOCOL_MANAGEMENT)

This data structure is used to parametrize connectionless data transfer calls. Its default values are also obtained via a call to the RETURN_PROTOCOL_PARAMETERS procedure in the package LW_PROTOCOL_MANAGEMENT.

CONNECTION_ID (LW_CONNECTION_MANAGEMENT)

This is a handle to an XTP connection. CONNECTION_IDs are created by the SLA interface connection establishment calls and used thereafter by the application to refer to a connection.

4. Code Common to Every Application

All programs which transfer data should get the recommended values of the protocol parameters from the protocol implementation. In addition, all programs must discover the MAC and network address of their local host. Both tasks are easily performed with a single call to the RETURN_PROTOCOL_PARAMETERS primitive. The code in Figure 3 places local host MAC and network addresses in the variables MY_MAC and MY_NETWORK and gets the recommended values for the connection parameters, message options, and unitdata options in the variables, CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, and UNITDATA_DEFAULTS, respectively.

```
RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
   CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);
```

Figure 3

5. Communication Paradigms Supported

There are three basic communication paradigms in the SLA interface: connection-oriented, unitdata (datagram), and transaction.The next section contains programs which illustrate each of the paradigms..This section outlines at the conceptual level the sequence of calls used to program in each of the three communication paradigms.

Connection-oriented

In this paradigm one application acts as a connection opener, and another acts as a connection accepter. The connection opener performs the following actions.

- Bind source and destination addresses with calls to `BIND_ADDRESS`.
- Open a connection to the destination with a call to `OPEN_CONNECTION`.
- Send and receive data using the `SEND_MESSAGE`, and `GET_MESSAGE` procedures, respectively.
- Close the connection with a call to `CLOSE_CONNECTION` unless the connection accepter does so instead.

The connection accepter does the following.

- Bind receiving address with a call to `BIND_ADDRESS`.
- Accept a connection from the sender with a call to `ACCEPT_CONNECTION`.
- Send and receive data using the `SEND_MESSAGE`, and `GET_MESSAGE` procedures, respectively.
- Close the connection with a call to `CLOSE_CONNECTION` unless the connection acceptor does so instead.

Unitdata

In this paradigm one application acts as a unitdata sender, and another acts as a unitdata receiver. The unitdata sender performs the following actions.

- Bind source and destination addresses with calls to `BIND_ADDRESS`.
- Send data to the destination using the `SEND_UNITDATA` procedure.

The unitdata receiver does the following.

- Bind receiving address with a call to `BIND_ADDRESS`.
- Receive data from the sender using the `GET_MESSAGE` procedure.

Transactions

In this paradigm one application acts as a client, employing the services of another application which acts as a server. The client reforms the following actions.

- Bind source and destination addresses with calls to `BIND_ADDRESS`.
- Send a request message to the server and receive a response message both via a call to `TRANSACTION_REQUEST`.

The server does the following.

- Bind receiving address with a call to BIND_ADDRESS.
- Accept a request message from the client with a call to ACCEPT_TRANSACTION_REQUEST.
- Send a response message to the client with a call to SEND_TRANSACTION_RESPONSE.

6. Example Programs

In order to illustrate the use of the Ada binding to the SAFENET Lightweight Application Services the following example programs are given below.

- addresses.ada
- connection_sender.ada and connection_receiver.ada
- multicast_connection_sender.ada and multicast_connection_receiver.ada
- unitdata_sender.ada and unitdata_receiver.ada
- multicast_unitdata_sender.ada and multicast_unitdata_receiver.ada
- client.ada and server.ada
- file_sender.ada and file_receiver.ada

The programs are located in the directory /labhome/labjrm2k/sla/examples.

6.1. addresses.ada

This program demonstrates the use of the ISO_ADDRESSING and HOST_NAME packages by displaying the network address and name of the local host. The ISO_ADDRESSING and HOST_NAME packages are not part of the Ada binding, but are included as an aid to applications programmers using ISO addressing and UNIX host names. Note that since this program does not act in data transfers, it does not need a call to the RETURN_PROTOCOL_PARAMETERS procedure. The program is run by typing

```
addresses
```

at the shell prompt.

First the program uses the GET_HOST_NAME procedure defined in the package HOST_NAME and the GET_NETWORK_ADDRESS procedure defined in the package ISO_ADDRESSING to get the network address of the local host. It then prints this address on the terminal screen in dotted-decimal notation. The program then uses the network address in a call to the GET_HOST_NAME procedure defined in the package ISO_ADDRESSING. This yields the name of the host having the given network address. The program then prints this name, which is, of course, that of the local host. The output of the program looks something like this:

```
   Network address: 1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.71
   Host name: larry
```

```
with  TEXT_IO,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      ISO_ADDRESSING,
      HOST_NAME;


procedure ADDRESSES is
   ------------------------------------------------------------------
   -- This program displays the network addresses of its host. It
   -- also prints the host name associated with that address, i.e.
   -- what should be the host's name.
   ------------------------------------------------------------------

   NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   HOST : STRING (1 .. HOST_NAME.MAX_HOST_NAME_LENGTH);

   package INTEGER_TEXT_IO is new TEXT_IO.INTEGER_IO (INTEGER);
   use INTEGER_TEXT_IO;
begin
   ------------------------------------------------------------------
   -- Acquire the network addresses of the current host.
   ------------------------------------------------------------------
   GET_NETWORK_ADDRESS (GET_HOST_NAME, NETWORK);

   ------------------------------------------------------------------
   -- Display the network addresses.
   ------------------------------------------------------------------
   PUT ("Network address: ");
   for I in NETWORK'RANGE loop
      PUT (INTEGER (NETWORK (I)), 1);
      if I < NETWORK'LAST then
```

```
         PUT ('.');
      end if;
   end loop;
   NEW_LINE;

   ------------------------------------------------------------------
   -- Acquire and display the name of the host with the given network
   -- address. This name should of course be that of the current
   -- host!
   ------------------------------------------------------------------
   GET_HOST_NAME (NETWORK, HOST);
   PUT_LINE ("Host name: " & HOST);
end ADDRESSES;
```

6.2. connection_sender.ada and connection_receiver.ada

These programs illustrate the connection-oriented paradigm. The `connection_sender.ada` program is the connection opener, and `connection_receiver.ada` is the accepter.

6.2.1. connection_sender.ada

This program sends the message "Hello world" to a user-specified remote host. The program is run by typing

```
connection_sender
```

at the shell prompt, and when given the prompt

```
Enter receiving host name:
```

responding with the name of the receiving host, for example, `moe`.

First the program calls the `RETURN_PROTOCOL_PARAMETERS` procedure as discussed in section 4. It then binds the address from which it will send with the procedure `BIND_ADDRESS`, receiving an address identifier. Next, the program prompts the user for the name of the receiving host and receives its name. With the aid of the `GET_NETWORK_ADDRESS` procedure, the program binds the address of the receiver, receiving an address identifier. Then our example program opens a connection to the

receiving host using the OPEN_CONNECTION procedure and the address identifiers. Then it uses the SEND_MESSAGE procedure to send the message “Hello world” to the receiver. Finally it closes the connection with the CLOSE_CONNECTION procedure.

The output from a run of the program looks like this:

```
Enter receiving host name: moe
```

where the user input is printed in bold type.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure CONNECTION_SENDER is
   ------------------------------------------------------------------
   -- This program sends the message “Hello world” to a user-specified
   -- receiving host using the connection-oriented paradigm.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   DESTINATION_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
```

```
   DESTINATION_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘r’, ‘e’, ‘m’, ‘o’, ‘t’, ‘e’, others => ASCII.NUL);
   MY_ID, DESTINATION_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   CONNECTION : LWCM.CONNECTION_ID;
   INDEX : LWCS.ACTIVITY_INDEX;
   DESTINATION_HOST_NAME : STRING (1 .. MAX_HOST_NAME_LENGTH);
   LAST : NATURAL;
   MESSAGE : constant STRING := “Hello world”;
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the source address to be used in the connection open. The
   -- source address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Get the name of the receiving host as input, and bind the
   -- destination address to be used in the connection open to the
   -- address of the receiving host.
   ------------------------------------------------------------------
   PUT (“Enter receiving host name: “);
   GET_LINE (DESTINATION_HOST_NAME, LAST);
   GET_NETWORK_ADDRESS (DESTINATION_HOST_NAME
      (DESTINATION_HOST_NAME’FIRST .. LAST), DESTINATION_NETWORK);
   BIND_ADDRESS (DESTINATION_LOGICAL_NAME, DESTINATION_NETWORK,
      DESTINATION_PORT, DESTINATION_ID);

   ------------------------------------------------------------------
   -- Open a connection to the receiving host.
   ------------------------------------------------------------------
   OPEN_CONNECTION (DESTINATION_ID, MY_ID, ACTIVITY_PARAMETERS,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, CONNECTION, INDEX);

   ------------------------------------------------------------------
   -- Send the message over the connection.
   ------------------------------------------------------------------
   SEND_MESSAGE (CONNECTION, ACTIVITY_PARAMETERS, MESSAGE’ADDRESS,
      MESSAGE’LENGTH, INDEX);

   ------------------------------------------------------------------
   -- Close the connection gracefully.
   ------------------------------------------------------------------
   CLOSE_CONNECTION (CONNECTION, GRACEFUL, ACTIVITY_PARAMETERS, INDEX);
```

```
end CONNECTION_SENDER;
```

6.2.2. connection_receiver.ada

This program receives the message “Hello world” from a remote host and displays it. The program is run by typing

```
connection_receiver
```

at the shell prompt.

First the program calls the RETURN_PROTOCOL_PARAMETERS procedure as discussed in section 4. It then binds the address at which it will receive with the procedure BIND_ADDRESS, receiving an address identifier. Then our example program accepts a connection from the connection opener using the ACCEPT_CONNECTION procedure. Then it uses the GET_MESSAGE procedure to receive the message “Hello world” from the sender. Finally it displays the message.

The output from a run of the program looks like this:

```
Received message: Hello world
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure CONNECTION_RECEIVER is
   ------------------------------------------------------------------
   -- This program receives a message using the connection-oriented
```

```
   -- paradigm and then displays it.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   MY_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   CONNECTION : LWCM.CONNECTION_ID;
   INDEX : LWCS.ACTIVITY_INDEX;
   MESSAGE : STRING (1 .. LWCS.MAX_DATA_BUFFER_SIZE);
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the receiver address to be used in the connection open.
   -- The address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Accept a connection from the sending host.
   ------------------------------------------------------------------
   ACCEPT_CONNECTION (MY_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, CONNECTION, INDEX);

   ------------------------------------------------------------------
   -- Send the message over the connection.
   ------------------------------------------------------------------
   GET_MESSAGE (CONNECTION,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      MESSAGE’ADDRESS, MESSAGE’LENGTH, INDEX);

   ------------------------------------------------------------------
```

```
   -- Display the message.
   ------------------------------------------------------------------
   PUT_LINE ("Received message: " &
      MESSAGE (MESSAGE'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));

   ------------------------------------------------------------------
   -- The other host closes the connection.
   ------------------------------------------------------------------
end CONNECTION_RECEIVER;
```

6.3. multicast_connection_sender.ada and multicast_connection_receiver.ada

These programs illustrate multicast connections. The `multicast_connection_sender.ada` program is the connection opener, and `multicast_connection_receiver.ada` is the accepter.

6.3.1. multicast_connection_sender.ada

This program sends the message "Hello world" to a host group. The program is run by typing

```
multicast_connection_sender
```

at the shell prompt.

The program is very similar to the `connection_sender.ada` program above. It differs however in three important ways, however. First, rather than prompting the user for the name of the receiving host, the program instead uses the `GET_NETWORK_ADDRESS` procedure to get the address of the host group, `stooges`, and binds its address as that of the receiver. Second, it ensures that the sending address and receiving address use the same port, `GROUP_PORT`. Third, the program uses connection parameters based on the defaults returned by `RETURN_PROTOCOL_PARAMETERS`, but sets the `MULTICAST` field to `TRUE`, in contrast to its default of `FALSE`. It is clear that the use of multicast is very straightforward.

The program produces no output.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure MULTICAST_CONNECTION_SENDER is
   ------------------------------------------------------------------
   -- This program sends the message “Hello world” to a host group
   -- using the connection-oriented paradigm.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS, MY_CONNECTION_PARAMETERS : LWPM.
      CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   GROUP_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   DESTINATION_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘r’, ‘e’, ‘m’, ‘o’, ‘t’, ‘e’, others => ASCII.NUL);
   MY_ID, DESTINATION_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   CONNECTION : LWCM.CONNECTION_ID;
   INDEX : LWCS.ACTIVITY_INDEX;
   DESTINATION_HOST_GROUP_NAME : constant STRING := “stooges”;
   MESSAGE : constant STRING := “Hello world”;
begin
   ------------------------------------------------------------------
```

```
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the source address to be used in the connection open. The
   -- source address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, GROUP_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Bind the destination address to be used in the connection open
   -- to that of the destination host group.
   ------------------------------------------------------------------
   GET_NETWORK_ADDRESS (DESTINATION_HOST_GROUP_NAME,
      DESTINATION_NETWORK);
   BIND_ADDRESS (DESTINATION_LOGICAL_NAME, DESTINATION_NETWORK,
      GROUP_PORT, DESTINATION_ID);

   ------------------------------------------------------------------
   -- Set the connection parameter for multicast and open a connection
   -- to the receiving host group.
   ------------------------------------------------------------------
   MY_CONNECTION_PARAMETERS := CONNECTION_DEFAULTS;
   MY_CONNECTION_PARAMETERS.MULTICAST := TRUE;
   OPEN_CONNECTION (DESTINATION_ID, MY_ID, ACTIVITY_PARAMETERS,
      MY_CONNECTION_PARAMETERS, MESSAGE_DEFAULTS, CONNECTION, INDEX);

   ------------------------------------------------------------------
   -- Send the message over the connection.
   ------------------------------------------------------------------
   SEND_MESSAGE (CONNECTION, ACTIVITY_PARAMETERS, MESSAGE'ADDRESS,
      MESSAGE'LENGTH, INDEX);

   ------------------------------------------------------------------
   -- Close the connection gracefully.
   ------------------------------------------------------------------
   CLOSE_CONNECTION (CONNECTION, GRACEFUL, ACTIVITY_PARAMETERS, INDEX);
end MULTICAST_CONNECTION_SENDER;
```

6.3.2. multicast_connection_receiver.ada

This program receives the message “Hello world” from a remote host and displays it. The program is run by typing

```
multicast_connection_receiver
```

at the shell prompt.

The program is very similar to the `connection_receiver.ada` program above. It differs in two important ways, however. Instead of binding the receiving address to that of the local host, it binds to the host group, `stooges`. In addition is sets the `MULTICAST` connection parameter to `TRUE`. This allows several receiver programs, running on different hosts to receive the single transmission from the sender.

The output from a run of the program looks like this:

```
   Received message: Hello world
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure MULTICAST_CONNECTION_RECEIVER is
   ------------------------------------------------------------------
   -- This program receives a message using the connection-oriented
   -- paradigm and then displays it.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS, MY_CONNECTION_PARAMETERS : LWPM.
      CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, GROUP_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   GROUP_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
```

```
      ('l', 'o', 'c', 'a', 'l', others => ASCII.NUL);
   GROUP_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   CONNECTION : LWCM.CONNECTION_ID;
   INDEX : LWCS.ACTIVITY_INDEX;
   MY_HOST_GROUP_NAME : constant STRING := "stooges";
   MESSAGE : STRING (1 .. LWCS.MAX_DATA_BUFFER_SIZE);
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the receiver address to be used in the connection open.
   -- The address is that of this host group.
   ------------------------------------------------------------------
   GET_NETWORK_ADDRESS (MY_HOST_GROUP_NAME, GROUP_NETWORK);
   BIND_ADDRESS (MY_LOGICAL_NAME, GROUP_NETWORK, GROUP_PORT, GROUP_ID);

   ------------------------------------------------------------------
   -- Set the connection parameter for multicast and accept a
   -- connection from the sending host.
   ------------------------------------------------------------------
   MY_CONNECTION_PARAMETERS := CONNECTION_DEFAULTS;
   MY_CONNECTION_PARAMETERS.MULTICAST := TRUE;
   ACCEPT_CONNECTION (GROUP_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      MY_CONNECTION_PARAMETERS, MESSAGE_DEFAULTS, CONNECTION, INDEX);

   ------------------------------------------------------------------
   -- Send the message over the connection.
   ------------------------------------------------------------------
   GET_MESSAGE (CONNECTION,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      MESSAGE'ADDRESS, MESSAGE'LENGTH, INDEX);

   ------------------------------------------------------------------
   -- Display the message.
   ------------------------------------------------------------------
   PUT_LINE ("Received message: " &
      MESSAGE (MESSAGE'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));

   ------------------------------------------------------------------
   -- The other host closes the connection.
   ------------------------------------------------------------------
end MULTICAST_CONNECTION_RECEIVER;
```

6.4. unitdata_sender.ada and unitdata_receiver.ada

These programs illustrate the unitdata paradigm. The `unitdata_sender.ada` program is the unitdata sender, and `unitdata_receiver.ada` is the unitdata receiver.

6.4.1. unitdata_sender.ada

This program sends the message “Hello world” to a user-specified remote host. The program is run by typing

```
unitdata_sender
```

at the shell prompt, and when given the prompt

```
Enter receiving host name:
```

responding with the name of the receiving host, for example, `moe`.

First the program calls the `RETURN_PROTOCOL_PARAMETERS` procedure as discussed in section 4. It then binds the address from which it will send with the procedure `BIND_ADDRESS`, receiving an address identifier. Next, the program prompts the user for the name of the receiving host and receives its name. With the aid of the `GET_NETWORK_ADDRESS` procedure, the program binds the address of the receiver, receiving an address identifier. Finally, our example program sends the message “Hello world” to the receiver using the address identifiers and the `SEND_UNITDATA` procedure.

The output from a run of the program looks like this:

```
Enter receiving host name: moe
```

where the user input is printed in bold type

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;
```

```
use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure UNITDATA_SENDER is
   ------------------------------------------------------------------------
   -- This program sends the message “Hello world” to a user-specified
   -- receiving host using the connectionless paradigm.
   ------------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   DESTINATION_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   DESTINATION_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘r’, ‘e’, ‘m’, ‘o’, ‘t’, ‘e’, others => ASCII.NUL);
   MY_ID, DESTINATION_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   INDEX : LWCS.ACTIVITY_INDEX;
   DESTINATION_HOST_NAME : STRING (1 .. MAX_HOST_NAME_LENGTH);
   LAST : NATURAL;
   MESSAGE : constant STRING := “Hello world”;
begin
   ------------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------------
   -- Bind the source address to be used in the send. The source
   -- address is that of this host.
   ------------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);
```

```
   ------------------------------------------------------------------
   -- Get the name of the receiving host as input, and bind the
   -- destination address to be used in the send to the address of the
   -- receiving host.
   ------------------------------------------------------------------
   PUT (“Enter receiving host name: “);
   GET_LINE (DESTINATION_HOST_NAME, LAST);
   GET_NETWORK_ADDRESS (DESTINATION_HOST_NAME
      (DESTINATION_HOST_NAME’FIRST .. LAST), DESTINATION_NETWORK);
   BIND_ADDRESS (DESTINATION_LOGICAL_NAME, DESTINATION_NETWORK,
      DESTINATION_PORT, DESTINATION_ID);

   ------------------------------------------------------------------
   -- Send the message to the receiving host.
   ------------------------------------------------------------------
   SEND_UNITDATA (DESTINATION_ID, MY_ID, ACTIVITY_PARAMETERS,
      MESSAGE’ADDRESS, MESSAGE’LENGTH, UNITDATA_DEFAULTS, INDEX);
end UNITDATA_SENDER;
```

6.4.2. unitdata_receiver.ada

This program receives the message “Hello world” from a remote host and displays it. The program is run by typing

```
unitdata_receiver
```

at the shell prompt.

First the program calls the RETURN_PROTOCOL_PARAMETERS procedure as discussed in section 4. It then binds the address at which it will receive with the procedure BIND_ADDRESS, receiving an address identifier. Finally our example program receives the message “Hello world” from the unitdata sender using the address identifier and the GET_UNITDATA procedure and displays it.

The output from a run of the program looks like this:

```
Received message: Hello world
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
```

```
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure UNITDATA_RECEIVER is
   ------------------------------------------------------------------------
   -- This program receives a message using the connectionless
   -- paradigm and then displays it.
   ------------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      ('l', 'o', 'c', 'a', 'l', others => ASCII.NUL);
   MY_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   INDEX : LWCS.ACTIVITY_INDEX;
   MESSAGE : STRING (1 .. LWCS.MAX_INITIAL_DATA_BUFFER_SIZE);
begin
   ------------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------------
   -- Bind the receiver address to be used in the get. The address is
   -- that of this host.
   ------------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------------
```

```
   -- Get the message from the sending host.
   ---------------------------------------------------------------
   GET_UNITDATA (MY_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      MESSAGE'ADDRESS, MESSAGE'LENGTH, UNITDATA_DEFAULTS, INDEX);

   ---------------------------------------------------------------
   -- Display the message.
   ---------------------------------------------------------------
   PUT_LINE ("Received message: " &
      MESSAGE (MESSAGE'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));
end UNITDATA_RECEIVER;
```

6.5. multicast_unitdata_sender.ada and multicast_unitdata_sender.ada

These programs illustrate the unitdata paradigm with multicast. The `unitdata_sender.ada` program is the unitdata sender, and `unitdata_receiver.ada` is the unitdata receiver.

6.5.1. multicast_unitdata_sender.ada

This program sends the message "Hello world" to a host group. The program is run by typing

```
multicast_unitdata_sender
```

at the shell prompt.

The program is very similar to the `unitdata_sender.ada` program above. It differs however in three important ways, however. First, rather than prompting the user for the name of the receiving host, the program instead uses the `GET_NETWORK_ADDRESS` procedure to get the address of the host group, `stooges`, and binds its address as that of the unitdata receiver. Second, it ensures that the sending address and receiving address use the same port, `GROUP_PORT`. Third, the program uses connection parameters based on the defaults returned by `RETURN_PROTOCOL_PARAMETERS`, but sets the `MULTICAST` field to `TRUE`, in contrast to its default of `FALSE`. Note that the changes required for the of multicast unitdata directly parallel those for the use of multicast connections.

The program produces no output.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure MULTICAST_UNITDATA_SENDER is
   ------------------------------------------------------------------
   -- This program sends the message “Hello world” to a receiving host
   -- group using the connectionless paradigm.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS, MY_UNITDATA_OPTIONS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   GROUP_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   DESTINATION_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘r’, ‘e’, ‘m’, ‘o’, ‘t’, ‘e’, others => ASCII.NUL);
   MY_ID, DESTINATION_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   INDEX : LWCS.ACTIVITY_INDEX;
   DESTINATION_HOST_GROUP_NAME : constant STRING := “stooges”;
   MESSAGE : constant STRING := “Hello world”;
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
```

```
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the source address to be used in the send. The source
   -- address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, GROUP_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Bind the destination address to be used in the send to the
   -- address of the receiving host group.
   ------------------------------------------------------------------
   GET_NETWORK_ADDRESS (DESTINATION_HOST_GROUP_NAME,
      DESTINATION_NETWORK);
   BIND_ADDRESS (DESTINATION_LOGICAL_NAME, DESTINATION_NETWORK,
      GROUP_PORT, DESTINATION_ID);

   ------------------------------------------------------------------
   -- Set the unitdata options for multicast and send the message to
   -- the receiving host group.
   ------------------------------------------------------------------
   MY_UNITDATA_OPTIONS := UNITDATA_DEFAULTS;
   MY_UNITDATA_OPTIONS.MULTICAST := TRUE;
   SEND_UNITDATA (DESTINATION_ID, MY_ID, ACTIVITY_PARAMETERS,
      MESSAGE'ADDRESS, MESSAGE'LENGTH, MY_UNITDATA_OPTIONS, INDEX);
end MULTICAST_UNITDATA_SENDER;
```

6.5.2. multicast_unitdata_receiver.ada

This program receives the message “Hello world” from a remote host and displays it. The program is run by typing

```
multicast_unitdata_receiver
```

at the shell prompt.

The program is very similar to the `unitdata_receiver.ada` program above. It differs in two important ways, however. Instead of binding the receiving address to that of the local host, it binds to the host group, `stooges`. In addition it sets the `MULTICAST` connection parameter to `TRUE`. This allows several receiver programs, running on different hosts to receive the single transmission from the sender. Note again that the changes required for the of multicast unitdata directly parallel those for the use of multicast connections.

The output from a run of the program looks like this:

```
Received message: Hello world
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      ISO_ADDRESSING,
      HOST_NAME;


procedure MULTICAST_UNITDATA_RECEIVER is
   ------------------------------------------------------------------------
   -- This program receives a message using the connectionless
   -- paradigm and then displays it.
   ------------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS, MY_UNITDATA_OPTIONS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, GROUP_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   GROUP_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   GROUP_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   INDEX : LWCS.ACTIVITY_INDEX;
   MY_HOST_GROUP_NAME : constant STRING := “stooges”;
   MESSAGE : STRING (1 .. LWCS.MAX_INITIAL_DATA_BUFFER_SIZE);
begin
   ------------------------------------------------------------------------
```

```
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the receiver address to be used in the get. The address is
   -- that of this host group.
   ------------------------------------------------------------------
   GET_NETWORK_ADDRESS (MY_HOST_GROUP_NAME, GROUP_NETWORK);
   BIND_ADDRESS (MY_LOGICAL_NAME, GROUP_NETWORK, GROUP_PORT, GROUP_ID);

   ------------------------------------------------------------------
   -- Set the unitdata options for multicast and get the message from
   -- the sending host.
   ------------------------------------------------------------------
   MY_UNITDATA_OPTIONS := UNITDATA_DEFAULTS;
   MY_UNITDATA_OPTIONS.MULTICAST := TRUE;
   GET_UNITDATA (GROUP_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      MESSAGE'ADDRESS, MESSAGE'LENGTH, MY_UNITDATA_OPTIONS, INDEX);

   ------------------------------------------------------------------
   -- Display the message.
   ------------------------------------------------------------------
   PUT_LINE ("Received message: " &
      MESSAGE (MESSAGE'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));
end MULTICAST_UNITDATA_RECEIVER;
```

6.6. client.ada and server.ada

These programs illustrate the transaction paradigm. The `client.ada` program acts as the client, and `server.ada` acts as the server.

6.6.1. client.ada

This program performs a transaction sending the request message "Hello server" to a user-specified server host and receiving and displaying the response "Hello world". The program is run by typing

```
client
```

at the shell prompt, and when given the prompt

```
Enter server host name:
```

responding with the name of the serving host, for example, `moe`.

First the program calls the `RETURN_PROTOCOL_PARAMETERS` procedure as discussed in section 4. It then binds the address from which it will send the transaction request with the procedure `BIND_ADDRESS`, receiving an address identifier. Next, the program prompts the user for the name of the serving host and receives its name. With the aid of the `GET_NETWORK_ADDRESS` procedure, the program binds the address of the server, receiving an address identifier. Then our example program sends a transaction request to the receiving host using the `TRANSACTION_REQUEST` procedure and the address identifiers. This procedure also returns the server's response message "Hello world". Finally the program displays this response message.

The output from a run of the program looks like this:

```
Enter receiving host name: moe
Received response: Hello world
```

where the user input is printed in bold type.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_TRANSACTION_SERVICES,
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_TRANSACTION_SERVICES,
      ISO_ADDRESSING,
      HOST_NAME;


procedure CLIENT is
   ------------------------------------------------------------------
   -- This program is a client to the "Hello world" server. It
   -- illustrates the transaction paradigm.
   ------------------------------------------------------------------
```

```
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWTS renames LW_TRANSACTION_SERVICES;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, SERVER_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   SERVER_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : LWCS.LOGICAL_NAME :=
      ('l', 'o', 'c', 'a', 'l', others => ASCII.NUL);
   SERVER_LOGICAL_NAME : LWCS.LOGICAL_NAME :=
      ('r', 'e', 'm', 'o', 't', 'e', others => ASCII.NUL);
   MY_ID, SERVER_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   INDEX : LWCS.ACTIVITY_INDEX;
   SERVER_HOST_NAME : STRING (1 .. MAX_HOST_NAME_LENGTH);
   LAST : NATURAL;
   REQUEST : constant STRING := "Hello server";
   RESPONSE : STRING (1 .. LWCS.MAX_DATA_BUFFER_SIZE);
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the source address to be used in the transaction. The
   -- source address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Get the name of the server host as input, and bind the server
   -- address to be used in the transaction to the address of the
   -- server host.
   ------------------------------------------------------------------
   PUT ("Enter server host name: ");
   GET_LINE (SERVER_HOST_NAME, LAST);
   GET_NETWORK_ADDRESS (SERVER_HOST_NAME
      (SERVER_HOST_NAME'FIRST .. LAST), SERVER_NETWORK);
   BIND_ADDRESS (SERVER_LOGICAL_NAME, SERVER_NETWORK,
      SERVER_PORT, SERVER_ID);

   ------------------------------------------------------------------
   -- Perform the transaction with the server host.
```

```
   ------------------------------------------------------------------
   TRANSACTION_REQUEST (SERVER_ID, MY_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      REQUEST'ADDRESS, REQUEST'LENGTH, RESPONSE'ADDRESS,
      RESPONSE'LENGTH, UNITDATA_DEFAULTS, INDEX);

   ------------------------------------------------------------------
   -- Display the server's response.
   ------------------------------------------------------------------
   PUT_LINE ("Received response: " &
      RESPONSE (RESPONSE'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));
end CLIENT;
```

6.6.2. server.ada

This program performs a transaction receiving and displaying the request message "Hello server" from a remote client sending the response message "Hello world". The program is run by typing

```
server
```

at the shell prompt.

First the program calls the RETURN_PROTOCOL_PARAMETERS procedure as discussed in section 4. It then binds the address at which it will receive with the procedure BIND_ADDRESS, receiving an address identifier. Then our example program accepts a transaction request message, "Hello server", from the client using the ACCEPT_TRANSACTION_REQUEST procedure. It then displays the request and uses the SEND_TRANSACTION_RESPONSE procedure to send the response message "Hello world" back to the client, completing the transaction.

The output from a run of the program looks like this:

```
Received request: Hello server
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_TRANSACTION_SERVICES,
```

```
      ISO_ADDRESSING,
      HOST_NAME;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_TRANSACTION_SERVICES,
      ISO_ADDRESSING,
      HOST_NAME;


procedure SERVER is
   ------------------------------------------------------------------
   -- This program accepts a request for “Hello world” service,
   -- displays it and responds with “Hello world”. This illustrates
   -- the transaction paradigm.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWTS renames LW_TRANSACTION_SERVICES;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : LWCS.LOGICAL_NAME :=
      (‘l’, ‘o’, ‘c’, ‘a’, ‘l’, others => ASCII.NUL);
   MY_ID : LWAM.ADDRESS_ID;
   ACTIVITY_PARAMETERS : constant LWCS.ACTIVITY_BLOCK :=
      (END_OF_MESSAGE => TRUE,
      BYTES_RECEIVED => 0,
      MODE => SYNCHRONOUS,
      PRIORITY => 0);
   TRANSACTION : LWTS.TRANSACTION_ID;
   INDEX : LWCS.ACTIVITY_INDEX;
   REQUEST : STRING (1 .. LWCS.MAX_INITIAL_DATA_BUFFER_SIZE);
   RESPONSE : constant STRING := “Hello world”;
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the receiver address to be used in the accept of the
   -- request. The address is that of this host.
   ------------------------------------------------------------------
```

```
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Get the request from the sending host.
   ------------------------------------------------------------------
   ACCEPT_TRANSACTION_REQUEST (MY_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY_PARAMETERS),
      REQUEST'ADDRESS, REQUEST'LENGTH, UNITDATA_DEFAULTS, TRANSACTION,
      INDEX);

   ------------------------------------------------------------------
   -- Display the request.
   ------------------------------------------------------------------
   PUT_LINE ("Received request: " &
      REQUEST (REQUEST'FIRST .. ACTIVITY_PARAMETERS.BYTES_RECEIVED));

   ------------------------------------------------------------------
   -- Send the response.
   ------------------------------------------------------------------
   SEND_TRANSACTION_RESPONSE (TRANSACTION, ACTIVITY_PARAMETERS,
      RESPONSE'ADDRESS, RESPONSE'LENGTH, INDEX);
end SERVER;
```

6.7. file_sender.ada and file_receiver.ada

These programs illustrate some of the advanced features of the Ada binding. The advanced features illustrated are asynchronous activities, and data transfer as part of the connection setup and tear down procedure.The `file_sender.ada` program sends a file to a remote host running the `file_receiver.ada` program.

6.7.1. file_sender.ada

This program sends a user-specified file to a user-specified remote host. The program is run by typing

```
file_sender
```

at the shell prompt, and when given the prompts

```
Enter file to send:
Enter receiving host name:
```

responding with the name of the file and receiving host, respectively for example, `file_sender.ada` and `moe`.

Unlike the previous programs, this program uses asynchronous procedure calls. Hence, the execution of many primitives continues asynchronously after the point of call. This allows higher performance at a cost of greater program complexity. First the program calls the RETURN_PROTOCOL_PARAMETERS procedure as discussed in section 4. Then the program prompts for the name of the file to send and opens it for reading. It then binds the address from which it will send with the procedure BIND_ADDRESS, receiving an address identifier. Next, the program prompts the user for the name of the receiving host and receives its name. With the aid of the GET_NETWORK_ADDRESS procedure, the program binds the address of the receiver, receiving an address identifier. At this point the program is ready to open a connection and transfer the file.

Note that there is a maximum size for data buffers used to send messages. Hence if our file exceeds this size it must be transferred in sends of several buffers. Our program contains code to handle this complexity. First it opens a connection to the receiving host with the OPEN_CONNECTION_WITH_DATA procedure. In the process it sends as much data as possible. If the entire file was sent in the open, the program closes the connection with the CLOSE_CONNECTION procedure. On the other hand, if there are any more data left to send, the program acquires an unused buffer, fills it with data from the file, and transmits it with the SEND_MESSAGE procedure, repeating as necessary. When the program has read the last buffer of data to send, it closes the connection, with the CLOSE_CONNECTION_WITH_DATA procedure, sending the last buffer in the process. Finally the program checks to make sure that all of its asynchronous activities have completed. The program prints an error warning if any activity results in error. Whenever the program needs the state of an asynchronous activity or the error associated with it, the program calls the functions GET_ACTIVITY_STATE and GET_ERROR, respectively. Finally, the program closes the file.

The output from a run of the program looks like this:

```
Enter file to send: file_sender.ada
Enter receiving host name: moe
```

where the user input is printed in bold type.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      LW_ERROR_MANAGEMENT,
      ISO_ADDRESSING,
      HOST_NAME,
      UNIX_TYPES,
      UNIX_IMPLEMENTATION_TYPES,
      SYSTEM;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      LW_ERROR_MANAGEMENT,
      ISO_ADDRESSING,
      HOST_NAME,
      UNIX_TYPES,
      UNIX_IMPLEMENTATION_TYPES;


procedure FILE_SENDER is
   ------------------------------------------------------------------
   -- This program sends a user-specified file to a user-specified
   -- host over a connection using asynchronous primitives and sending
   -- data in the connection open and close.
   ------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;
   package LWEM renames LW_ERROR_MANAGEMENT;

   package LW_ERROR_TEXT_IO is new TEXT_IO.ENUMERATION_IO
      (LWEM.LW_ERROR);
   use LW_ERROR_TEXT_IO;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
```

```
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 1);
   DESTINATION_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      ('l', 'o', 'c', 'a', 'l', others => ASCII.NUL);
   DESTINATION_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      ('r', 'e', 'm', 'o', 't', 'e', others => ASCII.NUL);
   MY_ID, DESTINATION_ID : LWAM.ADDRESS_ID;
   CONNECTION : LWCM.CONNECTION_ID;
   MAX_FILE_NAME_LENGTH : constant POSITIVE := 255;
   FILE_NAME : STRING (1 .. MAX_FILE_NAME_LENGTH);
   FILE_NAME_LENGTH : NATURAL;
   DESTINATION_HOST_NAME : STRING (1 .. MAX_HOST_NAME_LENGTH);
   DESTINATION_HOST_NAME_LENGTH : NATURAL;
   O_RDONLY : constant INTEGER := 0;
   INPUT_FILE : INTEGER;

   ------------------------------------------------------------------------
   -- UNIX system call interfaces:
   ------------------------------------------------------------------------
   function OPEN (PATH : in C_STRING_POINTER; FLAGS, MODE : in INTEGER)
      return INTEGER;
   pragma INTERFACE (C, OPEN);

   function READ (FD : in INTEGER; BUF : in SYSTEM.ADDRESS; NBYTE : in
      INTEGER) return INTEGER;
   pragma INTERFACE (C, READ);

   function CLOSE (FD : in INTEGER) return INTEGER;
   pragma INTERFACE (C, CLOSE);
begin
   ------------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------------
   -- Bind the source address to be used in the connection open. The
   -- source address is that of this host.
   ------------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------------
   -- Get the name of the file to send.
   ------------------------------------------------------------------------
   PUT ("Enter file to send: ");
   GET_LINE (FILE_NAME, FILE_NAME_LENGTH);

   ------------------------------------------------------------------------
   -- Open the file to be sent as a file of unsigned bytes.
   ------------------------------------------------------------------------
```

```
INPUT_FILE := OPEN (ADA_STRING_TO_C_STRING_POINTER (FILE_NAME (1 ..
FILE_NAME_LENGTH)), O_RDONLY, 0);
if (INPUT_FILE = - 1) then
   PUT (“can’t open “);
   PUT (FILE_NAME);
   NEW_LINE;
   raise CONSTRAINT_ERROR;
end if;

-----------------------------------------------------------------
-- Get the name of the receiving host as input, and bind the
-- destination address to be used in the connection open to the
-- address of the receiving host.
-----------------------------------------------------------------
PUT (“Enter receiving host name: “);
GET_LINE (DESTINATION_HOST_NAME, DESTINATION_HOST_NAME_LENGTH);
GET_NETWORK_ADDRESS (DESTINATION_HOST_NAME (
   1 .. DESTINATION_HOST_NAME_LENGTH), DESTINATION_NETWORK);
BIND_ADDRESS (DESTINATION_LOGICAL_NAME, DESTINATION_NETWORK,
   DESTINATION_PORT, DESTINATION_ID);

-----------------------------------------------------------------
-- To perform the primitives asynchronously requires that we keep
-- the various state information declared below.
-----------------------------------------------------------------
declare
   BYTES_READ : INTEGER;
   MAX_PENDING : constant NATURAL := 4; -- Must be at least 2.
   type INDEX_ARRAY is array (INTEGER range <>) of LWCS.ACTIVITY_INDEX;
   INDEX : INDEX_ARRAY (1 .. MAX_PENDING) := (others => 0);
   type BUFFER_ARRAY is array (INTEGER range <>) of LWCS.DATA_BUFFER
      (1 .. LWCS.MAX_DATA_BUFFER_SIZE);
   FILE_BUFFER : BUFFER_ARRAY (1 .. MAX_PENDING);
   type ACTIVITY_ARRAY is array (INTEGER range <>) of
      LWCS.ACTIVITY_BLOCK;
   ACTIVITY : ACTIVITY_ARRAY (1 .. MAX_PENDING) := (others =>
      (END_OF_MESSAGE => FALSE,
      BYTES_RECEIVED => 0,
      MODE => ASYNCHRONOUS,
      PRIORITY => 0));
   NEXT : NATURAL;
   STATE : LWCS.ACTIVITY_STATE;
   PENDING : BOOLEAN;
begin
   -----------------------------------------------------------------
   -- Now we may start to send the file. First we fill our buffer
   -- with the at most the number of bytes that may be sent on a
   -- connection open.
   -----------------------------------------------------------------
   NEXT := 1;
   BYTES_READ := READ (INPUT_FILE, FILE_BUFFER (NEXT)’ADDRESS,
      LWCS.MAX_INITIAL_DATA_BUFFER_SIZE);

   -----------------------------------------------------------------
   -- Open a connection to the receiving host sending several bytes
```

```
-- of the file in the process.
-----------------------------------------------------------------
OPEN_CONNECTION_WITH_DATA (DESTINATION_ID, MY_ID, ACTIVITY (NEXT),
   FILE_BUFFER (NEXT)'ADDRESS, BYTES_READ, CONNECTION_DEFAULTS,
   MESSAGE_DEFAULTS, CONNECTION, INDEX (NEXT));

-----------------------------------------------------------------
-- Send the remaining data (if any) over the connection.
-----------------------------------------------------------------
if BYTES_READ < LWCS.MAX_INITIAL_DATA_BUFFER_SIZE then
   -----------------------------------------------------------
   -- We have reached the end of the file.
   -----------------------------------------------------------
   NEXT := NEXT mod MAX_PENDING + 1;
   CLOSE_CONNECTION (CONNECTION, GRACEFUL, ACTIVITY (NEXT),
      INDEX (NEXT));
else
   loop
      -----------------------------------------------------------
      -- Find index of the next buffer.
      -----------------------------------------------------------
      NEXT := NEXT mod MAX_PENDING + 1;

      -----------------------------------------------------------
      -- Ensure that it is not still in use.
      -----------------------------------------------------------
      loop
         STATE := GET_ACTIVITY_STATE (INDEX (NEXT));
         case STATE is
            when ERROR =>
               PUT ("Error during send: ");
               PUT (GET_ERROR (INDEX (NEXT)));
               NEW_LINE;
               exit;
            when NONEXISTENT | SUCCESS =>
               exit;
            when IN_PROGRESS =>
               null;
         end case;
      end loop;

      -----------------------------------------------------------
      -- The buffer is now free for use.
      -----------------------------------------------------------
      BYTES_READ := READ (INPUT_FILE, FILE_BUFFER (NEXT)'ADDRESS,
         LWCS.MAX_DATA_BUFFER_SIZE);

      -----------------------------------------------------------
      -- When BYTES_READ < LWCS.MAX_DATA_BUFFER_SIZE, we have
      -- reached the end of the file.
      -----------------------------------------------------------
      exit when BYTES_READ < LWCS.MAX_DATA_BUFFER_SIZE;

      -----------------------------------------------------------
      -- Send the buffer.
```

```
            ------------------------------------------------------------
            SEND_MESSAGE (CONNECTION, ACTIVITY (NEXT), FILE_BUFFER
               (NEXT)'ADDRESS, BYTES_READ, INDEX (NEXT));
         end loop;

         if BYTES_READ > 0 then
            ------------------------------------------------------------
            -- We have just read in the last buffer, so send it with
            -- the connection close.
            ------------------------------------------------------------
            CLOSE_CONNECTION_WITH_DATA (CONNECTION, ACTIVITY (NEXT),
               FILE_BUFFER (NEXT)'ADDRESS, BYTES_READ, INDEX (NEXT));
         else
            ------------------------------------------------------------
            -- There is no data left to send, so just close the
            -- connection.
            ------------------------------------------------------------
            CLOSE_CONNECTION (CONNECTION, GRACEFUL, ACTIVITY (NEXT),
               INDEX (NEXT));
         end if;
      end if;

      ------------------------------------------------------------------
      -- Do a final check on all activities.
      ------------------------------------------------------------------
      PENDING := TRUE;
      while PENDING loop
         PENDING := FALSE;
         for I in 1 .. MAX_PENDING loop
            STATE := GET_ACTIVITY_STATE (INDEX (I));
            case STATE is
               when ERROR =>
                  PUT ("Error during send: ");
                  PUT (GET_ERROR (INDEX (I)));
                  NEW_LINE;
               when NONEXISTENT | SUCCESS =>
                  null;
               when IN_PROGRESS =>
                  PENDING := TRUE;
            end case;
         end loop;
      end loop;
   end;

   ---------------------------------------------------------------------
   -- Close the input file.
   ---------------------------------------------------------------------
   INPUT_FILE := CLOSE (INPUT_FILE);
end FILE_SENDER;
```

6.7.2. file_receiver.ada

This program receives a file from a remote host. The program is run by typing

```
file_receiver
```

at the shell prompt, and when given the prompt

```
Enter file to receive:
```

responding with the name of the file, for example, `fs.ada`.

Like the `file_sender.ada` program, this program uses asynchronous procedure calls. Hence, the execution of many primitives continues asynchronously after the point of call. This allows higher performance at a cost of greater program complexity. First the program calls the `RETURN_PROTOCOL_PARAMETERS` procedure as discussed in section 4. Then the program prompts for the name of the file to receive, creates it and opens it for writing. It then binds the address from which it will receive with the procedure `BIND_ADDRESS`, receiving an address identifier. At this point the program is ready to accept a connection and receive the file.

Since there is a maximum size for the data buffers used to send messages, if our file exceeds this size it must be transferred in several buffers. Our program contains code to handle this complexity. First it accepts a connection to the receiving host with the `ACCEPT_CONNECTION_WITH_DATA` procedure. In the process it receives as much data as possible. Then the program does a `GET_MESSAGE` for each unused buffer. It queries the state of each activity with the `GET_ACTIVITY_STATE` function and, upon success, writes the received data to the file. Care is taken such that the activities are queried in the order that they were performed. This is to make sure that the file arrives in order. When the program discovers via an exception that the connection has been closed by the sender, it waits for the completion of any pending receiving primitives and writes their data to the file. The program prints an error warning with the aid of the `GET_ERROR` function if any activity results in error. Finally, the program closes the data file.

The output from a run of the program looks like this:

```
   Enter file to receive: fs.ada
```

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      LW_ERROR_MANAGEMENT,
      ISO_ADDRESSING,
      HOST_NAME,
      UNIX_TYPES,
      UNIX_IMPLEMENTATION_TYPES,
      SYSTEM;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_PROTOCOL_MANAGEMENT,
      LW_ADDRESS_MANAGEMENT,
      LW_CONNECTION_MANAGEMENT,
      LW_DATA_TRANSFER,
      LW_ERROR_MANAGEMENT,
      ISO_ADDRESSING,
      HOST_NAME,
      UNIX_TYPES,
      UNIX_IMPLEMENTATION_TYPES;


procedure FILE_RECEIVER is
   ------------------------------------------------------------------------
   -- This program receives a user-specified file from another host
   -- over a connection using asynchronous primitives and receiving
   -- data in the connection accept.
   ------------------------------------------------------------------------

   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWCM renames LW_CONNECTION_MANAGEMENT;
   package LWDT renames LW_DATA_TRANSFER;
   package LWEM renames LW_ERROR_MANAGEMENT;

   package LW_ERROR_TEXT_IO is new TEXT_IO.ENUMERATION_IO
      (LWEM.LW_ERROR);
   use LW_ERROR_TEXT_IO;

   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
   MY_MAC : ISO_ADDRESSING.MAC_ADDRESS;
   MY_NETWORK, DESTINATION_NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   MY_PORT : ISO_ADDRESSING.TRANSPORT_ADDRESS := (0, 0, 0, 2);
   MY_LOGICAL_NAME : constant LWCS.LOGICAL_NAME :=
      ('l', 'o', 'c', 'a', 'l', others => ASCII.NUL);
```

```
   MY_ID : LWAM.ADDRESS_ID;
   CONNECTION : LWCM.CONNECTION_ID;
   MAX_FILE_NAME_LENGTH : constant POSITIVE := 255;
   FILE_NAME : STRING (1 .. MAX_FILE_NAME_LENGTH);
   FILE_NAME_LENGTH : NATURAL;
   O_CREAT_TRUNC_WRONLY : constant INTEGER := 16#200# + 16#400# + 16#1#;
   RW_ALL : constant INTEGER := 8#666#;
   OUTPUT_FILE : INTEGER;

   ------------------------------------------------------------------
   -- UNIX system call interfaces:
   ------------------------------------------------------------------
   function OPEN (PATH : in C_STRING_POINTER; FLAGS, MODE : in INTEGER)
      return INTEGER;
   pragma INTERFACE (C, OPEN);

   function WRITE (FD : in INTEGER; BUF : in SYSTEM.ADDRESS; NBYTE : in
      INTEGER) return INTEGER;
   pragma INTERFACE (C, WRITE);

   function CLOSE (FD : in INTEGER) return INTEGER;
   pragma INTERFACE (C, CLOSE);
begin
   ------------------------------------------------------------------
   -- Gain the protocol parameter values recommended by the
   -- lightweight protocol implementors.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MY_MAC, MY_NETWORK,
   CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Bind the source address to be used in the connection open. The
   -- source address is that of this host.
   ------------------------------------------------------------------
   BIND_ADDRESS (MY_LOGICAL_NAME, MY_NETWORK, MY_PORT, MY_ID);

   ------------------------------------------------------------------
   -- Get the name of the file to receive.
   ------------------------------------------------------------------
   PUT (“Enter file to receive: “);
   GET_LINE (FILE_NAME, FILE_NAME_LENGTH);

   ------------------------------------------------------------------
   -- Create the file to be received as a file of unsigned bytes.
   ------------------------------------------------------------------
   OUTPUT_FILE := OPEN (ADA_STRING_TO_C_STRING_POINTER (FILE_NAME (1 ..
   FILE_NAME_LENGTH)), O_CREAT_TRUNC_WRONLY, RW_ALL);
   if (OUTPUT_FILE = - 1) then
      PUT (“can’t open “);
      PUT (FILE_NAME);
      NEW_LINE;
      raise CONSTRAINT_ERROR;
   end if;

   ------------------------------------------------------------------
```

```
-- To perform the primitives asynchronously requires that we keep
-- the various state information declared below.
-----------------------------------------------------------------
declare
   BYTES_WRITTEN : INTEGER;
   MAX_PENDING : constant NATURAL := 4;
   type INDEX_ARRAY is array (INTEGER range <>) of LWCS.ACTIVITY_INDEX;
   INDEX : INDEX_ARRAY (1 .. MAX_PENDING) := (others => 0);
   type BUFFER_ARRAY is array (INTEGER range <>) of LWCS.DATA_BUFFER
      (1 .. LWCS.MAX_DATA_BUFFER_SIZE);
   FILE_BUFFER : BUFFER_ARRAY (1 .. MAX_PENDING);
   type ACTIVITY_ARRAY is array (INTEGER range <>) of
      LWCS.ACTIVITY_BLOCK;
   ACTIVITY : ACTIVITY_ARRAY (1 .. MAX_PENDING) := (others =>
      (END_OF_MESSAGE => FALSE,
      BYTES_RECEIVED => 0,
      MODE => ASYNCHRONOUS,
      PRIORITY => 0));
   NEXT, FIRST_NEXT : NATURAL;
   STATE : LWCS.ACTIVITY_STATE;
   PENDING : BOOLEAN;
begin
   ------------------------------------------------------------------
   -- Accept a connection to the receiving host receiving as many
   -- bytes as possible in the process.
   ------------------------------------------------------------------
   NEXT := 1;
   ACCEPT_CONNECTION_WITH_DATA (MY_ID,
      GET_ACTIVITY_BLOCK_POINTER (ACTIVITY (NEXT)),
      FILE_BUFFER (NEXT)'ADDRESS, LWCS.MAX_INITIAL_DATA_BUFFER_SIZE,
      CONNECTION_DEFAULTS, MESSAGE_DEFAULTS, CONNECTION, INDEX
      (NEXT));

   ------------------------------------------------------------------
   -- Get the remaining data (if any) over the connection and write
   -- it to the output file.
   ------------------------------------------------------------------
   loop
      ---------------------------------------------------------------
      -- Find index of the next buffer.
      ---------------------------------------------------------------
      NEXT := NEXT mod MAX_PENDING + 1;

      ---------------------------------------------------------------
      -- Ensure that it is not still in use.
      ---------------------------------------------------------------
      loop
         STATE := GET_ACTIVITY_STATE (INDEX (NEXT));
         case STATE is
            when ERROR =>
               PUT ("Error during send: ");
               PUT (GET_ERROR (INDEX (NEXT)));
               NEW_LINE;
               exit;
            when SUCCESS =>
```

```
            BYTES_WRITTEN := WRITE (OUTPUT_FILE, FILE_BUFFER
               (NEXT)'ADDRESS, ACTIVITY (NEXT).BYTES_RECEIVED);
         when NONEXISTENT =>
            exit;
         when IN_PROGRESS =>
            null;
      end case;
   end loop;

   ------------------------------------------------------------
   -- The buffer is now free for use.
   ------------------------------------------------------------
   begin
      GET_MESSAGE (CONNECTION, GET_ACTIVITY_BLOCK_POINTER
         (ACTIVITY (NEXT)), FILE_BUFFER (NEXT)'ADDRESS,
         LWCS.MAX_DATA_BUFFER_SIZE, INDEX (NEXT));
   exception
      when CONNECTION_UNKNOWN_ERROR =>
      ------------------------------------------------------------
      -- We have reached the end of the file.
      ------------------------------------------------------------
      exit;
   end;
end loop;

------------------------------------------------------------------
-- Do a final check on all activities.
------------------------------------------------------------------
FIRST_NEXT := NEXT;
loop
   ------------------------------------------------------------
   -- Find index of the next buffer.
   ------------------------------------------------------------
   NEXT := NEXT mod MAX_PENDING + 1;

   ------------------------------------------------------------
   -- Exit the loop when we have checked all pending buffers.
   ------------------------------------------------------------
   exit when NEXT = FIRST_NEXT;

   STATE := GET_ACTIVITY_STATE (INDEX (NEXT));
   case STATE is
      when ERROR =>
         PUT ("Error during send: ");
         PUT (GET_ERROR (INDEX (NEXT)));
         NEW_LINE;
      when SUCCESS =>
         BYTES_WRITTEN := WRITE (OUTPUT_FILE, FILE_BUFFER
            (NEXT)'ADDRESS, ACTIVITY (NEXT).BYTES_RECEIVED);
      when NONEXISTENT =>
         null;
      when IN_PROGRESS =>
         begin
            WAIT_ON_ACTIVITY (INDEX (NEXT));
            BYTES_WRITTEN := WRITE (OUTPUT_FILE, FILE_BUFFER
```

```
                    (NEXT)'ADDRESS, ACTIVITY (NEXT).BYTES_RECEIVED);
                exception
                   when others =>
                      PUT ("Error during send: ");
                      PUT (GET_ERROR (INDEX (NEXT)));
                      NEW_LINE;
                end;
          end case;
       end loop;
   end;

   ------------------------------------------------------------------
   -- Close the output file.
   ------------------------------------------------------------------
   OUTPUT_FILE := CLOSE (OUTPUT_FILE);
end FILE_RECEIVER;
```

7. Demonstration Programs

The following programs demonstrate the services provided by the Ada binding to the SAFENET Lightweight Application Services.

- `send_unit` and `get_unit`
- `file_sender and file_receiver`
- `send_pics and get_pics`
- `spm and gpm`
- `Npics`

The programs are located in the directory `/labhome/labjrm2k/sla/demo`. Source code is available for the first four sets of demos and is also located in this directory. The following subsections describe what each program does and how to run it.

7.1. send_unit and get_unit

These programs demonstrate the use of unitdata for sending a message given on the command line of `send_unit` to a user on a remote host or host group also specified on the command line. On the receiving host, `get_unit` receives and displays the unitdata message. The two programs may be run in asynchronous mode. The syntax for the `send_unit` program is as follows.

`send_unit` **`[-m] [-a]`** *receiving_host_name quoted_message*

The *receiving_host_name* argument is the name of the host on which `get_unit` is run and *quoted_message* is the message, in quotes, that is to be sent. If the *receiving_host_name* is that of a host group, then the message is multicast to that host group. The optional argument **`-a`** specifies that the message is to be sent asynchronously. The `get_unit` program has the following syntax.

```
get_unit [-m] [-a]
```

The optional argument **`-m`** specifies that the program should use the address for the multicast group `stooges` as its receiving address. The optional argument **`-a`** specifies that the program be run asynchronously.

As an example, suppose the two programs are run on the two hosts, `larry`, the sender, and `moe`, the receiver. On `larry`, type the following at the shell prompt.

```
send_unit moe “Hello Admiral, are you there?”
```

On `moe`, type the following at the shell prompt.

```
get_unit
```

As a result, the `get_unit` program displays the following output.

```
Message: Hello Admiral, are you there?
```

7.2. file_sender and file_receiver

These programs are described in the previous section.

7.3. send_pics and get_pics

These programs demonstrate the use of a connection for sending very large (one megabyte) images specified on the command line of `send_pics` to a remote host or host group also specified on the command line. On the receiving host, `get_pics` receives on its screen the number of images specified on the command line. The two programs may be run in asynchronous mode. The syntax for the `send_pics` program is as follows.

`send_pics` **`[-a]`** *receiving_host_name rasterfile ...*

The *receiving_host_name* argument is the name of the host or host group on which `get_pics` is run and *rasterfile ...* is the series of 1252x900x8 Sun rasterfile images in the order in which they will be sent. If the *receiving_host_name* is that of a host group, then the images are multicast to that host group. The optional argument **`-a`** specifies that the images are to be sent asynchronously. The `get_pics` program has the following syntax.

`get_pics` **`[-m] [-a]`** *number*

The *number* argument indicates how may images are to be received. This of course must be the same as the number of images sent. The optional argument **`-m`** specifies that the program should use the address for the multicast group `stooges` as its receiving address. The optional argument **`-a`** specifies that the program be run asynchronously.

As an example, suppose the two programs are run on the two hosts, `larry`, the sender, and `moe`, the receiver. On `larry`, type the following at the shell prompt.

```
send_pics moe images/*
```

The `images` directory contains the six images to be sent. When the program prompts with the message:

```
Hit return to see the rasters...
```

hit the return key to allow sending to begin. On `moe`, type the following at the shell prompt.

```
get_pics 6
```

When the program prompts with the message:

```
Hit return to see the rasters...
```

hit the return key to allow the images to appear on `moe`’s screen.

7.4. spm and gpm

These programs demonstrate the use of a connection for sending relatively small images specified on the command line of `spm` to a remote host or host group also specified on the command line. On the receiving host, `gpm` receives on its screen the number of images specified on the command line. Each image is cycled through and sent several times just to keep the demonstration running long enough to be observed. The two programs may be run in asynchronous mode. Note that the `gpm` and `spm` programs are very similar to `send_pics` and `get_pics`, respectively. The syntax for the `send_pics` program is as follows.

`spm` **`[-a]`** *receiving_host_name rasterfile ...*

The *receiving_host_name* argument is the name of the host or host group on which `gpm` is run and *rasterfile ...* is the series of 320x200x8 Sun rasterfile images in the order in which they will be sent. If the *receiving_host_name* is that of a host group, then the images are multicast to that host group. The optional argument **`-a`** specifies that the images are to be sent asynchronously. The `gpm` program has the following syntax.

`get_pics` **`[-m] [-a]`** *number*

The *number* argument indicates how may images are to be received. This of course must be the same as the number of images sent. The optional argument **`-m`** specifies that the program should use the address for the multicast group `stooges` as its receiving address. The optional argument **`-a`** specifies that the program be run asynchronously, in which case the latest image to arrive is the one displayed.

As an example, suppose the two programs are run on the two hosts, `larry`, the sender, and `moe`, the receiver. On `larry`, type the following at the shell prompt.

```
spm moe 3d/*
```

The `3d` directory contains the eight images to be sent. When the program prompts with the message:

```
Hit return to see the rasters...
```

hit the return key to allow sending to begin. On `moe`, type the following at the shell prompt.

```
gpm 8
```

When the program prompts with the message:

```
Hit return to see the rasters...
```

hit the return key to allow the images to appear on `moe`'s screen.

7.5. Npics

`Npics` demonstrates the capability afforded by XTP to construct a reliable, ordered atomic multicast service. Such services have been shown to be useful for distributed applications where messages sent to the communicants must arrive in the same order at all sites in order to insure the consistency of state information replicated at each site. A number of protocols for such atomic broadcast service have appeared in the literature, and there continues to be active research in this area.

The reliable multicast service in XTP offers the opportunity for constructing an efficient, powerful, yet relatively simple atomic multicast service within the SAFENET communications environment. The demo illustrates the construction of such a service as an application program running above XTP. With this service a set of nodes can asynchronously send messages among themselves with the guarantee that: (1) messages are reliably delivered to all members of the multicast group and (2) each member of the multicast receives the messages sent to the group (including its own) in exactly the same order. In a full implementation, policies defining "atomic delivery" of messages would most likely be considered more carefully than in the `Npics` demo.

Figure 4 illustrates the basic communication pattern. Each participant in the multicast group that will send and receive messages is termed a Slave. Slaves send their messages to a centralized node, the Master of the group, over unicast XTP connections established at set-up time. The Master continually scans for messages sent by the Slaves and, when a message arrives, the Master sends it to the Slave group over an XTP multicast connection. Since the XTP multicast channel is reliable and it preserves the order of data sent over it, the relayed messages arrive at each Slave with a consistent ordering and at approximately the same time. The code for this service as a user program is short and straightforward, with the power of the service coming from the capabilities of the XTP multicast channel.

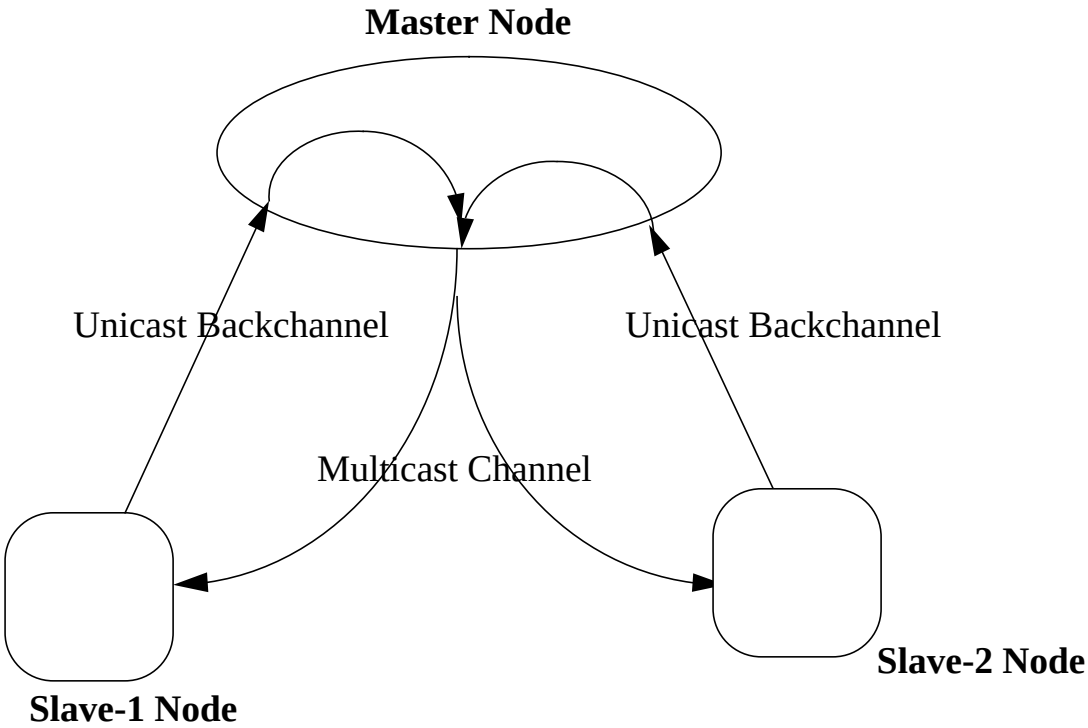

Figure 4—Data Flow in Npics Demo

The `Npics` demo illustrates consistent message ordering by making the "messages" in this case segments of 1252x900x8 Sun rasterfile images. At group set-up, the Master waits for some number of slaves (there can only be one per node) to establish

unicast channels to the Master. Then the Master establishes a multicast channel with the set of Slaves. After this NxN connection establishment completes, each slave begins sending a full screen image to the multicast group; that is, each slave sends a 1252x900x8 Sun rasterfile image to the Master. The Master relays messages arriving on the Slave's unicast backchannels to the group as discussed above. In this case the effect is to interleave the parts of the different rasterfiles being sent from the set of Slaves. Slaves display messages received from the Master as they arrive. The `Npics` demo thus creates on each Slave's screen a composite image made up of the interleaved pieces of all the Slaves' images. The point here is that, **due to consistent message ordering for the group, each Slave's screen is exactly the same.**

The following illustrates the syntax of the `Npics` demo. This information is also available by typing at the system prompt just `Npics` with no arguments. We suppose here that a two-member group is desired. Slaves are on the machines, `curly` and `moe`, while the Master node will be the third machine, `larry`. (In fact, one Slave can be on the same machine as the Master.) Then the following must be typed at each prompt:

```
curly% Npics S 1 rasterfile1
moe% Npics S 0 rasterfile2
larry% Npics M 2
```

The commands to start the Slaves should be entered before that to start the Master. At each Slave, the 'S' flag indicates this is a Slave, and *rasterfile* indicates the 1252x900x8 Sun rasterfile that this Slave will send to the group. The parameter just before *rasterfile* indicates whether this rasterfile's color map will be sent to the group as well (1 means yes, 0 no). **Exactly one Slave must use a 1 for this parameter and all other Slaves in the group must use 0.** The command line for the Master indicates 'M' for Master and '2' for the number of Slaves participating in this NxN connection.

8. System Administration

There are a number of system administration issues regarding the SAFENET lightweight system. These include protocol initialization and termination, and the modification of protocol parameter defaults on a system-wide basis. In addition, a system administrator must maintain the `/etc/fddi` file. There are a number of programs provided in the `/labhome/labjrm2k/sla/adm` directory for use in system administration. These programs are described below.

8.1. Protocol Initialization

Before any use of the SAFENET Lightweight system may be performed, the XTP protocol must be initialized. We have provided a system administration command, `init_xtp`, that initializes the protocol. To run this command, simply type

```
init_xtp
```

at the shell prompt. This command should be performed at the boot time of the host computer system. A simple way to ensure that this is the case is for the system administrator to place the command in the SunOS system file `/etc/rc.local`, a shell script of boot-time commands.

8.2. Protocol Termination

To disable the use of XTP, we have provided a command, `terminate_xtp`, that terminates the protocol. To run this command, simply type

```
terminate_xtp
```

at the shell prompt.

8.3. Setting System-Wide Protocol Parameter Defaults

When the protocol is initialized it sets up protocol parameter default values which are recommended for use by the protocol implementors. These protocol parameters are those returned by the RETURN_PROTOCOL_PARAMETERS primitive described in reference [2]. The system administrator may customize these values if necessary through the use of the UPDATE_PROTOCOL_PARAMETERS primitive also described in reference [2]. To illustrate this we have provided the following sample program update_xtp.ada.

The program gets the recommended values for the protocol parameters with the RETURN_PROTOCOL_PARAMETERS procedure and then modifies the MBUCKETS field for both the connection parameters and unitdata options, leaving the other fields unchanged. It then uses the UPDATE_PROTOCOL_PARAMETERS procedure to cause these modifications to take effect on the host. To run the program simply type

```
update_xtp
```

at the shell prompt.

```
with  TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_ADDRESS_MANAGEMENT,
      LW_PROTOCOL_MANAGEMENT,
      ISO_ADDRESSING,
      DEBUG;

use   TEXT_IO,
      LW_COMMUNICATIONS_SUPPORT,
      LW_ADDRESS_MANAGEMENT,
      LW_PROTOCOL_MANAGEMENT,
      ISO_ADDRESSING,
      DEBUG;


procedure UPDATE_XTP is
   ------------------------------------------------------------------
   -- Updates the MBUCKETS parameter of XTP.
   ------------------------------------------------------------------
```

```
   package LWCS renames LW_COMMUNICATIONS_SUPPORT;
   package LWAM renames LW_ADDRESS_MANAGEMENT;
   package LWPM renames LW_PROTOCOL_MANAGEMENT;

   MAC : ISO_ADDRESSING.MAC_ADDRESS;
   NETWORK : ISO_ADDRESSING.NETWORK_ADDRESS;
   CONNECTION_DEFAULTS : LWPM.CONNECTION_PARAMETERS;
   MESSAGE_DEFAULTS : LWPM.MESSAGE_OPTIONS;
   UNITDATA_DEFAULTS : LWPM.UNITDATA_OPTIONS;
begin -- UPDATE_XTP
   ------------------------------------------------------------------
   -- Acquire present defaults.
   ------------------------------------------------------------------
   RETURN_PROTOCOL_PARAMETERS (MAC, NETWORK, CONNECTION_DEFAULTS,
      MESSAGE_DEFAULTS, UNITDATA_DEFAULTS);

   ------------------------------------------------------------------
   -- Modify MBUCKETS.
   ------------------------------------------------------------------
   CONNECTION_DEFAULTS.MBUCKETS := 10;
   UNITDATA_DEFAULTS.MBUCKETS := 10;

   PUT_LINE (“Setting XTP parameters to the following values:”);
   PUT_CONNECTION_PARAMETERS (CONNECTION_DEFAULTS);
   PUT_MESSAGE_OPTIONS (MESSAGE_DEFAULTS);
   PUT_UNITDATA_OPTIONS (UNITDATA_DEFAULTS);

   UPDATE_PROTOCOL_PARAMETERS (CONNECTION_DEFAULTS, MESSAGE_DEFAULTS,
      UNITDATA_DEFAULTS);
end UPDATE_XTP;
```

8.4. Viewing System-Wide Protocol Parameter Defaults

To display the MAC and network addresses of the host computer (in hexadecimal) and the present values of the protocol parameters, we have provided a command, `show_xtp`. To run this command, simply type

```
show_xtp
```

at the shell prompt.

8.5. Updating the `/etc/fddi` File

Each time a new host is added to the network, a system administrator must add an entry for that host to the `/etc/fddi` file. The format of this file is described in section 2. For example, if we were to add a new host named `shimp` to the network having MAC

address `00:80:D8:30:00:67` and network address `0101:0101:0101:0101:0101:0101:0101:0101:0101:0148`, we would add the following line to `/etc/fddi` with a text editor.

```
00:80:d8:30:00:67 shimp 0101:0101:0101:0101:0101:0101:0101:0101:0101:0148
```

9. References

[1] *TeleGen2 for Sun-4 Workstations User Guide*, UG-1461N-V1.1a (SUN-4), Telesoft, July 1990.

[2] Dempsey, Bert J., Fenton, John C., Michel, Jeffrey R., Waterman, Alexander S., and Weaver, Alfred C., “Ada Binding Reference Manual - SAFENET Lightweight Application Services”, University of Virginia, November 1992.