18.4 The Universal Internet Communication Interface (UICI)

The Universal Internet Communication Interface (UICI) library, summarized in Table 18.1, provides a simplified interface to connection-oriented communication in UNIX. UICI is not part of any UNIX standard. The interface was designed by the authors to abstract the essentials of network communication, while hiding the details of the underlying network protocols. UICI has been placed in the public domain and is available on the book website. Programs that use UICI should include the `uici.h` header file.

This section introduces the UICI library. The next two sections implement several client-server strategies in terms of UICI. Section 18.7 discusses the implementation of UICI using sockets, and Appendix C provides a complete UICI implementation.

When using sockets, a server creates a communication endpoint (a socket) and associates it with a well-known port (binds the socket to the port). Before waiting for client requests, the server sets the socket to be passive so that it can accept client requests (sets the socket to listen). Upon detection of a client connection request on this endpoint, the server generates a new communication endpoint for private two-way communication with the client. The client and server access their communication endpoints by reading and writing using file descriptors. When finished, both parties close the file descriptors, releasing the resources associated with the communication channel.

UICI prototype	description (assuming no errors)
`int u_open(u_port_t port)`	creates a TCP socket bound to `port` and sets the socket to be passive returns a file descriptor for the socket
`int u_accept(int fd, int hostnsize) char *hostn)`	waits for connection request on `fd`, and on return `hostn` has first `hostname-1` characters of the client's host name returns a communication file descriptor
`int u_connect(u_port_t port, char *hostn)`	initiates a connection to server on port `port` and host `hostn`. returns a communication file descriptor

Table 18.1: The UICI API. If unsuccessful, UICI functions return –1 and set `errno`.

Figure 18.6 depicts a typical sequence of UICI calls used in client-server communication. The server creates a communication endpoint (`u_open`) and waits for a client to send a request (`u_accept`). The `u_accept` returns a private communication file descriptor. The client creates a communication endpoint for communicating with the server (`u_connect`).

Once they have established a connection, a client and server can communicate over the network using the ordinary `read` and `write` functions. Alternatively they can use the

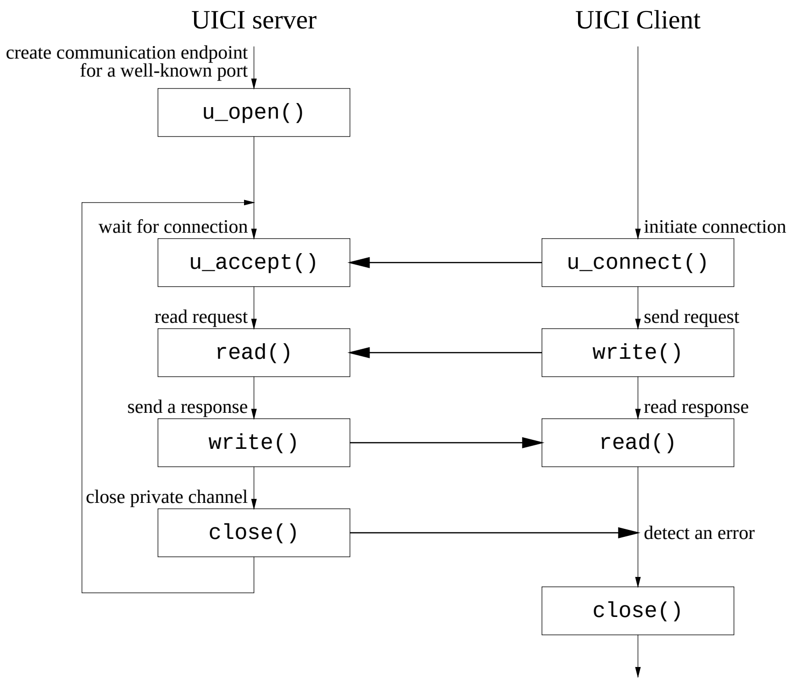

Figure 18.6: A typical interaction of a UICI client and server.

more robust `r_read` and `r_write` from the restart library of Appendix B. Either side can terminate communication by calling `close` or `r_close`. After the `close`, the remote end detects an end-of-file when reading or an error when writing. The diagram in Figure 18.6 shows a single request followed by a response, but more complicated interactions might involve several exchanges followed by a `close`.

In summary, UICI servers are organized as follows.

- Open a well-known listening port (`u_open`). The `u_open` returns a *listening file descriptor*.
- Wait for a connection request on the listening file descriptor (`u_accept`). The `u_accept` blocks until a client requests a connection and then returns a *communication file descriptor* to use as a handle for private, two-way client-server communication.
- Communicate with the client using the communication file descriptor (`read` and `write`).
- Close the communication file descriptor (`close`).

UICI clients use the following steps.

- Connect to a specified host and port (`u_connect`). The connection request returns the communication file descriptor used for two-way communication with the server.
- Communicate with the server (`read` and `write`) using the communication file descriptor.
- Close the communication file descriptor (`close`).

18.4.1 Error handling

A major design issue for UICI was how to handle errors. UNIX library functions generally report errors by returning –1 and setting `errno`. In order to keep the UICI interface simple and familiar, UICI functions also return –1 and set `errno`. None of the UICI functions display error messages. Applications using UICI should test for errors and display error messages where appropriate. Since UICI functions always set `errno` when a UICI function returns an error, applications can use `perror` to display the error message. POSIX does not specify an error code corresponding to the inability to resolve a host name. The `u_connect` function returns –1 and sets `errno` to `EINVAL`, indicating an invalid parameter when it cannot resolve the host name.

18.4.2 Reading and writing

Once they have obtained an open file descriptor from `u_connect` or `u_accept`, UICI clients and servers can use the ordinary `read` and `write` functions to communicate. We use the functions from the restart library since they are more robust and simplify the code.

Recall that `r_read` and `r_write` both restart themselves after being interrupted by a signal. Like `read`, `r_read` returns the number of bytes read or 0 if it encounters an end-of-file. If unsuccessful, `r_read` returns –1 and sets `errno`. If successful, `r_write` returns the number of bytes requested to write. The `r_write` function returns –1 and sets `errno` if an error occurred or if it could not write all of the requested bytes without error. The `r_write` function restarts itself if not all of the requested bytes have been written. This chapter also uses the `copyfile` function from the restart library, introduced in Program 4.6 on page 100 and `copy2files` introduced in Program 4.13 on page 111.

The restart library supports only blocking I/O. That is, an `r_read` or `r_write` may cause the caller to block. An `r_read` blocks until some information is available to be read. The meaning of blocking for `r_write` is less obvious. In the present context, blocking means that `r_write` returns when the output has been transferred to a buffer used by the transport mechanism. Returning does not imply that the message has actually been delivered to the destination. Writes may also block because of message delivery problems in the lower protocol layers or if all of the buffers for the network protocols are full. Fortunately, the issues of blocking and buffering are transparent for most applications.