DRCIPC 2.10 User's Manual

by

L. Leao

EDRC-05-02-86 2

September 1986

CARNEGIE-MELLON UNIVERSITY

DRCIPC 2.10 User's Manual

by

Luiz Alberto Villaga Leão

Engineering Design Research Center
Pittsburgh, Pennsylvania
August, 1986

Table of Contents

List of Figures

Acknowledgements

I am glad to have the chance of working with professor Sarosh Talukdar as my advisor. I am very thankful for his advice and friendship.

I would like to thank Mr. José Carlos Pereira Lima, of Banco Itaú S. A., for his constant support and attention since I joined Itaudata Ltda.

I would specially like to thank the directors of Banco Itaú S. A. for the opportunity and support that allowed my presence in the Carnegie-Mellon University and gave me the chance to conduct this work. I am particularly grateful to Dr. Sérgio Augusto Sawaya, for his trust, consideration and care.

Summary

DRCIPC is a package of routines that provide Inter-Process Communications in a network of computers executing the Unix 4.2 BSD Operating System. It provides global naming, using a centralized name-server process, guaranteed delivery of arbitrarily sized string messages and synchronous handling of multiple open sockets. It uses connected stream sockets to implement basic communications, hiding most of the details of the handling of sockets. The global naming facilities can be used independently from the message handling routines.

This manual describes **the** use of the **DRCIPC package.**

1. Introduction

DRCIPC was developed to allow COPS[1,2,3] to be operational in the Unix 4.2 BSD[4] environment. Although it was built with this specific objective in mind, it was also made general enough to be of potential interest to other applications.

DRCIPC tries to hide most of the details of the handling of Unix 4.2 BSD sockets[5], potentially allowing for the fast implementation of distributed systems using messages for interprocess communications, running in networks of computers.

1.1 Overview

The DRCIPC package has two reasonably distinct parts: the connection related routines and the message handling routines. The two parts are somewhat interdependent but can be used separately.

The connection related routines allow a process to assign itself one or more global names of its choice, to initiate connections, to accept connections, to close connections and to unassign names previously assigned to it

The message routines send and receive messages, in order and with guaranteed delivery. Those messages are strings of ascii characters, terminated by a null (\0). All 8 bits codes are transmitted (including the terminating null in a message). The *drcjecv* routine also handles the accepting of new connections, working with *drejeonnect* to keep a list of the local connected sockets. This routine allows a process to perform a receive without specifying the particular connection, in non-blocked, blocked or blocked with timeout modes.

An illustrative example of use can be found in figure 1-1. Parts of two programs are shown: a server that sends back whatever it receives and a program that reads lines, sends them to the echo-server and prints them back.

In this case, different clones of the printing program can be simultaneously connected to the echo-server. If the echo-server receives no message for more then 120 seconds, *drcjecv* will timeout with *retcode* 0.

This figure exemplifies the bidirectionality of the connections, the use of *drcjvaitjime,* the use of *drcjecv socket* and the use of most of the DRCIPC calls. The tests on *retcode* where not shown to simplify the example.

Echo-server program:

```
...
#include<drcipc.h>
#define LINESIZE      512
...
int    retcode;
char   line[LINESIZE];
...
     retcode = drc_checkin("echo-server");
...
     while (true) {
          drc_wait_time = 120; (seconds)
          retcode = drc_recv(line,LINESIZE);
...
          retcode = drc_send(drc_recv_socket,line);
...
     }
```

Program that requests echoes from the echo-server:

```
...
#include<drcipc.h>
#define LINESIZE      512
...
int    retcode;
int    echosocket;
char   line[LINESIZE];
...
     retcode = drc_connect("echo-server");
...
     echosocket = retcode;
...
     drc_wait_time = DRC_BLOCK;
     while (true) {
          printf("Line: ");
          scanf("%s\n",line);
          retcode = drc_send(echosocket,line);
...
          retcode = drc_recv(line,LINESIZE);
...
          printf("Echo: %s.\n",line);
     }
```

Figure 1-1: Example of use of drcipc.

2. DRCIPC reference text

2.1 DRCIPC entry points

It follows a description of the DRCIPC calls.

2.1.1 drc.checkin(name)

Drc.checkin assigns a *name* to a process (figure 2-1). This name will be globally known and can be used by other processes to *connect* to a process that has *checked-in* with that name.

```
...
char    namc[64]
...
        strcpy(name,argv[l]);
        drc.checkin(name);
...
```

Figure 2-1: Example of use of drc.checkin (from COPS).

Its only argument is:

- name: name to be assigned to the process.

Its execution will result in the creation of a passive listening stream socket, in the generation of an entry in the DRCNAMESERVER corresponding to the named process (containing its name, host name, network address and PID) and in the reception of an acknowledgement from the DRCNAMESERVER with its error code, if any.

It returns:

- -1, when error;
- the created passive socket, otherwise.

The returning passive socket can be ignored by most applications.

A process can *checkin* more than once with different names; only one passive socket will be created, though.

A *checkin* with a name already in use will cause an error, if a process with the PID present in the entry with

that name is alive in the correspondent host. If this is not the case, the name will be redefined and a new entry will be generated in the drenameserver (Note: this is not true in the present version; vide chapter 3).

There is a maximum number of sockets permitted within a process by the present Unix 4.2 implementation: this number is site dependent, less than 32 and guaranteed to be at least 20.

2.1.2 drc.checkout(name)

Drc.checkout unassigns a *name* to a process (figure 2-2).

```
...
char   name[64]
...
      drc.chcckout(name);
...
```

Figure 2-2: Example of use of drc.checkout (from **COPS).**

Its only argument is:

- **name:** name to be unassigned to the process.

Its execution will result in the deletion of the entry corresponding to the given name present in the DRCNAMESERVER, in the reception of an acknowledgement from the DRCNAMESERVER with its error code, if any and in the closing of the listening socket, if the process has no *checked-in* global name left

It returns:

- -1, when error;
- 0, otherwise.

There is a maximum number of sockets permitted within a process by the present Unix 4.2 implementation: this number is site dependent, less than 32 and guaranteed to be at least 20.

2.1.3 drc.connect(name)

Drcconnect connects to a named, *checked-in* remote process (figure 2-3).

Its only argument is:

- **name:** name of the process to be connected **to.**

```
...
char   copsrclaynamc[128j;
int    copsrclaysockct;
...
       strcpy(copsrclayname/fcopsrelay-w);
       strcat(copsrclayname,host);
       copsrclaysockct = drc.conncct(copsrelayname);
...
```

Figure 2-3: Example of use of drc.connect (from COPS).

Its execution will result in the fetching of the host and address of the remote process from the DRCNAMESERVER, in the creation of a new stream socket, in the connection of this new socket to the remote process and in the inclusion of this new socket in the active socket list

It returns:

- -1, when error;
- the new, connected socket, otherwise.

The returning connected socket should be stored to be used in subsequent *drcjends* (but it is not necessary on *drcjecvs)*. It can also be used in any other Unix 4.2 call that manipulates connected stream sockets (like *send, recv,* etc).

There is a maximum number of sockets permitted within a process by the present Unix 4.2 implementation: this number is site dependent, less than 32 and guaranteed to be at least 20.

2.1.4 drc.close(socket)

Drc.close closes connections (figure 2-4).

```
...
int    copsrelaysocket;
...
       drc_close(copsrelaysockeO;
```

Figure 2-4: Example of use of drcclose (from COPS).

Its only argument is:

- **socket**: connected socket corresponding to the connection to be terminated.

Its execution will result in the closing of the connected socket and in its removal from the active socket list.

It returns:

- 0, always.

A *drc_close* in a non-active socket will result in no action (therefore a 0 to 31 *drc_close* loop will result in the closing of all connections established via *drc_connect*, either locally or remotelly, with no harm done to any other socket).

It can be executed from both ends of a connection. If executed only from one end, it will eventually generate a condition on the remote side of the connection that will be synchronously treated by *drc_recv*, which will automatically close the socket associated with the remote side of the connection. If *drc_recv* is not used, the user must provide the closing in both sides of the connection.

There is a maximum number of sockets permitted within a process by the present Unix 4.2 implementation: this number is site dependent, less than 32 and guaranteed to be at least 20.

2.1.5 drc_send(socket,string_message)

Drc_send sends string_messages (null terminated arrays of characters) of arbitrary length to connected stream sockets (figure 2-5).

```
...
char   * command;
int    copsrelaysocket;
...
     if (drc_send(copsrelaysocket,command) = = -1)
          return(errno);
...
```

Figure 2-5: Example of use of drc_send (from COPS).

Its arguments are:

- **socket**: connected socket corresponding to the receiving remote process; **string_message**: a pointer to a null terminated array of characters (string) that will be sent to the remote process.

Its execution will result in sending the string, from its beginning to the first null encountered, including the null, to the remote process connected to socket

It returns:

- -1, when error;
- number of characters sent, otherwise.

It can be executed from both ends of a connection.

2.1.6 drc.recv(string_message, string_message.length)

Drc.recv receives stringjnessages (null terminated arrays of characters) of arbitrary length from connected stream sockets (figure 2-6).

```
...
char    messagefMESSAGESIZE]
...
      drc_wait_time = DRC.BLOCK;
      if (drc_recv(message,MESSAGESIZE) = = -1)
            retura(errno);
...
```

Figure 2-6: Example of use of drcjecv (from COPS).

Its arguments are:

- stringjnessage: a pointer to an area that will be filled by *drcjecv* with the complete contents of the received message, including terminating null.
- string_message_length: an integer that should contain the size of the area pointed by string message.

Its related global variables are:

- drc.waitjime: a long integer containing DRC.BLOCK, DRC.DONTBLOCK or the timeout for waiting for a message, in seconds;
- drcjecv.socket: an integer returning the socket from where the message arrived

Its execution will result in the blocking until there is a message to receive or a connection to accept, in the accepting of pending connects, if any, with correspondent update of the active socket list, until all pending

connects are accepted, and in the receiving of pending messages, if any, beginning with the one associated with the lowest numbered active socket, if stringjnessagejength is greater or equal to the message size, including terminating null.

It returns:

- -1, when error;
- 0, when timed out;
- message length, otherwise.

2.2 Error codes

A call returning -1 has an associated error code, present in the global variable *errno*. An error code smaller than 1023 is a Unix 4.2 BSD error code[4] generated by an internal call to Unix routines and transferred back from DRCIPC, to be handled outside the package.

Error codes greater than 1023 are DRCIPC error codes:

- **1024:** DRC.NONSHOST - nameserver host not **found;**
- **1025:** DRC.NOTFOUND - entry for name not found in by the nameserver;
- **1026:** DRQNAMEINUSE - name already in use;
- **1028:** DRQMSGISLARGER - message is larger than the *string-message* area provided in the call.

The global variable *errno* is shared with other Unix 4.2 system calls and is defined *extern* in the *drcipc.h* file.

3. Features and limitations of the present version

The present version of DRCIPC is the 2.10 version.

Its components are:

- **DRCIPC object code package, to be linked to the programs accessing the message system (drcipc.o);**
- **DRCIPC include file, to be included by code utilizing the package (drcipc.h);**
- **DRCIPC name server, a stand alone program, residing in a particular machine (gould.drc.cmu.edu), that provides the naming and identification of processes (drcnameserv). A log file of all transactions is generated during its execution (drcnameserv.log).**

The DRCIPC routines execute under Unix 4.2 BSD and UNIX 4.2/4.1 on VAXen and SUNs, and under the UTX/32 operating system on GOULD 9080 machines. This package has been tested on those systems and is expected to be transportable to other Unix 4.2 BSD systems or computers running compatible versions of this operating system.

The present implementation has the following known limitations:

- **there is a maximum number of sockets, guaranteed to be at least 20, that a process is allowed to have open simultaneously (this is a socket mechanism limitation and a Unix generation parameter);**
- **the timeout mechanism in the *drcjecv* is reset when a connection is accepted; therefore, it is not guaranteed that the timeout will always timeout in the expected amount of time: it may take longer if a connection is accepted during the no message period.**
- **the present implementation of DRCIPC has a temporary (and potentially dangerous) modification to allow for some peculiarities of the CS Unix 4.2 environment: a *drc-checkin* with a name already in use will cause the redefinition of the entry for that name and the generation of a warning message in the drcnameserver log file.**

References

1. Leão, Luiz V., "COPS: A Distributed Production System", Master's thesis, Carnegie-Mellon University, April 1986.

2. Talukdar, S. R, E. Cardozo and L. V. Leão, "TOAST: The Power System Operators' Assistant", *IEEE Computer*, Vol. 19, No. 7, July 1986, pp. 53-60.

3. Leão, Luiz V., "The COPS User's Manual", Tech. report EDRC-??-??-86, Engineering Design Research Center, Carnegie-Mellon University, August 1986.

4. Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, *UNIX Programmer's Manual 4.2 Berkeley Software Distribution*, Berkeley, California, 1983.

5. Leffler, Samuel J., Robert S. Fabry and William N. Joy, "A 4.2bsd Interprocess Communications Primer", Tech. report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 1983.