Kalpa Publications in Computing

Volume 17, 2023, Pages 69–80

Proceedings of 2023 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Race-Condition-Robust Hardware-Software Equivalence in *n*x

Lawrence J. Dickson[1]

Space Sciences Corporation
Lemitar, New Mexico
larry@spacesciencescorp.com

Abstract

Classic CSP communication channels behave equivalently whether connecting software processes or hardware devices. The static OCCAM language and the Transputer processor, both based on finite CSP, exhibit this property, which results in Hardware-Software Equivalence between implementation of the same programs (including binaries). Programs written in OCCAM and run on the Transputer can be proven correct and their behavior characterized down to cycle count.

This Fringe presentation shows a technique for extending this HSE to *n*x processes and devices in certain applications. Wherever certain capabilities of the ssh suite of programs are found, it works to allow communication using sockets as the communicators known to the programs. An ssh tunnel between sockets on separate devices allows this to work in the hardware case, without any change in the programs including their binaries. Among the *n*x varieties we have shown to work are Linux, Mac BSD, and Termux over Android.

The technique passes short messages via a server program. An investigation of race conditions, using the property of select() that it always finds a winner among file descriptor communication races, proves that the communication is robust across all possible timing differences between communicating client pairs.

The current code works if two of the clients are communicating at a time. It requires further development before handling three independent racing systems.

1 Introduction

Our system required at least two Raspberry Pis and possibly an Android tablet, messaging promptly and reliably to set up multi-megabyte file copies for a sequence of operations including camera uptake, image stitching, and critique based on machine learning. Since everything was *n*x, mostly Linux, sockets were convenient if supplemented by ssh tunneling between hardware.

The analogy with Transputer soft and hard channels inspired a design, related to NMEA-0183 (GPS text) message passing, in which all the client communications could be handled by single-line calls in bash scripts. The modest inefficiency of renegotiating the connections each time is insignificant in human time, and the programming is of interest because of two features:

(1) The server and clients, with the sockets, exhibit hardware-software equivalence (HSE) using unchanged binary code and same-name sockets for either. The hardware link is achieved by an ssh -nNT call between two sockets of the same name but on different hardware.

(2) Analysis of the code flow, based on the fact that *n*x select() always picks a winner from racing inputs on file descriptors, proves that the message flow is robust against timing differences between the communication calls made by the clients passing a message.

The code is contained in an Appendix to this Fringe.

2 Setup

Before running the trigonal set of programs with hardware channels, ssh tunneling must be set up, preferably without passwords needed every time.

What is shown below assumes the Rosebrock (or R) Raspberry Pi is at 192.168.10.3, the Camera Raspberry Pi (or C) is at 192.168.10.4, and the Tablet running Termux is at 192.168.10.5.

They should form a small local network, as implemented by a switch and a little yellow router in our lab. Your IP addresses may come out different.

Setup once only:

This is done on the Rosebrock RPi once only. It creates files that work to allow scp to function without a password between C and R.

```
ssh-keygen -t ecdsa -b 521
ssh-copy-id pi@192.168.10.4
```

Take the defaults (including no password) for ssh-keygen. This causes public and private keys to be created and matched on the camera RPi. When scp is attempted on the Rosebrock RPi, including in a script, it finds these and is able to proceed without asking for a password.

A test ssh run would be run on C, not R:

```
ssh -i id_ecdsa pi@192.168.10.3
```

And similarly on R for T if T is implemented in hardware, run the following on R:

```
ssh-keygen -t rsa -b 2048 -f id_rsa
ssh-copy-id pi@192.168.10.5
```

A test ssh run in this case would be run on T (Termux window), not R:

```
ssh -i id_rsa pi@192.168.10.3
```

Since both ssh-keygen runs are on R, I used a different key flavor so the defaults would not collide.

3 Run

One terminal must be dedicated per system to a background or server program. Suppose we are running C in its own system but R and T on the same system. Then we start on terminal R1 with the server

```
./trigonal
```

and we create the ssh tunnel by running one line on C1

```
ssh -nNT -L \
/tmp/9Lcamera.socket:/tmp/9Lcamera.socket \
pi@192.168.10.3
```

In both cases, the program will not terminate (informative stuff will come up on the R1 screen). THIS IS CORRECT BEHAVIOR. So leave those terminals and do stuff in the other terminals.

If the `ssh -nNT` command does not work, it may require `-i id_ecdsa` (surrounded by spaces) to be inserted before the `-nNT`.

In C2 run:

```
./trigcsr '!CRABCD'
```

And in R2 run:

```
./trigrsr
```

The message will pass through and appear in R2. It does not matter which of the clients is triggered first. Thus, you can generate an acknowledge as follows:

In C2 run:

```
./trigcsr
```

And in R2 run:

```
./trigrsr '!RCA'
```

The two characters immediately after the exclamation point are important, as the program reads them and interprets them as message source and target.

To shut down, hit terminal C1 with a Ctrl-C to kill the `ssh -nNT` and follow that in terminal C1 with

```
rm /tmp/9Lcamera.socket
```

Watch terminal R1. It will probably say 2 connections left. Finish the others from terminal R2 with

```
./trigtsr DOWN
./trigrsr DOWN
```

If it didn't say 2 connections left, you will also need

```
./trigcsr DOWN
```

and ./trigonal will quit.

4 Restrictions and scripting

The techniques developed here make use of the behavior of *n*x sockets, which establish a bidirectional communication (called a *socket link*) between *n*x processes. Here we solely use "Unix sockets" (soft sockets) and let the ssh tunnel take care of any hardware connection. Thus, the processes may be on the same hardware (one socket) or on different hardware (two sockets connected by one ssh tunnel). The behavior is equivalent.

Each socket link is driven solely by the client programs `trig?sr`, called on each process in a single sequential script or equivalent. The calls are grouped in *timing anchors*, two calls each side per anchor, with no interleaving, according to the scheme shown below.

Table 1: A timing anchor from C to R

process C script	process R script
`trigcsr '!CRthename'`	`trigrsr`
	`action`
`trigcsr`	`trigrsr '!RCA'`

Here `thename` is some ASCII information useful, for instance, in building filenames, and `action` is some appropriate shared action, such as an scp pull of a few megabytes of data from C to R. The calls to `trig?sr` without a parameter are *client reads,* while those with a parameter are *client writes.* Because of the sequential restriction, the horizontal pairs in the table always communicate with each other.

There is no timing restriction, except for sequentiality, so in each horizontal pair either communicator may come first. Every data packet is sent using a socket from the client write to the trigonal server, then from the trigonal server to the client read. The trigonal server has a one-message-deep buffer. Hence if the client write comes first, it closes quickly and leaves the data pending with the server, ahd when its mated client read arrives the buffer is then passed on. If the client read comes first, its communication channel is kept open until data arrives from its mated client write, and then the communication is quickly completed and both read and write are quickly closed.

The consequence is that the client read closes after the message is passed (at essentially the same time from a human point of view), and if it is known that the client write happened after the mated client read, then the client write also closes after the message is passed. In the timing anchor, given that the action takes more than a few milliseconds, the second client read (the one on the C side) will always start before the action is completed, and hence before the second client write (the one on the R side). Thus the timing anchor will finish at essentially the same time on both sides, soon after the action is done. But it is possible for both the write and read on the C side to begin long before the action.

It is thus deceptive in the table that `trigcsr` appears below the level of `action`, as it will normally start (but not finish) before or at the same time as `action` starts.

5 Robust communication

As long as the trigonal server code is able to determine the correct data to write in the pending case, which always works (see next section) if there is only a single link (e.g. C $\leftrightarrow$ R) supported by the sequential timing anchors, the select()-based server code is completely robust across all timing differences between communicating client pairs.

This is not obvious, because client reads generate one kind of server select() hit, the accept, while client writes generate two kinds, both accept and data read. It would seem possible that just the wrong timing could lose data. But not so, because the race is between source client data and target client accept. Source client accept does not enter into it.

Take for instance the top line of the table, C client write data vs R client read accept. The select() must choose one or another as the winner.

If C client write data is the winner, then R client read accept has not happened, hence current_Rosebrock==connection_Rosebrock, not data_Rosebrock. This results in setting pending_Rosebrock to 1 and putting the data into todo_Rosebrock. The later R client read accept

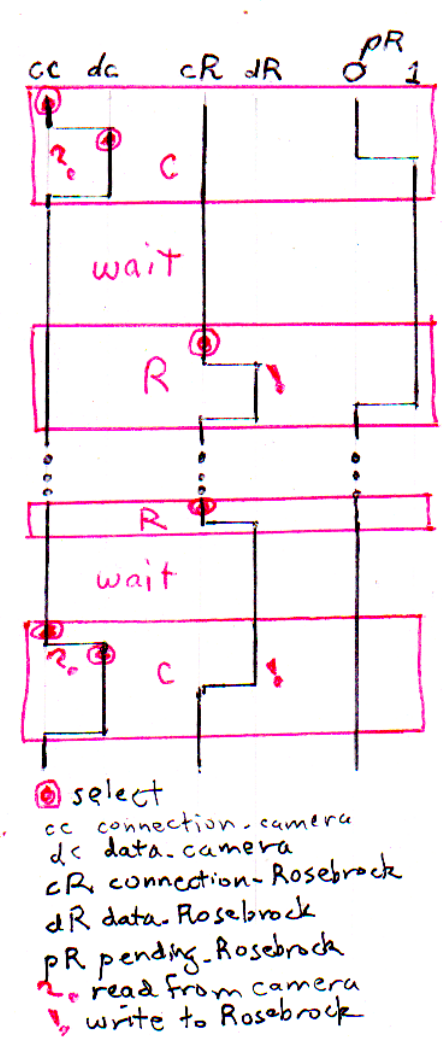

Figure 1: Two races

then outputs this data.

If R client read accept is the winner, then C client write data has not happened, so pending_Rosebrock is 0 and the code causes current_Rosebrock to equal data_Rosebrock at select loopback. The later C client write data then outputs its data on this channel.

6 Multiple timing anchor sequence troubles

This code needs further work if it is to function with a two-step data flow, for example C to R and R to T. It may not suffice to insist that R scripting make the CR-anchors and the RT-anchors be in sequence without overlapping. This is because C, R, and T all may have independent timing, so that there is nothing preventing both C and T from writing pending

data to R. An R accept select() win would then not know which of the two sequences of anchors it was serving.

I have no time to rework the code before the conference, though I suspect the problem is soluble by having the `trigrsr` call (without parameter) replaced with something like `trigrsr '?C'` giving the data source, and permitting two todo buffers to accept from different sources.

7 Appendix

The Appendix consists of the code, which is in C. Two of the clients are omitted, because they are easily constructed from the third by changing a few words (see the diffs immediately below). This leaves one client, trigcsr.c, and the server, trigonal.c.

The messages are an imitation of NMEA-0183 (GPS text) format. The shared header file uscocrlf.h determines their size and three names of sockets corresponding to the three clients. In the system where trigonal runs, these three sockets are initialized by the trigonal run. In cases where one or more clients run on separate systems, the ssh -nNT call creates a socket of the same name on the client system, and connects it with the server system socket via an ssh tunnel.

7.1 uscocrlf.h

```
#define SOCKET_NAME "/tmp/9L226.socket"
#define SOCKET_NAMC "/tmp/9Lcamera.socket"
#define SOCKET_NAMT "/tmp/9Ltablet.socket"
#define BUFFER_SIZE 40
```

7.2 trigrsr.c diff

```
diff trigcsr.c trigrsr.c
22c22
<     /*{{{  Create and connect local socket: CAMERA.*/
---
>     /*{{{  Create and connect local socket: ROSEBROCK.*/
39c39
<     strncpy(addr.sun_path, SOCKET_NAMC, sizeof(addr.sun_path) - 1);
---
>     strncpy(addr.sun_path, SOCKET_NAME, sizeof(addr.sun_path) - 1);
```

7.3 trigtsr.c diff

```
diff trigcsr.c trigtsr.c
22c22
<     /*{{{  Create and connect local socket: CAMERA.*/
---
>     /*{{{  Create and connect local socket: TABLET.*/
39c39
<     strncpy(addr.sun_path, SOCKET_NAMC, sizeof(addr.sun_path) - 1);
---
>     strncpy(addr.sun_path, SOCKET_NAMT, sizeof(addr.sun_path) - 1);
```

7.4 trigcsr.c

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include "uscocrlf.h"
/* Client:
 * Connects immediately
```

```
 * If argc>1 it sends argv[1] contents to server
 * Otherwise it awaits buffer from server and outputs it to stdout
 * Then closes the connection
 */
int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int ret;
    int data_socket;
    char buffer[BUFFER_SIZE];
    /*{{{  Create and connect local socket: CAMERA.*/
    /* Create local socket. */
    data_socket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (data_socket == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "socket-returned-%d\n", data_socket);
    }
    /*
     * For portability clear the whole structure, since some
     * implementations have additional (nonstandard) fields in
     * the structure.
     */
    memset(&addr, 0, sizeof(addr));
    /* Connect socket to socket address. */
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_NAMC, sizeof(addr.sun_path) - 1);
    ret = connect(data_socket, (const struct sockaddr *) &addr,
                  sizeof(addr));
    if (ret == -1) {
        fprintf(stderr, "The-server-is-down.\n");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "connect-returned-%d\n", ret);
    }
    /*}}}  */
    /*{{{  Send argv[1] or receive and output to stdout*/
    /* Send if argc>1. */
    if (argc > 1) {
        ret = write(data_socket, argv[1], strlen(argv[1]) + 1);
        if (ret == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        } else {
            fprintf(stderr, "write-returned-%d,-content-%s\n", ret, argv[1]);
        }
        sleep(1);
    } else {
        /* Receive result. */
        ret = read(data_socket, buffer, sizeof(buffer));
        if (ret == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        } else {
            /* Ensure buffer is 0-terminated. */
            buffer[sizeof(buffer) - 1] = 0;
            fprintf(stderr,
              "read-returned-%d,-buffer-size-with-trailing-null-%d,-content-%s\n",
              ret, strlen(buffer)+1, buffer);
            printf(buffer);
        }
    }
    /*}}}  */
    /* Close socket. */
    close(data_socket);
    exit(EXIT_SUCCESS);
}
```

7.5 trigonal.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include "uscocrlf.h"
int
main(int argc, char *argv[])
{
    fd_set rfds;
    struct sockaddr_un name_camera, name_Rosebrock, name_tablet;
    int ret;
    int valid_camera, valid_Rosebrock, valid_tablet;
    int connection_camera, connection_Rosebrock, connection_tablet;
    int data_camera, data_Rosebrock, data_tablet, fd_size;
    int current_camera, current_Rosebrock, current_tablet;
    int pending_camera, pending_Rosebrock, pending_tablet;
```

```
int nconnecs, result_camera, result_Rosebrock, result_tablet, retval;
char buffer[BUFFER_SIZE];
char todo_camera[BUFFER_SIZE];
char todo_Rosebrock[BUFFER_SIZE];
char todo_tablet[BUFFER_SIZE];
/*{{{  Create connection socket, bind, listen: CAMERA*/
/* Create local socket. */
connection_camera = socket(AF_UNIX, SOCK_STREAM, 0);
if (connection_camera == -1) {
    perror("camera-socket");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "camera-socket-returned-%d\n", connection_camera);
}
/*
 * For portability clear the whole structure, since some
 * implementations have additional (nonstandard) fields in
 * the structure.
 */
memset(&name_camera, 0, sizeof(name_camera));
/* Bind socket to socket name_camera. */
name_camera.sun_family = AF_UNIX;
strncpy(name_camera.sun_path, SOCKET_NAMC, sizeof(name_camera.sun_path) - 1);
ret = bind(connection_camera, (const struct sockaddr *) &name_camera,
           sizeof(name_camera));
if (ret == -1) {
    perror("camera-bind");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "camera-bind-returned-%d\n", ret);
}
/*
 * Prepare for accepting connections. The backlog size is set
 * to 20. So while one request is being processed other requests
 * can be waiting.
 */
ret = listen(connection_camera, 20);
if (ret == -1) {
    perror("camera-listen");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "camera-listen-returned-%d\n", ret);
}
/*}}}  */
/*{{{  Create connection socket, bind, listen: ROSEBROCK*/
/* Create local socket. */
connection_Rosebrock = socket(AF_UNIX, SOCK_STREAM, 0);
if (connection_Rosebrock == -1) {
    perror("Rosebrock-socket");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Rosebrock-socket-returned-%d\n", connection_Rosebrock);
}
/*
 * For portability clear the whole structure, since some
 * implementations have additional (nonstandard) fields in
 * the structure.
 */
memset(&name_Rosebrock, 0, sizeof(name_Rosebrock));
/* Bind socket to socket name_Rosebrock. */
name_Rosebrock.sun_family = AF_UNIX;
strncpy(name_Rosebrock.sun_path, SOCKET_NAME, sizeof(name_Rosebrock.sun_path) - 1);
ret = bind(connection_Rosebrock, (const struct sockaddr *) &name_Rosebrock,
           sizeof(name_Rosebrock));
if (ret == -1) {
    perror("Rosebrock-bind");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Rosebrock-bind-returned-%d\n", ret);
}
/*
 * Prepare for accepting connections. The backlog size is set
 * to 20. So while one request is being processed other requests
 * can be waiting.
 */
ret = listen(connection_Rosebrock, 20);
if (ret == -1) {
    perror("Rosebrock-listen");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Rosebrock-listen-returned-%d\n", ret);
}
/*}}}  */
/*{{{  Create connection socket, bind, listen: TABLET*/
/* Create local socket. */
connection_tablet = socket(AF_UNIX, SOCK_STREAM, 0);
if (connection_tablet == -1) {
    perror("tablet-socket");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "tablet-socket-returned-%d\n", connection_tablet);
}
/*
 * For portability clear the whole structure, since some
 * implementations have additional (nonstandard) fields in
 * the structure.
```

```
 */
memset(&name_tablet, 0, sizeof(name_tablet));
/* Bind socket to socket name_tablet. */
name_tablet.sun_family = AF_UNIX;
strncpy(name_tablet.sun_path, SOCKET_NAMT, sizeof(name_tablet.sun_path) - 1);
ret = bind(connection_tablet, (const struct sockaddr *) &name_tablet,
           sizeof(name_tablet));
if (ret == -1) {
    perror("tablet-bind");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "tablet-bind-returned-%d\n", ret);
}
/*
 * Prepare for accepting connections. The backlog size is set
 * to 20. So while one request is being processed other requests
 * can be waiting.
 */
ret = listen(connection_tablet, 20);
if (ret == -1) {
    perror("tablet-listen");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "tablet-listen-returned-%d\n", ret);
}
/*}}}  */
current_camera = connection_camera;
current_Rosebrock = connection_Rosebrock;
current_tablet = connection_tablet;
pending_camera = pending_Rosebrock = pending_tablet = 0;
valid_camera = 1;
valid_Rosebrock = 1;
valid_tablet = 1;
fd_size = current_camera + 1;
if (current_Rosebrock >= fd_size) fd_size = current_Rosebrock + 1;
if (current_tablet >= fd_size) fd_size = current_tablet + 1;
nconnecs = 3;
/* This is the main loop for handling connections. */
for (;;) {
    FD_ZERO(&rfds);
    if (valid_camera) FD_SET(current_camera, &rfds);
    if (valid_Rosebrock) FD_SET(current_Rosebrock, &rfds);
    if (valid_tablet) FD_SET(current_tablet, &rfds);
    retval = select(fd_size, &rfds, NULL, NULL, NULL);
    if (retval == -1) {
        perror("select()");
        exit(EXIT_FAILURE);
    }
    /*{{{  camera.c*/
    if (valid_camera && FD_ISSET(current_camera, &rfds)) {
        int down_flag = 0; /* if hit, close connection and reduce nconnecs */
        int detect_end = 0;
        /*{{{  all the stuff as yet unassigned*/
        if (current_camera == connection_camera) {
            /*{{{  Wait for incoming connection on connection socket.*/
            /* Wait for incoming connection. */
            data_camera = accept(connection_camera, NULL, NULL);
            if (data_camera == -1) {
                perror("accept");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "accept-returned-%d\n", data_camera);
                current_camera = data_camera;
                if (data_camera >= fd_size) fd_size = data_camera + 1;
                if (pending_camera) {
                    /*{{{  Send result.*/
                    ret = write(data_camera, todo_camera, strlen(todo_camera)+1);
                    if (ret == -1) {
                        perror("pending-camera-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "pending-camera-write-returned-%d\n", ret);
                    }
                    pending_camera = 0;
                    detect_end = 1;
                    /*}}}  */
                }
            }
            /*}}}  */
        } else { /* data_camera */
            /*{{{  Wait for next data packet, until DOWN.*/
            /* Wait for next data packet. */
            ret = read(data_camera, buffer, sizeof(buffer));
            if (ret == -1) {
                perror("camera-read");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "camera-read-returned-%d", ret);
                if ((ret>=0)&&(ret<BUFFER_SIZE)) {
                    buffer[ret] = '\0';
                    fprintf(stderr, "camera-<<<%s>>>\n", buffer);
                } else {
                    buffer[(BUFFER_SIZE)-1] = '\0';
                    fprintf(stderr, "truncated-camera-<<<%s>>>\n", buffer);
                }
            }
```

```
            /* Ensure buffer is 0-terminated. */
            buffer[sizeof(buffer) - 1] = 0;
            detect_end = 1;
            /*}}}  */
            /* Handle commands. */
            if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                down_flag = 1;
            } else if (current_Rosebrock == data_Rosebrock) {
                /*{{{  Send result.*/
                ret = write(data_Rosebrock, buffer, strlen(buffer)+1);
                if (ret == -1) {
                    perror("camera-to-Rosebrock-write");
                    exit(EXIT_FAILURE);
                } else {
                    fprintf(stderr, "camera-to-Rosebrock-write-returned-%d\n", ret);
                }
                /* Close socket. */
                close(data_Rosebrock);
                current_Rosebrock = connection_Rosebrock;
                /*}}}  */
            } else {
                strcpy(todo_Rosebrock, buffer);
                fprintf(stderr, "camera-to-Rosebrock-pending-content-%s\n",
                  todo_Rosebrock);
                pending_Rosebrock = 1;
            }
        }
        if (detect_end) {
            /* Close socket. */
            close(data_camera);
            current_camera = connection_camera;
            if (down_flag) {
                /* Quit on DOWN command. */
                close(connection_camera);
                /* Unlink the socket. */
                unlink(SOCKET_NAMC);
                valid_camera = 0;
                nconnecs--;
                fprintf(stderr, "camera-closed,-nconnecs-%d\n", nconnecs);
                if (nconnecs<1) break;
            }
        }
        /*}}}  */
    }
    /*}}}  */
    /*{{{  Rosebrok.c*/
    if (valid_Rosebrock && FD_ISSET(current_Rosebrock, &rfds)) {
        int down_flag = 0; /* if hit, close connection and reduce nconnecs */
        int detect_end = 0;
        /*{{{  all the stuff as yet unassigned*/
        if (current_Rosebrock == connection_Rosebrock) {
            /*{{{  Wait for incoming connection on connection socket.*/
            /* Wait for incoming connection. */
            data_Rosebrock = accept(connection_Rosebrock, NULL, NULL);
            if (data_Rosebrock == -1) {
                perror("accept");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "accept-returned-%d\n", data_Rosebrock);
                current_Rosebrock = data_Rosebrock;
                if (data_Rosebrock >= fd_size) fd_size = data_Rosebrock + 1;
                if (pending_Rosebrock) {
                    /*{{{  Send result.*/
                    ret = write(data_Rosebrock, todo_Rosebrock, strlen(todo_Rosebrock)+1);
                    if (ret == -1) {
                        perror("pending-Rosebrock-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "pending-Rosebrock-write-returned-%d\n", ret);
                    }
                    pending_Rosebrock = 0;
                    detect_end = 1;
                    /*}}}  */
                }
            }
            /*}}}  */
        } else { /* data_Rosebrock */
            /*{{{  Wait for next data packet, until DOWN.*/
            /* Wait for next data packet. */
            ret = read(data_Rosebrock, buffer, sizeof(buffer));
            if (ret == -1) {
                perror("Rosebrock-read");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "Rosebrock-read-returned-%d", ret);
                if ((ret>=0)&&(ret<BUFFER_SIZE)) {
                    buffer[ret] = '\0';
                    fprintf(stderr, "Rosebrock-<<<%s>>>\n", buffer);
                } else {
                    buffer[(BUFFER_SIZE)-1] = '\0';
                    fprintf(stderr, "truncated-Rosebrock-<<<%s>>>\n", buffer);
                }
            }
            /* Ensure buffer is 0-terminated. */
            buffer[sizeof(buffer) - 1] = 0;
            detect_end = 1;
```

```
            /*}}}  */
            /* Handle commands. */
            if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                down_flag = 1;
            } else if (buffer[2] == 'C') {
                if (current_camera == data_camera) {
                    /*{{{  Send result.*/
                    ret = write(data_camera, buffer, strlen(buffer)+1);
                    if (ret == -1) {
                        perror("Rosebrock-to-camera-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "Rosebrock-to-camera-write-returned-%d\n", ret);
                    }
                    /* Close socket. */
                    close(data_camera);
                    current_camera = connection_camera;
                    /*}}}  */
                } else {
                    strcpy(todo_camera, buffer);
                    fprintf(stderr, "Rosebrock-to-camera-pending-content-%s\n",
                        todo_camera);
                    pending_camera = 1;
                }
            } else if (buffer[2] == 'T') {
                if (current_tablet == data_tablet) {
                    /*{{{  Send result.*/
                    ret = write(data_tablet, buffer, strlen(buffer)+1);
                    if (ret == -1) {
                        perror("Rosebrock-to-tablet-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "Rosebrock-to-tablet-write-returned-%d\n", ret);
                    }
                    /* Close socket. */
                    close(data_tablet);
                    current_tablet = connection_tablet;
                    /*}}}  */
                } else {
                    strcpy(todo_tablet, buffer);
                    fprintf(stderr, "Rosebrock-to-tablet-pending-content-%s\n",
                      todo_tablet);
                    pending_tablet = 1;
                }
            }
        }
        if (detect_end) {
            /* Close socket. */
            close(data_Rosebrock);
            current_Rosebrock = connection_Rosebrock;
            if (down_flag) {
                /* Quit on DOWN command. */
                close(connection_Rosebrock);
                /* Unlink the socket. */
                unlink(SOCKET_NAME);
                valid_Rosebrock = 0;
                nconnecs--;
                fprintf(stderr, "Rosebrock-closed,-nconnecs-%d\n", nconnecs);
                if (nconnecs<1) break;
            }
        }
        /*}}}  */
    }
    /*}}}  */
    /*{{{  tablet.c*/
    if (valid_tablet && FD_ISSET(current_tablet, &rfds)) {
        int down_flag = 0; /* if hit, close connection and reduce nconnecs */
        int detect_end = 0;
        /*{{{  all the stuff as yet unassigned*/
        if (current_tablet == connection_tablet) {
            /*{{{  Wait for incoming connection on connection socket.*/
            /* Wait for incoming connection. */
            data_tablet = accept(connection_tablet, NULL, NULL);
            if (data_tablet == -1) {
                perror("accept");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "accept-returned-%d\n", data_tablet);
                current_tablet = data_tablet;
                if (data_tablet >= fd_size) fd_size = data_tablet + 1;
                if (pending_tablet) {
                    /*{{{  Send result.*/
                    ret = write(data_tablet, todo_tablet, strlen(todo_tablet)+1);
                    if (ret == -1) {
                        perror("pending-tablet-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "pending-tablet-write-returned-%d\n", ret);
                    }
                    pending_tablet = 0;
                    detect_end = 1;
                    /*}}}  */
                }
            }
            /*}}}  */
        } else { /* data_tablet */
```

```
                /*{{{  Wait for next data packet, until DOWN.*/
                /* Wait for next data packet. */
                ret = read(data_tablet, buffer, sizeof(buffer));
                if (ret == -1) {
                    perror("tablet-read");
                    exit(EXIT_FAILURE);
                } else {
                    fprintf(stderr, "tablet-read-returned-%d", ret);
                    if ((ret>=0)&&(ret<BUFFER_SIZE)) {
                        buffer[ret] = '\0';
                        fprintf(stderr, "tablet-<<<%s>>>\n", buffer);
                    } else {
                        buffer[(BUFFER_SIZE)-1] = '\0';
                        fprintf(stderr, "truncated-tablet-<<<%s>>>\n", buffer);
                    }
                }
                /* Ensure buffer is 0-terminated. */
                buffer[sizeof(buffer) - 1] = 0;
                detect_end = 1;
                /*}}}  */
                /* Handle commands. */
                if (!strncmp(buffer, "DOWN", strlen(buffer))) {
                    down_flag = 1;
                } else if (current_Rosebrock == data_Rosebrock) {
                    /*{{{  Send result.*/
                    ret = write(data_Rosebrock, buffer, strlen(buffer)+1);
                    if (ret == -1) {
                        perror("tablet-to-Rosebrock-write");
                        exit(EXIT_FAILURE);
                    } else {
                        fprintf(stderr, "tablet-to-Rosebrock-write-returned-%d\n", ret);
                    }
                    /* Close socket. */
                    close(data_Rosebrock);
                    current_Rosebrock = connection_Rosebrock;
                    /*}}}  */
                } else {
                    strcpy(todo_Rosebrock, buffer);
                    fprintf(stderr, "tablet-to-Rosebrock-pending-content-%s\n",
                      todo_Rosebrock);
                    pending_Rosebrock = 1;
                }
            }
            if (detect_end) {
                /* Close socket. */
                close(data_tablet);
                current_tablet = connection_tablet;
                if (down_flag) {
                    /* Quit on DOWN command. */
                    close(connection_tablet);
                    /* Unlink the socket. */
                    unlink(SOCKET_NAMT);
                    valid_tablet = 0;
                    nconnecs--;
                    fprintf(stderr, "tablet-closed,-nconnecs-%d\n", nconnecs);
                    if (nconnecs<1) break;
                }
            }
            /*}}}  */
        }
        /*}}}  */
    }
    exit(EXIT_SUCCESS);
}
```