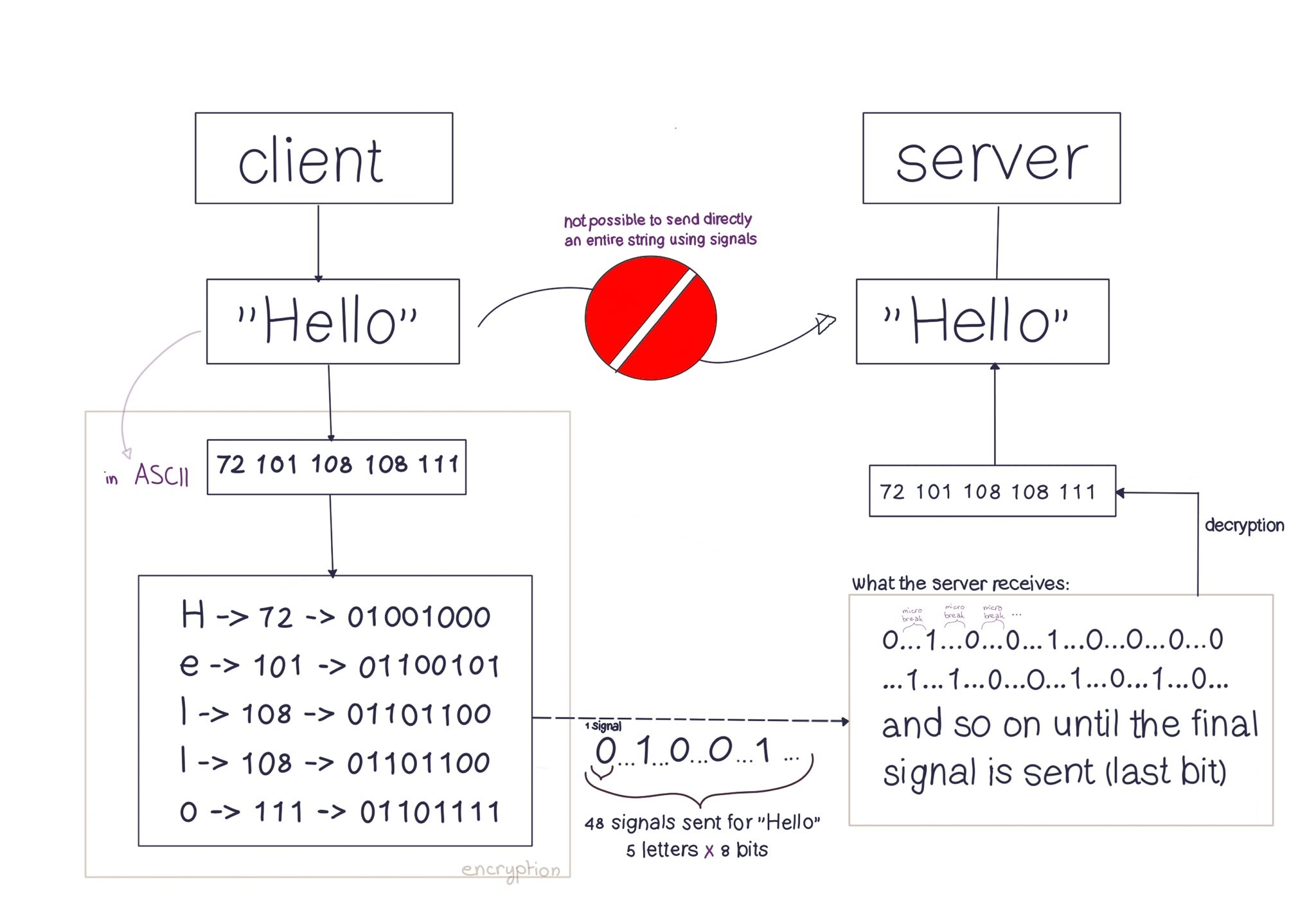
Minitalk

Communication program in the form of a client and server. The goal of the Minitalk project is to develop a simple program that allows processes (= programs running on a computer) to communicate with each other using a communication protocol called "minitalk".

Guidelines:

1. The server must be started first. After its launch, it has to print its PID.
2. The client takes two parameters: 1) The server PID 2) The string to send.
3. The client must send the string passed as a parameter to the server. Once the string has been received, the server must print it.
4. The server has to display the string quickly. Quickly means that if you think it takes too long, then it is probably too long.
5. Your server should be able to receive strings from several clients in a row without needing to restart.
6. The communication between your client and your server has to be done only using UNIX signals.
7. You can only use these two signals: SIGUSR1 and SIGUSR2.



Processes and signals are the most important terms to know in this project. Remember: a signal is just telling a process to do a certain thing!

Imagine you have a program that is running on your computer. This program is Google Chrome running in the background on your computer. When you have multiple windows open on your browser, you have multiple programs running. These programs are called processes. Now, suppose that you want to stop this program. You can do this by simply closing the browser window. We want to close the windows, that will be the signal that we want to send to the processes. For this, you might use the "kill" command in a terminal window to send a signal to the processes. This signal tells the process to terminate itself.

To send a signal to a certain process you need the PID of it. They will be useful, because that's how you will know which signal to send to which process.

kill <PID>

killall chrome

New Functions Used

Signal()

sighandler\_t signal(int signum, sighandler\_t handler);

The signal function in C allows a program to handle specific signals (e.g., SIGINT, SIGTERM) by assigning a custom signal handler function. When a signal is received, the handler executes, enabling the program to respond appropriately (e.g., cleanup before termination, ignoring a signal, or custom actions).

Sigemptyset()

int sigemptyset(sigset\_t \*set);

The sigemptyset function is used to initialize a signal set to the empty set, which means that it does not contain any signals. The sigemptyset function takes a pointer to a set of signals as an argument and empties this set by adding no signal to it.

Sigaddset()

int sigaddset(sigset\_t \*set, int signum);

This function allows to add a signal to a set of signals. The sigaddset function takes two arguments: a pointer to a set of signals and the number of the signal to add to the set.

Sigaction

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);

The sigaction function in C is used to specify the action to be taken when a specific signal is received by a process. It is defined in the signal.h header file.

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

void signal\_handler(int signum) {

printf("Received signal %d\n", signum);

}

int main(void) {

struct sigaction action;

action.sa\_handler = signal\_handler;

sigemptyset(&action.sa\_mask);

action.sa\_flags = 0;

sigaction(SIGINT, &action, NULL);

while (1) {

// Do some work

}

return 0;

}

| Line | Purpose |
| --- | --- |

|  |  |
| --- | --- |
| struct sigaction sa; | Declares a signal action structure |

|  |  |
| --- | --- |
| sa.sa\_flags = 0; | Uses default signal handling behavior |

|  |  |
| --- | --- |
| sa.sa\_handler = handle\_ack; | Assigns the handle\_ack function to handle SIGUSR1 |

|  |  |
| --- | --- |
| sigemptyset(&sa.sa\_mask); | Clears signal mask (does not block any signals) |

|  |  |
| --- | --- |
| sigaction(SIGUSR1, &sa, NULL); | Registers the handler for SIGUSR1 |

Kill()

int kill(pid\_t pid, int sig);

In C, the kill function is a system call that sends a signal to a process.

The pid argument specifies the process ID of the process you want to communicate with. The sig argument specifies the signal to be sent to the process.

Getpid()

pid\_t getpid(void);

In C, the getpid function returns the process ID of the current process. It is declared in the unistd.h header file.

#include <stdio.h>

#include <unistd.h>

int main(void) {

pid\_t pid;

pid = getpid();

printf("The process ID is %d\n", pid);

return 0;

}

This program will print the process ID of the current process to the console. The process ID is a unique identifier assigned to each process by the operating system. It is used to identify and track processes within the system.

Pause()

int pause(void);

pause() is a function in the C standard library that causes the calling process to sleep until a signal is received. The process remains blocked until a signal handler is executed or the signal is ignored

#include <stdio.h>

#include <unistd.h>

int main(void) {

printf("Entering pause...\n");

pause();

printf("Exiting pause.\n");

return 0;

}

sleep()

unsigned int sleep(unsigned int seconds);

sleep() is also a function in the C standard library that causes the calling process to sleep for a specified number of seconds

#include <stdio.h>

#include <unistd.h>

int main(void) {

printf("Sleeping for 3 seconds...\n");

sleep(3); // The program waits 3 seconds

printf("Done sleeping.\n");

return 0;

}

usleep()

int usleep(useconds\_t usec);

usleep() is a function in the C standard library that causes the calling process to sleep for a specified number of microseconds.

exit()

exit() is a function in the C standard library that terminates the calling process immediately. It takes an integer argument that specifies the exit status of the process. A value of 0 indicates successful termination, while non-zero values indicate an error.

#include <stdio.h>

#include <stdlib.h>

int main(void) {

printf("Exiting with status 0...\n");

exit(0);

}

Note: for handling signal acknowledgments and synchronization between the client and server. Since signal handlers can be executed asynchronously, using sig\_atomic\_t ensures that shared variables are updated safely without race conditions.