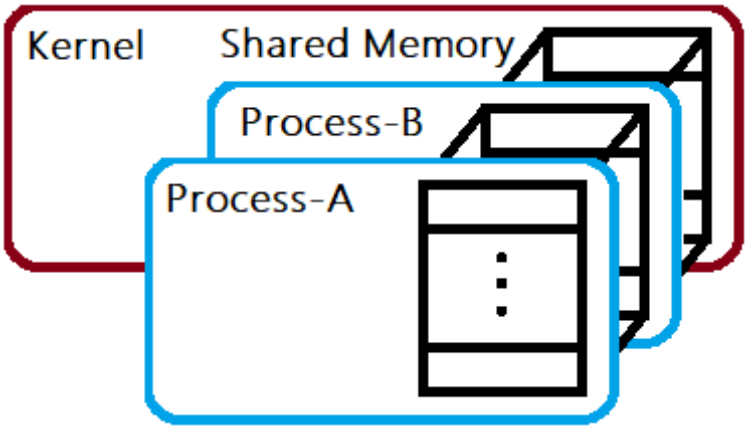
Four Ways of Inter Process Messaging

Well done! You have your first Linux program running. Now you are ready to code the second one but stop! You noticed that they have to communicate and send/receive messages. At that moment, I will tell you about four possible ways of Inter Process Communication on Linux 😎

Shared Memory

On a Linux, the easiest way of communicating to another process is via shared memory. Operating system creates a special file for you and all the processes that wants to communicate to each other only have to access that file. Yet the kernel does much more for you by mapping it into the memory of your process and you only have to use it like any other memory region.



Creating Shm

It is very simple. The first process has to create shared memory object at a predefined location with the correct permissions. The following line creates a file named `/dev/shm/ipc` with full permission.

```C++
int fd = shm_open("ipc", O_RDWR | O_CREAT, 0777);
```

The file descriptor `fd` will be non-negative on success. The second step is mapping the shared memory area to RAM so that it has an address in memory for reading and writing. Use `mmap()` for this.

```C++
mmap(0, SIZE_OF_MEMORY, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

This line maps the file into memory with read and write permissions and returns the address on success.

Messaging over Mapped Shared Memory in C++

I prepared a short demo that sends and receives a simple message made of `messageId` and `messageBody`.

shared_memory_ipc.cpp C++

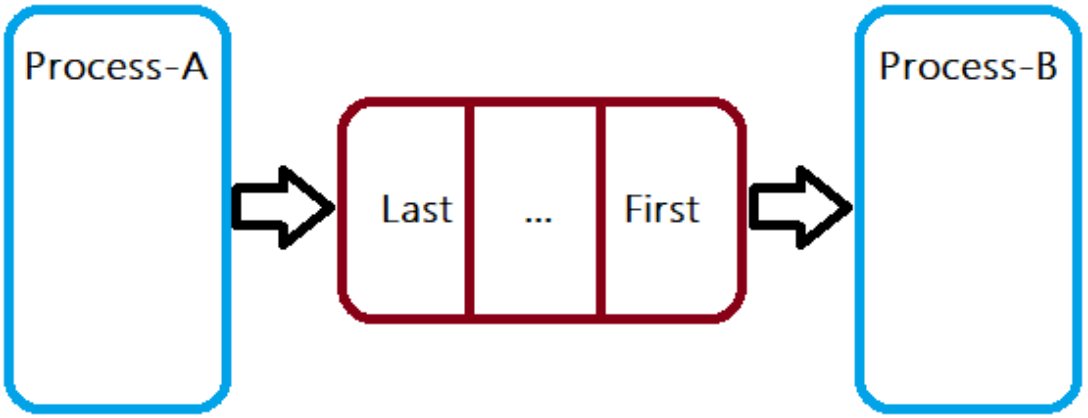
```cpp
/**
 * @file      shared_memory_ipc.cpp
 * @author    Atakan S.
 * @date      01/01/2019
 * @version   1.0
 * @brief     Example project for Linux Shared Memory IPC Messaging tutorial.
 *
 * @copyright Copyright (c) 2019 Atakan SARIOGLU ~ www.atakansarioglu.com
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a
 *  copy of this software and associated documentation files (the "Software"),
 *  to deal in the Software without restriction, including without limitation
 *  the rights to use, copy, modify, merge, publish, distribute, sublicense,
 *  and/or sell copies of the Software, and to permit persons to whom the
 *  Software is furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 *  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 *  DEALINGS IN THE SOFTWARE.
 */

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <iostream>
#include <mutex>

struct Message {
    std::mutex mutex;
    int messageId;
    int messageBody;
};

int main() {
    // Remove the old file if exists.
    shm_unlink("ipc");

    // Start 2nd process and initialize random number generator.
    if(fork()) sleep(1);
    srand(getpid());

    // Create shared memory object.
    int shm;
    if((shm = shm_open("ipc", O_RDWR | O_CREAT, 0777)) < 0) {
        std::cout << "Cannot open shm" << std::endl;
        return -1;
    }
```

```
57
58      // Create an empty message.
59      if(lseek(shm, 0, SEEK_END) < sizeof(Message)) {// Fill with zero
60          lseek(shm, 0, SEEK_SET);
61          Message dummy{};
62          if(write(shm, &dummy, sizeof(Message)) < sizeof(Message)) {
63              std::cout << "Cannot initialize shm" << std::endl;
64              return -1;
65          }
66      }
67
68      // Map the file to memory and obtain a pointer to that region.
69      Message * message;
70      if((message = (Message *)mmap(0, sizeof(Message), PROT_READ | PROT_WRITE, MAP_
71          std::cout << "Cannot create mapping" << std::endl;
72          return -1;
73      }
74
75      // Receive and send messages.
76      while(true) {
77          // Use mutex to send and receive the message inact.
78          std::cout << "Waiting for lock... " << std::endl;
79          while(!message->mutex.try_lock());
80          std::cout << "Has lock... " << std::endl;
81
82          // Read from and write to the memory as usual.
83          std::cout << "Process " << getpid() << " Received message" << std::endl;
84          std::cout << "   Got messageId=" << message->messageId << std::endl;
85          std::cout << "   Got messageBody=" << message->messageBody << std::endl;
86          std::cout << "Process " << getpid() << " Sending message" << std::endl;
87          std::cout << "   Set messageId=" << (message->messageId = getpid()) << std
88          std::cout << "   Set messageBody=" << (message->messageBody = rand()) << s
89
90          // Release the lock and wait.
91          message->mutex.unlock();
92          sleep(5);
93      }
94
95      // Unmap and unlink the shared memory.
96      munmap(message, sizeof(Message));
97      close(shm);
98      shm_unlink("ipc");
99
100     // Exit.
101     return 0;
102 }
```

You can compile with `g++ shared_memory_ipc.cpp -o shared_memory_ipc -lrt -std=c++11` as noted here. In the example code you will find two processes, one is `fork()`'ed from the parent process, and both share the same memory area to read/write the message. Usage of mutex is important since there is no other mechanism to prevent concurrent access to the data.

First In First Out Pipes

A FIFO object in Linux is a special type of named pipe which provides first-in-first-out buffering which makes it ideal for inter process messaging.



Creating FIFO Messaging Object

First created by calling `mkfifo()` and later `open()` it and use it for `read()` and `write()` operations.

Messaging over Linux FIFO in C++

The example project creates two processes and the parent process sends messages while the child process receives and prints them. Note that `write()` is non-blocking as long as the FIFO has space but `read()` blocks the current thread. Thats why I create an `alarm(1)` that creates `SIGALRM` every second to interrupt ongoing or blocking system calls.

linux_fifo_ipc.cpp C++

```cpp
/**
 * @file      linux_fifo_ipc.cpp
 * @author    Atakan S.
 * @date      01/01/2019
 * @version   1.0
 * @brief     Example for Linux IPC using Named Pipe FIFO Messaging.
 *
 * @copyright Copyright (c) 2019 Atakan SARIOGLU ~ www.atakansarioglu.com
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a
 *  copy of this software and associated documentation files (the "Software"),
 *  to deal in the Software without restriction, including without limitation
 *  the rights to use, copy, modify, merge, publish, distribute, sublicense,
 *  and/or sell copies of the Software, and to permit persons to whom the
 *  Software is furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 *  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 *  DEALINGS IN THE SOFTWARE.
 */

#include <stdlib.h>
#include <stdio.h>
```

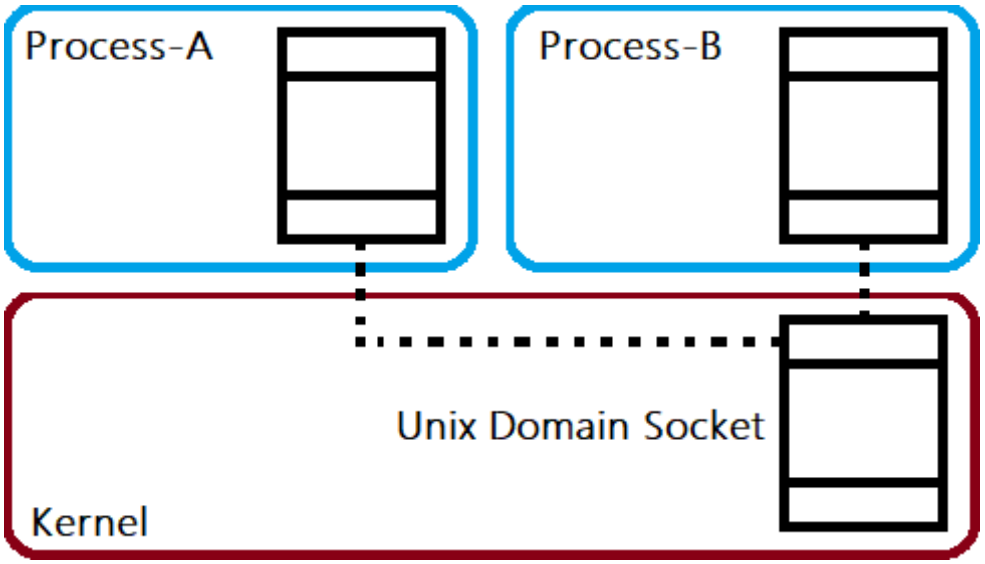
```cpp
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <chrono>
#include <sys/stat.h>
#include <signal.h>

// Alarm signal handler.
void alarmHandler(int) {}

// Simple message type.
typedef std::chrono::time_point<std::chrono::high_resolution_clock> Data;

int main() {
    // Create FIFO PIPE.
    int fd;
    struct stat buffer;
    if(stat("fifo", &buffer) != 0 && mkfifo("fifo", 0777) < 0) {
        std::cout << "Cannot create fifo" << std::endl;
        return -1;
    }

    // Parent process.
    if(fork()) {
        // Open FIFO in write only mode.
        if((fd = open("fifo", O_WRONLY)) < 0) {
            std::cout << "Parent: Cannot open fifo" << std::endl;
            return -1;
        }

        // Always send message.
        while(true) {
            // Send current time, write to the pipe.
            Data data = std::chrono::system_clock::now();
            std::cout << "Parent: Sending: " << data.time_since_epoch().count() <<

            // This will send immediately.
            if(write(fd, &data, sizeof(Data)) < sizeof(Data)) {
                std::cout << "Parent: Cannot write to fifo" << std::endl;
                return -1;
            }
            std::cout << "Parent: Sent." << std::endl;
            sleep(5);
        }

    // Child process.
    } else {
        // Register signal handler for SIGALRM.
        struct sigaction signalConfiguration{};
        signalConfiguration.sa_handler = alarmHandler;
        sigaction(SIGALRM, &signalConfiguration, 0);

        // Open FIFO in read only mode.
        if((fd = open("fifo", O_RDONLY)) < 0) {
            std::cout << "Child: Cannot open fifo" << std::endl;
            return -1;
        }

        // Always receive message.
        while(true) {
            // Receive message.
            Data data;
            std::cout << "Child: Waiting..." << std::endl;

            // Set alarm that will interrupt the read if no message arrives for lo
            alarm(1);

```

```
            // This will block until the parent writes to the pipe.
            if(read(fd, &data, sizeof(Data)) == sizeof(Data)) {
                std::cout << "Child: Received: " << data.time_since_epoch().count(
            }
        }
    }

    // Close the pipe.
    close(fd);

    // Exit.
    return 0;
}
```

Please compile with `g++ linux_fifo_ipc.cpp -o linux_fifo_ipc -std=c++11` command.

Unix Domain Socket

The most complex but the most professional way of Linux IPC is socket communication. Unix Domain Socket is not so different than server/client network socket communication but it is intended for local file system use. I will only cover the basics here to showcase its usage.



Creating a Unix Domain Socket

The socket is created just like a file in Linux returning a file handle.

```
C++
int fd = socket(AF_UNIX, SOCK_TYPE_SEE_BELOW, 0);
```

Socket file path is passed with `sockaddr_un` struct. Here I use file named `socket`.

```
C++
struct sockaddr_un address;
```

```
address.sun_family = AF_UNIX;
strcpy(address.sun_path, "socket");
```

Connected Server-Client Messaging

Server Side

The server is responsible for socket binding. This will create the socket file on the disk.

```C++
bind(fd, (struct sockaddr *)&address, sizeof(sockaddr_un));
```

Later the server starts listening, allowing multiple connections and accepts connections to serve them.

```C++
listen(fd, NUM_MAX_CONNECTIONS);
int connection = accept(fd, (struct sockaddr*)NULL, NULL);
```

Now the returned object from `accept()` is a new file descriptor for `read()` and `write()` operations between the server and the newly connected client. If the server is intended to serve multiple connections, it is a good practice to `fork()` and serve in the child process while the parent process can accept other clients. For IPC purpose, this is rarely necessary.

Client Side

Clients has to create socket and connect to a server. Rest of the operation is `read()` and `write()` just like any other file operation. To close the connection use `shutdown()` function.

IPC Example via Unix Domain Socket in C++

Here is a short example which summarizes the above concept.

domain_socket_ipc.cpp

```C++
/**
 * @file      domain_socket_ipc.cpp
 * @author    Atakan S.
 * @date      01/01/2019
 * @version   1.0
 * @brief     Example IPC Messaging using Unix Domain Socket Connection.
 *
 * @copyright Copyright (c) 2019 Atakan SARIOGLU ~ www.atakansarioglu.com
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a
 *  copy of this software and associated documentation files (the "Software"),
 *  to deal in the Software without restriction, including without limitation
 *  the rights to use, copy, modify, merge, publish, distribute, sublicense,
```

```
 *  and/or sell copies of the Software, and to permit persons to whom the
 *  Software is furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 *  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 *  DEALINGS IN THE SOFTWARE.
 */

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <chrono>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string>
#include <memory.h>

int main() {
    // Unix domain socket file address.
    struct sockaddr_un address;
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "socket");

    // Server process.
    if(fork()) {
        // Delete the old socket file.
        unlink("socket");

        // Create a unix domain socket.
        int fd;
        if((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
            std::cout << "Server: Cannot create socket" << std::endl;
            return -1;
        }

        // Bind the socket to the address.
        if(bind(fd, (struct sockaddr *)&address, sizeof(sockaddr_un)) != 0) {
            std::cout << "Server: Cannot bind socket" << std::endl;
            return -1;
        }

        // Listen up to 100 connections.
        if(listen(fd, 100) != 0) {
            std::cout << "Server: Cannot listen" << std::endl;
            return -1;
        }

        // Serve.
        while(true) {
            // Accept incoming connections.
            std::cout << "Server: Waiting connection..." << std::endl;
            int connection;
            if((connection = accept(fd, (struct sockaddr*)NULL, NULL)) < 0) {
                std::cout << "Server: Connection cannot be accepted" << std::endl;
                return -1;
            }
            std::cout << "Server: Connected." << std::endl;
```

```
81
82              // After accpting a connection, fork to child process to serve it.
83              if(!fork()) {
84                  // Read as long as the connection is alive.
85                  while(true) {
86                      std::cout << "Server: Waiting..." << std::endl;
87
88                      // Receive data.
89                      char buffer[100];
90                      if(read(connection, buffer, 100) > 0) {
91                          std::cout << "Server: Received: " << buffer << std::endl;
92                      } else break;
93                  }
94
95                  // Close he connection.
96                  close(connection);
97              }
98          }
99
100         // Close the socket.
101         close(fd);
102
103     // Client process.
104     } else {
105         // Initial delay until the server is ready.
106         sleep(1);
107
108         // Create a unix domain socket.
109         int fd;
110         if((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
111             std::cout << "Client: Cannot create socket" << std::endl;
112             return -1;
113         }
114
115         // Connect to the server.
116         while(connect(fd, (struct sockaddr *)&address, sizeof(sockaddr_un)) != 0)
117             std::cout << "Client: Connecting... " << std::endl;
118             sleep(1);
119         }
120
121         // Send timestamp.
122         int counter = 5;
123         while(counter--) {
124             auto now = std::to_string(std::chrono::system_clock::now().time_since_
125             std::cout << "Client: Requesting with: " << now << std::endl;
126
127             // Write data to the server.
128             if(write(fd, now.c_str(), now.size()) < now.size()) {
129                 std::cout << "Client: Cannot send request" << std::endl;
130                 return -1;
131             }
132
133             // Delay.
134             sleep(1);
135         }
136
137         // Shutdown and close the connection.
138         shutdown(fd, SHUT_RDWR);
139         close(fd);
140     }
141
142     // Exit.
143     return 0;
144 }
```

The client connects to the server and sends timestamp but also the server can send data back. The above code compiles with `g++ domain_socket_ipc.cpp -o domain_socket_ipc -std=c++11` command.

Connectionless Datagram Messaging

Different than the above approach, here there is no connection and let's say no server/client. One of the pairs has to `bind()` the socket and that is the only difference between them. If you are messaging full-duplex, be careful not to read the messages that you have sent yourself (take a look at `MSG_PEEK` flag).

Simple IPC Unix Domain Socket Messaging in C++

This is much simpler than the connected method I mentioned above. Using `sendto()` and `recvfrom()` methods, you can pass messages between processes. I won't call the processes server and client instead they have nearly identical roles.

socket_datagram_ipc.cpp C++

```cpp
/**
 * @file      socket_datagram_ipc.cpp
 * @author    Atakan S.
 * @date      01/01/2019
 * @version   1.0
 * @brief     Example IPC Messaging using Unix Domain Socket Datagram.
 *
 * @copyright Copyright (c) 2019 Atakan SARIOGLU ~ www.atakansarioglu.com
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a
 *  copy of this software and associated documentation files (the "Software"),
 *  to deal in the Software without restriction, including without limitation
 *  the rights to use, copy, modify, merge, publish, distribute, sublicense,
 *  and/or sell copies of the Software, and to permit persons to whom the
 *  Software is furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 *  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 *  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 *  DEALINGS IN THE SOFTWARE.
 */

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <chrono>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string>
```

```cpp
#include <memory.h>

int main() {
    // Unix domain socket file address.
    struct sockaddr_un address;
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "socket");

    // Receiver process.
    if(fork()) {
        // Delete the old socket file.
        unlink("socket");

        // Create a unix domain socket.
        int fd;
        if((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
            std::cout << "Receiver: Cannot create socket" << std::endl;
            return -1;
        }

        // Bind the socket to the address.
        if(bind(fd, (struct sockaddr *)&address, sizeof(sockaddr_un)) != 0) {
            std::cout << "Receiver: Cannot bind socket" << std::endl;
            return -1;
        }

        // Receive data.
        while(true) {
            // Check for incoming data, non-blocking.
            char buffer[100];
            if(recvfrom(fd, buffer, 100, MSG_DONTWAIT, NULL, NULL) > 0) {
                std::cout << "Receiver: Read: " << buffer << std::endl;
            }

            // Delay.
            sleep(1);
        }

        // Close the socket.
        close(fd);

    // Sender process.
    } else {
        // Initial delay until the socket is ready.
        sleep(1);

        // Create a unix domain socket.
        int fd;
        if((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
            std::cout << "Sender: Cannot create socket" << std::endl;
            return -1;
        }

        // Send timestamp.
        while(true) {
            auto now = std::to_string(std::chrono::system_clock::now().time_since_
            std::cout << "Sender: Writing: " << now << std::endl;

            // Write data to the socket.
            if(sendto(fd, now.c_str(), now.size(), 0, (struct sockaddr *)&address,
                std::cout << "Client: Cannot send" << std::endl;
                return -1;
            }

            // Delay.
            sleep(5);
        }
```

```
106
107         // Close the socket.
108         close(fd);
109     }
110
111     // Exit.
112     return 0;
113 }
```

Compile with `g++ socket_datagram_ipc.cpp -o socket_datagram_ipc -std=c++11` command.

Final Words

I tried to show very basic examples but the IPC topic is huge. Keep this as a starting point and refer to here for shared memory, here for named pipes and here for unix domain socket messaging. Try hard 👍

TAGS: C++, FEATURED3, FIFO, INTER PROCESS COMMUNICATION, INTER PROCESS MESSAGING, IPC, LINUX, NAMED PIPE, SHARED MEMORY, SOCKET COMMUNICATION, SYSTEM CALL, UNIX DOMAIN SOCKET