Systems programming

Week 2 – Lab 4

ZeroMQ

ZeroMQ is a high level communication library that uses existing IPC mechanism to establish communication between components in a systems.

Although ZeroMQ uses existing communication mechanism it offers a better programming interface and extra functionalities.

In this laboratory students will use the ZeroMQ library to implement a simple client-server and broadcast communication between different processes.

1 ZeroMQ

https://zeromq.org/get-started/

ZeroMQ (also spelled ØMQ, 0MQ or ZMQ) is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middle-ware, a ZeroMQ system can run without a dedicated message broker.

ZeroMQ supports common messaging patterns (pub/sub, request/reply, client/server and others) over a variety of transports (TCP, in-process, inter-process, multi-cast, Web-socket and more), making inter-process messaging, independently on the communication scope (local to a machine or in the internet)

1.1 Installation and compilation

There are official packages to most of the Linux distributions and Mac OS X. To install is is only necessary to follow the instruction on the next page:

https://zeromq.org/download/

MAC OS X

In mac os X the correct way to install this library is running the commanda:

```
brew install zeromq
```

this willinstall all the necessary files, but, unfortunately they are not copied to the correct directories. In order to compile programs with zeromq follow the next steps

1 - Verify if the files are in the correct directory (/opt/homebrew/Cellar/zeromq/4.3.5_1)

```
% ls /opt/homebrew/Cellar/zeromq/4.3.5_1/
AUTHORS                 INSTALL_RECEIPT.json  NEWS                    bin/                    lib/
ChangeLog               LICENSE               README.md               include/                share/
```

2 – If they are in the expected directory, specify those directories when compiling:

```
gcc client-sub.c -o client-sub -lzmq
                 -L /opt/homebrew/Cellar/zeromq/4.3.5_1/lib
                 -I /opt/homebrew/Cellar/zeromq/4.3.5_1/include
```

If the package was installed in a different directory the command brew info zeromq whill print that location

WSL

When using WSL it is necessary to install the correct packages running the packages manager in the WSL terminal as described in the ZeroMQ documentation.

After installation of the libraries it is possible to use the provided API by including the **zmq.h** header file in the C source code.

```
#include <zmq.h>
```

and linking against the **zmq** library files:

```
gcc example1.c -o example1 -lzmq
```

1.2 Programming model

To understand the functioning of ZeroMQ read the first chapter of the documentation:

https://zguide.zeromq.org/docs/chapter1/

In this chapter the following concepts were introduced:

- Context and sockets
- Communication patterns

- The API

1.2.1 socket creation

Communication between two processes is accomplished in ZeroMQ through sockets. These ZeroMQ socket are different from the Internet sockets (as some of you learned in other courses), although use similar concepts.

The way ZeroMQ sockets work is described in the following document:

https://zeromq.org/socket-api/ (sections Socket API, Key differences to conventional sockets, Socket lifetime, Bind vs Connect).

To establish communication between processes it is necessary for each process to create a context and then use such context to create all necessary sockets.

After socket creation, it can either be binded (to an address) or connected (to another socket) using the correct addresses. Usually it is the socket in the more stable process (for instance in a server) that is binded, while the clients do connect. Usually the address of the socket that is binded is the one that is known by all the clients.

For two sockets to communicate, it is necessary that one is binded and the other connected.

After the connection is established the processes can transmit data between them.

The creation and establishing of a channel between two sockets is done as follows:

Process 1 (server)	Process 2 (clients)

void * context = zmq_ctx_new ();
 	
void *responder = zmq_socket (context, ZMQ_TYPE_A);
 	
zmq_bind (responder, ADDRESS_A);
	void *context = zmq_ctx_new ();
	
	void *requester = zmq_socket (context, ZMQ_TYPE_B);
	
	zmq_connect (requester, ADDRESS_A);

The first line creates a context, that can be used to manage various channels/sockets in each process, the second line creates a socket using the previous context and assigns it a type, and the third line established a connection between both sockets.

As opposed from regular sockets, in ZeroMQ the bind and the connect are asynchronous, the server does not wait for the establishing of connections from the clients, all this is hidden inside the library and handled by the context.

1.2.2 Communication API

https://libzmq.readthedocs.io/en/latest/zmq_recv.html

https://libzmq.readthedocs.io/en/latest/zmq_send.html

After the establishing of the communication channel between two sockets it is now possible to exchange messages.

The two basic functions are **zmq_send** and **zmq_recv**, that work in a similar fashion as the UNIX recv/send or read/write:

```
int zmq_send (void '*socket', const void '*buf', size_t 'len', int 'flags');
int zmq_recv (void '*socket', void '*buf', size_t 'len', int 'flags');
```

The first argument is the socket that will be used to communicate, and the next two arguments represent the buffer (with corresponding length) that will be send, or where the message will be stored.

The flags can modify the behavior of the functions, for instance to do a non blocking read in the case of ZMQ_DONTWAIT.

The writing and reading of these messages is atomic, i.e. the library guarantees that the bytes that were provided in the send, and delivered in a single write.

1.2.2.1 Working with strings

https://zeromq.org/messages/#working-with-strings

ZeroMQ sockets handle binary data, if a C programm transmitts strings, it is advised to encode in a way compatible with clients written in other languages by omitting the ‘\0’in the transmission.

The following code shows how to send/receive strings in C using ZeroMQ, and demonstrates the use of ZeroMQ functions to send and receive messages:

```
static int s_send (void *socket, char *string) {
    int size = zmq_send (socket, string, strlen (string), 0);
    return size;
}
```

The s_send function just sends the bytes in the char array up to (not including) the '\0', it is the receiver that needs to append to the receive message the correct '\0'.

The s_recv message will call the zmq_recv and will read all the bytes sent on the other socket, put it on a buffer, add the ´\0' and duplicate the string.

```
static char * s_recv (void *socket) {
    char buffer [256];
    int size = zmq_recv (socket, buffer, 255, 0);
    if (size == -1)
        return NULL;
    if (size > 255)
        size = 255;
    buffer [size] = '\0';
    return strdup (buffer);
}
```

These functions are available in the **zhelpers.h** file.

1.2.2.2 Multipart messages

https://libzmq.readthedocs.io/en/latest/zmq_send.html

A ZeroMQ message is composed of 1 or more message parts, and ZeroMQ ensures atomic delivery of messages: peers shall receive either all *message parts* of a message or none at all.

Sometimes it is better for the programmer to have messages composed of multiple parts, i.e. call several times the zmq_send instead of calling zmq_send once, and have all that data sent as a whole.

To accomplish this it is necessary to use the ZMQ_SNDMORE flag to all the zmq_send calls with the exception of the final one. ZeroMQ ensures atomic delivery of messages: recipients shall receive either all *message parts* of a message or none at all.

The next example sends a message composed of three parts. The message is only transmitted to to the recipients after execution of the zmq_send without the ZMQ_SNDMORE flag.

```
zmq_send (send_socket, &part_1, len_part_1, ZMQ_SNDMORE);
zmq_send (send_socket, &part_2, len_part_2, ZMQ_SNDMORE);
zmq_send (send_socket, &final_part, len_final_part,);
```

The recipience of the message will need to do the same number of zmq_recv (one for each part):

```
zmq_recv (recv_socket, &part_1, len_part_1, 0);
zmq_recv (recv_socket, &part_2, len_part_2, 0);
zmq_recv (recv_socket, &final_part, len_final_part, 0);
```

The corresponding message to sens strings is

```
static int s_sendmore (void *socket, char *str) {
   int size = zmq_send (socket, str, strlen (str), ZMQ_SNDMORE);
   return size;
}
```

1.3 Communication patterns

https://zeromq.org/socket-api/#messaging-patterns

https://zguide.zeromq.org/docs/chapter1/

When creating ZeroMQ sockets it is necessary to define their type or what communication patterns they will implement. These patterns provide extra functionalities (for instance indirect communication) that the regular IPC mechanism do not.

Depending on the type of interaction the programmer wants for each channel, he must select the appropriate pattern and assign to the sockets the suitable types as presented in the following table. ZeroMQ offers more patterns but these two (Request/reply and Publish/subscribe) are most basic and useful.

Pattern	Socket 1 (server)	Socket 2 (client)

Request/ Reply
REQ-REP
	`zmq_socket (ctx, ZMQ_REP);`
	`zmq_bind(...)`
	
	`// replies to the clients`
		`zmq_socket (ctx, ZMQ_REQ);`
		`zmq_connect(...)`
		
		`// makes requests to the server`

Publish/ Subscribe
PUB-SUB
	`zmq_socket (ctx, ZMQ_PUB);`
	`zmq_bind(...)`
	
	`// publishes messages to the clients`
		`zmq_socket (ctx, ZMQ_SUB);`
		`zmq_connect(...)`
		
		`// subscribes messages from the server`

For any of the available communication patters offered by ZeroMQ it is important to understand what type of sockets can be connected, their responsibilities and message patterns.

1.3.1 Request-Reply (REQ/REP) pattern

https://zeromq.org/socket-api/#request-reply-pattern

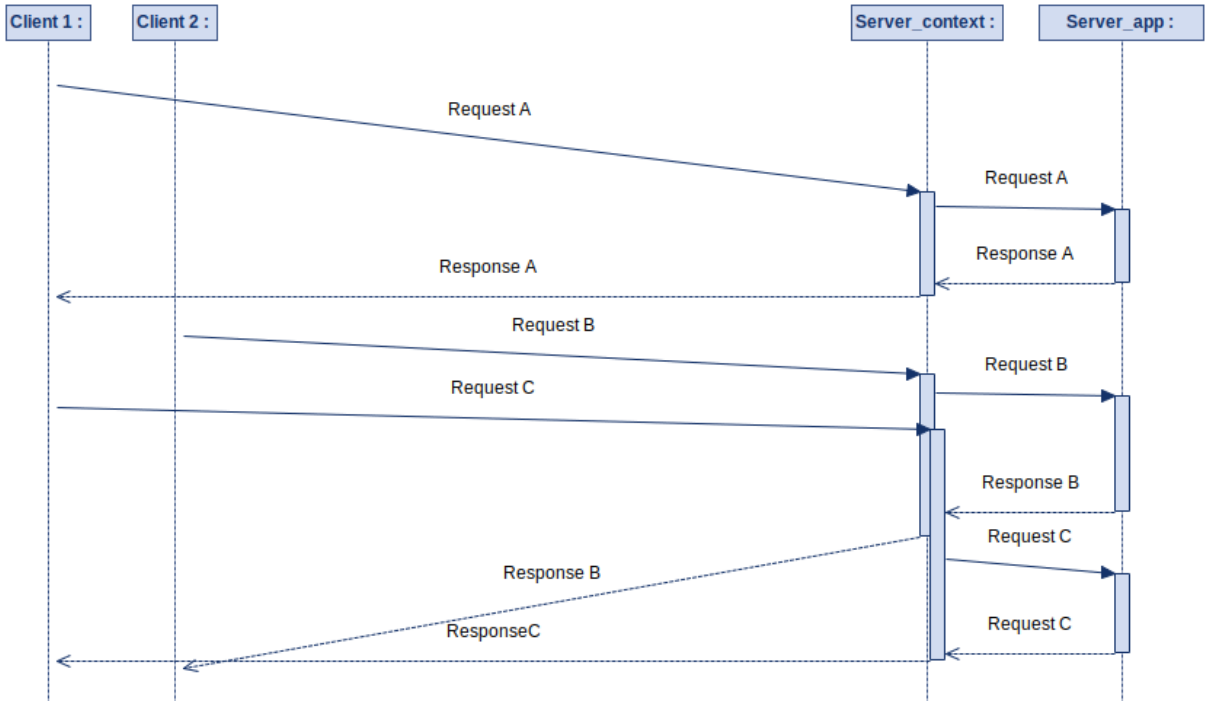
https://zguide.zeromq.org/docs/chapter1/#Ask-and-Ye-Shall-Receive

In order to implement the client/server model (a client sends a request to the server, and the server replies to it), it is necessary to use the REQ/REP pattern. The fundamental characteristics are:

Server	Client
the server should have a well known address	the client should know the server address
The server should have a socket of type ZMQ_REP	The client should have a socket of type ZMQ_REQ

The server should do a bind to the well known socket	The client should do a connect to the address of the server
To handle requests the server should have a loop that • receives a message (using the suitable socket) • replies using the same socket	To make a request to the server the client should: • send a requester • receive the reply

Various clients can be connected to a single server and ZeroMQ will handle such concurrent usage, as ilustrated in the next diagram:



The request A is sent by Client_1 and received by the server, after being processes the server replies to the client. After a few moments, Client_2 sends a request that is received by the server application immediately.

During the processing of the Request B, another message (Request C) is received by the Server_context. Since the application is still processing Request B, Request C will be put on wait. After the server application processes Request B, it can start processing Request C.

All the messages sent by the various clients are received by the server application on a single sockets. Until they are received by the application these messages are buffered by the context. The context also store information about the sender of every message, and, although does not pass this information to the server application, the context uses it to redirect the replies.

To correctly accomplish this type of interaction and functionality with various concurrent clients, the server should send the reply to every message received. By doing this, the context will forward the reply to the correct client, since the context knows who sent the corresponding request.

The REQ-REP socket pair is in lockstep. The client issues zmq_send() and then zmq_recv(), in a loop (or once if that's all it needs). Doing any other sequence (e.g., sending two messages in a row) will result in a return code of -1 from the send or recv call. Similarly, the server executes zmq_recv() and then zmq_send() in that order, as often as it needs to.

The following code shows a server that correctly handles messages from various clients. The server receives and integer and returns to all clients the World string:

```
int main (void)
{
    context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");
    while (1) {
        int n;
        zmq_recv (responder, &n, sizeof(n), 0);
        printf ("Received number %d\n", n);
        sleep(1);
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

Here the server sends a reply for every request it receives. When sending a string (on the zmq_send) the '\0' is not transmitted.

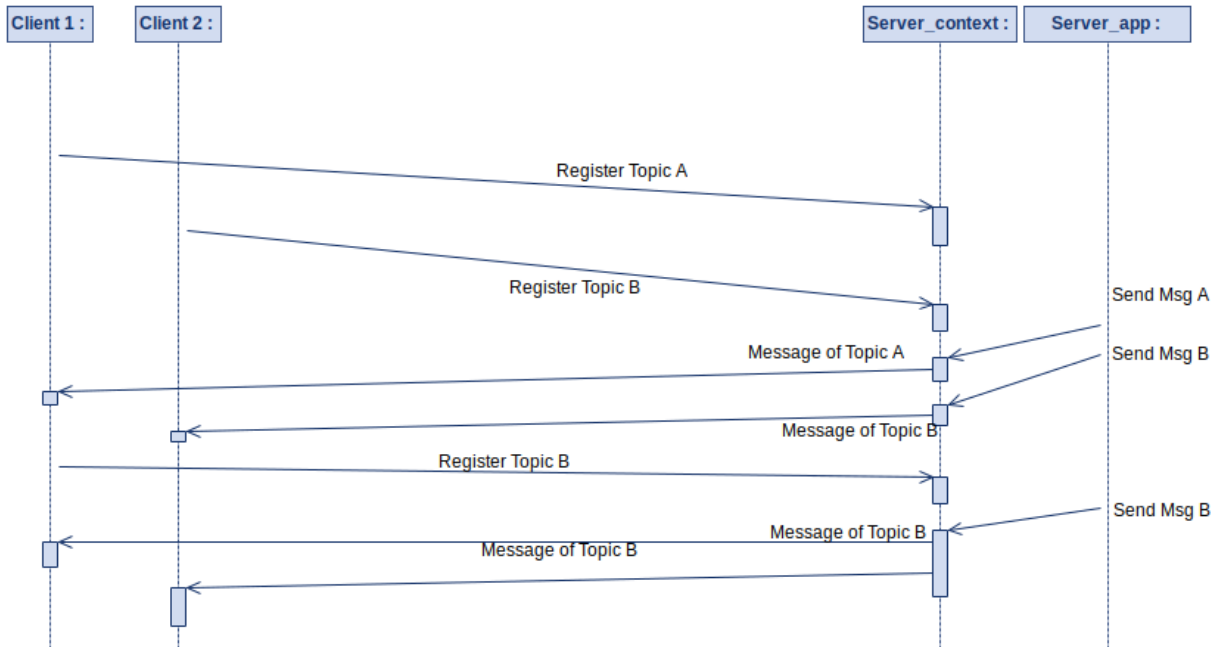
1.3.2 Publish/Subscribe (PUB/SUB) pattern

https://zeromq.org/socket-api/#publish-subscribe-pattern

https://zguide.zeromq.org/docs/chapter1/#Getting-the-Message-Out

http://api.zeromq.org/master:zmq-setsockopt

Pub/Sub is another classic pattern where the publisher sends messages that are received by various subscribers. This is similar to broadcast but it is possible for subscribers to filter messages that are deliver to them, based on topics.



In this model every message send by the server (using a ZMQ_PUB sockets) should have a topic assigned (in the previous example topic A and topic B). The client should have a ZMQ_SUB sockets, and specify what topics he is interested on before receiving messages (ilustrated as the Register Topic messages). Whenever the server sends a messaged (tagged with a topic) ZeroMQ filters messages delivered to each client based on that topics, a copy of the server message is delivered to every client that register such topic.

The general programming/execution steps are as follows:

Server	Client
the server should have a well known address	the client should know the server address
The server should have a socket of type ZMQ_PUB	The client should have a socket of type ZMQ_SUB
the server should do a bind to the well known socket address	The client should do a connect to the address of the server
	The client should register the topic of messages he is interested on.
The server can start sending messages tagged with a certain topic	The client starts receiving just the messages whose topic is one of those registed.

In order to specify the topics a client is interested on, it is necessary to use the zmq-setsockopt function as presented in the following code:

```
zmq_setsockopt (sub_socket, ZMQ_SUBSCRIBE, topic, strlen (topic));
```

Since a newly created ZMQ_SUB sockets filters out all incoming messages (i.e. the client does not receive any message), it is fundamental to call this function to start receiving messages. Every call to the zmq_setsockopt/ZMQ_SUBSCRIBE function will create a new message filter, allowing more messages to be delivered. Multiple filters can be attached to a single ZMQ_SUB socket, in which case a message is accepted if it matches at least one filter.

If the client wants to receive all messages published by a server, an empty topic with zero length should be used.

The server should specify the topic of each message sent in a ZMQ_PUB sockets. This is accomplished by placing a message topic in the beginning of every message, either when creating the buffer, or sending a multipart messages. The following code shows a server that send numbers prefixed with a topic (odd or even):

```
int main (void){
   void *context = zmq_ctx_new ();
```

```
    void *publisher = zmq_socket (context, ZMQ_PUB);
    int rc = zmq_bind (publisher, "tcp://*:5556");
    while (1) {
        char message[100];
        int n = rand();
        if (n%2 == 0){
            sprintf (message, "even %d", n);
        }else{
            sprintf (message, "odd %d", n);
        }
        printf("%s\n", message);
        s_send (publisher, message);
    }
    return 0;
}
```

and the next one a client that only receives the odd numbers sent by the server:

```
int main (int argc, char *argv [])
{
    void *context = zmq_ctx_new ();
    void *subscriber = zmq_socket (context, ZMQ_SUB);
    zmq_connect (subscriber, "tcp://localhost:5556");
    zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "odd", 3);
    while(1) {
        char *string = s_recv (subscriber);
        printf("received message - %s\n", string);
        free(string);
    }
    return 0;
}
```

Although the examples use strings as topics any byte sequence can be used. The provided example sends the topic as the first part of the message and a number in binary on the second part.

1.4 Communication transports

https://zguide.zeromq.org/docs/chapter2/#Plugging-Sockets-into-the-Topology

https://libzmq.readthedocs.io/en/latest/zmq_tcp.html

https://libzmq.readthedocs.io/en/latest/zmq_ipc.html

ZeroMQ can use various underlying communication transport protocols. To this course only the Unix sockets and TCP sockets will be used.

These two transport protocols mainly differ in the scope of communications and the type of addresses.

zmq_ipc

- Scope - Processes in the same machine can communicate using zmq_ipc sockets
- Addresses of ZMQ_ipc sockets have the following form ipc://<path>. The path should be replace by a real file system path that can be accessed by all the processes server and clients (usually in the /tmp/ directory).

zmq_tcp

- Scope - Processes in different machines in the internet can communicate using zmq_tcp sockets
- Addresses of ZMQ_tcp sockets have the following form tcp://<addr>:<port> . The addr part can be a wild-card * (meaning all available interfaces and addresses) or the numeric representation of the IPv4 or IPv6 address. The port number may be specified by a numeric value, usually above 1024.

When using ZMQ_ipc sockets the address on the bind and connect should be the same, while on a ZMQ_tcp socket the server can bind using the wildcard (*), but the clients should connect with an actual address.

1.5 Resources management

https://zguide.zeromq.org/docs/chapter1/#Making-a-Clean-Exit

https://man7.org/linux/man-pages/man3/assert.3.html

All objects should be created with suitable functions and when destroyed with the corresponding functions as presented in the following table

Creation	Destruction
void *ctx = zmq_ctx_new ();	zmq_ctx_shutdown (ctx);
void *s = zmq_socket (ctx, ZMQ_XYZ);	zmq_close (s);
char *string = s_recv (subscriber);	free(string);

All ZeroMQ functions return an error code that should be verified in order to guarantee that the operation executed correctly. Although these values are not verified in most of the examples, students should verify them:

```
void *context = zmq_ctx_new ();
assert (rc != NULL);
void *responder = zmq_socket (context, ZMQ_REP);
assert (responder != NULL);
int rc = zmq_bind (responder, "tcp://*:5555");
assert (rc == 0);
```

Student can use the assert function, that immediately aborts the program or any other verification to allow other error recuperation.

2 Important functions

This list includes the function fundamental for writing programs with ZeroMQ. Each function is linked to the corresponding man mage (online). Students can also execute the Linux man command to read every function documentation

zmq	zmq_ipc
zmq_ctx_new	zmq_tcp
zmq_ctx_shutdown	
	zmq_recv
zmq_socket	zmq_send
zmq_close	
zmq_bind	
zmq_connect	
zmq_getsockopt	

3 Exercise 1

Read provided PDF or the following ZeroMQ pages and study the various examples.

https://zguide.zeromq.org/docs/chapter1/

https://zeromq.org/get-started/

https://zeromq.org/download/

https://zeromq.org/messages/

https://zeromq.org/socket-api/

4 Exercise 2

1 - Observe the files in the **example-REQ-REP** directory, compile both programs and run one client and one server. Find the **/tmp/s1** file.

2 - execute the following combinations in different windows and observe the various behaviors:

- one server → one client → other client
- one server → one client → other server → other client
- one client + send numbers → other clients+send numbers → one server

3 - Comment the zmq_send in the server and the zmq_recv in the client and observe the behavior.

4 - Remove the comment in the server (i.e. the server sends the reply, but the client does not receive it) and observe the behavior.

5 - Correct the client and server code in order to verify the values returned by the zmq_sen and zmq_recv functions

5 Exercise 3

1 - Observe the files in the **example-PUB-SUB** directory, compile both programs.

2 - Execute the following combinations in different windows and observe the various behaviors (kill all process between experiments:

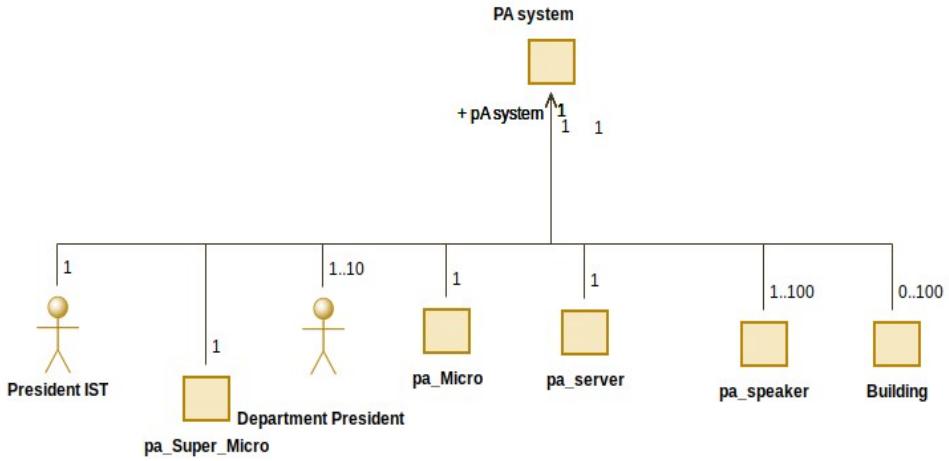
- one server → one client
- one client → one server
- one server → one client → other client

3 – Create a new client so that it now receives the even numbers

6 public address system (PA system)

In this exercise students will implement a simple PA system (https://en.wikipedia.org/wiki/Public_address_system) like those in America High schools, wher ethe principal talks to the students.

The block diagram of this system is as follows:



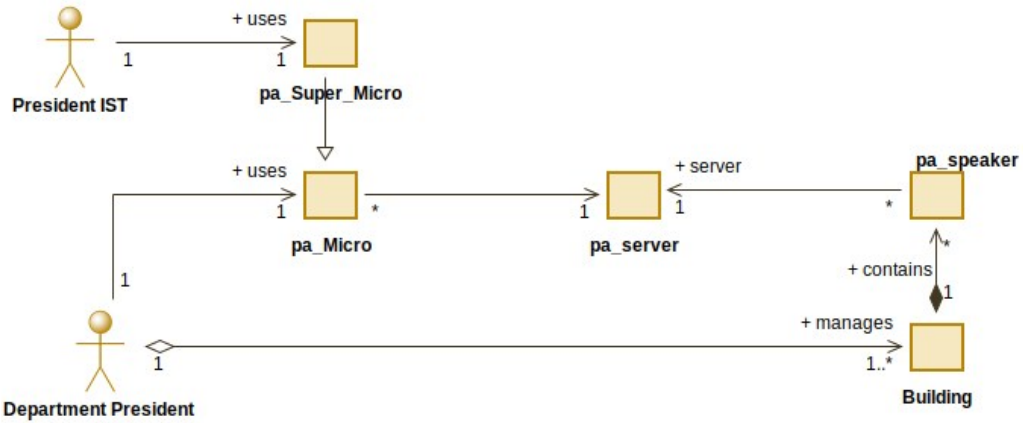
The entities relevant to this system are:

- the IST president
- the department presidents
- the various microphones (used by the presidents)
- the server that redirects messages to every building
- the buildings managed by each department president

- the speakers at the buildings

Each President will use his own microphone to broadcast messages, with the following limitations, the president of a department can only send messages to the buildings associated with his department, while the president of IST can send messages to any building.

The block diagram can further be detailed:



The president of IST uses a Super Microphone, while the Presidents of the Departments use regular ones. The Presidents of the Departments manage buildings that have speakers installed. The microphones and speakers connect to the server to forward and receive messages.

Students should implement 4 different applications that mimic the microphones (the super and the regular ones) the server and the speakers.

The messages written in the super-microphone should be presented in all speakers, while the messages of the Department Presidents should only be presented in the buildings managed by him/her.

The server should implement two ZeroMQ sockets: one to receive the messages from the microphones and another to forward them to the speakers.

Skeletons of all the components are provided, student will have to complete the 4 applications following the comments there. It is advisable that students start to

guarantee the communication between the microphones and the server, and afterwards implement the publishing of messages to the microphones.