# Zeromq tutorial in C

## Installation

Follows the steps on <http://zeromq.org/area:download> to install zeromq in your local machine.

## Basic message patterns

We assumption that you have succeed to install zeromq into your local machine.

* 1. Request – reply pattern

The most basic pattern in ZeroMQ is req-rep model, where client sends a request and waits for replies from server. Socket ZMQ\_REQ will block on send unless it has successfully received a reply back. Socket ZMQ\_REP will block on recv unless it has received a request.

**reqrep\_server.c**

Server is created with a socket type “ZMQ\_REP”

1. **void** \*context = zmq\_ctx\_new ();
2. **void** \*responder = zmq\_socket (context, ZMQ\_REP);
3. **int** rc = zmq\_bind (responder, "tcp://\*:5555");

It will block on recv() to get a request before it can send a reply.

1. zmq\_recv (responder, buffer, 10, 0);
2. printf ("Received Hello\n");
3. sleep (1);          //  Do some 'work'
4. zmq\_send (responder, "World", 5, 0);

**reqrep\_client.c**

Client is created with a socket type “ZMQ\_REQ”.

1. **void** \*context = zmq\_ctx\_new ();
2. **void** \*requester = zmq\_socket (context, ZMQ\_REQ);
3. zmq\_connect (requester, "tcp://localhost:5555");

They have to send a request and then wait for reply.

1. **for** (request\_nbr = 0; request\_nbr != 10; request\_nbr++) {
2. **char** buffer [10];
3. printf ("Sending Hello %d…\n", request\_nbr);
4. zmq\_send (requester, "Hello", 5, 0);
5. zmq\_recv (requester, buffer, 10, 0);
6. printf ("Received World %d\n", request\_nbr);
7. }

Any attempt to send another message to the socket (ZMQ\_REQ, ZMQ\_REP), without having received a reply, request will result in an error.

* 1. Publish – subscribe pattern

Pub/Sub is another classic pattern where senders of messages, called publishers do not program the messages to be sent directly to specific receivers, called subscribers. Messages are published without the knowledge of what or if any subscriber of that knowledge exists.

The general pattern where multiple subscribers subscribes to messages/topics being published by a publisher is introduced below.

**Pub\_server.c**

Publisher is created with “ZMQ\_PUB” socket type.

1. **void** \*context = zmq\_ctx\_new ();
2. **void** \*publisher = zmq\_socket (context, ZMQ\_PUB);
3. **int** rc = zmq\_bind (publisher, "tcp://\*:5556");

Data is publised along with a topic. The subscribers usually sets a filter on these topics for topic of their interests.

1. **while** (1) {
2. //  Send message to all subscribers
3. **char** update [100];
4. sprintf (update, "%d Hello from server", rand() % 3 + 1);
5. printf("Sending: %s\n", update);
6. zmq\_send (publisher, update, strlen(update), 0);
7. sleep(2);
8. }

**sub\_client.c**

Subcribers are created with “ZMQ\_SUB” socket type. A zmq subscriber can connect to many publishers.

1. **void** \*context = zmq\_ctx\_new ();
2. **void** \*subscriber = zmq\_socket (context, ZMQ\_SUB);
3. **int** rc = zmq\_connect (subscriber, "tcp://localhost:5556");

The current version of zm1 supports filtering of messages based on topics at subscriber side. This is usually set via socketoption.

1. **char** \*filter = (argc > 1)? argv [1]: "1";
2. rc = zmq\_setsockopt (subscriber, ZMQ\_SUBSCRIBE,
3. filter, strlen (filter));

Pub/sub communication is asynchronous. If a “publish” service has been started already and then when you start “subscribe” service, it would not receive a number of message that was published already by the pub services. Starting “publisher” and “subscriber” is independent of each other.

* 1. Push – pull pattern

Push/pull sockets let you distribute messages to multiple workers, arranged in a pipeline. A Push socket will distribute sent messages to its Pull clients evenly. The results computed by Pull clients are not sent upstream but downstream to another pull socket.

**Master.c**

Master is created with “ZMQ\_PUSH” socket type. Master is bound to well known port to which worker can connect to.

1. **void** \*context = zmq\_ctx\_new ();
3. //  Socket to send messages on
4. **void** \*sender = zmq\_socket (context, ZMQ\_PUSH);
5. zmq\_bind (sender, "tcp://\*:5557");
6. //  Send 100 tasks
7. **int** task\_nbr;
8. **int** total\_msec = 0;     //  Total expected cost in msecs
9. **for** (task\_nbr = 0; task\_nbr < 100; task\_nbr++) {
10. **int** workload;
11. //  Random workload from 1 to 100msecs
12. workload = rand() % 100 + 1;
13. total\_msec += workload;
14. **char** string [10];
15. sprintf (string, "%d", workload);
16. zmq\_send (sender, string, strlen(string), 0);
17. }

**Worker.c**

Workers are created with “ZMQ\_PULL” socket type to pull requests from master and uses a push socket to connect and push result to result collector.

1. **void** \*context = zmq\_ctx\_new ();
2. **void** \*receiver = zmq\_socket (context, ZMQ\_PULL);
3. zmq\_connect (receiver, "tcp://localhost:5557");
5. //  Socket to send messages to
6. **void** \*sender = zmq\_socket (context, ZMQ\_PUSH);
7. zmq\_connect (sender, "tcp://localhost:5558");
9. //  Process tasks forever
10. **char** string[100] = {0};
11. **int** len = 0;
12. **while** (1) {
13. len = zmq\_recv (receiver, string, 100, 0);
14. string[len] = '\0';
15. printf ("%s.", string);     //  Show progress
16. fflush (stdout);
17. usleep (atoi (string) \* 1000);    //  Do the work
18. zmq\_send (sender, "", 0, 0);        //  Send results to sink
19. }

**Sink.c**

Result collector is created with “ZMQ\_PULL” socket type and collects results from intermediate workers. They also are bound to well known port so that intermediate workers can connect to it.

1. //  Prepare our context and socket
2. **void** \*context = zmq\_ctx\_new ();
3. **void** \*receiver = zmq\_socket (context, ZMQ\_PULL);
4. zmq\_bind (receiver, "tcp://\*:5558");
6. //  Wait for start of batch
7. **char** string[100] = {0};
8. **int** len = 0;
9. len = zmq\_recv (receiver, string, 100, 0);
11. //  Process 100 confirmations
12. **int** task\_nbr;
13. **for** (task\_nbr = 0; task\_nbr < 100; task\_nbr++) {
14. len = zmq\_recv (receiver, string, 100, 0);
15. **if** ((task\_nbr / 10) \* 10 == task\_nbr)
16. printf (":");
17. **else**
18. printf (".");
19. fflush (stdout);
20. }