Other

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 middleware, 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, multicast, WebSocket and more), making inter-process messaging as simple as inter-thread messaging. This keeps your code clear, modular and extremely easy to scale.

ZeroMQ is developed by a large community of contributors. There are third-party bindings for many popular programming languages and native ports for C# and Java.

The Zero in ZeroMQ

The philosophy of ZeroMQ starts with the zero. The zero is for zero broker (ZeroMQ is brokerless), zero latency, zero cost (it's free), and zero administration.

More generally, "zero" refers to the culture of minimalism that permeates the project. We add power by removing complexity rather than by exposing new functionality.

The Guide

The guide explains how to use ØMQ, covers basic, intermediate and advanced use with 60+ diagrams and 750 examples in 28 languages.

Also available as a book O'Reilly.

Libzmq - the low level library

Libzmq (https://github.com/zeromq/libzmq) is the low-level library behind most of the different language bindings. Libzmq expose C-API and implemented in C++. You will rarely use libzmq directly, however if you want to contribute to the project or learn the internals of zeromq, that is the place to start.

First example

So let's start with some code, the "Hello world" example (of course).

Copy

```
//  Hello World server
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>

int main (void)
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind (responder, "tcp://*:5555");
    assert (rc == 0);

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
        printf ("Received Hello\n");
        sleep (1);          //  Do some 'work'
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
```

The server creates a socket of type response (you will read more about request-response later), binds it to port 5555 and then waits for messages. You can also see that we have zero configuration, we are just sending strings.

Copy

```
//  Hello World client
#include <zmq.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main (void)
{
    printf ("Connecting to hello world server…\n");
    void *context = zmq_ctx_new ();
    void *requester = zmq_socket (context, ZMQ_REQ);
    zmq_connect (requester, "tcp://localhost:5555");
```

```
        int request_nbr;
        for (request_nbr = 0; request_nbr != 10; request_nbr++) {
            char buffer [10];
            printf ("Sending Hello %d…\n", request_nbr);
            zmq_send (requester, "Hello", 5, 0);
            zmq_recv (requester, buffer, 10, 0);
            printf ("Received World %d\n", request_nbr);
        }
        zmq_close (requester);
        zmq_ctx_destroy (context);
        return 0;
    }
```

The client creates a socket of type request, connects and starts sending messages.

Both the `send` and `receive` methods are blocking (by default). For the receive it is simple: if there are no messages the method will block. For sending it is more complicated and depends on the socket type. For request sockets, if the high watermark is reached or no peer is connected the method will block.

Other

Socket API

Sockets are the de facto standard API for network programming. That's why ZeroMQ presents a familiar socket-based API. One thing that make ZeroMQ especially tasty to developers is that it uses different socket types to implement any arbitrary messaging pattern. Furthermore ZeroMQ sockets provide a clean abstraction over the underlying network protocol which hides the complexity of those protocols and makes switching between them very easy.

Key differences to conventional sockets

Generally speaking, conventional sockets present a synchronous interface to either connection-oriented reliable byte streams (SOCK_STREAM), or connection-less unreliable datagrams (SOCK_DGRAM). In comparison, ZeroMQ sockets present an abstraction of an asynchronous message queue, with the exact queueing semantics depending on the socket type in use. Where conventional sockets transfer streams of bytes or discrete datagrams, ZeroMQ sockets transfer discrete messages.

ZeroMQ sockets being asynchronous means that the timings of the physical connection setup and tear down, reconnect and effective delivery are transparent to the user and organized by ZeroMQ itself. Further, messages may be queued in the event that a peer is unavailable to receive them.

Conventional sockets allow only strict one-to-one (two peers), many-to-one (many clients, one server), or in some cases one-to-many (multicast) relationships. With the exception of PAIR sockets, ZeroMQ sockets may be connected to multiple endpoints, while simultaneously accepting incoming connections from multiple endpoints bound to the socket, thus allowing many-to-many relationships.

Socket lifetime

ZeroMQ sockets have a life in four parts, just like BSD sockets:

- Creating and destroying sockets, which go together to form a karmic circle of socket life
- Configuring sockets by setting options on them and checking them if necessary
- Plugging sockets into the network topology by creating ZeroMQ connections to and from them.
- Using the sockets to carry data by writing and receiving messages on them.

Bind vs Connect

With ZeroMQ sockets it doesn't matter who binds and who connects. In the above you may have noticed that the server used Bind while the client used Connect. Why is this, and what is the difference?

ZeroMQ creates queues per underlying connection. If your socket is connected to three peer sockets, then there are three messages queues behind the scenes.

With Bind, you allow peers to connect to you, thus you don't know how many peers there will be in the future and you cannot create the queues in advance. Instead, queues are created as individual peers connect to the bound socket.

With Connect, ZeroMQ knows that there's going to be at least a single peer and thus it can create a single queue immediately. This applies to all socket types except ROUTER, where queues are only created after the peer we connect to has acknowledge our connection.

Consequently, when sending a message to bound socket with no peers, or a ROUTER with no live connections, there's no queue to store the message to.

When should I use bind and when connect?

As a general rule use bind from the most stable points in your architecture, and use connect from dynamic components with volatile endpoints. For request/reply, the service provider might be the point where you bind and the clients are using connect. Just like plain old TCP.

If you can't figure out which parts are more stable (i.e. peer-to-peer), consider a stable device in the middle, which all sides can connect to.

You can read more about this at the ZeroMQ FAQ under the "Why do I see different behavior when I bind a socket versus connect a socket?" section.

High-Water-Mark

The high water mark is a hard limit on the maximum number of outstanding messages ZeroMQ is queuing in memory for any single peer that the specified socket is communicating with.

If this limit has been reached the socket enters an exceptional state and depending on the socket type, ZeroMQ will take appropriate action such as blocking or dropping sent messages. Refer to the individual socket descriptions below for details on the exact action taken for each socket type.

Messaging Patterns

Underneath the brown paper wrapping of ZeroMQ's socket API lies the world of messaging patterns. ZeroMQ patterns are implemented by pairs of sockets with matching types.

The built-in core ZeroMQ patterns are:

- **Request-reply**, which connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Pub-sub**, which connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- **Pipeline**, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern.
- **Exclusive pair**, which connects two sockets exclusively. This is a pattern for connecting two threads in a process, not to be confused with "normal" pairs of sockets.

There are more ZeroMQ patterns that are still in draft state:

- **Client-server**, which allows a single ZeroMQ *server* talk to one or more ZeroMQ *clients*. The client always starts the conversation, after which either peer can send messages asynchronously, to the other.
- **Radio-dish**, which used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion.

Request-reply pattern

The request-reply pattern is intended for service-oriented architectures of various kinds. It comes in two basic flavors: synchronous (`REQ` and `REP` socket types), ~~and asynchronous socket types (`DEALER` and `ROUTER` socket types), which may be mixed in various ways.~~

The request-reply pattern is formally defined by RFC 28/REQREP.

REQ socket

A `REQ` socket is used by a client to send requests to and receive replies from a service. This socket type allows only an alternating sequence of *sends* and subsequent *receive* calls. A `REQ` socket may be connected to any number of `REP` or ~~`ROUTER` sockets~~. Each request sent is round-robined among all connected services, and each reply received is matched with the last issued request. It is designed for simple request-reply models where reliability against failing peers is not an issue.

If no services are available, then any send operation on the socket will block until at least one service becomes available. The `REQ` socket will not discard any messages.

Summary of characteristics:

Compatible peer sockets	`REP` , ~~`ROUTER`~~
Direction	Bidirectional
Send/receive pattern	Send, Receive, Send, Receive, …
Outgoing routing strategy	Round-robin
Incoming routing strategy	Last peer
Action in mute state	Block

REP socket

A `REP` socket is used by a service to receive requests from and send replies to a client. This socket type allows only an alternating sequence of *receive* and subsequent *send* calls. Each request received is fair-queued from among all clients, and each reply sent is routed to the client that issued the last request. If the original requester does not exist any more the reply is silently discarded.

Summary of characteristics:

Compatible peer sockets	`REQ` , ~~`DEALER`~~
Direction	Bidirectional
Send/receive pattern	Receive, Send, Receive, Send …
Outgoing routing strategy	Fair-robin
Incoming routing strategy	Last peer

~~DEALER socket~~

~~The DEALER socket type talks to a set of anonymous peers~~, sending and receiving messages using round-robin algorithms. It is reliable, insofar as it does not drop messages. DEALER works as an asynchronous replacement for REQ for clients that talk to REP or ROUTER servers. Message received by a DEALER are fair-queued from all connected peers.

When a DEALER socket enters the mute state due to having reached the high water mark for all peers, or if there are no peers at all, then any *send* operation on the socket will block until the mute state ends or at least one peer becomes available for sending; messages are not discarded.

When a DEALER socket is connected to a REP socket message sent must contain an empty frame as first part of the message (the delimiter), followed by one or more body parts.

Summary of characteristics:

Compatible peer sockets	ROUTER , REP , DEALER
Direction	Bidirectional
Send/receive pattern	Unrestricted
Outgoing routing strategy	Round-robin
Incoming routing strategy	Fair-queued
~~Action in mute state~~	Block

~~ROUTER socket~~

The ROUTER socket type talks to a set of peers, using explicit addressing so that each ~~outgoing message is sent to a specific peer connection. ROUTER works as an~~ asynchronous replacement for REP , and is often used as the basis for servers that talk to DEALER clients.

When receiving messages a ROUTER socket will prepend a message part containing the routing id of the originating peer to the message before passing it to the application. Messages received are fair-queued from among all connected peers. When sending messages a ROUTER socket will remove the first part of the message and use it to determine the *routing id* of the peer the message shall be routed to. If the peer does not exist anymore, or has never existed, the message shall be silently discarded.

When a ROUTER socket enters the mute state due to having reached the high water mark for all peers, then any messages sent to the socket will be dropped until the mute state ends. Likewise, any messages routed to a peer for which the individual high water mark has been reached will also be dropped.

~~When a `REQ` socket is connected to a~~ `ROUTER` socket, in addition to the *routing id* of the originating peer each message received shall contain an empty delimiter message part. Hence, the entire structure of each received message as seen by the application becomes: one or more routing id parts, delimiter part, one or more body parts. When sending replies to a `REQ` socket the application must include the delimiter part.

Summary of characteristics:

Compatible peer sockets	`DEALER` , `REQ` , `ROUTER`
Direction	Bidirectional
Send/receive pattern	Unrestricted
Outgoing routing strategy	See text
Incoming routing strategy	Fair-queued
Action in mute state	Drop (see text)

Publish-subscribe pattern

The publish-subscribe pattern is used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion.

The publish-subscribe pattern is formally defined by RFC 29/PUBSUB.

ZeroMQ comes with support for Pub/Sub by way of four socket types:

- `PUB` Socket Type
- ~~`XPUB` Socket Type~~
- `SUB` Socket Type
- ~~`XSUB` Socket Type~~

Topics

ZeroMQ uses multipart messages to convey topic information. Topics are expressed as an array of bytes, though you may use a string and with suitable text encoding.

A publisher must include the topic in the message's' first frame, prior to the message payload. For example, to publish a status message to subscribers of the status topic:

Copy

```
//  Send a message on the 'status' topic
zmq_send (pub, "status", 6, ZMQ_SNDMORE);
zmq_send (pub, "All is well", 11, 0);
```

Subscribers specify which topics they are interested in by setting the ZMQ_SUBSCRIBE option on the subscriber socket:

Copy

```
//  Subscribe to the 'status' topic
zmq_setsockopt (sub, ZMQ_SUBSCRIBE, "status", strlen ("status"));
```

A subscriber socket can have multiple subscription filters.

A message's topic is compared against subscribers' subscription topics using a prefix check.

That is, a subscriber who subscribed to `topic` would receive messages with topics:

- `topic`
- `topic/subtopic`
- `topical`

However it would not receive messages with topics:

- `topi`
- `TOPIC` (remember, it's a byte-wise comparison)

A consequence of this prefix matching behaviour is that you can receive all published messages by subscribing with an empty topic string.

PUB socket

A `PUB` socket is used by a publisher to distribute data. Messages sent are distributed in a fan out fashion to all connected peers. This socket type is not able to receive any messages.

When a `PUB` socket enters the mute state due to having reached the high water mark for a subscriber, then any messages that would be sent to the subscriber in question shall instead be dropped until the mute state ends. The send function does never block for this socket type.

Summary of characteristics:

Compatible peer sockets	`SUB` , ~~`XSUB`~~
Direction	Unidirectional
Send/receive pattern	Send only

Incoming routing strategy	N/A
Outgoing routing strategy	Fan out
Action in mute state	Drop

SUB socket

A `SUB` socket is used by a subscriber to subscribe to data distributed by a publisher. Initially a `SUB` socket is not subscribed to any messages. The `send` function is not implemented for this socket type.

Summary of characteristics:

Compatible peer sockets	`PUB` , `XPUB`
Direction	Unidirectional
Send/receive pattern	Receive only
Incoming routing strategy	Fair-queued
Outgoing routing strategy	N/A

XPUB socket

Same as `PUB` except that you can receive subscriptions from the peers in form of incoming messages. Subscription message is a byte 1 (for subscriptions) or byte 0 (for unsubscriptions) followed by the subscription body. Messages without a sub/unsub prefix are also received, but have no effect on subscription status.

Summary of characteristics:

Compatible peer sockets	ZMQ_SUB, ZMQ_XSUB
Direction	Unidirectional
Send/receive pattern	Send messages, receive subscriptions
Incoming routing strategy	N/A
Outgoing routing strategy	Fan out
Action in mute state	Drop

XSUB socket

Same as SUB except that you subscribe by sending subscription messages to the socket. Subscription message is a byte 1 (for subscriptions) or byte 0 (for unsubscriptions) followed by the subscription body. Messages without a sub/unsub prefix may also be sent, but have no effect on subscription status.

Summary of characteristics:

Compatible peer sockets	ZMQ_PUB, ZMQ_XPUB
Direction	Unidirectional
Send/receive pattern	Receive messages, send subscriptions
Incoming routing strategy	Fair-queued
Outgoing routing strategy	N/A
Action in mute state	Drop

Pipeline pattern

The pipeline pattern is intended for task distribution, typically in a multi-stage pipeline where one or a few nodes push work to many workers, and they in turn push results to one or a few collectors. The pattern is mostly reliable insofar as it will not discard messages unless a node disconnects unexpectedly. It is scalable in that nodes can join at any time.

The pipeline pattern is formally defined by RFC 30/PIPELINE.

ZeroMQ comes with support for pipelining by way of two socket types:

- PUSH Socket Type
- PULL Socket Type

PUSH socket

The PUSH socket type talks to a set of anonymous PULL peers, sending messages using a round-robin algorithm. The *receive* operation is not implemented for this socket type.

When a PUSH socket enters the mute state due to having reached the high water mark for all downstream nodes, or if there are no downstream nodes at all, then any *send* operations on the socket will block until the mute state ends or at least one downstream node becomes available for sending; messages are not discarded.

Summary of characteristics:

Compatible peer sockets	PULL
Direction	Unidirectional
Send/receive pattern	Send only
Incoming routing strategy	N/A
Outgoing routing strategy	Round-robin
Action in mute state	Block

PULL socket

The PULL socket type talks to a set of anonymous PUSH peers, receiving messages using a fair-queuing algorithm.

The *send* operation is not implemented for this socket type.

Summary of characteristics:

Compatible peer sockets	PUSH
Direction	Unidirectional
Send/receive pattern	Receive only
Incoming routing strategy	Fair-queued
Outgoing routing strategy	N/A
Action in mute state	Block

Exclusive pair pattern

PAIR is not a general-purpose socket but is intended for specific use cases where the two peers are architecturally stable. This usually limits PAIR to use within a single process, for inter-thread communication.

The exclusive pair pattern is formally defined by 31/EXPAIR.

PAIR socket

A socket of type `PAIR` can only be connected to a single peer at any one time. No message routing or filtering is performed on messages sent over a `PAIR` socket.

When a PAIR socket enters the mute state due to having reached the high water mark for the connected peer, or if no peer is connected, then any *send* operations on the socket will block until the peer becomes available for sending; messages are not discarded.

While PAIR sockets can be used over transports other than **inproc**, their inability to auto-reconnect coupled with the fact that new incoming connections will be terminated while any previous connections (including ones in a closing state) exist makes them unsuitable for TCP in most cases.

Client-server pattern

> **Note:** This pattern is still in draft state and thus might not be supported by the zeromq library you're using!

The client-server pattern is used to allow a single SERVER server talk to one or more CLIENT clients. The client always starts the conversation, after which either peer can send messages asynchronously, to the other.

The client-server pattern is formally defined by RFC 41/CLISRV.

CLIENT socket

The CLIENT socket type talks to one or more SERVER peers. If connected to multiple peers, it scatters sent messages among these peers in a round-robin fashion. On reading, it reads fairly, from each peer in turn. It is reliable, insofar as it does not drop messages in normal cases.

If the CLIENT socket has established a connection, *send* operations will accept messages, queue them, and send them as rapidly as the network allows. The outgoing buffer limit is defined by the high water mark for the socket. If the outgoing buffer is full, or if there is no connected peer, *send* operations will block, by default. The CLIENT socket will not drop messages.

Summary of characteristics:

Compatible peer sockets	SERVER
Direction	Bidirectional
Send/receive pattern	Unrestricted
Outgoing routing strategy	Round-robin

Incoming routing strategy	Fair-queued
Action in mute state	Block

SERVER socket

The SERVER socket type talks to zero or more CLIENT peers. Each outgoing message is sent to a specific peer CLIENT. A SERVER socket can only reply to an incoming message: the CLIENT peer must always initiate a conversation.

Each received message has a *routing_id* that is a 32-bit unsigned integer. To send a message to a given CLIENT peer the application must set the peer's *routing_id* on the message.

> Example `clisrv` is missing for `libzmq`. Would you like to contribute it? Then follow the steps below:
>
> Copy
>
> ```
>     git clone https://github.com/zeromq/zeromq.org
>     example_dir=content/docs/examples/c/libzmq
>     cd zeromq.org && mkdir -p $example_dir
>     [ -s $example_dir/index.md ] || cat >$example_dir/index.md <<'EOF
> ---
> headless: true
> ---
> EOF
>     cp archetypes/examples/clisrv.md
>     $example_dir/clisrv.md
> ```

If the *routing_id* is not specified, or does not refer to a connected client peer, the send call will fail. If the outgoing buffer for the client peer is full, the send call will block. The SERVER socket will not drop messages in any case.

Summary of characteristics:

Compatible peer sockets	CLIENT
Direction	Bidirectional
Send/receive pattern	Unrestricted

Outgoing routing strategy	See text
Incoming routing strategy	Fair-queued
Action in mute state	Fail

Radio-dish pattern

> **Note:** This pattern is still in draft state and thus might not be supported by the zeromq library you're using!

The radio-dish pattern is used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion.

Radio-dish is using groups (vs Pub-sub topics), Dish sockets can join a group and each message sent by Radio sockets belong to a group.

Groups are null terminated strings limited to 16 chars length (including null). The intention is to increase the length to 40 chars (including null). The encoding of groups shall be UTF8.

Groups are matched using exact matching (vs prefix matching of PubSub).

RADIO socket

A RADIO socket is used by a publisher to distribute data. Each message belong to a group. Messages are distributed to all members of a group. The *receive* operation is not implemented for this socket type.

> Example `raddsh_radio_example` is missing for `libzmq`. Would you like to contribute it? Then follow the steps below:
>
> Copy

```
    git clone https://github.com/zeromq/zeromq.org
    example_dir=content/docs/examples/c/libzmq
    cd zeromq.org && mkdir -p $example_dir
    [ -s $example_dir/index.md ] || cat >$example_dir/index.md <<'EOF
---
headless: true
---
EOF
```

```
    cp archetypes/examples/raddsh_radio_example.md
    $example_dir/raddsh_radio_example.md
```

When a RADIO socket enters the mute state due to having reached the high water mark for a subscriber, then any messages that would be sent to the subscriber in question will instead be dropped until the mute state ends. The *send* operation will never block for this socket type.

Summary of characteristics:

Compatible peer sockets	DISH
Direction	Unidirectional
Send/receive pattern	Send only
Incoming routing strategy	N/A
Outgoing routing strategy	Fan out
Action in mute state	Drop

DISH socket

A DISH socket is used by a subscriber to subscribe to groups distributed by a RADIO. Initially a DISH socket is not subscribed to any groups. The *send* operations are not implemented for this socket type.

Example `raddsh_dish_example` is missing for `libzmq`. Would you like to contribute it? Then follow the steps below:

Copy

```
    git clone https://github.com/zeromq/zeromq.org
    example_dir=content/docs/examples/c/libzmq
    cd zeromq.org && mkdir -p $example_dir
    [ -s $example_dir/index.md ] || cat >$example_dir/index.md <<'EOF
---
headless: true
---
EOF
```

```
    cp archetypes/examples/raddsh_dish_example.md
    $example_dir/raddsh_dish_example.md
```

Summary of characteristics:

Compatible peer sockets	RADIO
Direction	Unidirectional
Send/receive pattern	Receive only
Incoming routing strategy	Fair-queued
Outgoing routing strategy	N/A

Other

Messages

A ZeroMQ message is a discrete unit of data passed between applications or components of the same application. From the point of view of ZeroMQ itself messages are considered to be opaque binary data.

On the wire, ZeroMQ messages are blobs of any size from zero upwards that fit in memory. You do your own serialization using protocol buffers, msgpack, JSON, or whatever else your applications need to speak. It's wise to choose a data representation that is portable, but you can make your own decisions about trade-offs.

The simplest ZeroMQ message consist of one frame (also called message part). Frames are the basic wire format for ZeroMQ messages. A frame is a length-specified block of data. The length can be zero upwards. ZeroMQ guarantees to deliver all the parts (one or more) for a message, or none of them. This allows you to send or receive a list of frames as a single on-the-wire message.

A message (single or multipart) must fit in memory. If you want to send files of arbitrary sizes, you should break them into pieces and send each piece as separate single-part messages. Using multipart data will not reduce memory consumption.

Working with strings

Passing data as strings is usually the easiest way to get a communication going as serialization is rather straightforward. For ZeroMQ we established the rule that **strings are length-specified and are sent on the wire without a trailing null**.

The following function sends a string to a socket where the string's length equals frame's length.

Copy

```
static void
s_send_string (void *socket, const char *string) {
    zmq_send (socket, strdup(string), strlen(string), 0);
}
```

To read a string from a socket we have to provide a buffer and its length. The *zmq_recv* method write as much data into the buffer as possible. If there's more data it will get discarded. We use the returned frame's size to set appropriate null-terminator and return a duplicate of the retrieved string.

Copy

```
//  Receive string from socket and convert into C string
//  Chops string at 255 chars, if it's longer
static char *
s_recv_string (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;
    /* use strndup(buffer, sizeof(buffer)-1) in *nix */
    return strdup (buffer);
}
```

Because we utilise the frame's length to reflect the string's length we can send mulitple strings by putting each of them into a seperate frame.

The following function sends an array of string to a socket. The *ZMQ_SNDMORE* flag tells ZeroMQ to postpone sending until all frames are ready.

Copy

```
static void
s_send_strings (void *socket, const char[] *strings, int no_of_strings) {
    for (index = 0; index < no_of_strings; index++) {
        int FLAG = (index + 1) == no_of_strings ? 0 : ZMQ_SNDMORE;
        zmq_send (socket, strdup(strings[index]), strlen(strings[index]), F
    }
}
```

To retrieve a string frames from a multi-part messages we must use the *ZMQ_RCVMORE* `zmq_getsockopt()` option after calling `zmq_recv()` to determine if there are further parts to receive.

Copy

```
char *strings[25];
int rcvmore;
size_t option_len = sizeof (int);
int index = 0;
```

```
    do {
        strings[index++] = s_recv_string (socket);
        zmq_getsockopt (socket, ZMQ_RCVMORE, &rcvmore, &option_len);
    } while (rcvmore);
```

More

Find out more about working with messages:

- Initialise a message: zmq_msg_init(), zmq_msg_init_size(), zmq_msg_init_data().
- Sending and receiving a message: zmq_msg_send(), zmq_msg_recv().
- Release a message: zmq_msg_close().
- Access message content: zmq_msg_data(), zmq_msg_size(), zmq_msg_more().
- Work with message properties: zmq_msg_get(), zmq_msg_set().
- Message manipulation: zmq_msg_copy(), zmq_msg_move().