Search this site Search

Chapter Two

Table of Contents

Chapter 2 - Sockets and Patterns

top prev next

In Chapter 1 - Basics we took ZeroMQ for a drive, with some basic examples of the main ZeroMQ patterns: request-reply, pub-sub, and pipeline. In this chapter, we're going to get our hands dirty and start to learn how to use these tools in real programs.

We'll cover:

- How to create and work with ZeroMQ sockets.
- How to send and receive messages on sockets.
- How to build your apps around ZeroMQ's asynchronous I/O model.
- How to handle multiple sockets in one thread.
- How to handle fatal and nonfatal errors properly.
- How to handle interrupt signals like Ctrl-C.
- How to shut down a ZeroMQ application cleanly.
- How to check a ZeroMQ application for memory leaks.
- How to send and receive multipart messages.
- How to forward messages across networks.
- How to build a simple message queuing broker.
- How to write multithreaded applications with ZeroMQ.
- How to use ZeroMQ to signal between threads.
- How to use ZeroMQ to coordinate a network of nodes.
- How to create and use message envelopes for pub-sub.
- Using the HWM (high-water mark) to protect against memory overflows.

The Socket API

top prev next

To be perfectly honest, ZeroMQ does a kind of switch-and-bait on you, for which we don't apologize. It's for your own good and it hurts us more than it hurts you. ZeroMQ presents a familiar socket-based API, which requires great effort for us to hide a bunch of message-processing engines. However, the result will slowly fix your world view about how to design and write distributed software.

Sockets are the de facto standard API for network programming, as well as being useful for stopping your eyes from falling onto your cheeks. One thing that makes ZeroMQ especially tasty to developers is that it uses sockets and messages instead of some other arbitrary set of concepts. Kudos to Martin Sustrik for pulling this off. It turns "Message Oriented Middleware", a phrase guaranteed to send the whole room off to Catatonia, into "Extra Spicy Sockets!", which leaves us with a strange craving for pizza and a desire to know more.

Like a favorite dish, ZeroMQ sockets are easy to digest. 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 (see `zmq_socket()`, `zmq_close()`).
- Configuring sockets by setting options on them and checking them if necessary (see `zmq_setsockopt()`, `zmq_getsockopt()`).
- Plugging sockets into the network topology by creating ZeroMQ connections to and from them (see `zmq_bind()`, `zmq_connect()`).
- Using the sockets to carry data by writing and receiving messages on them (see `zmq_msg_send()`, `zmq_msg_recv()`).

Note that sockets are always void pointers, and messages (which we'll come to very soon) are structures. So in C you pass sockets as-such, but you pass addresses of messages in all functions that work with messages, like `zmq_msg_send()` and `zmq_msg_recv()`. As a mnemonic, realize that "in ZeroMQ, all your sockets are belong to us", but messages are things you actually own in your code.

Creating, destroying, and configuring sockets works as you'd expect for any object. But remember that ZeroMQ is an asynchronous, elastic fabric. This has some impact on how we plug sockets into the network topology and how we use the sockets after that.

Plugging Sockets into the Topology

top prev next

To create a connection between two nodes, you use `zmq_bind()` in one node and `zmq_connect()` in the other. As a general rule of thumb, the node that does `zmq_bind()` is a "server", sitting on a well-known network address, and the node which does `zmq_connect()` is a "client", with unknown or arbitrary network addresses. Thus we say that we "bind a socket to an endpoint" and "connect a socket to an endpoint", the endpoint being that well-known network address.

ZeroMQ connections are somewhat different from classic TCP connections. The main notable differences are:

- They go across an arbitrary transport (`inproc`, `ipc`, `tcp`, `pgm`, or `epgm`). See `zmq_inproc()`, `zmq_ipc()`, `zmq_tcp()`, `zmq_pgm()`, and `zmq_epgm()`.
- One socket may have many outgoing and many incoming connections.
- There is no `zmq_accept`() method. When a socket is bound to an endpoint it automatically starts accepting

connections.

- The network connection itself happens in the background, and ZeroMQ will automatically reconnect if the network connection is broken (e.g., if the peer disappears and then comes back).

- Your application code cannot work with these connections directly; they are encapsulated under the socket.

Many architectures follow some kind of client/server model, where the server is the component that is most static, and the clients are the components that are most dynamic, i.e., they come and go the most. There are sometimes issues of addressing: servers will be visible to clients, but not necessarily vice versa. So mostly it's obvious which node should be doing `zmq_bind()` (the server) and which should be doing `zmq_connect()` (the client). It also depends on the kind of sockets you're using, with some exceptions for unusual network architectures. We'll look at socket types later.

Now, imagine we start the client *before* we start the server. In traditional networking, we get a big red Fail flag. But ZeroMQ lets us start and stop pieces arbitrarily. As soon as the client node does `zmq_connect()`, the connection exists and that node can start to write messages to the socket. At some stage (hopefully before messages queue up so much that they start to get discarded, or the client blocks), the server comes alive, does a `zmq_bind()`, and ZeroMQ starts to deliver messages.

A server node can bind to many endpoints (that is, a combination of protocol and address) and it can do this using a single socket. This means it will accept connections across different transports:

```
zmq_bind (socket, "tcp://*:5555");
zmq_bind (socket, "tcp://*:9999");
zmq_bind (socket, "inproc://somename");
```

With most transports, you cannot bind to the same endpoint twice, unlike for example in UDP. The `ipc` transport does, however, let one process bind to an endpoint already used by a first process. It's meant to allow a process to recover after a crash.

Although ZeroMQ tries to be neutral about which side binds and which side connects, there are differences. We'll see these in more detail later. The upshot is that you should usually think in terms of "servers" as static parts of your topology that bind to more or less fixed endpoints, and "clients" as dynamic parts that come and go and connect to these endpoints. Then, design your application around this model. The chances that it will "just work" are much better like that.

Sockets have types. The socket type defines the semantics of the socket, its policies for routing messages inwards and outwards, queuing, etc. You can connect certain types of socket together, e.g., a publisher socket and a subscriber socket. Sockets work together in "messaging patterns". We'll look at this in more detail later.

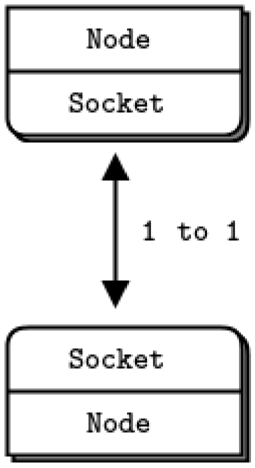
It's the ability to connect sockets in these different ways that gives ZeroMQ its basic power as a message queuing system. There are layers on top of this, such as proxies, which we'll get to later. But essentially, with ZeroMQ you define your network architecture by plugging pieces together like a child's construction toy.

Sending and Receiving Messages

top prev next

To send and receive messages you use the `zmq_msg_send()` and `zmq_msg_recv()` methods. The names are conventional, but ZeroMQ's I/O model is different enough from the classic TCP model that you will need time to get your head around it.

Figure 9 - TCP sockets are 1 to 1



Let's look at the main differences between TCP sockets and ZeroMQ sockets when it comes to working with data:

- ZeroMQ sockets carry messages, like UDP, rather than a stream of bytes as TCP does. A ZeroMQ message is length-specified binary data. We'll come to messages shortly; their design is optimized for performance and so a little tricky.
- ZeroMQ sockets do their I/O in a background thread. This means that messages arrive in local input queues and are sent from local output queues, no matter what your application is busy doing.
- ZeroMQ sockets have one-to-N routing behavior built-in, according to the socket type.

The `zmq_send()` method does not actually send the message to the socket connection(s). It queues the message so that the I/O thread can send it asynchronously. It does not block except in some exception cases. So the message is not necessarily sent when `zmq_send()` returns to your application.

Unicast Transports

top prev next

ZeroMQ provides a set of unicast transports (`inproc`, `ipc`, and `tcp`) and multicast transports (epgm, pgm). Multicast is an advanced technique that we'll come to later. Don't even start using it unless you know that your fan-out ratios will make 1-to-N unicast impossible.

For most common cases, use **`tcp`**, which is a *disconnected TCP* transport. It is elastic, portable, and fast enough for most cases. We call this disconnected because ZeroMQ's `tcp` transport doesn't require that the endpoint exists before you connect to it. Clients and servers can connect and bind at any time, can go and come back, and it remains transparent to applications.

The inter-process `ipc` transport is disconnected, like `tcp`. It has one limitation: it does not yet work on Windows. By convention we use endpoint names with an ".ipc" extension to avoid potential conflict with other file names. On UNIX systems, if you use `ipc` endpoints you need to create these with appropriate permissions otherwise they may not be shareable between processes running under different user IDs. You must also make sure all processes can access the files, e.g., by running in the same working directory.

The inter-thread transport, **`inproc`**, is a connected signaling transport. It is much faster than `tcp` or `ipc`. This transport has a specific limitation compared to `tcp` and `ipc`: **the server must issue a bind before any client issues a connect**. This is something future versions of ZeroMQ may fix, but at present this defines how you use `inproc` sockets. We create and bind one socket and start the child threads, which create and connect the other sockets.

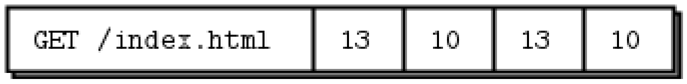
ZeroMQ is Not a Neutral Carrier

top prev next

A common question that newcomers to ZeroMQ ask (it's one I've asked myself) is, "how do I write an XYZ server in ZeroMQ?" For example, "how do I write an HTTP server in ZeroMQ?" The implication is that if we use normal sockets to carry HTTP requests and responses, we should be able to use ZeroMQ sockets to do the same, only much faster and better.

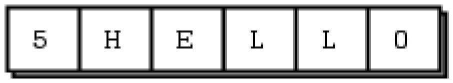
The answer used to be "this is not how it works". ZeroMQ is not a neutral carrier: it imposes a framing on the transport protocols it uses. This framing is not compatible with existing protocols, which tend to use their own framing. For example, compare an HTTP request and a ZeroMQ request, both over TCP/IP.

Figure 10 - HTTP on the Wire



The HTTP request uses CR-LF as its simplest framing delimiter, whereas ZeroMQ uses a length-specified frame. So you could write an HTTP-like protocol using ZeroMQ, using for example the request-reply socket pattern. But it would not be HTTP.

Figure 11 - ZeroMQ on the Wire



Since v3.3, however, ZeroMQ has a socket option called `ZMQ_ROUTER_RAW` that lets you read and write data without the ZeroMQ framing. You could use this to read and write proper HTTP requests and responses. Hardeep Singh contributed this change so that he could connect to Telnet servers from his ZeroMQ application. At time of writing this is still somewhat experimental, but it shows how ZeroMQ keeps evolving to solve new problems. Maybe the next patch will be yours.

I/O Threads

top prev next

We said that ZeroMQ does I/O in a background thread. One I/O thread (for all sockets) is sufficient for all but the most extreme applications. When you create a new context, it starts with one I/O thread. The general rule of thumb is to allow one I/O thread per gigabyte of data in or out per second. To raise the number of I/O threads, use the `zmq_ctx_set()` call *before* creating any sockets:

```
int io_threads = 4;
void *context = zmq_ctx_new ();
zmq_ctx_set (context, ZMQ_IO_THREADS, io_threads);
assert (zmq_ctx_get (context, ZMQ_IO_THREADS) == io_threads);
```

We've seen that one socket can handle dozens, even thousands of connections at once. This has a fundamental impact on how you write applications. A traditional networked application has one process or one thread per remote connection, and that process or thread handles one socket. ZeroMQ lets you collapse this entire structure into a single process and then break it up as necessary for scaling.

If you are using ZeroMQ for inter-thread communications only (i.e., a multithreaded application that does no external socket I/O) you can set the I/O threads to zero. It's not a significant optimization though, more of a curiosity.

Messaging Patterns

top prev next

Underneath the brown paper wrapping of ZeroMQ's socket API lies the world of messaging patterns. If you have a background in enterprise messaging, or know UDP well, these will be vaguely familiar. But to most ZeroMQ newcomers, they are a surprise. We're so used to the TCP paradigm where a socket maps one-to-one to another node.

Let's recap briefly what ZeroMQ does for you. It delivers blobs of data (messages) to nodes, quickly and efficiently. You can map nodes to threads, processes, or nodes. ZeroMQ gives your applications a single socket API to work with, no matter what the actual transport (like in-process, inter-process, TCP, or multicast). It automatically reconnects to peers as they come and go. It queues messages at both sender and receiver, as needed. It limits these queues to guard processes against running out of memory. It handles socket errors. It does all I/O in background threads. It uses lock-free techniques for talking between nodes, so there are never locks, waits, semaphores, or deadlocks.

But cutting through that, it routes and queues messages according to precise recipes called *patterns*. It is these patterns that provide ZeroMQ's intelligence. They encapsulate our hard-earned experience of the best ways to distribute data and work. ZeroMQ's patterns are hard-coded but future versions may allow user-definable patterns.

ZeroMQ patterns are implemented by pairs of sockets with matching types. In other words, to understand ZeroMQ patterns you need to understand socket types and how they work together. Mostly, this just takes study; there is little that is obvious at this level.

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.

We looked at the first three of these in Chapter 1 - Basics, and we'll see the exclusive pair pattern later in this chapter. The `zmq_socket()` man page is fairly clear about the patterns — it's worth reading several times until it starts to make sense. These are the socket combinations that are valid for a connect-bind pair (either side can bind):

- PUB and SUB
- REQ and REP
- REQ and ROUTER (take care, REQ inserts an extra null frame)
- DEALER and REP (take care, REP assumes a null frame)
- DEALER and ROUTER
- DEALER and DEALER
- ROUTER and ROUTER
- PUSH and PULL
- PAIR and PAIR

You'll also see references to XPUB and XSUB sockets, which we'll come to later (they're like raw versions of PUB and SUB). Any other combination will produce undocumented and unreliable results, and future versions of ZeroMQ will probably return errors if you try them. You can and will, of course, bridge other socket types via code, i.e., read from one socket type and write to another.

High-Level Messaging Patterns

top prev next

These four core patterns are cooked into ZeroMQ. They are part of the ZeroMQ API, implemented in the core C++ library, and are guaranteed to be available in all fine retail stores.

On top of those, we add *high-level messaging patterns*. We build these high-level patterns on top of ZeroMQ and implement them in whatever language we're using for our application. They are not part of the core library, do not come with the ZeroMQ package, and exist in their own space as part of the ZeroMQ community. For example the Majordomo pattern, which we explore in Chapter 4 - Reliable Request-Reply Patterns, sits in the GitHub Majordomo project in the ZeroMQ organization.

One of the things we aim to provide you with in this book are a set of such high-level patterns, both small (how to handle messages sanely) and large (how to make a reliable pub-sub architecture).

Working with Messages

top prev next

The `libzmq` core library has in fact two APIs to send and receive messages. The `zmq_send()` and `zmq_recv()` methods that we've already seen and used are simple one-liners. We will use these often, but `zmq_recv()` is bad at dealing with arbitrary message sizes: it truncates messages to whatever buffer size you provide. So there's a second API that works with zmq_msg_t structures, with a richer but more difficult API:

- 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()`.

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.

In memory, ZeroMQ messages are `zmq_msg_t` structures (or classes depending on your language). Here are the basic ground rules for using ZeroMQ messages in C:

- You create and pass around `zmq_msg_t` objects, not blocks of data.

- To read a message, you use `zmq_msg_init()` to create an empty message, and then you pass that to `zmq_msg_recv()`.

- To write a message from new data, you use `zmq_msg_init_size()` to create a message and at the same time allocate a block of data of some size. You then fill that data using `memcpy`, and pass the message to `zmq_msg_send()`.

- To release (not destroy) a message, you call `zmq_msg_close()`. This drops a reference, and eventually ZeroMQ will destroy the message.

- To access the message content, you use `zmq_msg_data()`. To know how much data the message contains, use `zmq_msg_size()`.

- Do not use `zmq_msg_move()`, `zmq_msg_copy()`, or `zmq_msg_init_data()` unless you read the man pages and know precisely why you need these.

- After you pass a message to `zmq_msg_send()`, ØMQ will clear the message, i.e., set the size to zero. You cannot send the same message twice, and you cannot access the message data after sending it.

- These rules don't apply if you use `zmq_send()` and `zmq_recv()`, to which you pass byte arrays, not message structures.

If you want to send the same message more than once, and it's sizable, create a second message, initialize it using `zmq_msg_init()`, and then use `zmq_msg_copy()` to create a copy of the first message. This does not copy the data but copies a reference. You can then send the message twice (or more, if you create more copies) and the message will only be finally destroyed when the last copy is sent or closed.

ZeroMQ also supports *multipart* messages, which let you send or receive a list of frames as a single on-the-wire message. This is widely used in real applications and we'll look at that later in this chapter and in Chapter 3 - Advanced Request-Reply Patterns.

Frames (also called "message parts" in the ZeroMQ reference manual pages) are the basic wire format for ZeroMQ messages. A frame is a length-specified block of data. The length can be zero upwards. If you've done any TCP programming you'll appreciate why frames are a useful answer to the question "how much data am I supposed to read of this network socket now?"

There is a wire-level protocol called ZMTP that defines how ZeroMQ reads and writes frames on a TCP connection. If you're interested in how this works, the spec is quite short.

Originally, a ZeroMQ message was one frame, like UDP. We later extended this with multipart messages, which are quite simply series of frames with a "more" bit set to one, followed by one with that bit set to zero. The ZeroMQ API then lets you write messages with a "more" flag and when you read messages, it lets you check if there's "more".

In the low-level ZeroMQ API and the reference manual, therefore, there's some fuzziness about messages versus frames. So here's a useful lexicon:

- A message can be one or more parts.
- These parts are also called "frames".
- Each part is a `zmq_msg_t` object.
- You send and receive each part separately, in the low-level API.
- Higher-level APIs provide wrappers to send entire multipart messages.

Some other things that are worth knowing about messages:

- You may send zero-length messages, e.g., for sending a signal from one thread to another.
- ZeroMQ guarantees to deliver all the parts (one or more) for a message, or none of them.
- ZeroMQ does not send the message (single or multipart) right away, but at some indeterminate later time. A multipart message must therefore fit in memory.
- 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.*
- You must call `zmq_msg_close()` when finished with a received message, in languages that don't automatically destroy objects when a scope closes. You don't call this method after sending a message.

And to be repetitive, do not use `zmq_msg_init_data()` yet. This is a zero-copy method and is guaranteed to create trouble for you. There are far more important things to learn about ZeroMQ before you start to worry about shaving off microseconds.

This rich API can be tiresome to work with. The methods are optimized for performance, not simplicity. If you start using these you will almost definitely get them wrong until you've read the man pages with some care. So one of the main jobs of a good language binding is to wrap this API up in classes that are easier to use.

Handling Multiple Sockets

top prev next

In all the examples so far, the main loop of most examples has been:

1. Wait for message on socket.
2. Process message.
3. Repeat.

What if we want to read from multiple endpoints at the same time? The simplest way is to connect one socket to all the endpoints and get ZeroMQ to do the fan-in for us. This is legal if the remote endpoints are in the same pattern, but it would be wrong to connect a PULL socket to a PUB endpoint.

To actually read from multiple sockets all at once, use `zmq_poll()`. An even better way might be to wrap `zmq_poll()` in a framework that turns it into a nice event-driven *reactor*, but it's significantly more work than we want to cover here.

Let's start with a dirty hack, partly for the fun of not doing it right, but mainly because it lets me show you how to do nonblocking socket reads. Here is a simple example of reading from two sockets using nonblocking reads. This rather confused program acts both as a subscriber to weather updates, and a worker for parallel tasks:

msreader: Multiple socket reader in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Java | Lua | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Haskell | Haxe | Node.js | ooc | Q | Racket

The cost of this approach is some additional latency on the first message (the sleep at the end of the loop, when there are no waiting messages to process). This would be a problem in applications where submillisecond latency was vital. Also, you need to check the documentation for nanosleep() or whatever function you use to make sure it does not busy-loop.

You can treat the sockets fairly by reading first from one, then the second rather than prioritizing them as we did in this example.

Now let's see the same senseless little application done right, using `zmq_poll()`:

mspoller: Multiple socket poller in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Haxe | ooc | Q | Racket

The items structure has these four members:

```
typedef struct {
    void *socket;       //  ZeroMQ socket to poll on
    int fd;             //  OR, native file handle to poll on
    short events;       //  Events to poll on
    short revents;      //  Events returned after poll
} zmq_pollitem_t;
```

Multipart Messages

top prev next

ZeroMQ lets us compose a message out of several frames, giving us a "multipart message". Realistic applications use multipart messages heavily, both for wrapping messages with address information and for simple serialization. We'll look at reply envelopes later.

What we'll learn now is simply how to blindly and safely read and write multipart messages in any application (such as a proxy) that needs to forward messages without inspecting them.

When you work with multipart messages, each part is a `zmq_msg` item. E.g., if you are sending a message with five parts, you must construct, send, and destroy five `zmq_msg` items. You can do this in advance (and store the `zmq_msg` items in an array or other structure), or as you send them, one-by-one.

Here is how we send the frames in a multipart message (we receive each frame into a message object):

```
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
…
zmq_msg_send (&message, socket, ZMQ_SNDMORE);
…
zmq_msg_send (&message, socket, 0);
```

Here is how we receive and process all the parts in a message, be it single part or multipart:

```
while (1) {
    zmq_msg_t message;
    zmq_msg_init (&message);
    zmq_msg_recv (&message, socket, 0);
    //  Process the message frame
    …
    zmq_msg_close (&message);
    if (!zmq_msg_more (&message))
        break;      //  Last message frame
}
```

Some things to know about multipart messages:

- When you send a multipart message, the first part (and all following parts) are only actually sent on the wire when you send the final part.
- If you are using `zmq_poll()`, when you receive the first part of a message, all the rest has also arrived.
- You will receive all parts of a message, or none at all.
- Each part of a message is a separate `zmq_msg` item.
- You will receive all parts of a message whether or not you check the more property.
- On sending, ZeroMQ queues message frames in memory until the last is received, then sends them all.
- There is no way to cancel a partially sent message, except by closing the socket.

Intermediaries and Proxies

top prev next

ZeroMQ aims for decentralized intelligence, but that doesn't mean your network is empty space in the middle. It's filled with message-aware infrastructure and quite often, we build that infrastructure with ZeroMQ. The ZeroMQ plumbing can range from tiny pipes to full-blown service-oriented brokers. The messaging industry calls this *intermediation*, meaning that the stuff in the middle deals with either side. In ZeroMQ, we call these proxies, queues, forwarders, device, or brokers, depending on the context.

This pattern is extremely common in the real world and is why our societies and economies are filled with intermediaries who have no other real function than to reduce the complexity and scaling costs of larger networks.

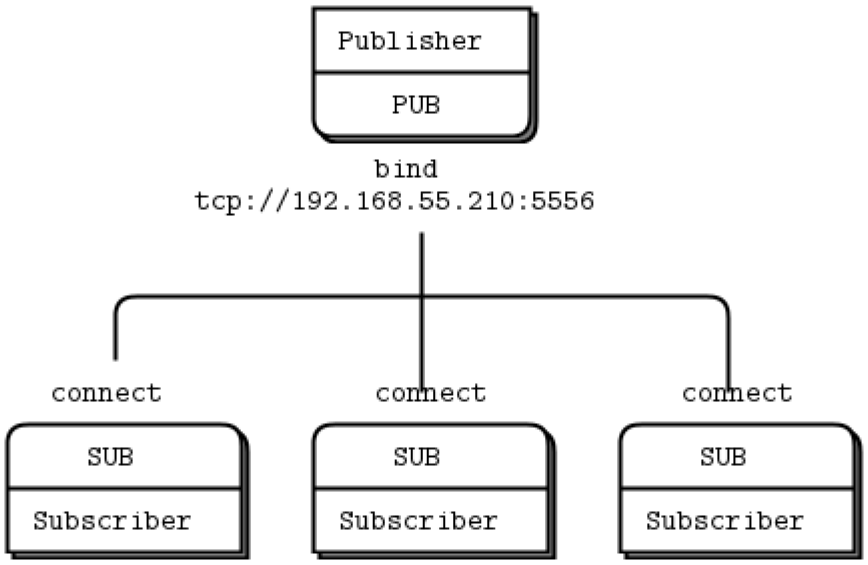
Real-world intermediaries are typically called wholesalers, distributors, managers, and so on.

The Dynamic Discovery Problem

One of the problems you will hit as you design larger distributed architectures is discovery. That is, how do pieces know about each other? It's especially difficult if pieces come and go, so we call this the "dynamic discovery problem".

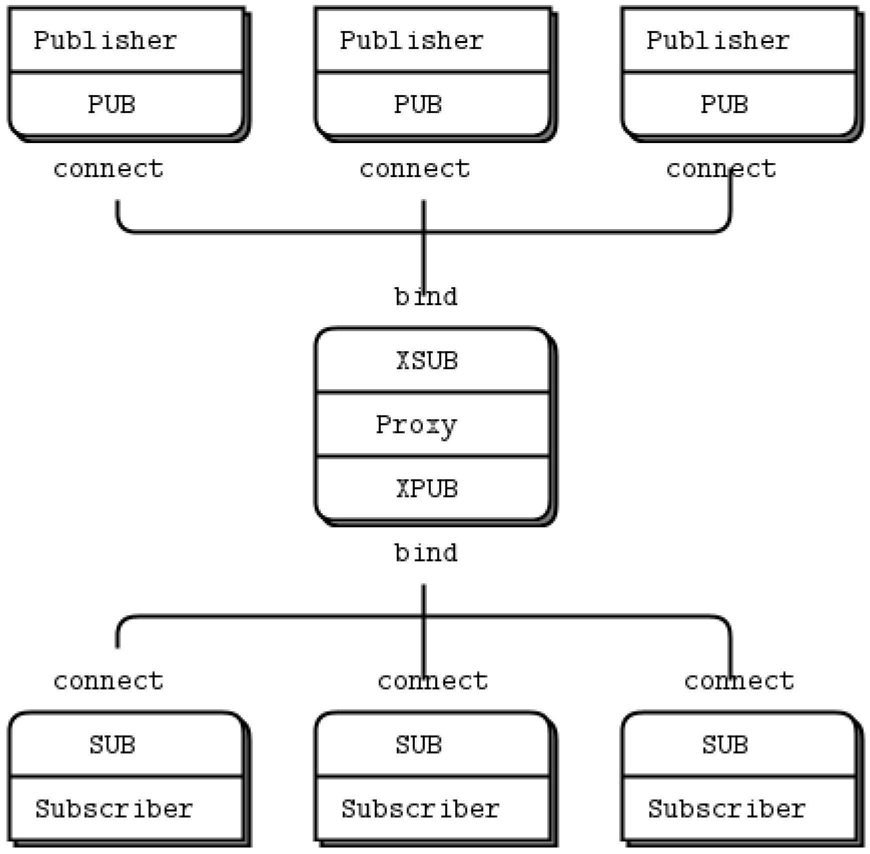
There are several solutions to dynamic discovery. The simplest is to entirely avoid it by hard-coding (or configuring) the network architecture so discovery is done by hand. That is, when you add a new piece, you reconfigure the network to know about it.

Figure 12 - Small-Scale Pub-Sub Network



In practice, this leads to increasingly fragile and unwieldy architectures. Let's say you have one publisher and a hundred subscribers. You connect each subscriber to the publisher by configuring a publisher endpoint in each subscriber. That's easy. Subscribers are dynamic; the publisher is static. Now say you add more publishers. Suddenly, it's not so easy any more. If you continue to connect each subscriber to each publisher, the cost of avoiding dynamic discovery gets higher and higher.

Figure 13 - Pub-Sub Network with a Proxy

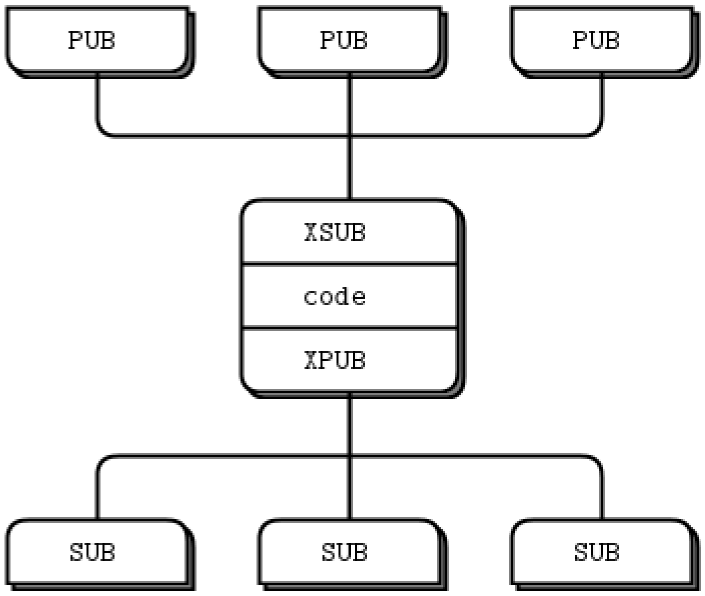


There are quite a few answers to this, but the very simplest answer is to add an intermediary; that is, a static point in the network to which all other nodes connect. In classic messaging, this is the job of the message broker. ZeroMQ doesn't come with a message broker as such, but it lets us build intermediaries quite easily.

You might wonder, if all networks eventually get large enough to need intermediaries, why don't we simply have a message broker in place for all applications? For beginners, it's a fair compromise. Just always use a star topology, forget about performance, and things will usually work. However, message brokers are greedy things; in their role as central intermediaries, they become too complex, too stateful, and eventually a problem.

It's better to think of intermediaries as simple stateless message switches. A good analogy is an HTTP proxy; it's there, but doesn't have any special role. Adding a pub-sub proxy solves the dynamic discovery problem in our example. We set the proxy in the "middle" of the network. The proxy opens an XSUB socket, an XPUB socket, and binds each to well-known IP addresses and ports. Then, all other processes connect to the proxy, instead of to each other. It becomes trivial to add more subscribers or publishers.

Figure 14 - Extended Pub-Sub



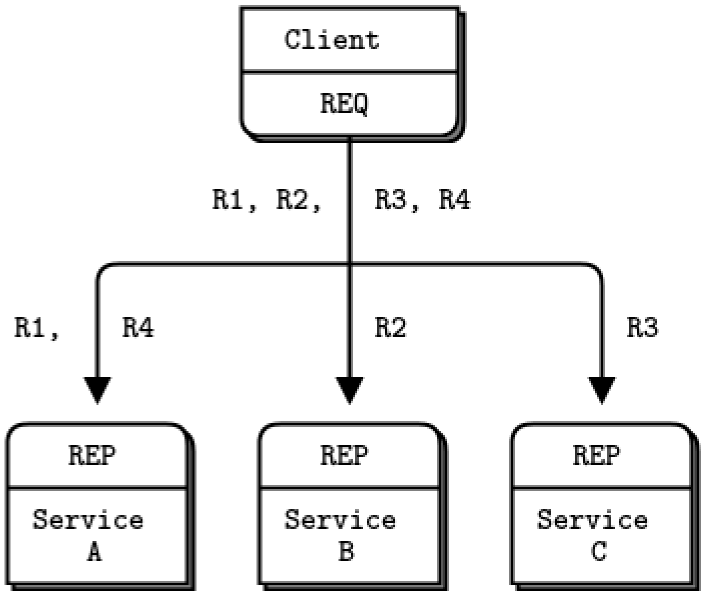
We need XPUB and XSUB sockets because ZeroMQ does subscription forwarding from subscribers to publishers. XSUB and XPUB are exactly like SUB and PUB except they expose subscriptions as special messages. The proxy has to forward these subscription messages from subscriber side to publisher side, by reading them from the XPUB socket and writing them to the XSUB socket. This is the main use case for XSUB and XPUB.

Shared Queue (DEALER and ROUTER sockets)

top prev next

In the Hello World client/server application, we have one client that talks to one service. However, in real cases we usually need to allow multiple services as well as multiple clients. This lets us scale up the power of the service (many threads or processes or nodes rather than just one). The only constraint is that services must be stateless, all state being in the request or in some shared storage such as a database.

Figure 15 - Request Distribution



There are two ways to connect multiple clients to multiple servers. The brute force way is to connect each client socket to multiple service endpoints. One client socket can connect to multiple service sockets, and the REQ socket will then distribute requests among these services. Let's say you connect a client socket to three service endpoints; A, B, and C. The client makes requests R1, R2, R3, R4. R1 and R4 go to service A, R2 goes to B, and R3 goes to service C.

This design lets you add more clients cheaply. You can also add more services. Each client will distribute its requests to the services. But each client has to know the service topology. If you have 100 clients and then you decide to add three more services, you need to reconfigure and restart 100 clients in order for the clients to know about the three new services.

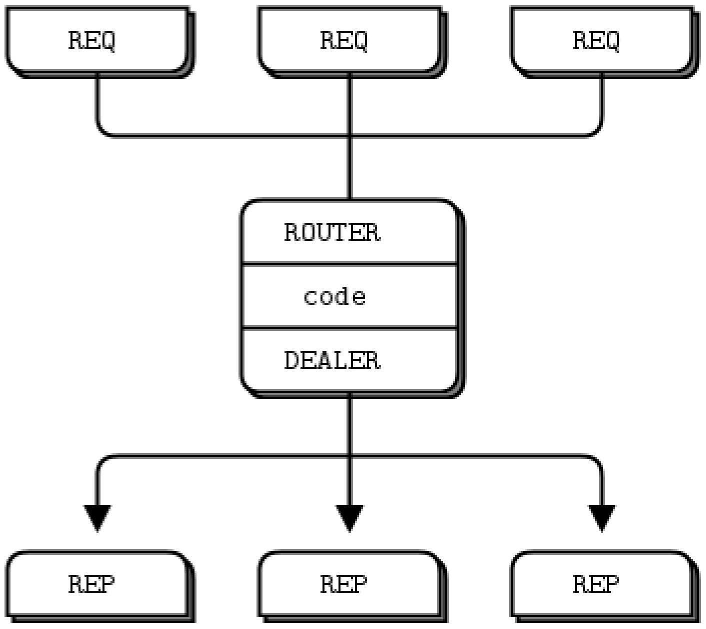
That's clearly not the kind of thing we want to be doing at 3 a.m. when our supercomputing cluster has run out of resources and we desperately need to add a couple of hundred of new service nodes. Too many static pieces are like liquid concrete: knowledge is distributed and the more static pieces you have, the more effort it is to change the topology. What we want is something sitting in between clients and services that centralizes all knowledge of the topology. Ideally, we should be able to add and remove services or clients at any time without touching any other part of the topology.

So we'll write a little message queuing broker that gives us this flexibility. The broker binds to two endpoints, a frontend for clients and a backend for services. It then uses `zmq_poll()` to monitor these two sockets for activity and when it has some, it shuttles messages between its two sockets. It doesn't actually manage any queues explicitly—ZeroMQ does that automatically on each socket.

When you use REQ to talk to REP, you get a strictly synchronous request-reply dialog. The client sends a request. The service reads the request and sends a reply. The client then reads the reply. If either the client or the service try to do anything else (e.g., sending two requests in a row without waiting for a response), they will get an error.

But our broker has to be nonblocking. Obviously, we can use `zmq_poll()` to wait for activity on either socket, but we can't use REP and REQ.

Figure 16 - Extended Request-Reply



Luckily, there are two sockets called DEALER and ROUTER that let you do nonblocking request-response. You'll see in Chapter 3 - Advanced Request-Reply Patterns how DEALER and ROUTER sockets let you build all kinds of asynchronous request-reply flows. For now, we're just going to see how DEALER and ROUTER let us extend REQ-REP across an intermediary, that is, our little broker.

In this simple extended request-reply pattern, REQ talks to ROUTER and DEALER talks to REP. In between the DEALER and ROUTER, we have to have code (like our broker) that pulls messages off the one socket and shoves them onto the other.

The request-reply broker binds to two endpoints, one for clients to connect to (the frontend socket) and one for workers to connect to (the backend). To test this broker, you will want to change your workers so they connect to the backend socket. Here is a client that shows what I mean:

rrclient: Request-reply client in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q

Here is the worker:

rrworker: Request-reply worker in PHP

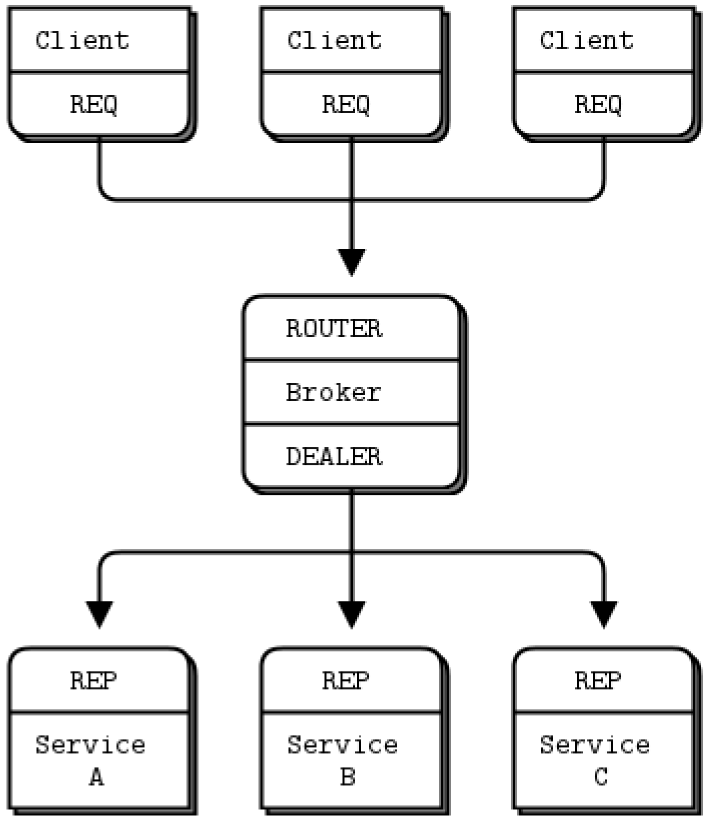
C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q

And here is the broker, which properly handles multipart messages:

rrbroker: Request-reply broker in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q | Racket

Figure 17 - Request-Reply Broker



Using a request-reply broker makes your client/server architectures easier to scale because clients don't see workers, and workers don't see clients. The only static node is the broker in the middle.

ZeroMQ's Built-In Proxy Function

top prev next

It turns out that the core loop in the previous section's `rrbroker` is very useful, and reusable. It lets us build pub-sub forwarders and shared queues and other little intermediaries with very little effort. ZeroMQ wraps this up in a single method, `zmq_proxy()`:

```
zmq_proxy (frontend, backend, capture);
```

The two (or three sockets, if we want to capture data) must be properly connected, bound, and configured. When we call the `zmq_proxy` method, it's exactly like starting the main loop of `rrbroker`. Let's rewrite the request-reply broker to call `zmq_proxy`, and re-badge this as an expensive-sounding "message queue" (people have charged houses for code that did less):

msgqueue: Message queue broker in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Q | Ruby | Tcl | Ada | Basic | Felix | Objective-C | ooc | Racket | Scala

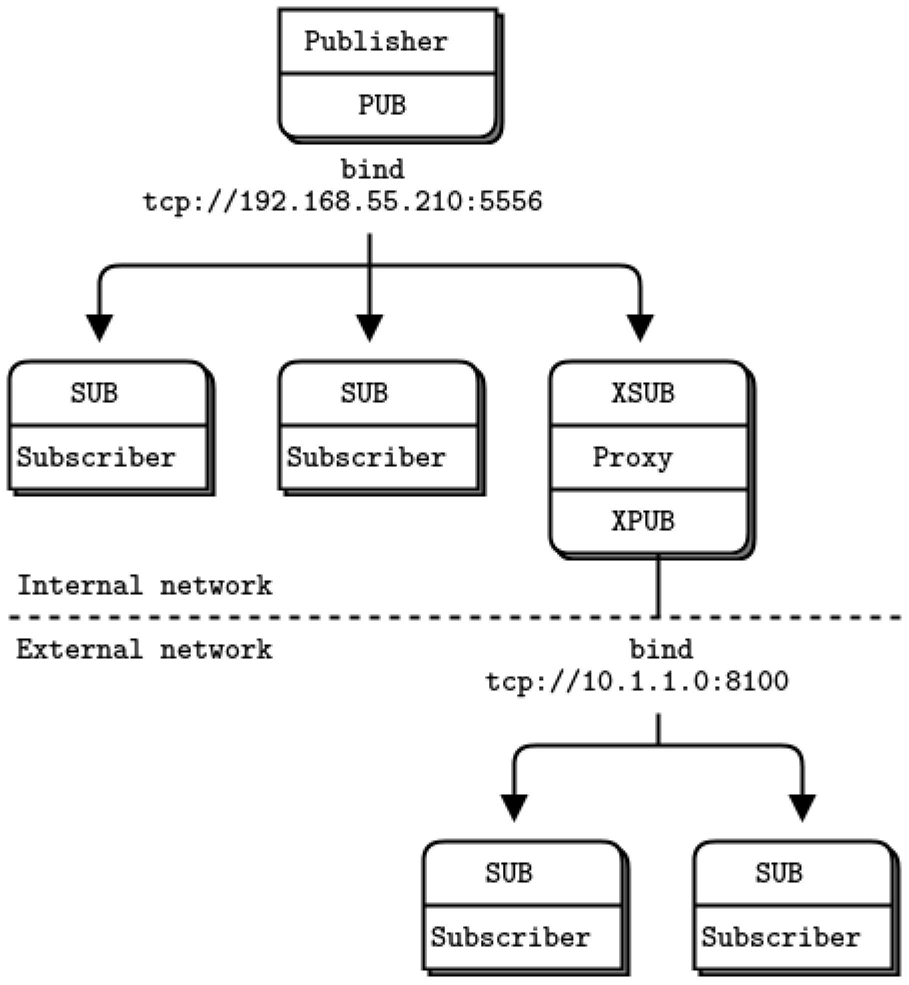
If you're like most ZeroMQ users, at this stage your mind is starting to think, "What kind of evil stuff can I do if I plug random socket types into the proxy?" The short answer is: try it and work out what is happening. In practice, you would usually stick to ROUTER/DEALER, XSUB/XPUB, or PULL/PUSH.

Transport Bridging

top prev next

A frequent request from ZeroMQ users is, "How do I connect my ZeroMQ network with technology X?" where X is some other networking or messaging technology.

Figure 18 - Pub-Sub Forwarder Proxy



The simple answer is to build a *bridge*. A bridge is a small application that speaks one protocol at one socket, and converts to/from a second protocol at another socket. A protocol interpreter, if you like. A common bridging problem in ZeroMQ is to bridge two transports or networks.

As an example, we're going to write a little proxy that sits in between a publisher and a set of subscribers, bridging two networks. The frontend socket (SUB) faces the internal network where the weather server is sitting, and the backend (PUB) faces subscribers on the external network. It subscribes to the weather service on the frontend socket, and republishes its data on the backend socket.

wuproxy: Weather update proxy in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q | Racket

It looks very similar to the earlier proxy example, but the key part is that the frontend and backend sockets are on two different networks. We can use this model for example to connect a multicast network (`pgm` transport) to a `tcp` publisher.

Handling Errors and ETERM

top prev next

ZeroMQ's error handling philosophy is a mix of fail-fast and resilience. Processes, we believe, should be as vulnerable as possible to internal errors, and as robust as possible against external attacks and errors. To give an analogy, a living cell will self-destruct if it detects a single internal error, yet it will resist attack from the outside by all means possible.

Assertions, which pepper the ZeroMQ code, are absolutely vital to robust code; they just have to be on the right side of the cellular wall. And there should be such a wall. If it is unclear whether a fault is internal or external, that is a design flaw to be fixed. In C/C++, assertions stop the application immediately with an error. In other languages, you may get exceptions or halts.

When ZeroMQ detects an external fault it returns an error to the calling code. In some rare cases, it drops messages silently if there is no obvious strategy for recovering from the error.

In most of the C examples we've seen so far there's been no error handling. **Real code should do error handling on every single ZeroMQ call**. If you're using a language binding other than C, the binding may handle errors for you. In C, you do need to do this yourself. There are some simple rules, starting with POSIX conventions:

- Methods that create objects return NULL if they fail.
- Methods that process data may return the number of bytes processed, or -1 on an error or failure.
- Other methods return 0 on success and -1 on an error or failure.
- The error code is provided in `errno` or `zmq_errno()`.
- A descriptive error text for logging is provided by `zmq_strerror()`.

For example:

```
void *context = zmq_ctx_new ();
assert (context);
void *socket = zmq_socket (context, ZMQ_REP);
assert (socket);
int rc = zmq_bind (socket, "tcp://*:5555");
if (rc == -1) {
    printf ("E: bind failed: %s\n", strerror (errno));
    return -1;
}
```

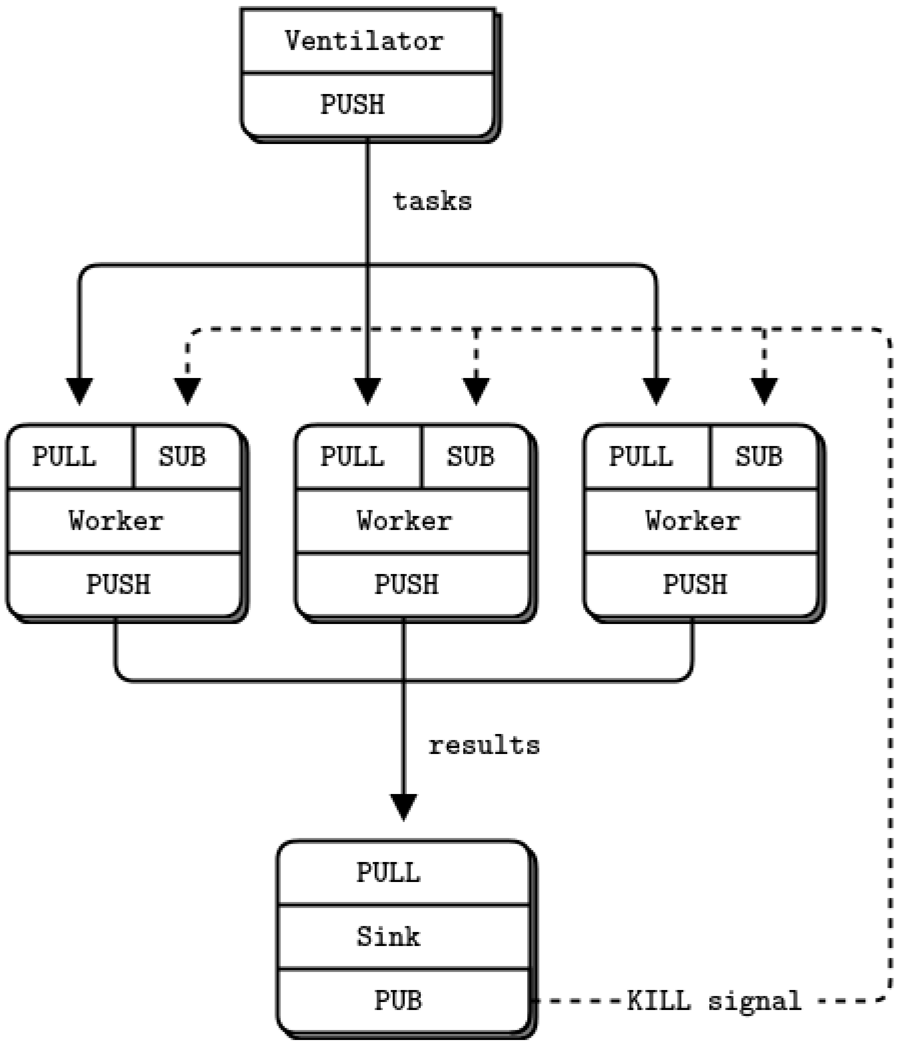
There are two main exceptional conditions that you should handle as nonfatal:

- When your code receives a message with the `ZMQ_DONTWAIT` option and there is no waiting data, ZeroMQ will return -1 and set `errno` to `EAGAIN`.

- When one thread calls `zmq_ctx_destroy()`, and other threads are still doing blocking work, the `zmq_ctx_destroy()` call closes the context and all blocking calls exit with -1, and `errno` set to `ETERM`.

In C/C++, asserts can be removed entirely in optimized code, so don't make the mistake of wrapping the whole ZeroMQ call in an `assert()`. It looks neat; then the optimizer removes all the asserts and the calls you want to make, and your application breaks in impressive ways.

Figure 19 - Parallel Pipeline with Kill Signaling



Let's see how to shut down a process cleanly. We'll take the parallel pipeline example from the previous section. If we've started a whole lot of workers in the background, we now want to kill them when the batch is finished. Let's do this by sending a kill message to the workers. The best place to do this is the sink because it really knows when the batch is done.

How do we connect the sink to the workers? The PUSH/PULL sockets are one-way only. We could switch to another socket type, or we could mix multiple socket flows. Let's try the latter: using a pub-sub model to send kill messages to the workers:

- The sink creates a PUB socket on a new endpoint.
- Workers connect their input socket to this endpoint.
- When the sink detects the end of the batch, it sends a kill to its PUB socket.

- When a worker detects this kill message, it exits.

It doesn't take much new code in the sink:

```
void *controller = zmq_socket (context, ZMQ_PUB);
zmq_bind (controller, "tcp://*:5559");
…
//  Send kill signal to workers
s_send (controller, "KILL");
```

Here is the worker process, which manages two sockets (a PULL socket getting tasks, and a SUB socket getting control commands), using the `zmq_poll()` technique we saw earlier:

taskwork2: Parallel task worker with kill signaling in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | ooc | Q | Racket

Here is the modified sink application. When it's finished collecting results, it broadcasts a kill message to all workers:

tasksink2: Parallel task sink with kill signaling in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | ooc | Q | Racket

Handling Interrupt Signals

top prev next

Realistic applications need to shut down cleanly when interrupted with Ctrl-C or another signal such as `SIGTERM`. By default, these simply kill the process, meaning messages won't be flushed, files won't be closed cleanly, and so on.

Here is how we handle a signal in various languages:

interrupt: Handling Ctrl-C cleanly in PHP

C | C++ | C# | Delphi | Erlang | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Ruby | Scala | Ada | Basic | Clojure | CL | F# | Felix | Objective-C | ooc | Q | Racket | Tcl

The program provides `s_catch_signals()`, which traps Ctrl-C (`SIGINT`) and `SIGTERM`. When either of these signals arrive, the `s_catch_signals()` handler sets the global variable `s_interrupted`. Thanks to your signal handler, your application will not die automatically. Instead, you have a chance to clean up and exit gracefully. You have to now explicitly check for an interrupt and handle it properly. Do this by calling `s_catch_signals()` (copy this from `interrupt.c`) at the start of your main code. This sets up the signal handling. The interrupt will affect ZeroMQ calls as follows:

- If your code is blocking in a blocking call (sending a message, receiving a message, or polling), then when a signal arrives, the call will return with `EINTR`.
- Wrappers like `s_recv()` return NULL if they are interrupted.

So check for an `EINTR` return code, a NULL return, and/or `s_interrupted`.

Here is a typical code fragment:

```
s_catch_signals ();
client = zmq_socket (...);
while (!s_interrupted) {
    char *message = s_recv (client);
    if (!message)
        break;          //  Ctrl-C used
}
zmq_close (client);
```

If you call `s_catch_signals()` and don't test for interrupts, then your application will become immune to Ctrl-C and `SIGTERM`, which may be useful, but is usually not.

Detecting Memory Leaks

top prev next

Any long-running application has to manage memory correctly, or eventually it'll use up all available memory and crash. If you use a language that handles this automatically for you, congratulations. If you program in C or C++ or any other language where you're responsible for memory management, here's a short tutorial on using valgrind, which among other things will report on any leaks your programs have.

- To install valgrind, e.g., on Ubuntu or Debian, issue this command:

```
sudo apt-get install valgrind
```

- By default, ZeroMQ will cause valgrind to complain a lot. To remove these warnings, create a file called `vg.supp` that contains this:

```
{
   <socketcall_sendto>
   Memcheck:Param
   socketcall.sendto(msg)
   fun:send
   ...
}
{
   <socketcall_sendto>
   Memcheck:Param
   socketcall.send(msg)
   fun:send
   ...
}
```

- Fix your applications to exit cleanly after Ctrl-C. For any application that exits by itself, that's not needed, but for long-running applications, this is essential, otherwise valgrind will complain about all currently allocated memory.
- Build your application with `-DDEBUG` if it's not your default setting. That ensures valgrind can tell you exactly where memory is being leaked.
- Finally, run valgrind thus:

```
valgrind --tool=memcheck --leak-check=full --suppressions=vg.supp someprog
```

And after fixing any errors it reported, you should get the pleasant message:

```
==30536== ERROR SUMMARY: 0 errors from 0 contexts...
```

Multithreading with ZeroMQ

top prev next

ZeroMQ is perhaps the nicest way ever to write multithreaded (MT) applications. Whereas ZeroMQ sockets require some readjustment if you are used to traditional sockets, ZeroMQ multithreading will take everything you know about writing MT applications, throw it into a heap in the garden, pour gasoline over it, and set it alight. It's a rare book that deserves burning, but most books on concurrent programming do.

To make utterly perfect MT programs (and I mean that literally), **we don't need mutexes, locks, or any other form of inter-thread communication except messages sent across ZeroMQ sockets.**

By "perfect MT programs", I mean code that's easy to write and understand, that works with the same design approach in any programming language, and on any operating system, and that scales across any number of CPUs with zero wait states and no point of diminishing returns.

If you've spent years learning tricks to make your MT code work at all, let alone rapidly, with locks and semaphores and critical sections, you will be disgusted when you realize it was all for nothing. If there's one lesson we've learned from 30+ years of concurrent programming, it is: *just don't share state*. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later, they're going to get into a fight. And the more drunkards you add to the table, the more they fight each other over the beer. The tragic majority of MT applications look like drunken bar fights.

The list of weird problems that you need to fight as you write classic shared-state MT code would be hilarious if it didn't translate directly into stress and risk, as code that seems to work suddenly fails under pressure. A large firm with world-beating experience in buggy code released its list of "11 Likely Problems In Your Multithreaded Code", which covers forgotten synchronization, incorrect granularity, read and write tearing, lock-free reordering, lock convoys, two-step dance, and priority inversion.

Yeah, we counted seven problems, not eleven. That's not the point though. The point is, do you really want that code running the power grid or stock market to start getting two-step lock convoys at 3 p.m. on a busy Thursday? Who cares what the terms actually mean? This is not what turned us on to programming, fighting ever more complex side effects with ever more complex hacks.

Some widely used models, despite being the basis for entire industries, are fundamentally broken, and shared state concurrency is one of them. Code that wants to scale without limit does it like the Internet does, by sending messages and sharing nothing except a common contempt for broken programming models.

You should follow some rules to write happy multithreaded code with ZeroMQ:

- Isolate data privately within its thread and never share data in multiple threads. The only exception to this are ZeroMQ contexts, which are threadsafe.
- Stay away from the classic concurrency mechanisms like as mutexes, critical sections, semaphores, etc. These are an anti-pattern in ZeroMQ applications.
- Create one ZeroMQ context at the start of your process, and pass that to all threads that you want to connect via `inproc` sockets.
- Use *attached* threads to create structure within your application, and connect these to their parent threads using PAIR sockets over `inproc`. The pattern is: bind parent socket, then create child thread which connects its socket.
- Use *detached* threads to simulate independent tasks, with their own contexts. Connect these over `tcp`. Later you can move these to stand-alone processes without changing the code significantly.
- All interaction between threads happens as ZeroMQ messages, which you can define more or less formally.
- Don't share ZeroMQ sockets between threads. ZeroMQ sockets are not threadsafe. Technically it's possible to migrate a socket from one thread to another but it demands skill. The only place where it's remotely sane to share sockets between threads are in language bindings that need to do magic like garbage collection on sockets.

If you need to start more than one proxy in an application, for example, you will want to run each in their own thread. It is easy to make the error of creating the proxy frontend and backend sockets in one thread, and then passing the sockets to the proxy in another thread. This may appear to work at first but will fail randomly in real use. Remember: *Do not use or close sockets except in the thread that created them.*

If you follow these rules, you can quite easily build elegant multithreaded applications, and later split off threads into separate processes as you need to. Application logic can sit in threads, processes, or nodes: whatever your scale needs.

ZeroMQ uses native OS threads rather than virtual "green" threads. The advantage is that you don't need to learn any new threading API, and that ZeroMQ threads map cleanly to your operating system. You can use standard tools like Intel's ThreadChecker to see what your application is doing. The disadvantages are that native threading APIs are not always portable, and that if you have a huge number of threads (in the thousands), some operating systems will get stressed.

Let's see how this works in practice. We'll turn our old Hello World server into something more capable. The original server ran in a single thread. If the work per request is low, that's fine: one ØMQ thread can run at full speed on a CPU core, with no waits, doing an awful lot of work. But realistic servers have to do nontrivial work per request. A single core may not be enough when 10,000 clients hit the server all at once. So a realistic server will start multiple worker threads. It then accepts requests as fast as it can and distributes these to its worker threads. The worker threads grind through the work and eventually send their replies back.

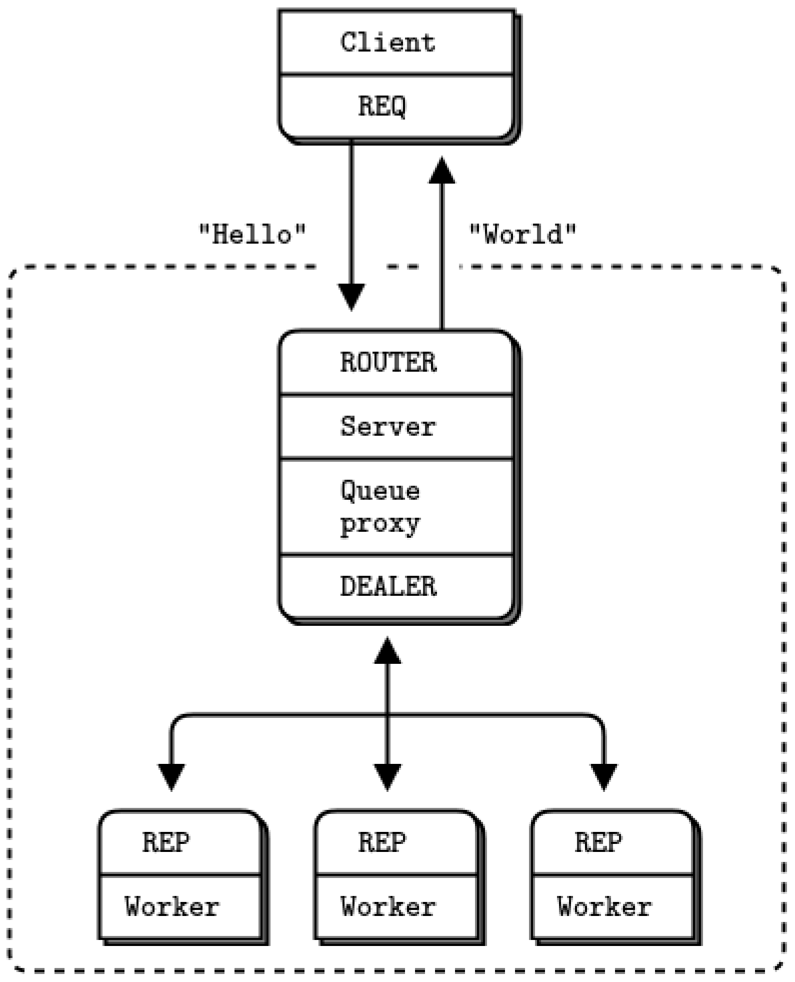
You can, of course, do all this using a proxy broker and external worker processes, but often it's easier to start one process that gobbles up sixteen cores than sixteen processes, each gobbling up one core. Further, running workers as threads will cut out a network hop, latency, and network traffic.

The MT version of the Hello World service basically collapses the broker and workers into a single process:

mtserver: Multithreaded service in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Perl | Python | Q | Ruby | Scala | Ada | Basic | Felix | Node.js | Objective-C | ooc | Racket | Tcl

Figure 20 - Multithreaded Server



All the code should be recognizable to you by now. How it works:

- The server starts a set of worker threads. Each worker thread creates a REP socket and then processes requests on this socket. Worker threads are just like single-threaded servers. The only differences are the transport (`inproc` instead of `tcp`), and the bind-connect direction.
- The server creates a ROUTER socket to talk to clients and binds this to its external interface (over `tcp`).
- The server creates a DEALER socket to talk to the workers and binds this to its internal interface (over `inproc`).
- The server starts a proxy that connects the two sockets. The proxy pulls incoming requests fairly from all clients, and distributes those out to workers. It also routes replies back to their origin.

Note that creating threads is not portable in most programming languages. The POSIX library is pthreads, but on Windows you have to use a different API. In our example, the `pthread_create` call starts up a new thread running the `worker_routine` function we defined. We'll see in Chapter 3 - Advanced Request-Reply Patterns how to wrap this in a portable API.

Here the "work" is just a one-second pause. We could do anything in the workers, including talking to other nodes. This is what the MT server looks like in terms of ØMQ sockets and nodes. Note how the request-reply chain is `REQ-ROUTER-queue-DEALER-REP`.

Signaling Between Threads (PAIR Sockets)

top prev next

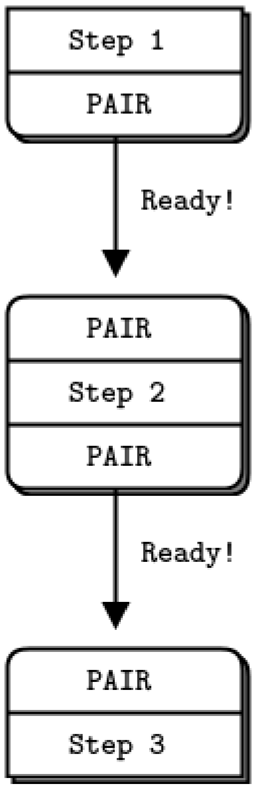
When you start making multithreaded applications with ZeroMQ, you'll encounter the question of how to coordinate your threads. Though you might be tempted to insert "sleep" statements, or use multithreading techniques such as semaphores or mutexes, **the only mechanism that you should use are ZeroMQ messages**. Remember the story of The Drunkards and The Beer Bottle.

Let's make three threads that signal each other when they are ready. In this example, we use PAIR sockets over the `inproc` transport:

mtrelay: Multithreaded relay in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Perl | Python | Q | Ruby | Scala | Ada | Basic | Felix | Node.js | Objective-C | ooc | Racket | Tcl

Figure 21 - The Relay Race



This is a classic pattern for multithreading with ZeroMQ:

1. Two threads communicate over `inproc`, using a shared context.
2. The parent thread creates one socket, binds it to an `inproc://` endpoint, and *then* starts the child thread, passing the context to it.
3. The child thread creates the second socket, connects it to that `inproc://` endpoint, and *then* signals to the parent thread that it's ready.

Note that multithreading code using this pattern is not scalable out to processes. If you use `inproc` and socket pairs, you are building a tightly-bound application, i.e., one where your threads are structurally interdependent. Do this when low latency is really vital. The other design pattern is a loosely bound application, where threads have their own context and communicate over `ipc` or `tcp`. You can easily break loosely bound threads into separate processes.

This is the first time we've shown an example using PAIR sockets. Why use PAIR? Other socket combinations might seem to work, but they all have side effects that could interfere with signaling:

- You can use PUSH for the sender and PULL for the receiver. This looks simple and will work, but remember that PUSH will distribute messages to all available receivers. If you by accident start two receivers (e.g., you already have one running and you start a second), you'll "lose" half of your signals. PAIR has the advantage of refusing more than one connection; the pair is *exclusive*.

- You can use DEALER for the sender and ROUTER for the receiver. ROUTER, however, wraps your message in an "envelope", meaning your zero-size signal turns into a multipart message. If you don't care about the data and treat anything as a valid signal, and if you don't read more than once from the socket, that won't matter. If, however, you decide to send real data, you will suddenly find ROUTER providing you with "wrong" messages. DEALER also distributes outgoing messages, giving the same risk as PUSH.

- You can use PUB for the sender and SUB for the receiver. This will correctly deliver your messages exactly as you sent them and PUB does not distribute as PUSH or DEALER do. However, you need to configure the subscriber with an empty subscription, which is annoying.

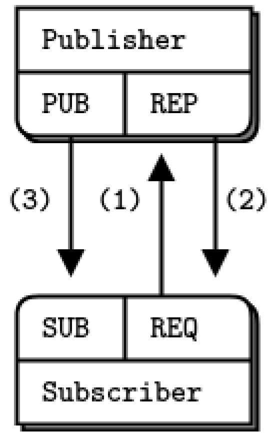
For these reasons, PAIR makes the best choice for coordination between pairs of threads.

Node Coordination

top prev next

When you want to coordinate a set of nodes on a network, PAIR sockets won't work well any more. This is one of the few areas where the strategies for threads and nodes are different. Principally, nodes come and go whereas threads are usually static. PAIR sockets do not automatically reconnect if the remote node goes away and comes back.

Figure 22 - Pub-Sub Synchronization



The second significant difference between threads and nodes is that you typically have a fixed number of threads but a more variable number of nodes. Let's take one of our earlier scenarios (the weather server and clients) and use node coordination to ensure that subscribers don't lose data when starting up.

This is how the application will work:

- The publisher knows in advance how many subscribers it expects. This is just a magic number it gets from somewhere.
- The publisher starts up and waits for all subscribers to connect. This is the node coordination part. Each subscriber subscribes and then tells the publisher it's ready via another socket.
- When the publisher has all subscribers connected, it starts to publish data.

In this case, we'll use a REQ-REP socket flow to synchronize subscribers and publisher. Here is the publisher:

syncpub: Synchronized publisher in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q

And here is the subscriber:

syncsub: Synchronized subscriber in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Racket | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q

This Bash shell script will start ten subscribers and then the publisher:

```
echo "Starting subscribers..."
for ((a=0; a<10; a++)); do
    syncsub &
done
echo "Starting publisher..."
syncpub
```

Which gives us this satisfying output:

```
Starting subscribers...
Starting publisher...
Received 1000000 updates
Received 1000000 updates
...
Received 1000000 updates
Received 1000000 updates
```

We can't assume that the SUB connect will be finished by the time the REQ/REP dialog is complete. There are no guarantees that outbound connects will finish in any order whatsoever, if you're using any transport except `inproc`. So, the example does a brute force sleep of one second between subscribing, and sending the REQ/REP synchronization.

A more robust model could be:

- Publisher opens PUB socket and starts sending "Hello" messages (not data).
- Subscribers connect SUB socket and when they receive a Hello message they tell the publisher via a REQ/REP socket pair.
- When the publisher has had all the necessary confirmations, it starts to send real data.

Zero-Copy

top prev next

ZeroMQ's message API lets you send and receive messages directly from and to application buffers without copying data. We call this *zero-copy*, and it can improve performance in some applications.

You should think about using zero-copy in the specific case where you are sending large blocks of memory (thousands of bytes), at a high frequency. For short messages, or for lower message rates, using zero-copy will make your code messier and more complex with no measurable benefit. Like all optimizations, use this when you know it helps, and *measure* before and after.

To do zero-copy, you use `zmq_msg_init_data()` to create a message that refers to a block of data already allocated with `malloc()` or some other allocator, and then you pass that to `zmq_msg_send()`. When you create the message, you also pass a function that ZeroMQ will call to free the block of data, when it has finished sending the message. This is the simplest example, assuming `buffer` is a block of 1,000 bytes allocated on the heap:

```
void my_free (void *data, void *hint) {
    free (data);
}
//  Send message from buffer, which we allocate and ZeroMQ will free for us
zmq_msg_t message;
zmq_msg_init_data (&message, buffer, 1000, my_free, NULL);
zmq_msg_send (&message, socket, 0);
```

Note that you don't call `zmq_msg_close()` after sending a message—`libzmq` will do this automatically when it's actually done sending the message.

There is no way to do zero-copy on receive: ZeroMQ delivers you a buffer that you can store as long as you wish, but it will not write data directly into application buffers.

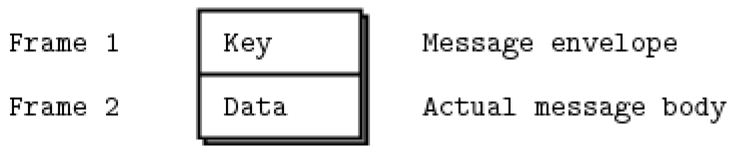
On writing, ZeroMQ's multipart messages work nicely together with zero-copy. In traditional messaging, you need to marshal different buffers together into one buffer that you can send. That means copying data. With ZeroMQ, you can send multiple buffers coming from different sources as individual message frames. Send each field as a length-delimited frame. To the application, it looks like a series of send and receive calls. But internally, the multiple parts get written to the network and read back with single system calls, so it's very efficient.

Pub-Sub Message Envelopes

top prev next

In the pub-sub pattern, we can split the key into a separate message frame that we call an *envelope*. If you want to use pub-sub envelopes, make them yourself. It's optional, and in previous pub-sub examples we didn't do this. Using a pub-sub envelope is a little more work for simple cases, but it's cleaner especially for real cases, where the key and the data are naturally separate things.

Figure 23 - Pub-Sub Envelope with Separate Key



Subscriptions do a prefix match. That is, they look for "all messages starting with XYZ". The obvious question is: how to delimit keys from data so that the prefix match doesn't accidentally match data. The best answer is to use an envelope because the match won't cross a frame boundary. Here is a minimalist example of how pub-sub envelopes look in code. This publisher sends messages of two types, A and B.

The envelope holds the message type:

psenvpub: Pub-Sub envelope publisher in PHP

C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q | Racket

The subscriber wants only messages of type B:

psenvsub: Pub-Sub envelope subscriber in PHP

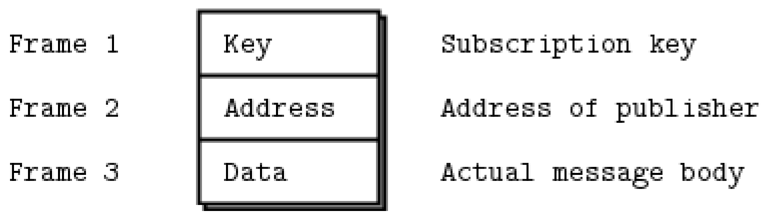
C | C++ | C# | Clojure | CL | Delphi | Erlang | F# | Go | Haskell | Haxe | Java | Lua | Node.js | Perl | Python | Ruby | Scala | Tcl | Ada | Basic | Felix | Objective-C | ooc | Q | Racket

When you run the two programs, the subscriber should show you this:

```
[B] We would like to see this
[B] We would like to see this
[B] We would like to see this
...
```

This example shows that the subscription filter rejects or accepts the entire multipart message (key plus data). You won't get part of a multipart message, ever. If you subscribe to multiple publishers and you want to know their address so that you can send them data via another socket (and this is a typical use case), create a three-part message.

Figure 24 - Pub-Sub Envelope with Sender Address



High-Water Marks

top prev next

When you can send messages rapidly from process to process, you soon discover that memory is a precious resource, and one that can be trivially filled up. A few seconds of delay somewhere in a process can turn into a backlog that blows up a server unless you understand the problem and take precautions.

The problem is this: imagine you have process A sending messages at high frequency to process B, which is processing them. Suddenly B gets very busy (garbage collection, CPU overload, whatever), and can't process the messages for a short period. It could be a few seconds for some heavy garbage collection, or it could be much longer, if there's a more serious problem. What happens to the messages that process A is still trying to send frantically? Some will sit in B's network buffers. Some will sit on the Ethernet wire itself. Some will sit in A's network buffers. And the rest will accumulate in A's memory, as rapidly as the application behind A sends them. If you don't take some precaution, A can easily run out of memory and crash.

It is a consistent, classic problem with message brokers. What makes it hurt more is that it's B's fault, superficially, and B is typically a user-written application which A has no control over.

What are the answers? One is to pass the problem upstream. A is getting the messages from somewhere else. So tell that process, "Stop!" And so on. This is called *flow control*. It sounds plausible, but what if you're sending out a Twitter feed? Do you tell the whole world to stop tweeting while B gets its act together?

Flow control works in some cases, but not in others. The transport layer can't tell the application layer to "stop" any more than a subway system can tell a large business, "please keep your staff at work for another half an hour. I'm too busy". The answer for messaging is to set limits on the size of buffers, and then when we reach those limits, to take some sensible action. In some cases (not for a subway system, though), the answer is to throw away messages. In others, the best strategy is to wait.

ZeroMQ uses the concept of HWM (high-water mark) to define the capacity of its internal pipes. Each connection out of a socket or into a socket has its own pipe, and HWM for sending, and/or receiving, depending on the socket type. Some sockets (PUB, PUSH) only have send buffers. Some (SUB, PULL, REQ, REP) only have receive buffers. Some (DEALER, ROUTER, PAIR) have both send and receive buffers.

In ZeroMQ v2.x, the HWM was infinite by default. This was easy but also typically fatal for high-volume publishers. In ZeroMQ v3.x, it's set to 1,000 by default, which is more sensible. If you're still using ZeroMQ v2.x, you should always set a HWM on your sockets, be it 1,000 to match ZeroMQ v3.x or another figure that takes into account your message sizes and expected subscriber performance.

When your socket reaches its HWM, it will either block or drop data depending on the socket type. PUB and ROUTER sockets will drop data if they reach their HWM, while other socket types will block. Over the `inproc` transport, the sender and receiver share the same buffers, so the real HWM is the sum of the HWM set by both sides.

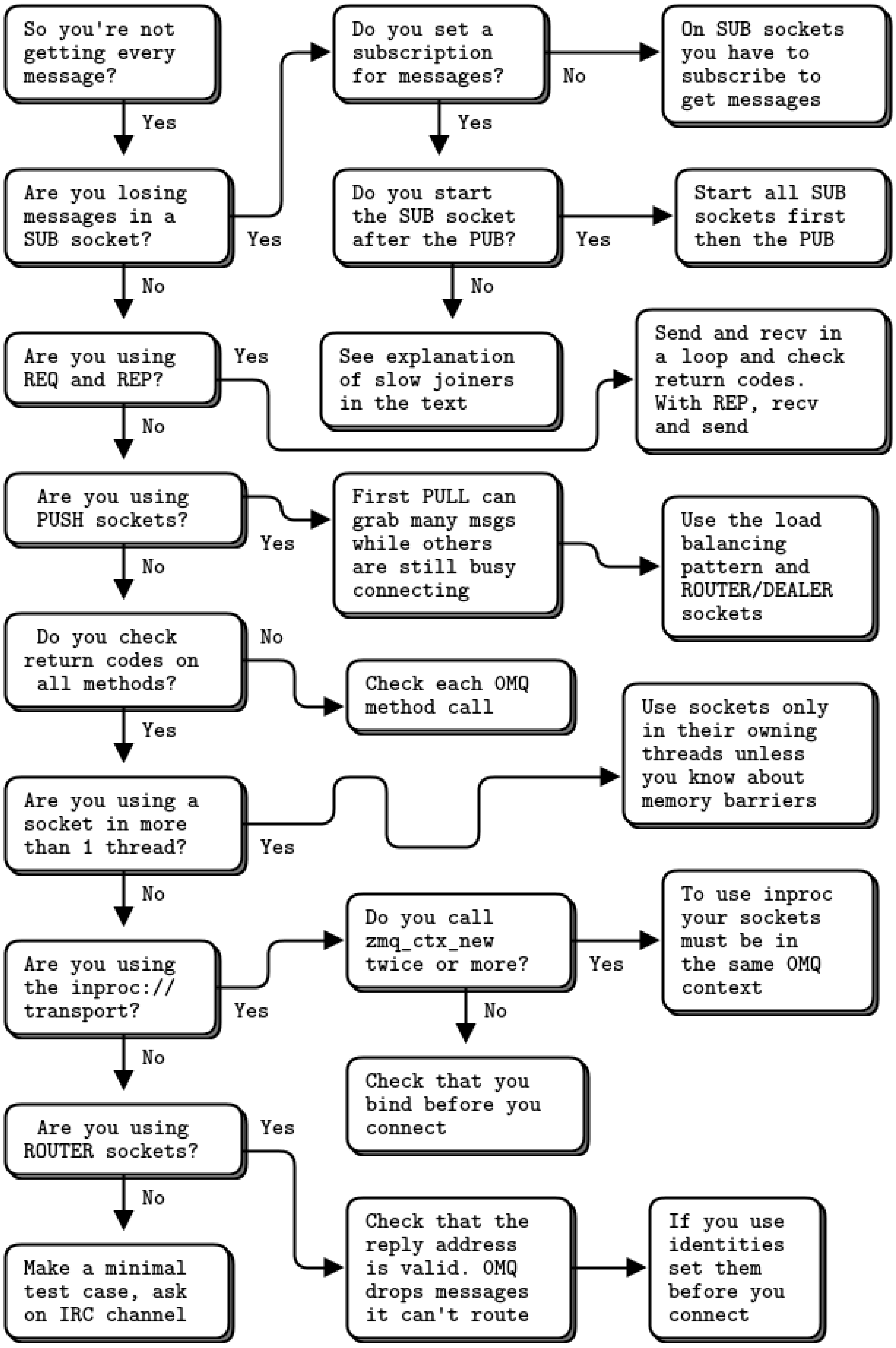
Lastly, the HWMs are not exact; while you may get *up to* 1,000 messages by default, the real buffer size may be much lower (as little as half), due to the way `libzmq` implements its queues.

Missing Message Problem Solver

top prev next

As you build applications with ZeroMQ, you will come across this problem more than once: losing messages that you expect to receive. We have put together a diagram that walks through the most common causes for this.

Figure 25 - Missing Message Problem Solver


So you're not getting every message?
Yes
Are you losing messages in a SUB socket?
Yes
Do you set a subscription for messages?
No
On SUB sockets you have to subscribe to get messages
Yes
Do you start the SUB socket after the PUB?
Yes
Start all SUB sockets first then the PUB
No
See explanation of slow joiners in the text
No
Are you using REQ and REP?
Yes
Send and recv in a loop and check return codes. With REP, recv and send
No
Are you using PUSH sockets?
Yes
First PULL can grab many msgs while others are still busy connecting
Use the load balancing pattern and ROUTER/DEALER sockets
No
Do you check return codes on all methods?
No
Check each OMQ method call
Yes
Are you using a socket in more than 1 thread?
Yes
Use sockets only in their owning threads unless you know about memory barriers
No
Are you using the inproc:// transport?
Yes
Do you call zmq_ctx_new twice or more?
Yes
To use inproc your sockets must be in the same OMQ context
No
Check that you bind before you connect
No
Are you using ROUTER sockets?
Yes
Check that the reply address is valid. OMQ drops messages it can't route
If you use identities set them before you connect
No
Make a minimal test case, ask on IRC channel

Here's a summary of what the graphic says:

- On SUB sockets, set a subscription using `zmq_setsockopt()` with `ZMQ_SUBSCRIBE`, or you won't get messages. Because you subscribe to messages by prefix, if you subscribe to "" (an empty subscription), you will get everything.

- If you start the SUB socket (i.e., establish a connection to a PUB socket) *after* the PUB socket has started sending out data, you will lose whatever it published before the connection was made. If this is a problem, set up your architecture so the SUB socket starts first, then the PUB socket starts publishing.

- Even if you synchronize a SUB and PUB socket, you may still lose messages. It's due to the fact that internal queues aren't created until a connection is actually created. If you can switch the bind/connect direction so the SUB socket binds, and the PUB socket connects, you may find it works more as you'd expect.

- If you're using REP and REQ sockets, and you're not sticking to the synchronous send/recv/send/recv order, ZeroMQ will report errors, which you might ignore. Then, it would look like you're losing messages. If you use REQ or REP, stick to the send/recv order, and always, in real code, check for errors on ZeroMQ calls.

- If you're using PUSH sockets, you'll find that the first PULL socket to connect will grab an unfair share of messages. The accurate rotation of messages only happens when all PULL sockets are successfully connected, which can take some milliseconds. As an alternative to PUSH/PULL, for lower data rates, consider using ROUTER/DEALER and the load balancing pattern.

- If you're sharing sockets across threads, don't. It will lead to random weirdness, and crashes.

- If you're using `inproc`, make sure both sockets are in the same context. Otherwise the connecting side will in fact fail. Also, bind first, then connect. `inproc` is not a disconnected transport like `tcp`.

- If you're using ROUTER sockets, it's remarkably easy to lose messages by accident, by sending malformed identity frames (or forgetting to send an identity frame). In general setting the `ZMQ_ROUTER_MANDATORY` option on ROUTER sockets is a good idea, but do also check the return code on every send call.

- Lastly, if you really can't figure out what's going wrong, make a *minimal* test case that reproduces the problem, and ask for help from the ZeroMQ community.