## [Chapter 2 - Sockets and Patterns](http://zguide.zeromq.org/page:all" \l "Chapter-Sockets-and-Patterns)

|  |
| --- |
| [Chapter 3 - Advanced Request-Reply Patterns](http://zguide.zeromq.org/page:all#Chapter-Advanced-Request-Reply-Patterns)[The Load Balancing Pattern](http://zguide.zeromq.org/page:all#The-Load-Balancing-Pattern) |

In the ROUTER to DEALER example, we saw a 1-to-N use case where one server talks asynchronously to multiple workers.

**Example : *lbbroker***

[**A High-Level API for ZeroMQ**](http://zguide.zeromq.org/page:all#A-High-Level-API-for-ZeroMQ)

Automatic handling of sockets.

Portable thread management.

Piping from parent to child threads.

Portable clocks.

A reactor to replace *[zmq\_poll()](http://api.zeromq.org/3-2:zmq_poll)*.

Proper handling of Ctrl-C.

**Example : *lbbroker2***

### [The Asynchronous Client/Server Pattern](http://zguide.zeromq.org/page:all#The-Asynchronous-Client-Server-Pattern)

N-to-1 architecture used when various clients talk to a single server, and do this asynchronously (DEALER to ROUTER).

To properly manage client state in a stateful asynchronous server, you have to:

* Do heartbeating from client to server. In our example, we send a request once per second, which can reliably be used as a heartbeat.
* Store state using the client identity (whether generated or explicit) as key.
* Detect a stopped heartbeat. If there's no request from a client within, say, two seconds, the server can detect this and destroy any state it's holding for that client.

We know the basic model well by now:

* The REQ client (REQ) threads create workloads and pass them to the broker (ROUTER).
* The REQ worker (REQ) threads process workloads and return the results to the broker (ROUTER).
* The broker queues and distributes workloads using the load balancing pattern.

A federated broker would be able to handle only one task at a time. The federation model is perfect for other kinds of routing, especially service-oriented architectures (SOAs), which route by service name and proximity rather than load balancing.

Instead of federation, let's look at a peering approach in which brokers are explicitly aware of each other and talk over privileged channels.

**Example : peering**

## [Chapter 4 - Reliable Request-Reply Patterns](http://zguide.zeromq.org/page:all#Chapter-Reliable-Request-Reply-Patterns)

ZeroMQ architectures:

* The *Lazy Pirate* pattern: reliable request-reply from the client side
* The *Simple Pirate* pattern: reliable request-reply using load balancing
* The *Paranoid Pirate* pattern: reliable request-reply with heartbeating
* The *Majordomo* pattern: service-oriented reliable queuing
* The *Titanic* pattern: disk-based/disconnected reliable queuing
* The *Binary Star* pattern: primary-backup server failover
* The *Freelance* pattern: brokerless reliable request-reply

[**Designing Reliability**](http://zguide.zeromq.org/page:all#Designing-Reliability)

There are, in my experience, roughly three ways to connect clients to servers. Each needs a specific approach to reliability:

* Multiple clients talking directly to a single server. Use case: a single well-known server to which clients need to talk.
* Multiple clients talking to a broker proxy that distributes work to multiple workers. Use case: service-oriented transaction processing.
* Multiple clients talking to multiple servers with no intermediary proxies. Use case: distributed services such as name resolution.

[**Client-Side Reliability (Lazy Pirate Pattern)**](http://zguide.zeromq.org/page:all#Client-Side-Reliability-Lazy-Pirate-Pattern)

**Example : lpserver & lpclient**

Handling failures only at the client works when we have a set of clients talking to a single server. It can handle a server crash, but only if recovery means restarting that same server.

[**Basic Reliable Queuing (Simple Pirate Pattern)**](http://zguide.zeromq.org/page:all#Basic-Reliable-Queuing-Simple-Pirate-Pattern)

**Example :** spworker & spqueue **& lpclient**

Our second approach extends the Lazy Pirate pattern with a queue proxy that lets us talk, transparently, to multiple servers, which we can more accurately call "workers". The basis for the queue proxy is the load balancing broker from [Chapter 3](http://zguide.zeromq.org/page:all#advanced-request-reply).

[**Robust Reliable Queuing (Paranoid Pirate Pattern)**](http://zguide.zeromq.org/page:all#Robust-Reliable-Queuing-Paranoid-Pirate-Pattern)

It’s like the Simple Pirate Queue bit with heartbeating in queue proxy and in workers. The queue extends the load balancing pattern with heartbeating of workers.

**Example :** ppworker & ppqueue **& lpclient**

[**Heartbeating**](http://zguide.zeromq.org/page:all#Heartbeating)

Heartbeating solves the problem of knowing whether a peer is alive or dead.

#### [One-Way Heartbeats](http://zguide.zeromq.org/page:all#One-Way-Heartbeats)

A second option is to send a heartbeat message from each node to its peers every second or so. For pub-sub, this does work, and it's the only model you can use.

#### [Ping-Pong Heartbeats](http://zguide.zeromq.org/page:all#Ping-Pong-Heartbeats)

This works for all ROUTER-based brokers. (treat any incoming data as a pong, and only send a ping when not otherwise sending data.)

Here are some tips for your own heartbeating implementation:

* Use zmq\_poll or a reactor as the core of your application's main task.
* Start by building the heartbeating between peers, test it by simulating failures, and *then* build the rest of the message flow. Adding heartbeating afterwards is much trickier.
* Use simple tracing, i.e., print to console, to get this working. To help you trace the flow of messages between peers, use a dump method such as zmsg offers, and number your messages incrementally so you can see if there are gaps.
* In a real application, heartbeating must be configurable and usually negotiated with the peer. Some peers will want aggressive heartbeating, as low as 10 msecs. Other peers will be far away and want heartbeating as high as 30 seconds.
* If you have different heartbeat intervals for different peers, your poll timeout should be the lowest (shortest time) of these. Do not use an infinite timeout.
* Do heartbeating on the same socket you use for messages, so your heartbeats also act as a *keep-alive* to stop the network connection from going stale (some firewalls can be unkind to silent connections).

[**Contracts and Protocols**](http://zguide.zeromq.org/page:all#Contracts-and-Protocols)

[rfc.zeromq.org](http://rfc.zeromq.org/)

[**Service-Oriented Reliable Queuing (Majordomo Pattern)**](http://zguide.zeromq.org/page:all#Service-Oriented-Reliable-Queuing-Majordomo-Pattern)

**Example :** mdcliapi & mdclient & mdwrkapi & mdworker & mdbroker

[**Asynchronous Majordomo Pattern**](http://zguide.zeromq.org/page:all#Asynchronous-Majordomo-Pattern)

**Example :** mdcliapi2 & mdclient2 & tripping

It has a fundamental weakness, namely that it cannot survive a broker crash without more work.

[**Service Discovery**](http://zguide.zeromq.org/page:all#Service-Discovery)

ZooKeeper for service discovery

**Example :** mmiecho

### [Idempotent Services](http://zguide.zeromq.org/page:all#Idempotent-Services)

To handle non-idempotent operations, use the fairly standard solution of detecting and rejecting duplicate requests. This means:

* The client must stamp every request with a unique client identifier and a unique message number.
* The server, before sending back a reply, stores it using the combination of client ID and message number as a key.
* The server, when getting a request from a given client, first checks whether it has a reply for that client ID and message number. If so, it does not process the request, but just resends the reply.

### [Disconnected Reliability (Titanic Pattern)](http://zguide.zeromq.org/page:all#Disconnected-Reliability-Titanic-Pattern)

So, here's the Titanic pattern, in which we write messages to disk to ensure they never get lost. We can implement our fire-and-forget reliability in a specialized worker. This is excellent for several reasons:

* It is *much* easier because we divide and conquer: the broker handles message routing and the worker handles reliability.
* It lets us mix brokers written in one language with workers written in another.
* It lets us evolve the fire-and-forget technology independently.

[Titanic Service Protocol (TSP)](http://rfc.zeromq.org/spec:9)

The simplest design of Titanic is a "proxy service". That is, Titanic doesn't affect workers at all. Client talks directly to Titanic. Titanic talks with the Brocker. Titanic replays to Client.

Titanic consists of three services:

* titanic.request: store a request message, and return a UUID for the request.
* titanic.reply: fetch a reply, if available, for a given request UUID.
* titanic.close: confirm that a reply has been stored and processed.