ZMQ (ZeroMQ) Concepts - Ramya

Before we start

1. **Sockets**:
   It is an endpoint for communication between 2 machines over a network. It acts as a door through which data is sent and received. There are 2 types of sockets: TCP and UDP.
   Stream Sockets (TCP): Requires handshake. Ensures all data arrives and in order. Ex: email and web browsing
   Datagram Sockets (UDP): Does not require handshake. Data may not arrive in order or not arrive at all. Ex: Video streaming and gaming

2. **TCP handshake**:
   A TCP handshake is the process used to establish a reliable connection between two devices over a network using the **Transmission Control Protocol (TCP)**. It's called a **3-way handshake** because it involves three steps:
    a. SYN (Synchronize)
        - The client sends a packet to the server with the **SYN** flag set.
        - This is basically the client saying, “Hey, I’d like to start a connection and here’s my sequence number.”
    b. SYN-ACK (Synchronize-Acknowledge)
        - The server responds with a packet that has both the **SYN** and **ACK** flags set.
        - It says, “Okay, I hear you, let’s connect. Here’s my sequence number and I’m acknowledging yours.”
    c. ACK (Acknowledge)
        - The client sends a final **ACK** to acknowledge the server’s sequence number.
        - After this, the connection is established and data can start flowing.

    Visual:

```
Client                    Server
  | ------- SYN -------> |
  | <--- SYN-ACK -----  |
  | ------- ACK ------>  |
Connection established
```

3. **broker vs broker less:**
    - ZeroMQ (ØMQ or ZMQ) is considered broker-less, and that’s a big part of what makes it different from traditional messaging systems like Kafka, RabbitMQ, or ActiveMQ.
    - In traditional messaging systems...there is a central message broker (like a server) that acts as a middleman:
        - Producers send messages to the broker
        - Consumers receive messages from the broker

      The broker manages:
        - Message queues
        - Routing
        - Persistence
        - Delivery guarantees
    - ZeroMQ is broker-less:

      With ZeroMQ, there’s no central broker by default. Instead, your applications (processes, nodes, services) talk directly to each other. You build the messaging patterns yourself, using ZMQ’s flexible sockets.

      You can use patterns like:
        - pub-sub (publish-subscribe)
        - req-rep (request-reply)
        - push-pull (pipeline)

      But the magic is: you don’t need a server in the middle.

      Example:
        - One service can `bind` a socket and another can `connect` to it — they’re talking peer-to-peer.
        - If you want, you *can* build a broker using ZMQ — but it's not required.
    - Broker-less = more flexibility, but more responsibility

      Pros:

        - Low latency (no middleman)
        - Simpler deployment (fewer moving parts)
        - Great for embedded systems, microservices, real-time apps

    Cons:

        - You have to handle things brokers usually take care of:

            - Message persistence
            - Message ordering
            - Backpressure
            - Failover/retries
    - How to handle without brokers?
        1. Message Persistence

           Default (ZMQ):

            - Messages are stored in memory only.
            - If the receiver is down or slow, messages can be lost.

           What You Should Do:

            - Log messages to disk (e.g. SQLite, LevelDB) before sending.
            - Implement a journal system: mark messages as “sent” only after ACK.
            - Retry unsent messages after app restarts.

           Pro tip: Use ZMQ_REQ/ZMQ_REP pattern with manual ACK logic.
        2. Message Ordering

           Default (ZMQ):

            - Guarantees FIFO order between one sender and one receiver.
            - No guaranteed order across multiple sources or consumers.

           What You Should Do:

            - Add a sequence number to each message.
            - Receiver checks order and reorders if needed.
            - Detect missing messages using the sequence gap.

           Pro tip: Structure messages with a field like {"seq": 42, "data": ...}.
        3. Backpressure

           Default (ZMQ):

            - ZMQ uses High Water Mark (HWM) to limit in-memory queues.
            - When full, sends may block (or drop, depending on socket config).

           What You Should Do:

            - Set and tune ZMQ_SNDHWM and ZMQ_RCVHWM.
            - Use PUSH/PULL with worker pools to spread load.
            - Implement a flow control mechanism (e.g. consumer signals "I'm ready").

           Pro tip: Monitor HWM usage and socket status for congestion signs.
        4. Failover / Retries

           Default (ZMQ):

            - If a peer dies, connection is broken, and unsent messages may be lost.
            - ZMQ doesn’t automatically retry or switch nodes.

           What You Should Do:

            - Use ZMQ_REQ + ZMQ_DEALER with timeouts + retry logic.
            - Implement heartbeat detection using:
                - ZMQ_HEARTBEAT_IVL
                - ZMQ_HEARTBEAT_TIMEOUT
                - ZMQ_HEARTBEAT_TTL
            - Maintain a list of fallback nodes in the app.

           Pro tip: Wrap ZMQ logic with your own connection manager module.

What is ZMQ

It is a messaging library that is available in 20 languages. This means, we include it (or import it) in our code to enable messaging. It is built in C++ and expands in Go (and also other languages).

It is different from RabbitMQ which is a server. We connect to RMQ as a client, to enable messaging. With ZMQ, we don't need to run a separate service, just import a library and we can get started.

It is broker-less unlike kafka, rabbitMQ etc.

It supports 5 messaging patterns:

a. Synchronous Request/Response
b. Asynchronous Request/Response (Router/Dealer)
c. Publish/Subscribe
d. Push/Pull
e. Exclusive Pair: 2 threads that talk to each other

It provides 6 socket types:

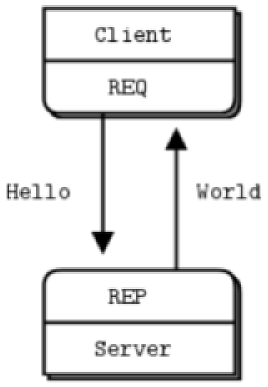
a. Req
b. Rep
c. Push
d. Pull
e. Dealer
f. Router

All 6 main socket types (REQ, REP, PUSH, PULL, DEALER, ROUTER) use **TCP by default**.

If you want **UDP-style transport**, look into **PUB/SUB** over **PGM/EPGM**.

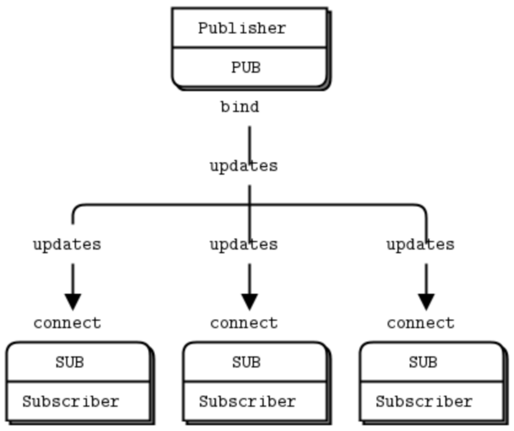
Messaging Patterns:

a. Request/Response:



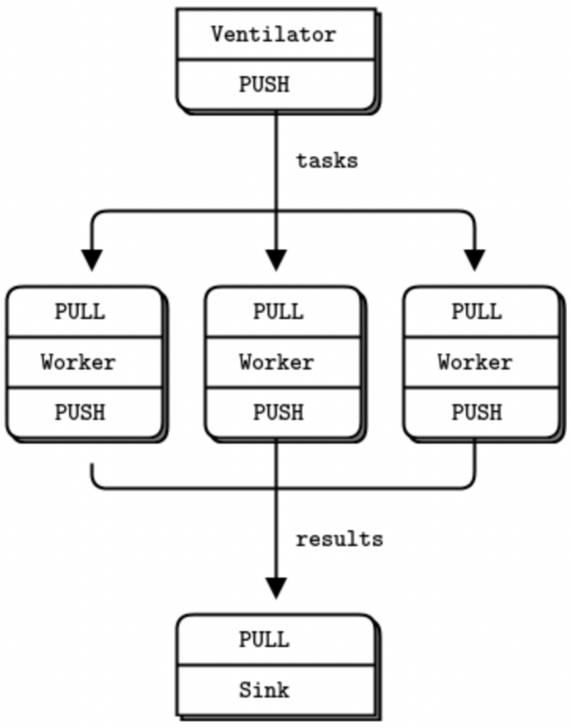
The client issues zmq_send() and zmq_recv() in a loop. That means, if it wants to send and receive multiple messages, it will all be sent in a pair of send() and receive(). If it issues more than 1 send() in a row, it will result in a return code of -1 from the respective call.

b. Publish/Subscribe:



The server pushes updates and the client can subscribe to any topic. Whatever topic the client has subscribed to, it will receive data related to that topic.
1 thing to note with pub/sub model is that if subscriber is started first and publisher is started a little later, subscriber will miss first few data because it will try to establish connection to publisher and miss the messages sent during that time.

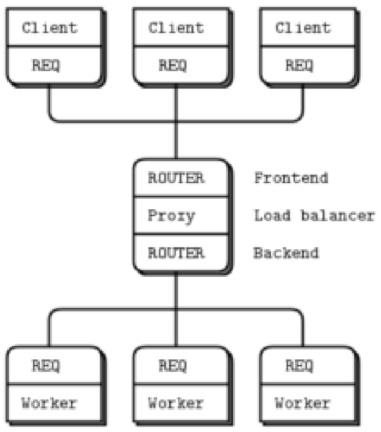
c. Push/Pull:



This contains:

i. A ventilator that produces tasks that can be done in parallel
ii. A set of workers that process tasks
iii. A sink that collects results back from the worker processes

d. Async request/response:



ZMQ doesnt define a broker and works on peer-peer by default but this messaging pattern allows us to defile broker in between to widen the Request/Response
pattern to send msgs even when its not received which was a draw back with Request/Response pattern if messaging.

This broker does the following:

- Accepts connections from a set of clients.
- Accepts connections from a set of workers.
- Accepts requests from clients and holds these in a single queue.
- Sends these requests to workers using the load balancing pattern.
- Receives replies back from workers.
- Sends these replies back to the original requesting client.

DEALER socket as broker: does not care about identity
ROUTER socket as broker: appends identity and expects identity

Bind v/s Connect: what to use when?

Both .Bind() and .Connect() are used to establish communication between sockets, but they serve different roles.

.Bind(): Creates an endpoint and waits for others to connect.

.Connect(): Initiates a connection to an existing bound socket.

General rule: Bind is used for stable and known endpoints i.e servers. Connect is used for dynamic clients that need to find and join a server.

In push/pull pattern:

a. Ventilator binds for workers with PUSH socket (port 5557) and connects to sink on PUSH socket (port 5558) (to send the 1st signal of task started)
b. Workers connect to ventilator on PULL socket (port 5557) and connect to sink on PUSH socket (port 5558)
c. Sink binds for workers and ventilator on PULL socket (port 5558)

Why use ZMQ:

a. Handles I/O asynchronously:
   The application does not wait for I/O to happen. These I/O operations continue in background threads and are handled by ZMQ.
b. Uses lock-free data structures:
   ZMQ ensures message passing is safe between multiple threads. This means no need for mutexes, locks or semaphores. This means multiple threads can access data simultaneously without waiting for each other, making the application faster.
c. Components can come and go dynamically and ZMQ can handle it.
d. It handles overflow of the queue, meaning it will either block senders or throw away messages, depending on the pattern implemented.

TCP v/s ZMQ:

ZeroMQ sockets are different from classic TCP/IP sockets:

a. ZeroMQ sockets have one-to-N routing behavior built-in, according to the socket type.
b. There is no 'accept' method. When a socket is bound to a connection, it automatically starts accepting connections.
c. 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).
d. ZeroMQ sockets carry messages, like UDP, rather than a stream of bytes as TCP does. A ZeroMQ message is length-specified binary data.
e. 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.

Why use tcp

When in code, we are using the word "tcp" to connect and bind sockets, we are only specifying the transport layer protocol that zmq uses internally. It provides other protocols too like ipc, inproc, pgm. tcp is the most commonly used protocol, hence the examples use that.
ZMQ differs from tcp in terms of how sockets are handled. These differences are listed above.

Code examples of all patterns

zmq-demo.zip