ØMQ

Prepared for Weta Digital, March 2012

Martin Lucina <martin@lucina.net>

What is ZeroMQ?

- … a replacement for AMQP?
- … a Message Queue?
- … BSD sockets with framing?

  Not quite all of the above...
- Lego bricks for building your own distributed systems.
- BSD sockets the way they might look if designed today.

Existing solutions

- Custom-built:
  - Roll your own messaging over BSD sockets.
- Proprietary/enterprise message-oriented-middleware:
  - Message queueing in a box.
  - Most current implementations are like SQL databases in the 1980s, or cater to niche markets (FT).
  - Usually big, slow, complex and *always* expensive.
- FOSS middleware:
  - AMQP: RabbitMQ, Redhat MRG, OpenAMQ.
  - Niche or domain-specific: OpenMPI, D-Bus.

ZeroMQ

- Is 100% Free Software, LGPL.
- Around 20k LOC of extremely conservative C++.
- Provides a lean and mean native C API inspired by BSD sockets.
- Cross platform
  - Linux, *BSD, Solaris and any other POSIX platform.
  - Win32/Win64
  - VMS
- Language agnostic
  - C++, Python, Ruby, Java, .NET CLR, Perl, Erlang, LISP, Haskell and more

ZeroMQ Sockets vs. BSD (TCP) sockets

- Messages vs. bytes.
- Transfer is atomic, either you get the whole message, or you get nothing.
- No reliability guarantees (at-most-once delivery).
- Sending is entirely asynchronous.
- No direct access to the individual underyling connections.
- A single socket can be connected and/or bound to *multiple* endpoints, potentially using *multiple* transports.

The API

- zmq_socket()
- zmq_connect(), zmq_bind()
- zmq_send(), zmq_recv()
- zmq_setsockopt(), zmq_getsockopt()
- zmq_poll() for integration with event loops.
- That's it!
- Well, not quite. A bunch of functions for housekeeping: zmq_init(), zmq_term() and zero-copy send/receive, manipulation of messages: zmq_sendmsg(), zmq_recvmsg(), zmq_msg_*().

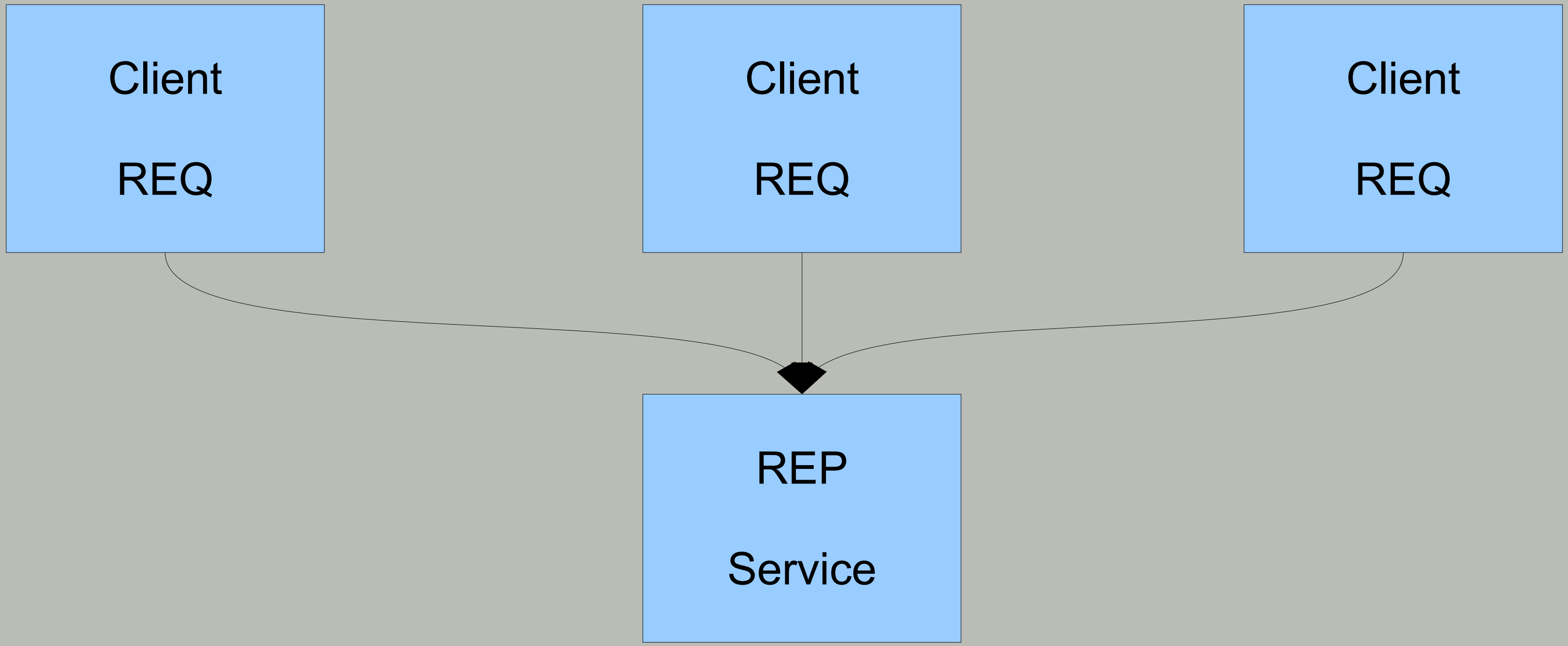
Sockets and patterns

- Request/reply
  - ZMQ_REQ, ZMQ_REP
  - An SQL client/server model.
- Publish/subscribe
  - ZMQ_PUB, ZMQ_SUB
  - A data distribution model. E.g. stock market quotes, media streaming, …
- Pipeline
  - ZMQ_PUSH, ZMQ_PULL
  - Work distribution. Eg, HPC worker nodes.

Transports

- tcp://...
  - plain old TCP
- ipc://...
  - Local inter-process communication
- inproc://...
  - Local in-process communication
- epgm://..., pgm://...
  - PGM “mostly-reliable” multicast

Request/reply topology



Client

REQ

Client

REQ

Client

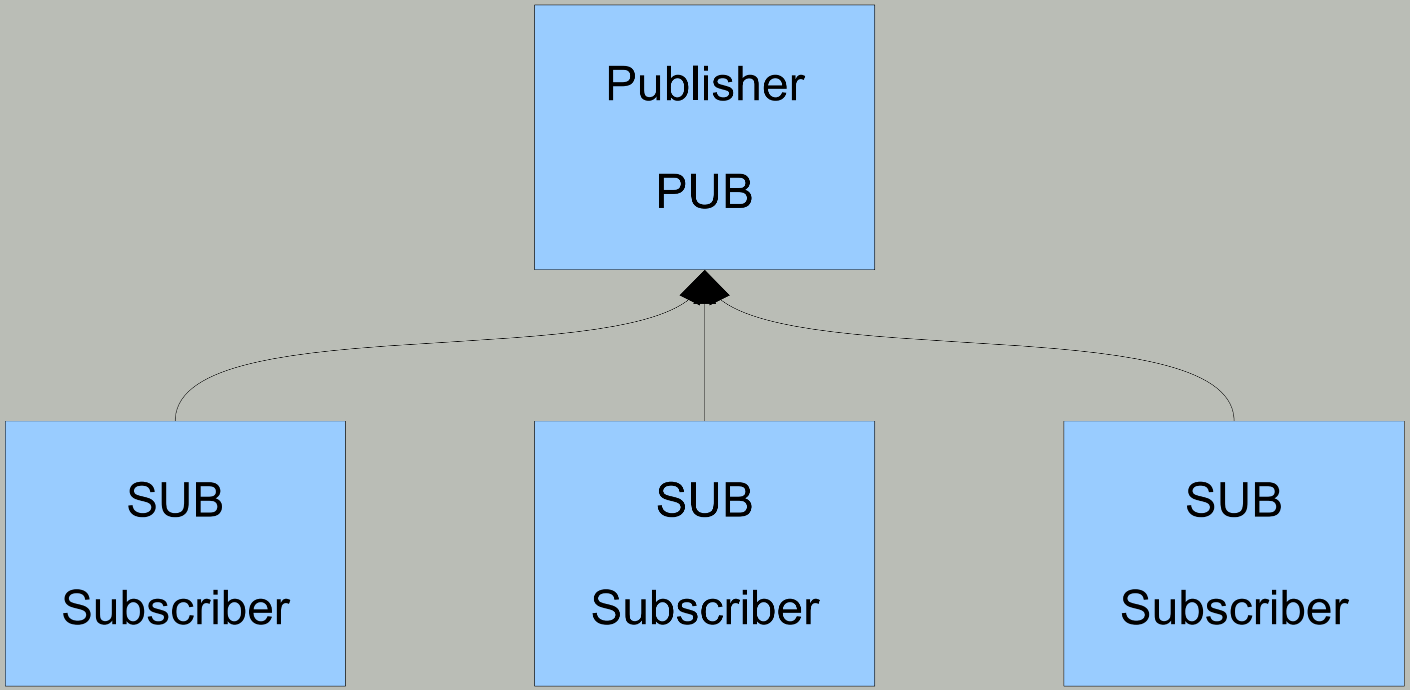
REQ

REP

Service

(Live demo examples shown here)

Publish/subscribe topology



(Live demo examples shown here)

The multithreaded problem

- Many cores, many threads.
- Classic MT code uses locks.
- Rearchitecting using message-passing is a really nice model for a lot of applications.
- Entire languages (Erlang) built around message passing integrated into the language.
- ZeroMQ lets you use this kind of model with any language we have a binding for.
- Example: Each thread owns an inproc:// socket “mailbox”.
- ZeroMQ actively encourages you to use this model:
  - Sockets may be migrated between threads, if you're not scared of full memory barriers.
  - Otherwise, always use a socket from a single thread.

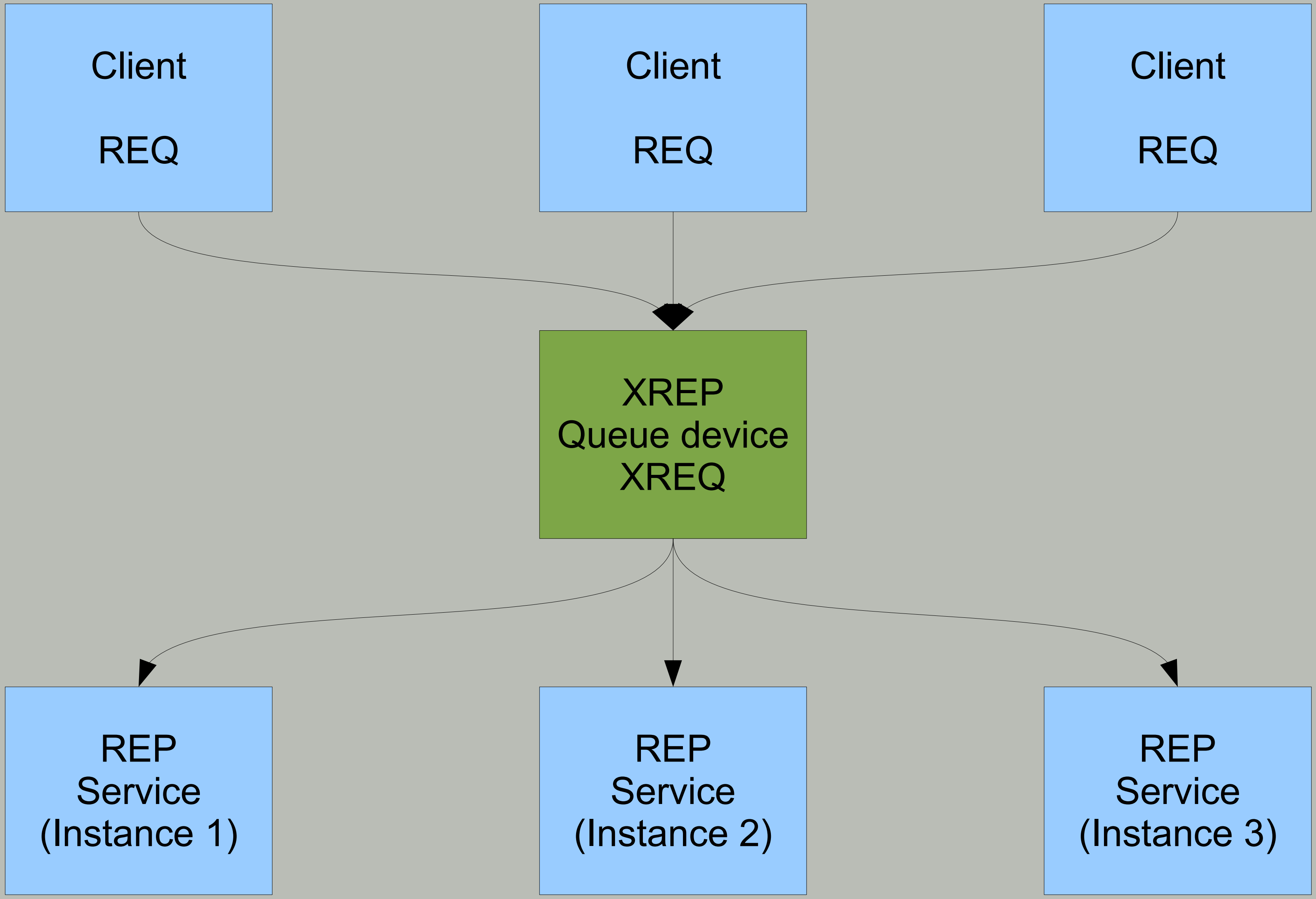
The reliability problem

- A lot of people ask “Does ZeroMQ do reliable/persistent/fault-tolerant messaging?”
- What is “reliable”, exactly?
  - At-most-once delivery (yup, out of the box).
  - At-least-once delivery (easy to implement, sufficient for most applications, potential project for someone to fix in-core REQ/REP).
  - Once and only once delivery (hard, don't believe the vendors, *impossible* without operator intervention).
  - “Reliable publish/subscribe” is a lie; a single hung consumer can kill the entire topology.
- Cost/benefit: Applications crashing? Servers crashing? HDD mangling your bytess on their way to a journal?

The scalability problem

- Scale out, infinitely.
- The Web has taught us to make services stateless and/or able to tolerate duplicate requests: at-least-once delivery.
- ZeroMQ actively encourages you to architect your distributed systems to be infinitely scalable.
  - No direct application access to underlying connections or information about individual instances.
  - All basic ZeroMQ patterns (socket types) are designed to be scalable.
- A long-term design goal is scaling out to global topologies:
  - End-to-end (REQ/REP, PUB/SUB)
  - vs. hop-by-hop (XREQ/XREP, XPUB/XSUB)
  - Global data distribution, devices as transparent middle nodes in the topology, etc.

Request/reply with Queue Device topology


Client

REQ

Client

REQ

Client

REQ

XREP
Queue device
XREQ

REP
Service
(Instance 1)

REP
Service
(Instance 2)

REP
Service
(Instance 3)

Queue device

- Implement a classic shared queue. Services can come and go, the clients do not need to know about individual instances.
- This is the “Enterprise Service Bus”.
- All the queue device does is:
    - Poll for input on its “in” and “out” sockets.
    - Receive requests from the “in” socket and forward them to the “out” socket.
    - Receive replies from the “out” socket and forward them to the “in” socket.
- Device is a transparent middle node. Node code changes at client or service are needed to use it, only your endpoints change.

Future directions

- API stability and simplicity. Implies keeping many potential features *out* of the core library.
- Pluggable transports, filtering mechanisms.
- Use over the Internet. The library must not crash, ever.
- Naming and discovery of distributed services.
- Authentication, encryption. Hard problems to get right.
- Management and monitoring infrastructure.
- Moving towards making ZeroMQ an integral part of the Internet stack
  - A nascent working group for eventual IETF standardisation of the concepts behind ZeroMQ, and a kernel-space implementation.

Questions?

- www.zeromq.org
- Extensive API reference and user guide
- Active mailing list with 1000+ members
- IRC chat at #zeromq on Freenode

Martin Lucina, <martin@lucina.net>