ØMQ and PyØMQ

Simple and Fast Messaging

Brian Granger

SciPy 2010

Message passing

- Message = binary data, csv, structured data, Python objects, files, XML, JSON, video frames, audio streams, etc.
- Passing = the act of sending messages between two endpoints. The endpoints are typically in different threads, processes or hosts.

General “messaging” features

- Security.
- Asynchronous send/recv’s
- Message queuing/routing/filtering.
- Persistence, reliability, transactions.
- Fault tolerance.
- Difficult to add features without:
  - Killing performance.
  - Adding massive complexity.

Existing options

- Sockets, pipes
- XMLRPC, JSONRPC, SOAP, CORBA
- Twisted, Pyro, RPyC, Eventlet, Concurrence, Gevent, Cogen, etc.
- Amazon SQS
- AMQP, XMPP, Java MS, Microsoft MQ
- Fast options lack messaging features.
- More fully featured options tend to be complex.

ØMQ

- LGPL C++ messaging LIBRARY developed by iMatix.
- Thin layer of messaging features on top of raw sockets: “Sockets on steroids”, “Pimped socket interface”.
- 15 Language bindings.
- Simple binary wire protocol
  - Freedom to choose serialization approach.
  - JSON, HTTP, XML, Protocol Buffers, Custom.
  - Easy to implement.

ØMQ Messaging

- Asynchronous atomic send/recv's.
- Basic messaging patterns:
  - publish/subscribe.
  - request/reply + load balancing + fair queueing.
  - point-to-point.
  - Reliable multicast using PGM.
- Flow control and quality of service logic.
- No security, no centralized broker or backbone.
- Multipart messages.

PyØMQ

- LGPL, Cython based bindings to ØMQ.
- Full and faithful coverage of the ØMQ API.
- Careful to release the GIL when calling ØMQ.
- Small set of extras:
  - Built-in, but optional JSON and pickle serialization.
  - Tornado web server integration.
  - Polling interfaces compatible with select.poll and select.select.

Messaging and the GIL

- Most of us are running C/C++ code that takes a long time and doesn’t release the GIL (numpy.linalg.eigvals).
- When non GIL-releasing C/C++ code runs in Python ALL network traffic in that process stops.
- Threads do not help at all.
- Multiple processes do not fully help.

A Simple example

```
def compute(n):
    import numpy as np
    a = np.random.rand(n,n)
    # doesn't release the GIL!
    return np.sum(np.linalg.eigvals(a))

server = SimpleXMLRPCServer(('localhost',10002))
server.register_function(compute)
server.serve_forever()
```

- Each time compute is called, it blocks and network traffic stops.
- Even if you put SimpleXMLRPC in a thread, np.linalg.eigvals will hold the GIL and network traffic will stop.
- The client will also block unless you use threads or a select/poll loop.
- BUT, even if you use select/poll, the client can't tell if the server is 1) busy or 2) dead. Server will be dead to other clients as well.

ØMQ Threading

- ØMQ Sockets use a pool of C++ IO threads to send, receive and queue messages.
- All of this logic continues even while:
    - GIL releasing or non-GIL releasing C/C++ code runs.
    - Python code runs.
    - The other endpoint is not connected!
- The Socket.send and Socket.recv functions that a user calls don’t block. They simply move the message to the IO threads to be handled.

Fast

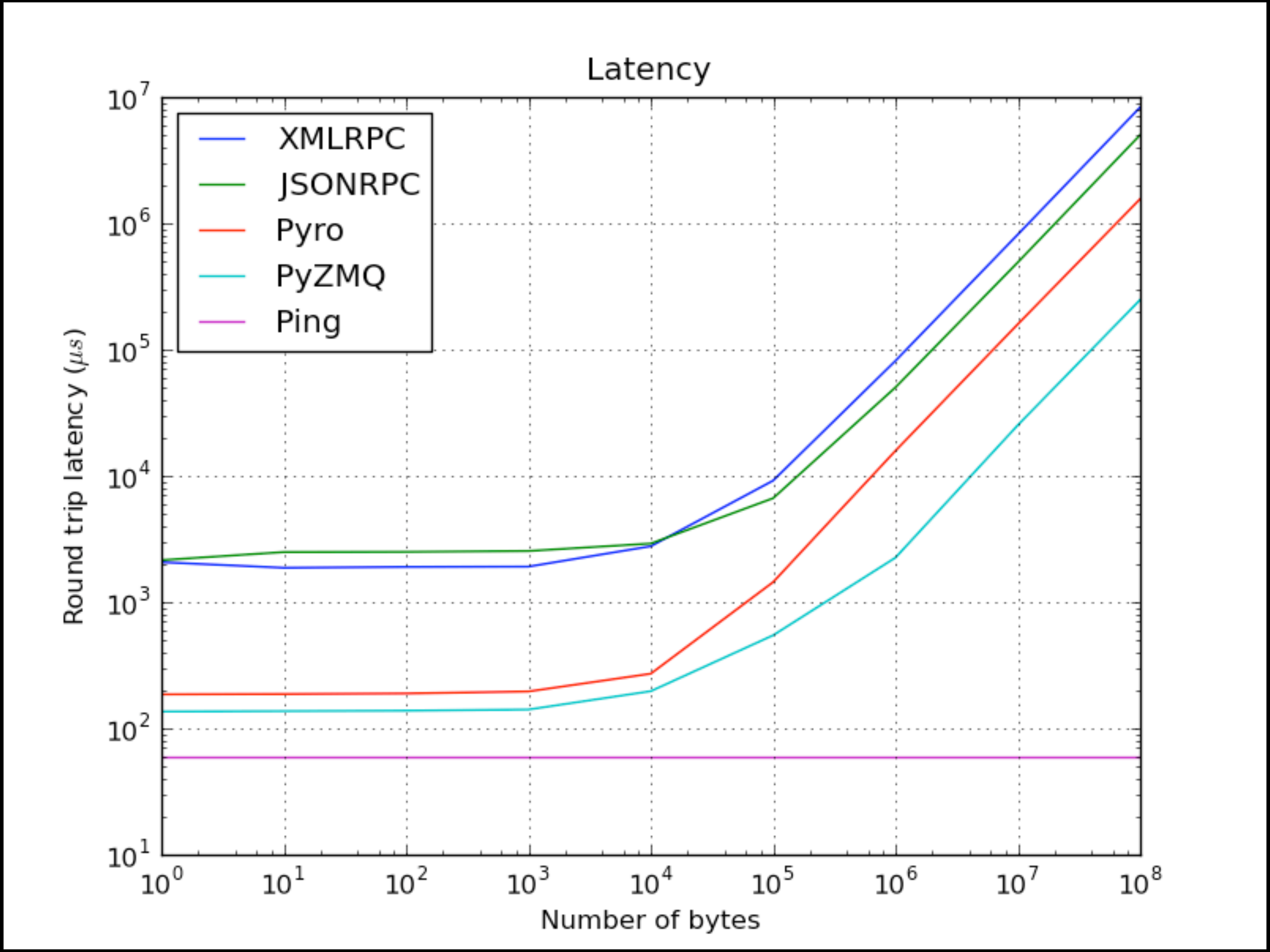
Why is ØMQ fast?

- Lightweight C++ library.
- The C++ IO thread pool enables ØMQ to scale to multicore.
- Dead simple wire protocol. Not chatty.
- Great care taken in ØMQ/PyØMQ to make sure messages are not copied.
- Does a few well defined things.

Benchmarks

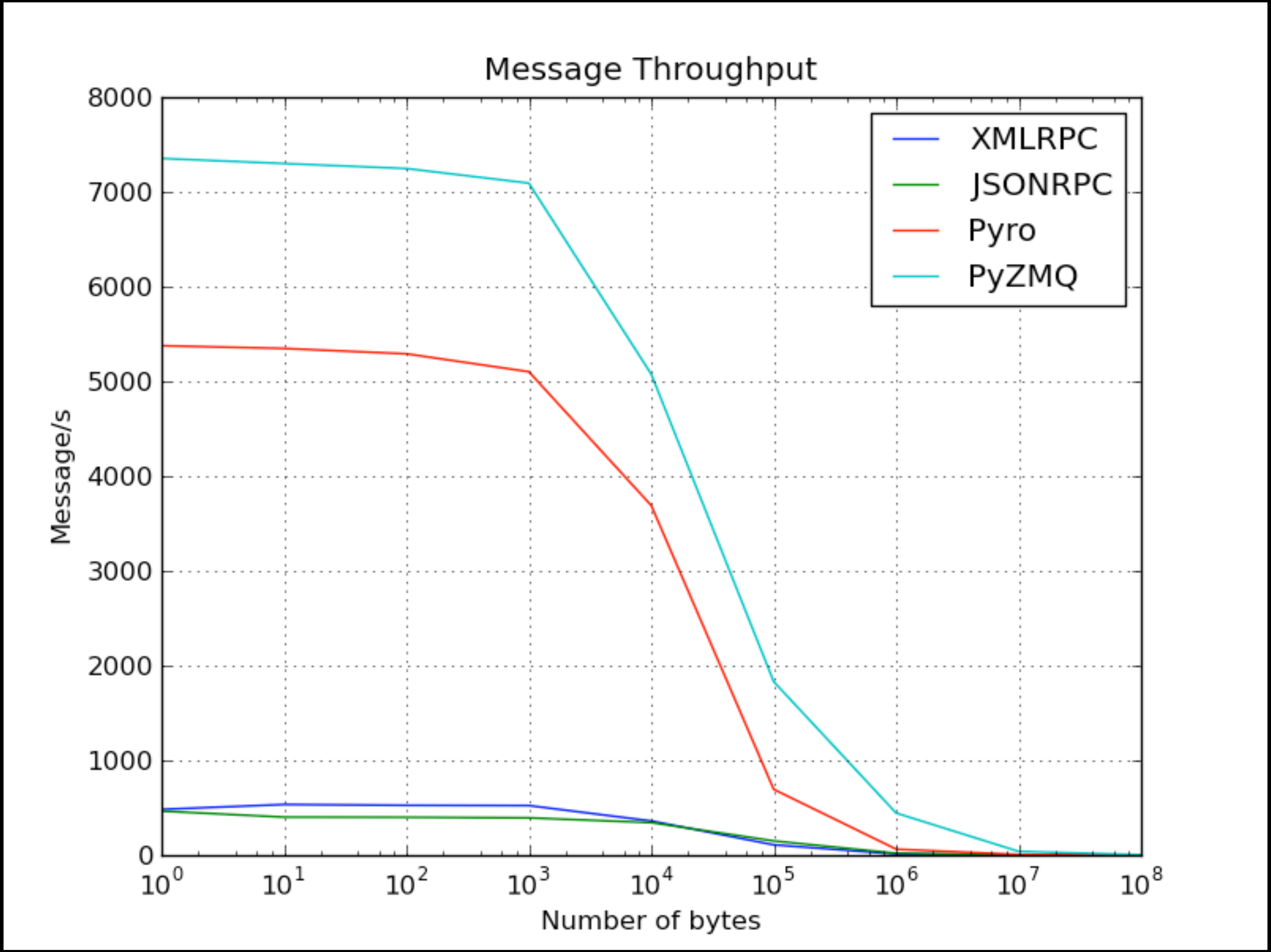
- Send a message on a round trip between two processes (TCP over loopback).
- Measure round trip latency in microseconds.
- Measure throughput in messages/s.
- Compare PyØMQ, Pyro, xmlrpclib, jsonrpclib.

Latency

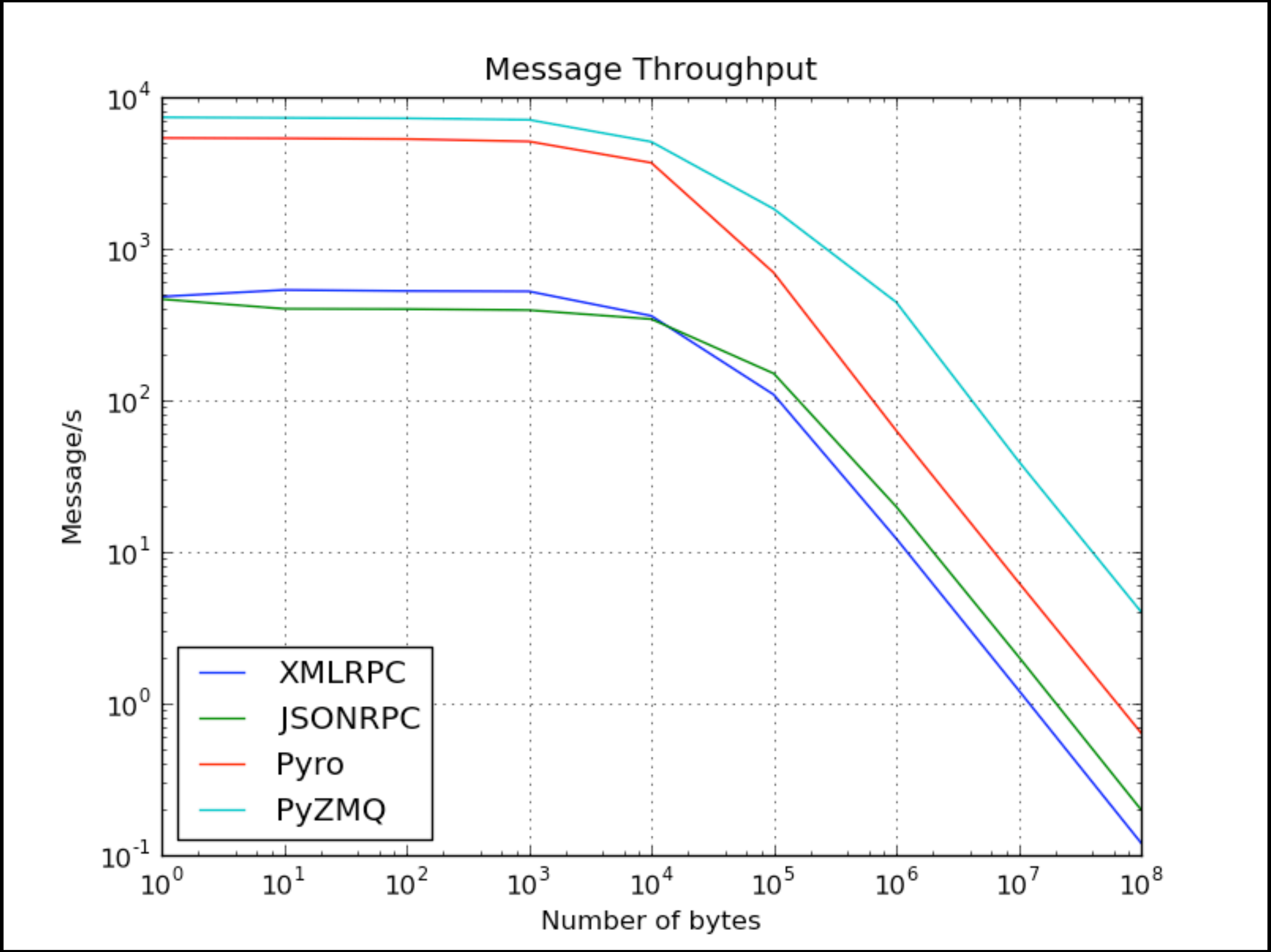


Latency
XMLRPC
JSONRPC
Pyro
PyZMQ
Ping
Round trip latency (μs)
Number of bytes
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^0
10^8

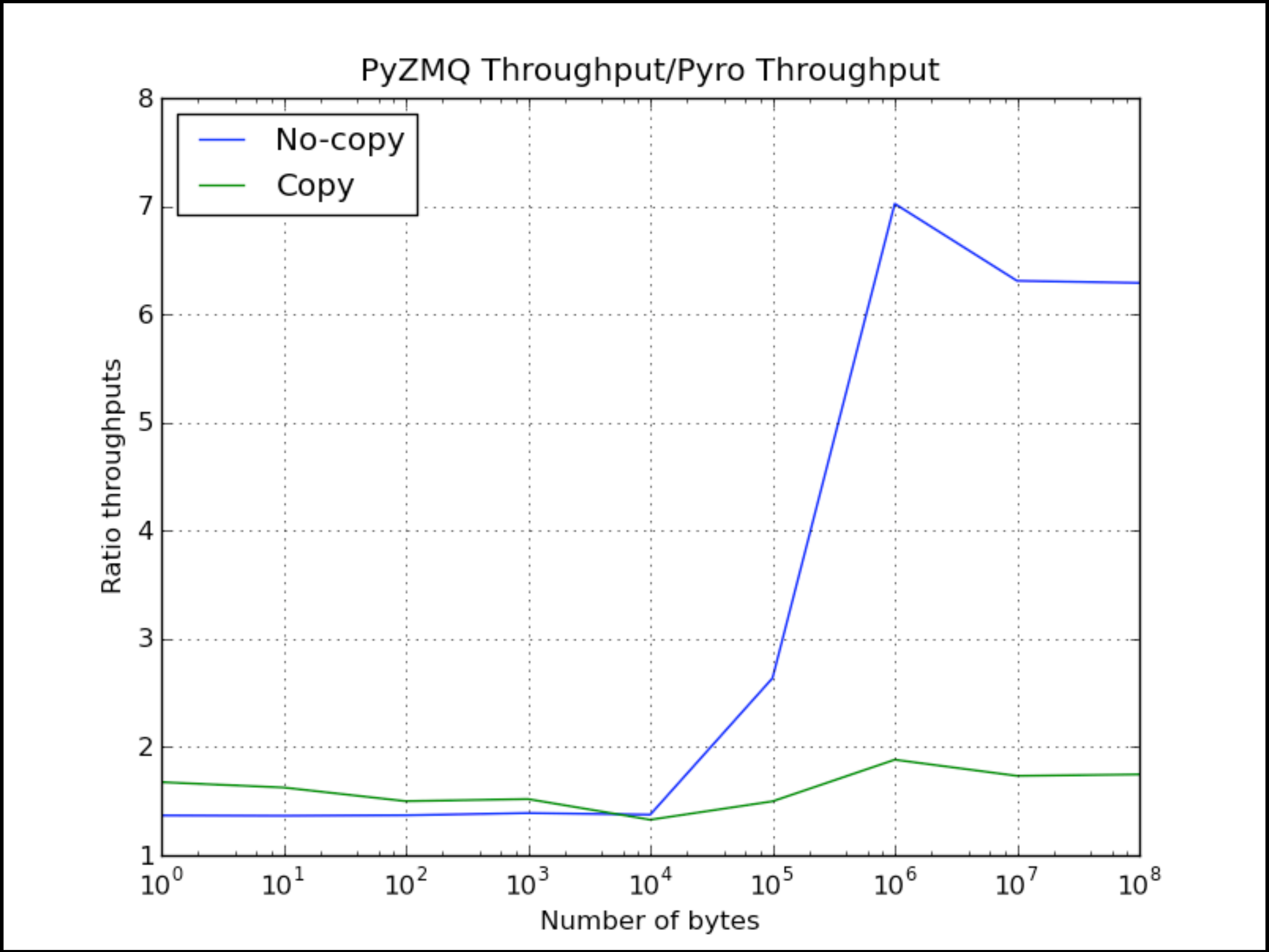
Throughput I



Throughput II



No-copy makes a difference



Simple but powerful

ØMQ is simple

- The philosophy is to do one thing well.
- Small API (afternoon sized).
- Sockets automatically connect and reconnect.
- A few simple abstractions that are repeated and combined to build non-trivial applications.

ØMQ/PyØMQ API

- Create a Context
- Create a Socket type
- Pick one or more transports
- Call Socket.send and Socket.recv

```
import zmq

c = zmq.Context()


s = c.socket(zmq.REP)


s.bind('tcp://127.0.0.1:10001')
s.bind('inproc://myendpoint')


while True:
    msg = s.recv()
    s.send(msg)
```

ØMQ Socket types

- REQ/REP: request/reply pattern
- XREQ/XREP: many-to-many load balanced, fair queueing request/reply.
- PUB/SUB: publish/subscribe. Think twitter.
- PAIR: point-to-point. Think texting.
- UPSTREAM/DOWNSTREAM: Load balanced, fair queuing for PUB/SUB data pipelines.

Code Examples

Learning more

- http://www.zeromq.org
- http://github.com/ellisonbg/pyzmq
- Blog post by Nicholas Peil:
  - http://nichol.as/zeromq-an-introduction