Messaging

Low Latency, High Throughput, Durable,
RESTful, Open, Standards, ..

Why bother?

- We've spent years making messaging
- Working on protocols like AMQP and RestMS
- Working on software like OpenAMQ and ZeroMQ
- Working with partners like HP, Intel, Cisco
- It is a lot of work... :-)
- The question is... *why*?

Messaging saves $$$

- Messaging saves application architects money
- Cheaper to design
- Cheaper to make
- Cheaper to deploy
- Cheaper to scale
- Cheaper to maintain

How it used to be...

- Option 1: build messaging as part of your application
- Option 2: buy a very expensive messaging “solution”
- Option 3: cludge something together
- Where is the open messaging stack?
  - Fast, efficient, scalable, powerful, etc.
  - Free software, open standards, competition...

Aiming for Lazy & Stupid

- Lazy & Stupid applications are best
- **Lazy**, because they only work when they have to
  - Applications are asynchronous, event-driven, queue-driven
- **Stupid**, because they know almost nothing about the rest of the world
  - All knowledge is in the message and only in the message

Why is this amazing?

- Total separation between developers and architects
- Messages are contracts
- Just add and remove pieces to scale
- Replace any piece at any time
- Lowest possible cost over whole app lifecycle

Do we need proof?

- International banking: SWIFT
- Based on messages (thousands of them)
- Each piece (bank) operates independently
- Now standard for European payments

Ideal messaging has...

- Messages as blobs
- Virtual addressing
- Queues all over the place
- A documented wire protocol
- Open source stacks
- … & the perfect QoS / performance tradeoff

QoS vs. Performance

- Store in memory or on disk?
- Flush to disk or use write cache?
- Use larger envelopes or smaller ones?
- Use binary framing or text framing?
- Multiplex connections or not?
- Heartbeating... max. message size... HTTP compatibility... and so on

What we learned

- There is no one answer. We found three:
  - Data center messaging
    - Low latency, small messages, multicast, thin envelopes, no security, no persistence.
  - Enterprise messaging
    - 10x larger messages, higher latency, unicast, fatter envelopes, basic security, persistence, transactions.
  - Internet messaging
    - high latency, massive scaleout, paranoid security

And thus we get

- Data centre messaging
  - 0MQ, library for peer-to-peer messaging using 'hypersocket' paradigm
  - No real protocol, just blob encoding
- Enterprise messaging
  - OpenAMQ, broker + client
  - AMQP protocol, 100 pages or so
- Internet messaging
  - RestMS protocol, XML-over-HTTP (RESTful)

A little about iMatix

- Started in 1996 to build open source stacks for Internet applications
- Develops open source products (0MQ, OpenAMQ, and others)
- Sells support licenses and services
- Primary designer of AMQP protocol
- In 2010, still supports OpenVMS clients!
  - Tour operator using ACMS and AST front-ends for LAT, VT220, Telnet and TN3270, built by iMatix.

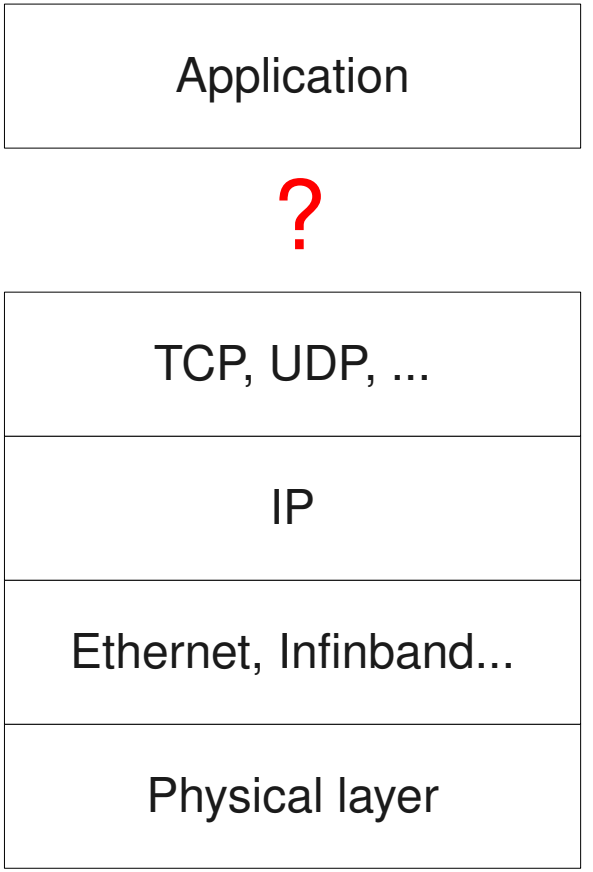
For more information

- 0MQ is at www.zeromq.org
- OpenAMQ is at www.openamq.org
- iMatix is at www.imatix.com
- Pieter is at ph@imatix.com

Fastest. Messaging. Ever.

Missing piece in networking stack

- What can I do?
- Build the missing piece myself.
- Buy a messaging product.



Building it yourself

- Doing the same work over and over again.
- Code has to be improved and maintained for a long time.
- Given enough development iterations, messaging code tends to become the most complex part of the application.

Buying a messaging product

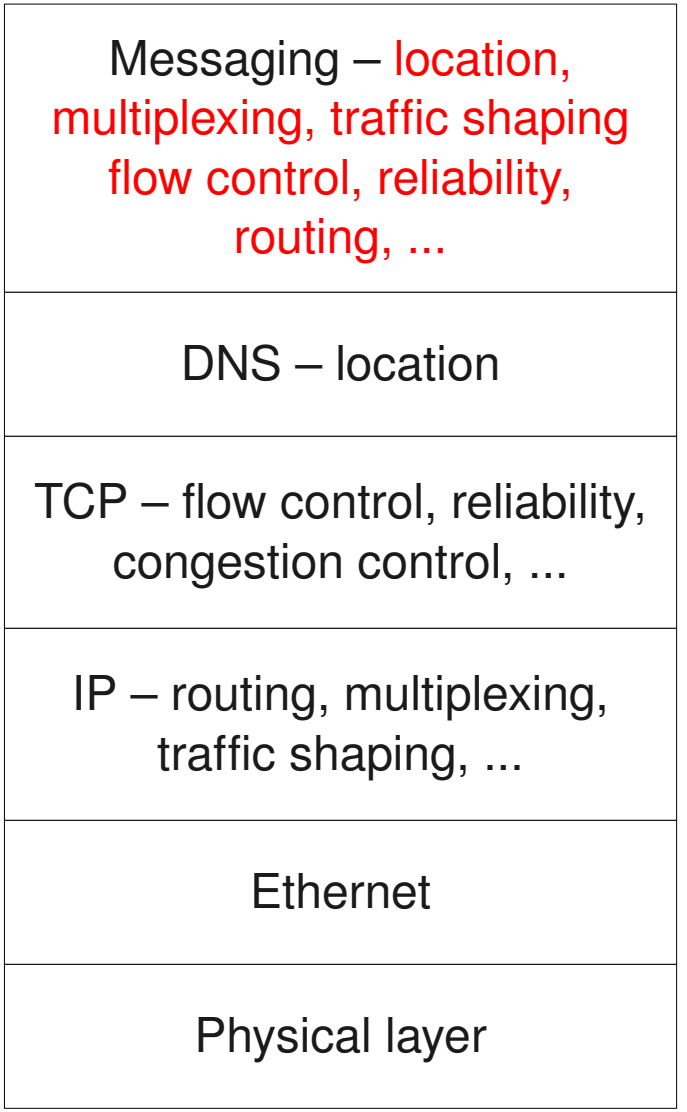
- It's expensive.
- It's terribly expensive.
- Even if it's free it's terribly expensive.
- Proof: Small projects never use messaging - the cost would kill them.
- To understand the cost comparing it with standard networking stack helps.

Duplication of networking stack

The roots of the evil are back in 80's when networking stack wasn't as ubiquitous as it is now.

Duplication causes:

- Lower quality of code.
- Lower performance.
- Constrained functionality.
- Requires parallel set of skills/admins/developers.



Complexity

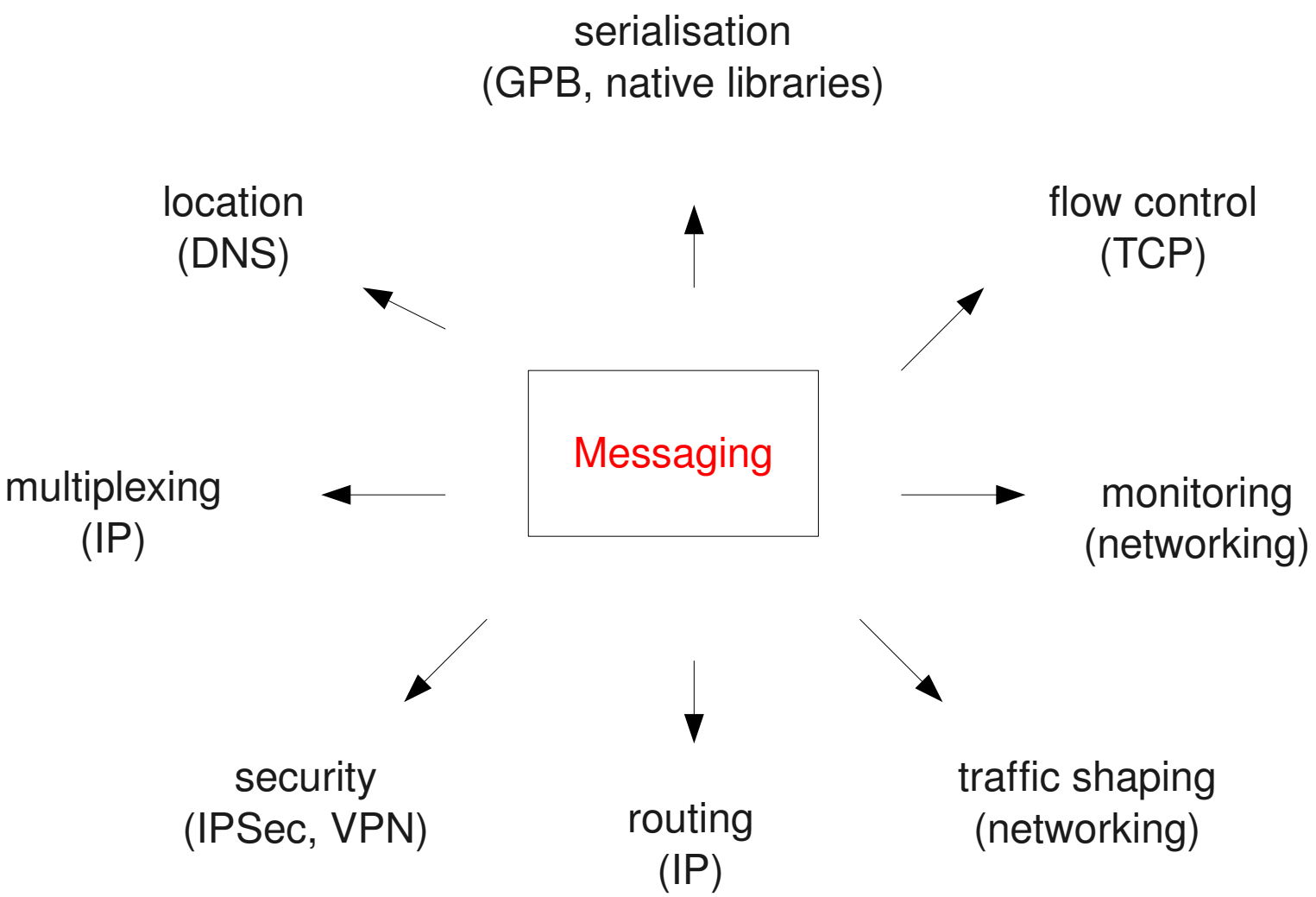
- Complex installation.
- Complex administration.
- Complex APIs.
- Complex protocols.
- Steep learning curve.
- Adoption of a messaging system is extremely expensive.

Niche solutions

- Messaging systems are focused on big customers that can afford the cost.
- They incorporate a lot of niche functionality that's not needed for common usage.
- That makes them even more complex and expensive.
- Additionally, it makes them very large. No way running it on a phone or an industrial sensor.

How to address the problem?

Offload everything!



What are the benefits?

- Almost no need for niche experts.
- Simple APIs and protocols.
- Lightweight system.
- Better performance (including HW support!).
- Broader functionality.
- Not tied to a specific niche.
- Ubiquitous.

Extended networking stack

Business logic	Application
Datatype marshalling/unmarshalling	GPB
Messaging patterns	ØMQ
Reliability, flow control	TCP
Routing, multiplexing, traffic shaping	IP
Data transmission, packaging	Ethernet
Signal transmission	Physical layer

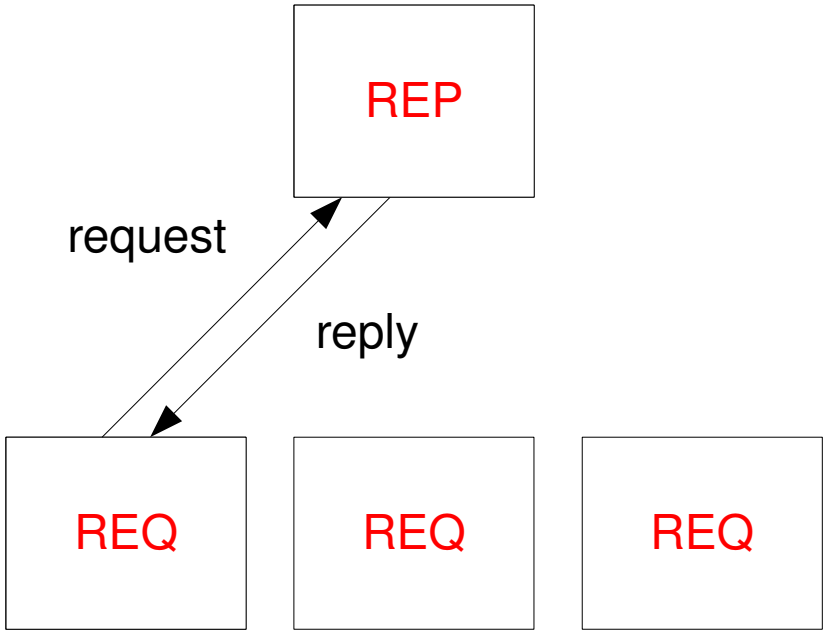
Huh? Messaging patterns?

- Patterns of how the messages are exchanged.
- Messaging pattern involves multiple connections between multiple endpoints.
- So far we've identified three messaging patterns: request/reply, publish/subscribe and message streaming.

Request/reply (a.k.a. RPC)

- Works like a web server.
- Client sends a request, server responds.
- Replies are routed to the right client automatically.

Simple request/reply (client/server)

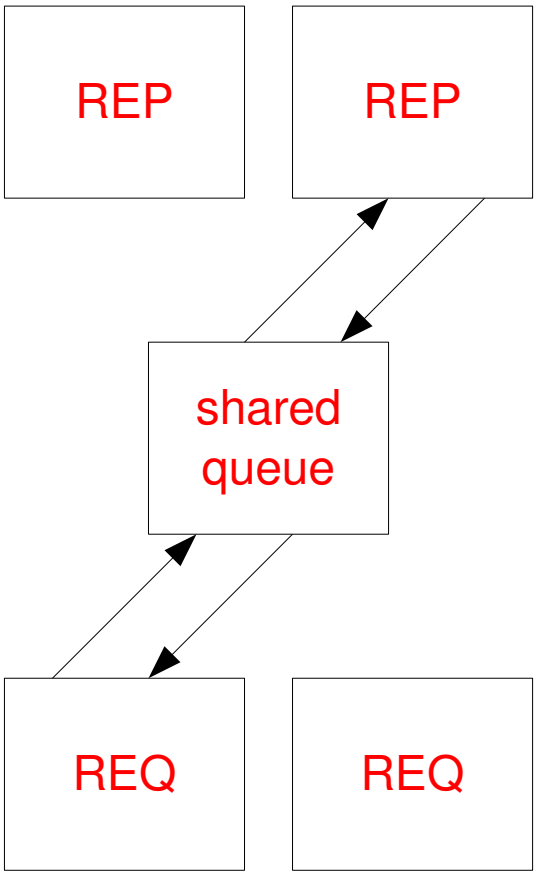



```
zmq::context_t ctx (1, 1);
zmq::socket_t s (ctx, ZMQ_REP);
s.bind (“tcp://eth0:5555”);
zmq::message_t request, reply;
while (true) {
    s.recv (&request);
    ...
    s.send (reply);
}
```

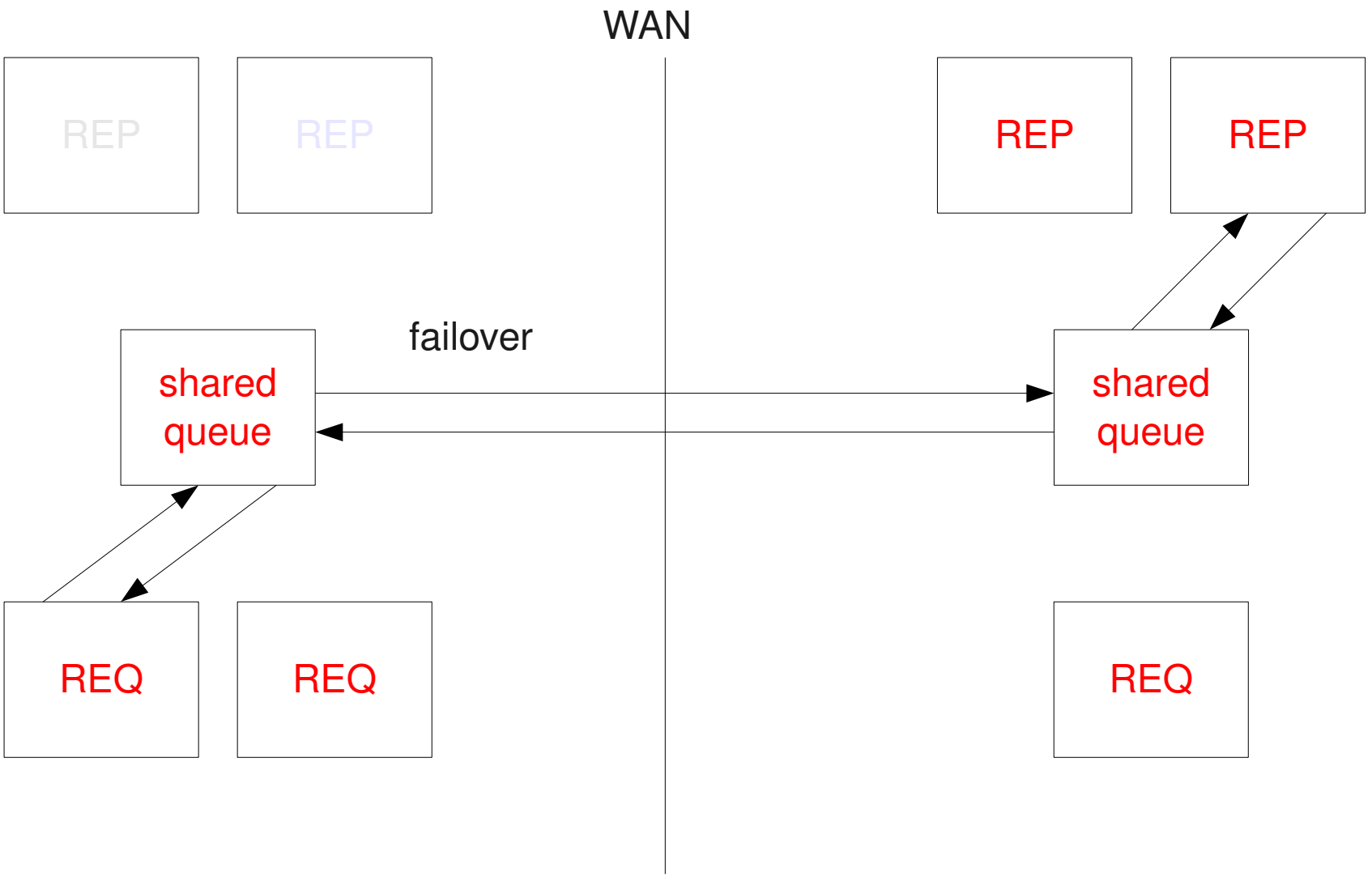
```
zmq::context_t ctx (1, 1);
zmq::socket_t s (ctx, ZMQ_REQ);
s.connect (“tcp://192.168.0.111:5555”);
zmq::message_t request, reply;
...
s.send (request);
s.recv (&reply);
...
```

Complex request/reply (SOA)

- Queue stores messages even if there's no “service” online.
- Load-balancing.
- Location transparency.
- “Multi-threaded server” scenario.



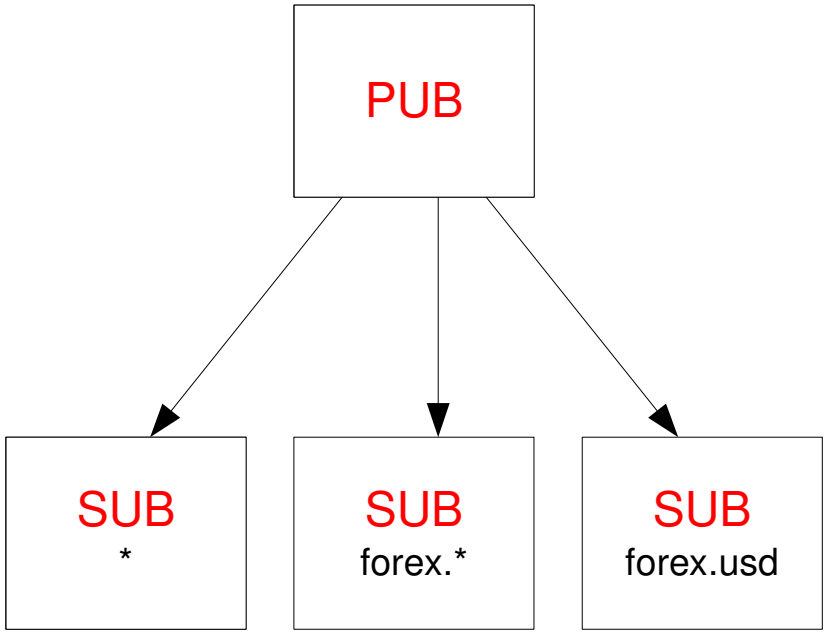
Very complex request/reply



Publish/subscribe

- It's like a radio transmission.
- Publisher sends continuous stream of messages, subscribers join the stream, receive messages, then leave the stream.
- Each message is delivered to every subscriber.
- Messages may be filtered w.r.t. “topics”.
- Maps to multicast nicely.

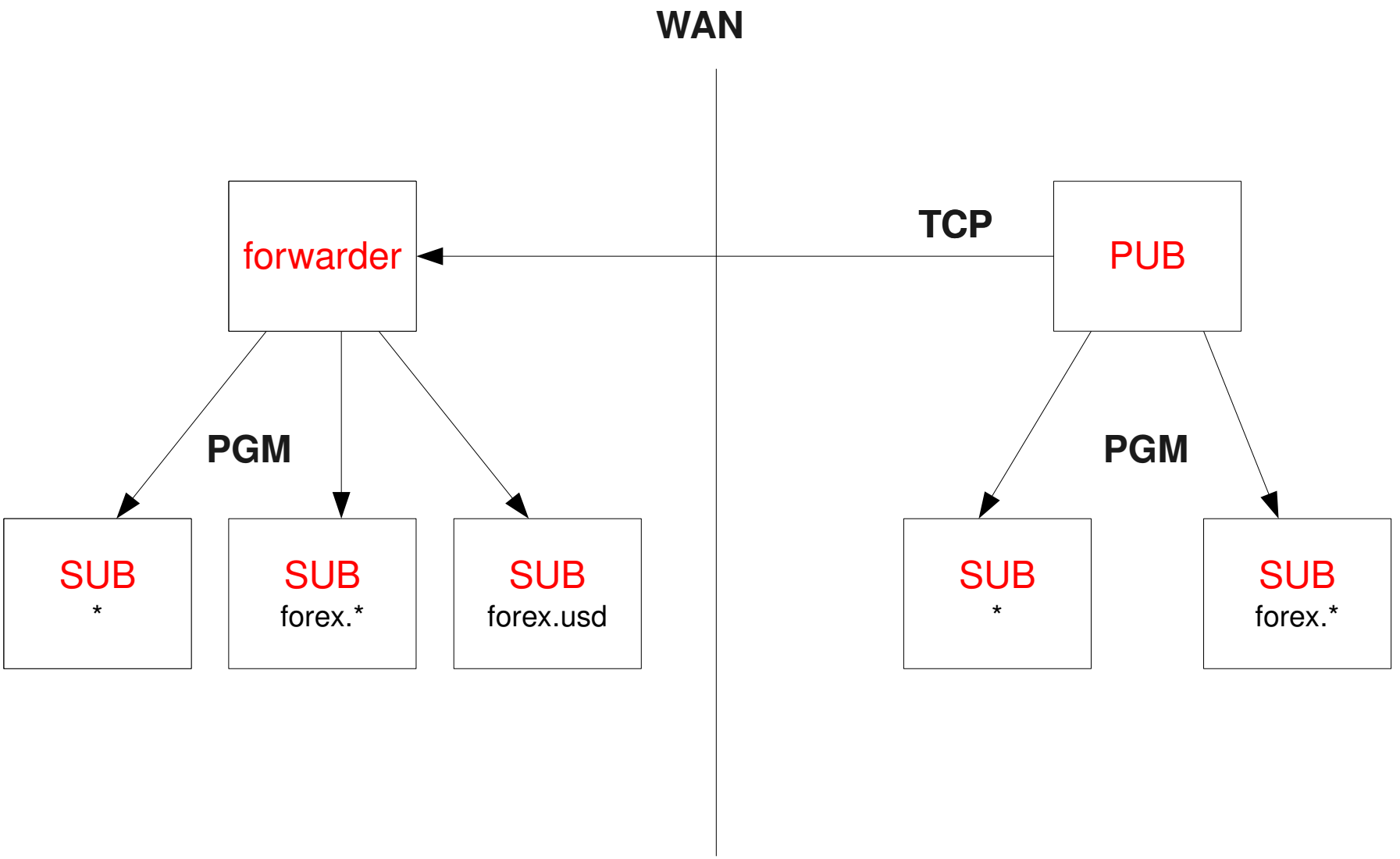
Simple publish/subscribe




```
zmq::context_t ctx (1, 1);
zmq::socket_t s (ctx, ZMQ_PUB);
s.bind (“pgm://eth0;224.0.0.1:5555”);
zmq::message_t quote;
while (true) {
    ...
    s.send (quote);
}
```

```
zmq::context_t ctx (1, 1);
zmq::socket_t s (ctx, ZMQ_SUB);
s.setsockopt (ZMQ_SUBSCRIBE,
    “forex.”, 6);
s.connect (“pgm://eth0;224.0.0.1:5555”);
zmq::message_t quote;
while (true) {
    s.recv (&quote);
    …
}
```

Complex publish/subscribe

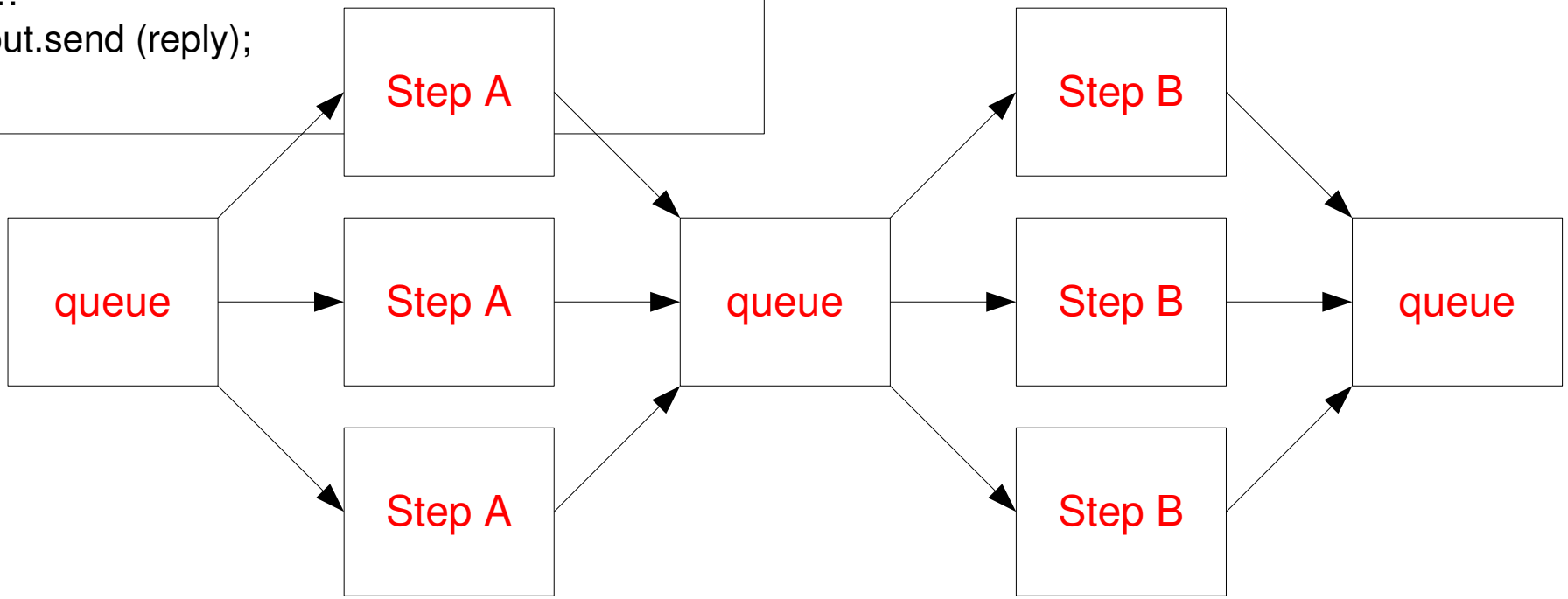


Message streaming

- Classic HPC or datacenter scenario.
- Solves the problem of distributing monolithic application on many boxes.
- Tasks are streamlined through one or more processing steps.
- Each step can be handled by multiple application instances (load-balancing).
- Application instances can be added dynamically to address actual needs.

Message streaming

```
zmq::context_t ctx (1, 1);
zmq::socket_t in (ctx, ZMQ_UPSTREAM);
s.connect (“tcp://192.168.0.111:5555”);
zmq::socket_t out (ctx, ZMQ_DOWNSTREAM);
s.connect (“tcp://192.168.0.112:5556”);
zmq::message_t request, reply;
while (true) {
    in.recv (&request);
    …
    out.send (reply);
}
```



Performance

- Linux exhibits end-to-end latencies of 30-40 us.
- Throughput on Linux is 3-4 million of small messages per second in a single stream.
- Getting the best possible performance on OpenVMS is a problem Brett will address in his presentation.

Martin Sústrik
sustrik@250bpm.com

OpenAMQ and ØMQ on OpenVMS

Ported to both OpenVMS Alpha and Integrity

No major challenges to porting

Standard C/C++ code

OpenVMS C RTL sufficiently comprehensive

Biggest job was porting Apache Portable Runtime (APR)

- OpenAMQ uses this extensively
- Port provided by Apache on OpenVMS very slow
- Probably took a day to port the bits of APR we needed

Found bug in TCP/IP services socketpair() function

- Has been fixed

Some limitations (but we're working on it)

Use of multicast (OpenPGM) with ØMQ

Still some work to do

Multi-threaded OpenAMQ broker still a little unstable (single-threaded is fine)

Performance optimizations

- Latencies down to 160 us with ØMQ
  - Room for improvement
- Can easily do 15000 messages per second with OpenAMQ
  - This will improve once we sort out a couple of issues
    - Threading
    - Polling

Support model

OpenVMS-specific AMQP API that can be called from any language

- Mostly done
  - Test programs in FORTRAN and COBOL

Futures

Working on providing support for these technologies on OpenVMS

Investigating inclusion of AMQP and/or ØMQ capabilities into RTR

Various other interesting plans that I really can't talk about...

- Watch this space!

Brett Cameron
Brett.Cameron@hp.com

Questions?