Towards messaging on an Internet scale

Martin Sústrik, <sustrik@250bpm.com>

Martin Lucina, <mato@kotelna.sk>

Problem statement

The Internet is a remarkably simple network. Data packets enter the network at one point, pass through a neutral infrastructure, and exit the network. While simplicity and neutrality are desirable, and indeed the worldwide success of the Internet can be attributed at least in part to these properties, communication patterns involving more than two peers are impossible to implement in an elegant, systematic and transparent fashion.

All stakeholders stand to benefit from elegant solutions to these problems. Content providers would be able to globally scale content delivery, service providers could provide distributed services on a global scale, ISPs could transparently provide better service to their customers and finally everyone benefits from lower infrastructure-related costs.

While one may object that mechanisms such as IP multicast and anycast already exist and can be used to solve these problems, their deployment on a wide scale on the Internet has been largely unsuccessful.[1] This is due to the fragile nature of the technology where minor error can lead to a large-scale collapse; therefore no administrator will allow end users to communicate using multicast unless absolutely necessary.

We propose to solve these problems by taking the largest systems providing many-to-many communication patterns that are used in production today – namely enterprise messaging middleware – as a starting point and gradually evolve these into a layer tightly integrated with existing Internet stack.

The long and winding road behind

The following is a point-by-point account of what we have achieved so far during the development of the ØMQ project.

Getting the messaging stack into order

Messaging systems have their origin in the mid-80's corporate environment. At that time the existence of a standard and fully functional network stack wasn't assumed. The design of corporate messaging took this fact into account and implemented almost all of the functionality that is available in the standard Internet stack today.

The relative isolation of corporate environments, combined with the long life cycles of enterprise software, resulted in a legacy that persists to this day. Functionality remains on the messaging level, virtually duplicating almost every feature of the Internet stack. Moreover, backward compatibility issues with legacy enterprise applications require any new messaging system to duplicate the design of the system being replaced.

In our design we take a completely opposite approach and instead of duplicating a traditional heavyweight messaging system we decouple individual features from the messaging layer, delegating them to the appropriate layer in the Internet stack, leaving ØMQ to implement only the

1 Previous experiments such as MBone have shown that it is practically impossible to implement a reliable infrastructure for multicast content delivery on an Internet scale using IP multicast alone.

core messaging functionality.

For example: Many legacy systems provide their own implementation of multiplexed virtual circuits. We delegate multiplexing to the IP layer, where it belongs. This approach eliminates the feared "head of line blocking" problem and it is network-friendly, meaning that existing networking components (routers, switches) can transparently apply sensible congestion control, traffic shaping, and so on.

Different example: We deliberately refrain from structured content. ØMQ messages are opaque binary data, to be contrasted with the rich system of data types provided by legacy messaging systems such as CORBA. Serialisation and de-serialisation of data can be easily implemented on the application layer using many existing libraries.

Yet another example: Flow control is delegated to the TCP layer, where it belongs. In this way we avoid broken models of flow control at the messaging layer as well as duplicate transmit and receive windows.

Fixing broken functionality

Delegating functionality to it's natural place in the Internet stack led us to compare the algorithms used by messaging systems with the algorithms used by the Internet stack.

We found that the heavily isolated nature of messaging system development prevents the flow of innovation between the Internet community and the world of messaging. For example, such a widely accepted algorithm as fair queueing[2] has as of 2010 not yet made it across the border.

Simplifying the wire protocol

Enterprise messaging is a world of proprietary protocols, products with closed protocols make up the majority of the market. Several attempts to introduce open protocols to the messaging sphere have been made over the years[3] however, none of these have yet succeeded.

We have participated in the standardisation of the AMQP protocol almost from it's inception in 2004, and over time we came to realise that the major obstacle to adoption of AMQP is it's inherent complexity. A complex specification makes implementation a costly prospect and compromises interoperability by the sheer amount of detail that may be misinterpreted by implementers.[4]

To counter this problem, we designed a minimalist wire protocol which can be described in a couple of paragraphs. Moreover, the protocol is designed as any well-behaved Internet protocol should be: It can be layered on top of an arbitrary underlying transport protocol.

Simplifying the API

The complexity of tasks performed by traditional messaging systems is naturally reflected by overgrown and convoluted APIs. Most of them are closed, the only widely accepted open API is called JMS, which is Java-specific and comes with a specification that is 138 pages long.

During the design of ØMQ we initially mimicked widely used messaging APIs. After delegating functionality to the appropriate place in the Internet stack as described above the original API no longer matched the functionality.

---

2 First described by John Nagle in RFC 970 in 1985 and extremely useful for preventing congestion caused by misbehaving senders.

3 Such as CORBA, AMQP, BEEP, XMPP.

4 To give reader some idea, the latest published AMQP specification is 280 pages long and for comparison the core CORBA specification is over 1000 pages long.

To solve this problem we designed a new API inspired by Berkeley sockets. This API is a simple extension of the concepts used by Berkeley sockets, mapping almost 1:1 to BSD socket API, flattening the learning curve almost to zero and leaving the way open towards the possible integration of both APIs.

Making it ubiquitous

For ØMQ as a solution which is steadily evolving towards a layer in the Internet stack, it is important to run everywhere. Our implementation is designed to be portable and currently runs on POSIX-compliant platforms, Microsoft Windows and OpenVMS.

For exactly the same reason it's important that the implementation runs – and runs well – on any available hardware, and scales across a range of options. On the top end of the range, it runs on large multi-core servers (we have tested scaling on machines with up to 16 CPU cores). On the low end it runs on embedded systems and vintage computers. Further, ØMQ has been designed and tested to scale from unreliable slow networks such as GPRS all the way to high performance data centre infrastructure such as InfiniBand and 10Gb Ethernet.

For ØMQ to be universally useful a key requirement is that it be language-agnostic. Our implementation thus provides the lowest common denominator, a C library. The ØMQ community is working steadily on providing bindings for a wide range of programming languages.[5]

Providing industrial strength

The ØMQ networking engine is extremely fast. By eliminating unnecessary calls to the underlying network stack, message throughput can exceed the network stack's performance by a factor of 3-4. For message latency, the engine adds only a couple of microseconds on top of the network stack.

The performance thus achieved is a prerequisite for ØMQ to be suitable for operation on a Internet scale, for example on the core backbone.

The long and winding road ahead

Today we are at the point where the Internet was in it's early beginnings. We have a design that works, and can be used in the real world in a controlled environment such as a data centre operated by a single enterprise. When interconnecting heterogeneous networks, the Internet is treated as a passive tunnel which transports data between endpoints.

The moment we wish to treat the Internet as an integral, active part of the messaging infrastructure a new set of requirements arises:

1. Intermediate network nodes should not be required to support or otherwise be aware of the messaging infrastructure. Conversely, this means that the infrastructure should be capable of operating using the existing status quo, i.e. treat the Internet as a passive tunnel by default.
2. Messaging infrastructure on intermediate network nodes should be completely transparent to endpoints, meaning that it would be possible to transparently install infrastructure on an intermediate network node without applications at the edge of the network even noticing the change.
3. Our current implementation treats the network as a trusted entity. For use on the Internet, suitable end to end confidentiality and authentication, denial of service avoidance and other security considerations must be addressed.

5 Including C++, Common Lisp, Haskell, Java, Lua, Python, Ruby and more.

4. The needed infrastructure should be developed as a fully integrated part of the Internet stack rather than a stand-alone domain-specific solution.

We believe that adhering to the points above, such an infrastructure will elevate messaging into the position of a first-class citizen on the Internet and that it's adoption will gradually expand from the edges of the Internet into it's core.