# MaXXlinks

Top

## Introduction

MaXX**desktop** relies on a robust high performance inter-process (support multi-threading as well) communication infrastructure. Message-driven architecture is a natural choice for fast, efficient and loosely couple communications, but yet simple. Enters MaXX**links**, a multi-languages, multi-protocols, broker free, stand-alone, high performance messaging and inter-process communication library.

**Here are the requirements for MaXXlinks:**

- support synchronous and asynchronous connections
- support multiple protocols
- support authentication
- support transport layer encryption
- support transport optimization for local messaging such as passing by reference using Shared Memory or zero-copy with ØMQ
- support message-driven architecture
- promote loosely coupling
- support connection patterns : one to one, publish /subscribe, router and dealer
- support distributed deployments
- provide a simplified implementation of the Enterprise Integration Patterns
- support for: C, C++, Python and Java
- provide an abstraction layer that could support different back-end providers
- support multi threading and signalling mechanism between threads
- must be light, low in dependency and broker free

## Build or buy

The eternal dilemma... With the current state of affair, very small pool of developers, limited free time and so much things to do, it makes no sense to decide to build from scratch an entire messaging sub-system that must comply to the above requirements. On the other end, the build option could make sense in a possible future. For that reason, MaXX**links** must be built with the future in mind and support some kind of abstraction layer allowing different back-end to be plugged in.

## The Choice

The decision was made to use ØMQ communication library as MaXX**links** back-end provider and built an

abstraction layer (wrapper) to ensure its ability to be back-end provider agnostic. ØMQ is a mature, robust and extremely fast messaging library to crossed on the boxes on our shopping list, and more. Applications build in C, C++, Python or Java (to name a few) can be integrated into the MaXX**desktop** environment with ease.

Now that the messaging back-end has been taken care of, we can focus on the real

# Messaging

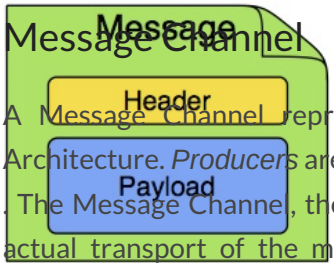
Since MaXX**desktop** is built as a multi layered architecture of inter-connected components (applications and services), messaging plays a critical role at different levels and helps bringing everything together into an homogeneous, yet simple loosely couple ecosystem. Advanced features such as: load balancing, tasks distribution, high availability, thread signalling, decoupling client-server with simple consumer/producer semantic are all possible (and in many cases rather simple to implement) through Messaging.

MaXX**links** will provide all the necessary features to connect components together regardless if they are running within the same process or not and on the same machine or across any network.

## Message

A *Message* is a generic wrapper to some metadata and a payload that is transported between *Endpoints* over a *Channel.* A Message contains headers and a payload. The payload can be anything technically (more on that later) and the headers holds important information like: unique identifier, destination, creation timestamp, expiration, payload type descriptor and optionally a reply-to *Channel*. Headers are containing context information regarding the message's intent and its content.

The payload can be of any types, but for performance and efficiency sake, MaXX**links** limits the payload type to *Text* , *Bytes* and *SharedMemoryLocator*. The *Text* type can be any sub-type of text such as XML, JSON or plain text. *Bytes* are used to transport raw information like an image or file, whereas *SharedMemoryLocator* allow to *pass-by-reference* larger chunk of data, within the same machine and without the Network-tax.

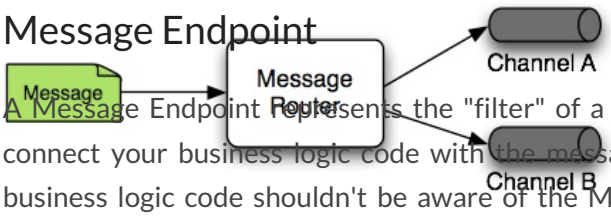



A Message Channel represents the "pipe" of *pipes-and-filters* high-level concepts of our Message Driven Architecture. *Producers* are sending Messages to Channels, and *Consumers* are receiving Messages from Channels . The Message Channel, therefore, decouples the endpoints from each other and provide an abstraction from the actual transport of the messaging sub-system. Channels also provide a convenient data collection points for intercepting and monitoring of Messages and the overall performance of the messaging layer.


Message Channel are either implementing Point-to-Point(queues), Publish/Subscribe(topics), Router or Dealer semantics. With Point-to-Point Message Channel, there will be at least one producer and a minimum of one consumer. There is no limit on the number of producer vs. consumer, but rather by the technical requirements and common sense. The most simplistic use-case is one producer and one consumer connected via a channel. This model can be pushed to one producer and many consumers connected via a channel that serves as a kind of load-balancer for parallel execution or router based on some Header based rule. Publish/Subscribe channels, on the other hand, will attempt to broadcast each Message to all of its subscribers, such as in a an event or notification

delivery feature.

## Message Endpoint




A Message Endpoint represents the "filter" of a pipes-and-filters architecture. The Endpoint's primary role is to connect your business logic code with the messaging framework in a non-invasive manner. In other words, the business logic code shouldn't be aware of the Message objects and Message Channels. Similarly to the a HTTP Controller class in the MVC paradigm, where the Controller handles a HTTP request, extracts incoming data, transforms it and forwards it to the business logic for execution, then does the reverse with the result of the execution, build a responses and sends it back. An Endpoint is doing pretty much the same things but under a standardized messaging framework.

As with the Model-View-Controller paradigm for web applications, MaXX**links** goal with the Message, Channel and Endpoint approach is to provide a thin standardize dedicated transport layer that extract and translate data from an inbound Message received from an incoming Channel, then invocation the business logic attached to the Endpoint and translates back the response into an outbound Message, that is sent to the outgoing Channel. That was a month full, but in essence pretty much it! In a successful Message Driven Architecture implementation, the value is created by this separation of concern and the measure of its success is by maintaining the business logic totally clueless of its surroundings.

Endpoints are asynchronous in nature and are ran in threads. Asynchronous multi-threading provides a much smoother and less spiky workload across all allocated CPUs/Cores. MaXX**monitor** and MaXX**scope** will ensure that is the case.

# Development

One of MaXX**links** main goal is to simplify the development of inter-connected components and applications through a set of already made components that accelerate the development cycle. Ultimately this means no one should ever have to re-implement low-level communication, transport layer, consumers nor the producers, but rather instantiate or extend existing components. Then plugin the business logic to Endpoints and go.

Refer to MaXXlinks Framework documentation for more detail.

---

Source:

https://maxxinteractive.com

https://www.enterpriseintegrationpatterns.com/

https://docs.spring.io/spring-integration/docs/current/reference/html/overview.html#spring-integration-introduction

Top

Revision #21
Created 10 January 2021 16:35:20 by Eric Masson
Updated 24 June 2021 12:24:32 by Eric Masson