MaXX Interactive Desktop Architecture

	Top	

Goals and Requirements

- Provide a secure modern and robust computational environment for smart applications and services.
- Support both local and distributed deployment.
- Be modular, expandable and flexible while supporting a loosely couple design.
- Do more with less. Meaning be smart in how resources are used and keep it in mind.
- Be as fast and efficient as possible. It is a balancing act and it always depends on the use-case.
- Support CPU architecture specific optimizations.
- Support Hardware Acceleration (GPU decode/encode & rendering).
- Support high performance asynchronous messaging for inter-application communications.
- Can run on different OS and CPU architectures (Intel/AMD, ARM and RISC-V).

Architecture

The MaXXdesktop's global architecture is based on the following five(5) principles in order to ensure the above goals and requirements are met.

- Follows the **SOLID Principles**
- Follow the **Clean Architecture**
- Support a Multi-layers architecture style for better/clearer separation with clear boundaries, responsibilities and dependency.
- Support a **Message-Driven** architecture for both inter-process communication and asynchronous event-driven execution.
- Support **Shared Memory** to prevent unnecessary copy/duplication of information when doing local (same machine) inter-process computations or when handling/displaying images over a local X11 session.

1.SOLID Principles

The SOLID principles were first conceptualized by Robert C. Martin in his 2000 paper, *Design Principles and Design Patterns*. These concepts were later built upon by Michael Feathers, who introduced us to the SOLID

acronym. And in the last 20 years, these 5 principles have revolutionized the world of object-oriented programming, changing the way that we write software.

The SOLID design principles encourage developers to create more maintainable, understandable, and flexible software by shifting their focus on the functionality rather than the low-level details. Consequently, **as an application grow in size, adeveloper can reduce its complexity** and save his sanity.

The following 5 concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. ~~**L**iskov Substitution~~ (not so much used anymore)
4. **I**nterface Segregation
5. **D**ependency Inversion

While some of these words may sound daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into what each of these principles means, along with a quick Java example to illustrate each one.

2.Clean Architecture

Clean Architecture is a way of developing software, such that just by looking at the source code of a program, you should be able to tell what the program does. The programming language, hardware and the software libraries used to achieve the objective of the program should become irrelevant. The aim of Clean Architecture is to make the following sentence possible: *'Hey, the arrangement of directories tells me this is a shopping cart app. I don't know which programming language or software library is used. I need to go into the directories and find out.'*

3.Multi-Layered Architecture

The architecture is inspired from multi-layered Enterprise class system where each layer defines precise responsibility and its application/service (a.k.a. component) exposes functionalities or behaviours. At the heart of the architecture are clear communications channels between components and their respective layers and well defined data-contracts exposed by each of them.

Overall, this design strategy will improve robustness of the Desktop experience, provide solid foundations to build upon, allow a clean separation of concern, modularity and overall re-usability. Another way to see MaXX's architecture design is to look at micro-services architecture, but from a Desktop application perspective where each layer of the architecture is composed of specialized micro-services performing specific tasks.

Layered Architecture by responsibility

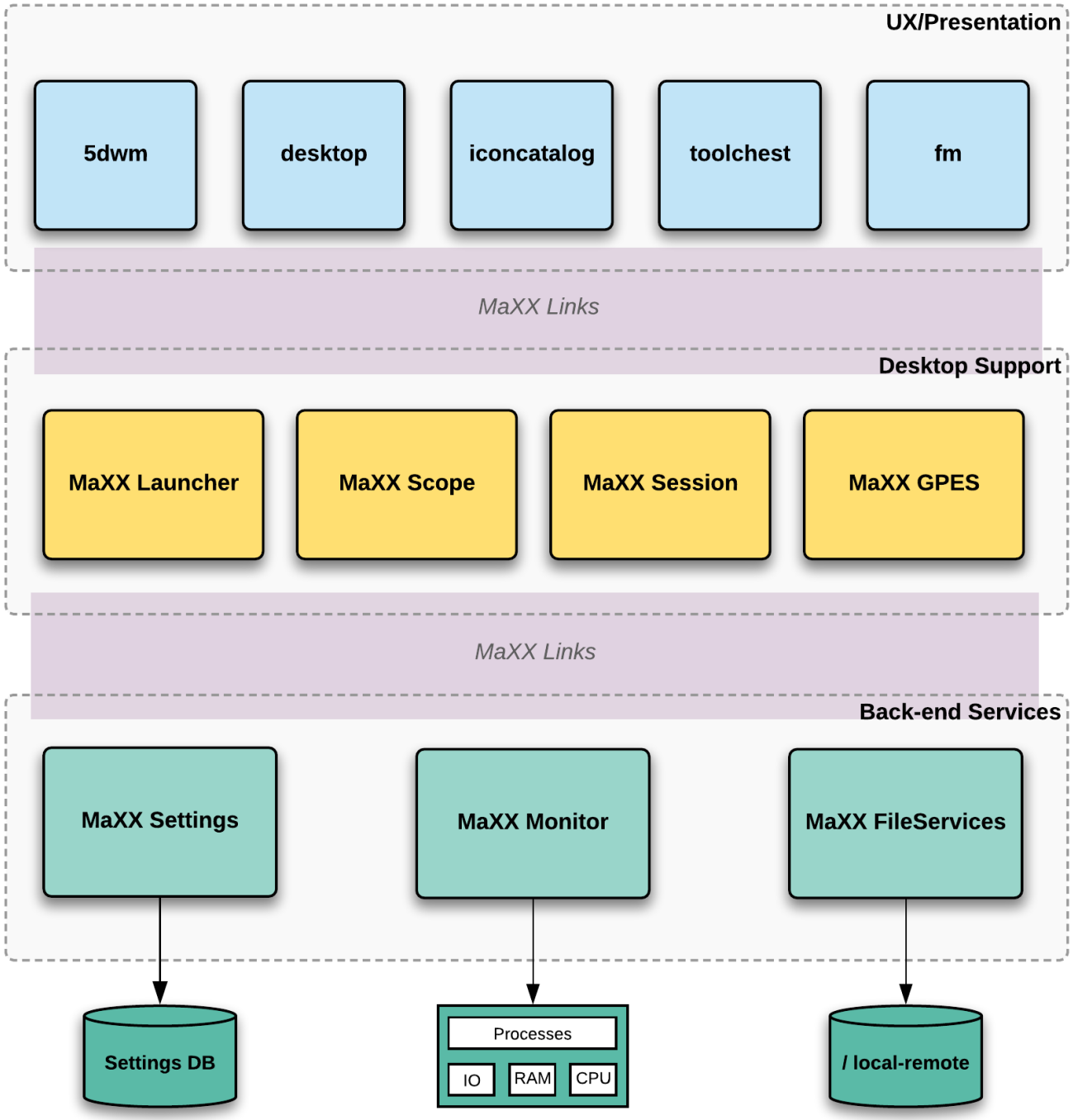
The MaXX**desktop** architecture is divided into three (3) responsibility layers from which MaXX's application and service can be built. Below are the layers in question with a short description for each:

- **User Experience/Presentation** - layer performing visual-oriented tasks like displaying User Interfaces

and/or capturing user's input. Components and/or applications on that layers are communicating with the Desktop Support layer for computation and resources access via proxy like services.

- **Desktop Support** - layer provides desktop computation support while providing an abstraction-layer to various Back-end Services, where most of the actions are taking place. This layer exposed some of those computations as functionally aggregator (composition and proxy design pattern) where a specific Desktop Support functionally is realized by utilizing one or more Back-end Service and an orchestration service that can coordinate the execution of other services/components.

- **Back-end Services** - layer is where most of the actual work is performed by low-level services/components. This layer can only communicate with components/services from the Desktop Support layer. Among the functionalities exposed by this layer are: hardware and application monitoring from MaXX**monitor**, file-system accesses and configuration management via MaXX**settings**.



The diagram below illustrates the current MaXX**desktop** architecture

MaXX Links (Inter-Layer Communication)

The MaXX**desktop** Architecture is planning tree(3) means of inter-layer communication mechanisms ensuring security and maintaining separation of responsibility.

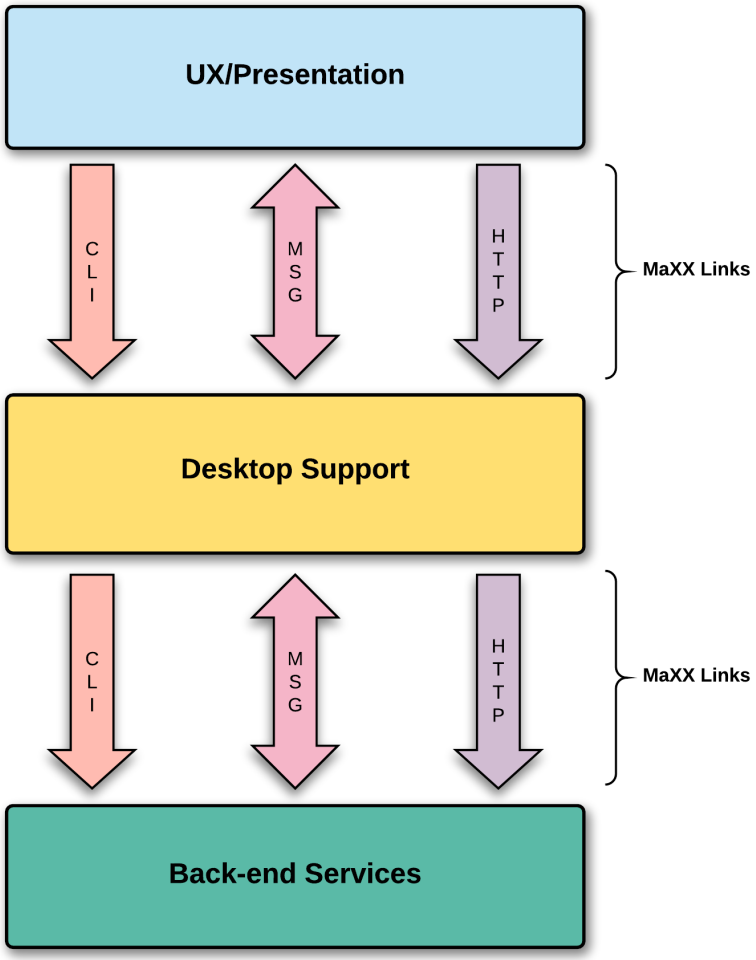
CLI or Command Line Interface - allows a command-line driven client/server like interaction where the *Client CLI* initiate a CLI session with the *Server CLI* counterpart and sends one or many CLI commands. The communication is obviously synchronous and based on a request/response semantic that is redirected through the

POSIX standard I/Os of the *CLI Client* application. Nothing prevents the *Server CLI* to emit an event during the processing of a CLI command. The CLI mechanism is great to provide a *fire-and-forget* interaction semantic.

HTTP or REST Interface - allows a Web/REST driven client/server like interaction where the *HTTP Client* sends a request to a *HTTP Service*. The communication is again synchronous and based on a request/response semantic using HTTP/S and predefined data types. Nothing prevents the *HTTP Service* to emit an event during the processing of the request. This mechanism is great to provide a a full features API style integration.

Messaging - allows synchronous, asynchronous or event-driven interactions where a *producer* create a message and send it to a destination via a *communication channel*. One or many *consumers* are at the receiving end of that channels allowing either *point-to-point* or *publish-subscribe delivery mechanism*. The payload of a message can either contain real data (passing by value) or provide a secured shared-memory location where the data can be retrieved (passing by reference).

A **message** is an item of data that is sent to a specific destination. An **event** is a signal emitted by a component upon reaching a given state. An event can be transported via a message, not the way around



The diagram below illustrates the inter-layer communication mechanisms.

4.Message-Driven

The Message-Driven architecture provides low dependency, no tight coupling and robust communication between components and services. MaXXlinks Framework will provide all the necessary features and abstractions to supports modern multi-protocols synchronous/asynchronous messaging communications. Things such as: load balancing, tasks distribution, high availability, decoupling client-server with simple consumer/producer semantic are all possible (and in many cases rather simple to implement) through Messaging.

Messaging also introduce the ability to support a polyglot code base where components integration are performed via high performance messaging channels while supporting both local and distributed environment. It does not really matter any more in which programming language the component/service is written, as long as it supports Messaging ØMQ and complies to predefined data-contracts.

Through MaXXlinks, any C/C++/Python or Java application can be integrated into the MaXX**desktop** environment or new modern features can be added to an existing application very easily.

The Messaging approach as been proven to yield excellent values by allowing ongoing improvement of components, total decoupling between producers and consumers and bring robust asynchronous and event-driven capability. As long as data-contracts are respected, changes documented+communicated, with versioning and backward compatibility supported, this is one of the best way to do inter-application communication.

The MaXX Interactive team bring more than 2 decades of real world expertise in High Performance Messaging Systems. That must count for something :)

5.Shared Memory

The use of Shared Memory for inter-process communication is not new, but it is not often considered due to its complexity and step learning curve. Shared Memory does not apply for distributed computing. Although Shared Memory has been part of X Windows (X11) as an Extension (MIT-SHM) since the early days of X11 as a was a very efficient mechanism for sharing information between local X11 applications, many choose the easy road with excuses such as 'the penalty is small' over doing things right and the most efficiently possible. The general idea is to avoid the X11 taxes (network stack and round-trips to the X Server) as much as possible, when sharing large data-sets like images.

This practice is used throughout the MaXX**desktop**, its applications and in the MaXXvue thanks to MaXXlinks.

Using Shared Memory in local Messaging is also a very important aspect of the architecture and help addressing the efficient and smart technical requirements. In this context, Shared Memory is used to avoid unnecessary copy/duplication of information when sending or receiving messages to and from another application. We call it, zero-copy transport. Instead of passing the information by value into the message's payload, only a secured Shared Memory locator is part of the payload (passing by reference).

We utilize this '*passing by reference*' strategy quite heavily in GPES (General Purpose Execution Service) as a very

efficient way to consume computation results without the net-impact. In some cases, all the work/computation, memory allocation and display are all performed by the GPU.

Sources
https://blog.flexiple.com/clean-architecture-build-software-like-an-artisan/
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

Top

Revision #46
Created 9 February 2020 23:04:09 by Eric Masson
Updated 23 June 2021 21:00:02 by Eric Masson