NULSTAR

1] Introduction

Nulstar is a C++ port of NULS protocol built using microservices as its base architecture, and a publish-subscription development pattern, both provide multiple advantages especially with medium-large scale deployments common in enterprises.

The main objective is to provide a lot more flexibility on how modules are developed and deployed. Modules will no longer be confined in a single environment or to a single programming language, so modules won't have hard dependencies between each other.

2] Simplified General Design

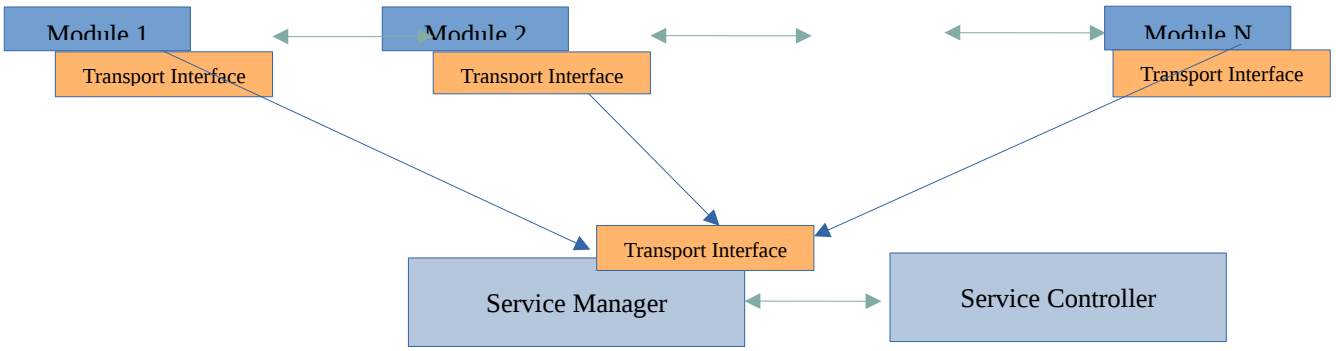

Fig 2.1

2.1] Modules

The modules are separate services with their own memory space. During execution each module registers its *ROLE* with respective API and connection information to the Service Manager for any other service (module) that needs its functionality. Upon receipt of the connection information each module is able to connect directly to the desired service.

Depending on the functionality, specifically if a module is stateless, it could be cloned to improve throughput. In this the case, the Service Manager could act as a load balancer.

For each type of input/output functionality available in a service, a separate layer of abstraction is created so that it becomes easier to create alternatives without changing any code in the service itself. For example if the module 'Store_Transactions' needs to store data to disk then a variety of *plugins* can be developed and loaded during runtime determining how and where you can perform this action. These plugins could include Mysql, Elastic search, LevelDB, MongoDB, File as CVS, etc. The *plugins* should be available to all modules.

2.2] Transport Interface

This interface define the mechanism by which components talk to each other. The initial implementation will be done using http/json because most developers know this technology and it is very easy to debug. If performance hit is too high, a binary plugin could be developed to minimize the penalty.

2.3] Service Manager

This component is responsible for registering the APIs of each ROLE and services connection information so that each module can query which service is offering the desired functionality in runtime. This information can change anytime without shutting down any component. For these reasons it is the first component that must be executed.

When developing an application around the core architecture, like a user interface or any other app, it will serve as the entry point exposing the APIs of the modules that are running currently. Therefore, the internal structure is completely transparent for applications because they will 'see' a single API.

2.4] Service Controller

This component is responsible to start and stop modules and to coordinate updates; updates can be done per module if the API is backward compatible

3] Features

3.1] Complete decoupling of modules

The architecture guarantees that there are no hard dependencies between modules, so it is extremely easy to add new services runtime to the platform. If a module can't find any service that can provide what it needs then it can gracefully report to other components about this situation without crashing the whole application.

3.2] Maximum deployment flexibility

As every module is effectively a separate application, it is possible to put each one in a separate environment according to specific needs of each company. Imagine for example that in a specific application (Module_2) is a performance intensive service then, in this case the company can just deploy several clones of it in multiple servers to scale accordingly (Fig 3.2)

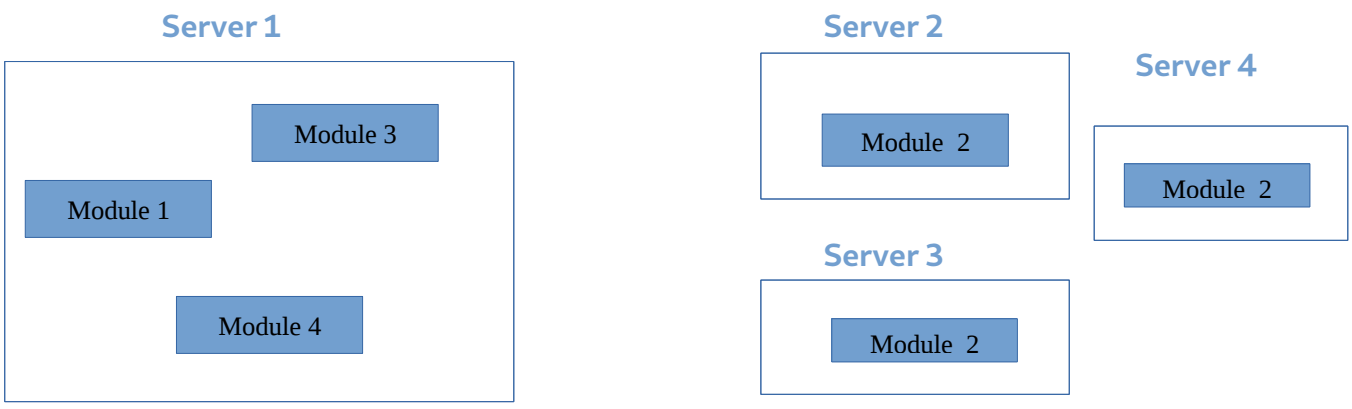

Fig 3.2

Microservices architecture blurs the line in defining what exactly a *node* is, for example assuming that (Module_4) is responsible of logging then in a traditional deployments this module should be present for each node currently running but with Nulstars just one instance of (Module_4) serving two or more nodes could be deployed and thereby reducing costs considerably.

3.3] Near zero down time

Due to its architecture, it is a lot easier to provide continuous uptime service. If a module crashes then it can gracefully be deployed again automatically using standard cloud tools. The messages that had to be delivered to it, are just put on hold for few seconds until another instance is deployed. Also if a module is critical then the enterprises can deploy multiple working units in parallel and Service Manager will act as load balancer.

Even upgrades could be done without restarting the whole infrastructure because each module can be replaced individually and the upgrades can be done in steps. Also, two versions of the same module could be deployed so that an enterprise can create a rule where it uses the new version just 5% of time to test real dynamics. This is done without risking the whole set of incoming commands, and of course if the upgrade fails then the previous version is restored gracefully and automatically.

3.4] Maximum development flexibility

The infrastructure is language agnostic so modules can be developed using different programming languages. The decision is made by developers themselves, and the only requirements are that modules use the same transport mechanism (http/json will be the first plugin developed) and that they expose an API corresponding to a specific ROLE and then send this info to the Service Manager.

This will allow developers to focus on a single module instead of learning the entire infrastructure. By focusing on just the functionality they are interested in, the learning curve will be a lot lower, and they can become productive in a shorter time span.

Enterprises can add modules without touching the core infrastructure to fulfill their specific needs. For example a module named 'CALCULATE TAXES' could be developed and integrated into the infrastructure run time!

It is also possible to integrate modules that are not necessarily crypto-related, such as a connector to their accounting system. The possibilities are in fact limitless and that is after all what NULS is about 'NULS is everything!'

3.5] Improved security and quality assurance

As the modules are compact and decoupled pieces of software, code reviews and tests can be done in an efficient manner. Every change just affects the integrity of the module being developed and the rest of infrastructure is not affected. It also allows for the flexibility to give different levels of testing to each module. For example, some critical modules can be audited by an external party without the need to review all infrastructure.

As these services may be executed in different servers, it is more difficult to compromise the whole platform

3.6] Publish - Subscription development pattern

Most applications in crypto world are designed and developed using traditional techniques that are not well suited when internal states are changing constantly and you will need to update your application accordingly. For example when a node is downloading transactions there is a function which returns how many transactions have been processed. This function needs to be polled constantly to update the application, but polling is one of the most inefficient ways to transfer information. With Publish - Subscription development pattern, this problem could be eliminated altogether because the function will update constantly at specific periods or events defined by the user of the function. When the application no longer needs the information it just 'unsubscribe' from the function.

3.7] Dedicated user interfaces

The user interfaces will be applications completely decoupled from the core architecture, and the only thing binding them is the transport protocol and an API. This is very important because user interfaces have very short development cycles, whereas the core has long development cycle; multiple developers and designers can work in parallel without the need to coordinate with each other or even with the core development team. This property will allow the development of multiple interfaces to fulfill different needs since companies may need powerful and complex interfaces, while others light and novice-friendly ones, some may prefer web applications while others natives ones.

This flexibility also allows the creation of user interfaces that controls a full set of nodes, so companies could have full management capabilities for each and every node using a single interface, including consolidated reports.

4] Considerations for the future

Every crypto out there needs to develop an organic ecosystem around their respective platforms in order to be viable long term. Lowering the learning curve and offering developers options to be more productive is a great start, but it is necessary to develop tools that allow people and companies to build businesses around our platform. For this to happen, two projects have been suggested that should be integrated to our platform.

4.1] Store

We have discussed in topic 3.4] how enterprises could easily develop and integrate their own modules, however there could be situations when it is more cost effective to simply hire an external developer to build the desired functionality, or alternatively just purchase an existing solution. For this reason, it will be important to build a store where developers can offer their modules at a price or even make it open sourced with zero fee.

4.2] Bounty site

Companies or individuals should be able to commit 'bounties' for a desired module to provide direct incentives for developers to start developing around our platform.