Summary: Software Define Network (SDN) is a recent paradigm based on the separation between the data plane and the control plane, allowing to handle the traffic network by means of software. The SDN ecosystem is fragmented due to the multitude of different controller platforms. This creates a danger of a controller lock-in for SDN application developers and for SDN network operator. In order to tackle this problem an innovative SDN architecture is presented whose aim is to execute SDN applications written for different controller in an unique network. This architecture presents an important problem to debug and analyze the SDN network. Therefore, a set of tool is design and developed with the purpose of solving this limitation and guarantee the proper operation of the network. Three of these tools are detailed in depth, a logger which displays the messages that cross the architecture, a profiler that shows information of the applications and parameters of the network and a model checker whose main task is to validate important properties of the network such as assure that there are no black holes.
Tools for a Multi-Controller SDN Architecture

Sergio N. Tamurejo Moreno
Universidad Carlos III de Madrid
IMDEA Networks Institute
Email:sergio.tamu@gmail.com

Abstract—Software Define Network (SDN) is a recent paradigm based on the separation between the data plane and the control plane, allowing to handle the traffic network by means of software. The SDN ecosystem is fragmented due to the multitude of different controller platforms. This creates a danger of a controller lock-in for SDN application developers and for SDN network operator. In order to tackle this problem an innovative architecture is presented whose aim is to execute SDN applications written for different controller in an unique network. This architecture presents an important problem to debug and analyze the SDN network. Therefore, a set of tool is design and developed with the purpose of solving this limitation and guarantee the proper operation of the network. Three of these tools are detailed in depth, a logger which displays the messages that cross the architecture, a profiler that shows information of the applications and parameters of the network and a model checker whose main task is to validate important properties of the network such as assure that there are no black holes.

I. INTRODUCTION

Software Defined Network (SDN) is an emerging paradigm which stands as the key to solve the limitations and difficulties of the current networks which has the data and control planes vertically integrated [1]. Its control plane (responsible of the logic that handles the traffic) and data plane (whose task is forwarding traffic according to the logic established by the prior plane) are bundled in the same forwarding element. Therefore, to introduce a protocol in this type of networks, each device must be configured individually.

SDN allows the separation between the control plane and the data plane, giving rise to a logical entity called controller, which implements into the underlying forwarding elements the necessary logic to handle the traffic of the network. Hence, the switches forwards the incoming packets according to the rules installed by that controller. If a switch do not have a rule for a packet, the packet will be sent to the controller and this logical entity will notify to the device what to do with it (drop the packet or forward the packet).

The separation between both planes can be realized by means of a well-defined programming interface between the forwarding elements and the SDN controller. The most notable example of such an API is OpenFlow [2]. The main advantages the SDN paradigm provides are: flexibility, simpler network management and faster evolution and innovation of the network [3]. In Fig.1 is represented the infrastructure of a SDN network.

The impact of this new technology has been of great importance. It began as an academic experiment and has become (in just a few years) in one of the best embraced paradigms by industry. In fact, some of the biggest companies such as Google, Facebook, Yahoo, Microsoft, Verizon and Deutsche Telekom founded the Open Networking Foundation (ONF)[11] with the aim to promote and adopt SDN through open standards development.

Nowadays, SDN paradigm is considered as a researching spotlight and a lot of work is being carried out by industry and academia. This effort of the networking community has resulted in a wide variety of open and closed-source controllers, e.g., Opendaylight [4], Ryu [5], Pox [6], Nox [7], etc. (Each one has its own programming model, execution model and capabilities). This has led to a fragmented control plane, that forces to network application developers as well as to network operators, to choose one control platform, which is a hard decision since all have their strengths and weaknesses.

Controller platform diversity would be an asset, if porting SDN applications among different controllers was an easy task. However, this is not the case and the danger of a controller lock-in arises. To tackle the problem and to enrich the flexibility of using SDN in a network, a platform is presented, whose main goal is supporting the execution of SDN applications across different controllers in the same network. Such platform allows to deploy multiple controllers called ‘Client Controllers’ inside a single network, each of them run their pertaining network applications. These Client
Controllers communicate to an unique controller (known as 'Server Controller'), responsible for the communication with the underlying Network Elements. This platform is denominated 'NetIDE architecture' and has been designed by the consortium of the FP7EU Project NetIDE [8] and described in detailed in [9].

Nevertheless, the NetIDE architecture presents one important issue that must be solved. How a network deployed using this platform can be debugged, since a bug in a SDN network application could suppose an Achilles heel to the performance of the whole network. Therefore, the main contribution in this paper is a set of tools which is completely integrated in the platform and whose purpose is diagnose, debug and troubleshoot a SDN network ensuring its proper operation.

The following section II introduces the design principles of the architecture. Section III presents a toolset for the proposed architecture. Section IV provides an overview of the state of the art. In section V open issues and relevant future directions are discussed. Section VI summarizes the main conclusions.

II. NetIDE ARCHITECTURE

The NetIDE Architecture is depicted in Fig. 2, which is breakdown in three layers: (i) the Integrated Development Environment (IDE) (ii) the application repository which stores simple modules and composed applications and (iii) the Network Engine, where controllers and applications are executed.

![Fig. 2. The NetIDE Architecture](image)

The IDE is basically composed by several code editors that support different network programming languages and a particular graphical editor to create topologies. It reads modules and applications from the repository and also can do the reverse path (writing applications and modules to it). Finally, the IDE is able to deploy those applications on the Network Engine to be executed. Similar to the architecture, the Network Engine is structured in layers, specifically in two. In the upper layer are the Client Controllers which will run their respective modules or applications. In the bottom layer a unique Server Controller will be the responsible of talking directly to the underlying data plane.

The main challenge lies in the communication between the Client Controller layer and the Server Controller layer. The upper one, must be able to transmit all the messages generated by the network applications to the bottom layer and vice-versa. This task is not trivial, due to the client’s South-bound interfaces (SBI) do not match with the server’s North-bound interfaces (NBI). Hence, there is a need of an intermediate protocol which make feasible the intercommunication between the above layers. To solve this issue, we develop two different modules, the former called 'Backend' integrated in each of the Client Controllers and the latter denominated 'Shim layer' incorporated to the Server Controller. These modules speak the same protocol and the interaction becomes possible.

A. Intermediate Protocol

The Intermediate Protocol covers three essential needs for the intercommunication between the Client Controller layer and the Server Controller layer. (i) Transport control messages between Backend and Shim layers, e.g., to start up/take down an application or a particular module fixing a unique identifier for each of them. (ii) Carry event and action messages between Backend and Shim layers and demultiplex properly those messages to the right modules or applications according to the unique identifier fixed previously. And finally, (iii) encapsulate messages specific to a particular SBI protocol version (e.g., OF 1.X, NETCONF, etc.) towards the Client Controllers with proper information to recognize these messages as such.

In the first prototypes of the Network Engine, we leveraged the Pyretic Protocol [10], although it was very useful to accomplish some preliminary proofs of concepts, its current version limits the functionality of the network applications running on top of the Engine Network, due to this protocol only supports a restricted subset of OpenFlow v.1.0 messages. Therefore, a new intermediate protocol was implemented from scratch. This new protocol uses TCP as transport and encapsulates the payload with the following header:

```
netide_ver | type | length  
xid       | module_id |          
datapath_id |
```
Where netide_ver is the version of the NetIDE protocol, length is the total length of the payload in bytes and type indicates the type of the message (e.g. NETIDE_HELLO, NETIDE_OPENFLOW, etc.). datapath_id is a 64-bits field that uniquely identifies the Network Elements. module_id is a 32-bits field that uniquely identifies the application modules running on top of each Client Controller. The composition mechanism in the core leverages on this field to implement the correct execution flow of these modules. Finally, xid is the transaction identifier associated to the each message. Replies must use the same value to facilitate the pairing.

III. TOOLS

In the architecture exposed before, exist two well differentiated classes of tools. The first one, groups all the set of tools whose purpose is to provide resources and facilities to the user for designing, creating and deploying a SDN network, whilst the second group aims to debug, inspection and diagnose the deployed network. The first class encompasses three essentials tools which are summarized in Table I:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Functionality</th>
<th>Execution Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology Editor</td>
<td>Graphical editor to create and modify network topologies.</td>
<td></td>
</tr>
<tr>
<td>Network Programming Language Editors</td>
<td>The editors to implement network applications support several programming languages as Python for Pyretic, POX and Ryu controllers; or C/C++ for NOX controller or Java for Floodlight, Beacon and OpenDaylight controllers.</td>
<td></td>
</tr>
<tr>
<td>Deployment Configuration</td>
<td>User interface in which developers can choose a previously specified topology model and the users can choose a platform and the path of the network application code for each controller in the topology model.</td>
<td></td>
</tr>
</tbody>
</table>

TABLE I
TOOLS FOR DESIGNING AND DEPLOYING A SDN NETWORK

The goal of the diagnosing set of tools is to provide a framework that enables the developer to systematically test, profile and troubleshoot the SDN network, besides tuning the network applications. The first efforts to achieve this objective in the project are focused on identifying the different tools needed in Software Defined Networking (SDN) environments. Currently, we have identified the following categories: Logger, Simulator, Model Checker, Resource Manager, Garbage Collector and Profiler. A taxonomy of the tools according to its functionality and its execution nature, is presented in Table II.

The Execution Nature field only can take two values, Runtime or Off-line depending on the moment the tool is executed. If the tool is executed when the SDN network is deployed and the Network Engine is running then we refer to it as Runtime tool. However, if the tool needs to be executed without being the Network Engine working, then the tool will be categorized as Off-line. The Functionality field provides a briefly description of the utility of each tool.

In the subsections III-A, III-B and III-C a detailed analysis will be made about the Logger, Profiler and Model Checker tools, respectively.

A. Logger

This tool captures the messages exchanged between the data plane (Network Elements) and the control plane (Client Controller, where a network application is executed) and print these messages into a screen, allowing to the user to visualize and filtering them accordingly to the type of message. It is based on RabbitMQ (a description of this technology is detailed in the Appendix). The Shim layer is modified to include a module that queue the messages required by the tool. In this specific case, those messages are all the ones that the Shim layer intercepts. The software architecture of the tool and its integration within the overall architecture are detailed as follows and illustrated in Fig.3.
The LogPub module, integrated inside the Shim layer, is responsible for intercepting all the messages that arrive to the Shim layer (i.e., the messages exchanged between the Networks Elements and the Client Controller), replicating them and sending back to the broker element. The second module called Logger, will retrieve the messages kept in the broker and will print them in a terminal.

The Fig.4 represents two instances of the second module, Logger (1) and Logger (2), which are consuming the messages kept in the broker and sent by the LogPub. Firstly, the first module establishes a connection with the RabbitMQ server (Broker) and captures the messages in order to send them to the Exchange element (called X in Fig.4). This new element is in charge of knowing what to do with the messages. In our case, it has been configured for sending the messages to the different queues depending on the type of message that the user wants to visualize. If there is no consumer (an instance of Logger module running) there will not be any queue, hence the Exchange element will drop the messages received. Finally, the second module (Logger) retrieves the messages of the queues and print them in a terminal. The user has the possibility to choose what kind of message wants to observe. There are two types of messages, in or out. The former, are messages sent by the Network Elements to the Client Controller and they are represented in green color. The latter, are the messages which do the reverse path (from the controller to the network) and are represented in yellow.

To simplify the complexity and limit the development of the tool in time, we started the practical work with a restricted version of the architecture. In particular, we made the following assumptions: (i) a single Client Controller scenario. (ii) The development has been performed for Ryu Shim layer.

In Fig.5, is represented the outcomes obtained when the Logger tool is running. The image displays the messages sent from the data plane to the controller (in green) and from the data plane to the controller (coloured in yellow). The structure of the messages depicted in the image is the following: the first field is a timestamp to know when the packet was sent, next information provided in the message is the direction of the message (to the controller or to data plane). The last one, is the content of the message.

B. Profiler

The Profiler tool is used to analyze programs, providing to the user relevant statistical information, such as the execution time of various parts of that program or the consumed RAM memory.

The Profiler has followed two different approaches. The first one (Application Profiler), is based on the classical behavior of the tool (described above). It is able to provide the execution time of the functions of the analyzed software. Exist two options, analyze only one SDN application or analyze all the applications which are being executed plus the controller framework. Moreover, the Application Profiler provides the memory consumption of the Client Controller and the executed applications. The second approach called Network Profiler focuses on the network and offers statistics of the forwarding elements (number of rules installed by the controller and number of removed rules).

1) Application Profiler: To accomplish the development of the first approach, two assumptions have been established: (i) The Client Controller is a Ryu controller, (ii) Only one application is executed. Moreover, we have used two different Python modules, one called cProfile for obtaining the execution times of the methods and other denominated memprof for getting the memory usage of the profiled functions. The logic process for obtaining the execution times of different methods is depicted below in Fig.6.

While the SDN application is being executed, we gather relevant data. Once the execution has stopped, the information contained in the binary file is dumped to a text file. This new file is processed and read with a software for statistical computing and graphics. Finally, three graphics are generated to show the outcomes. The procedure to get the memory usage is identical.

The outcomes produced are the followings: a text file where is specified the execution time of each analyzed function, a graphic which contains the same information but more visual and a histogram of the amount of functions that have a specific execution time. In the case of memory consumption it is shown a graphic that illustrates the memory consumed by the controller plus the executed application.
Fig. 6. Application Profiler flowchart

Fig. 7 and Fig. 8 are graphics obtained with the Profiler tool after having obtained data from the execution of the controller plus the SDN application.

Fig. 7 is an histogram and represents the amount of functions that share the same execution time. For instance, there are approximately about forty two functions with execution times closed to zero seconds and about ten functions which take longer than 300 seconds.

In Fig. 8 it is displayed the execution time of the sixty functions that consume more time when are executed. In abscissa axis are specified the names of those functions.

In both figures, there is a legend in the right side. It specifies the units and the range of values that ordinate axis can take. With these outcomes, the user can know whether or not is necessary a code optimization in the SDN applications that are executed in the SDN network.

2) Network Profiler: In this approach it is assumed that the Server Controller is a Ryu controller. Similar to the Logger, a module denominated ProfPub is added to the Shim layer which captures the messages denominated flow mod. Those messages are sent by the controller to remove or install rules in the switches. Once captured, the messages are parsed and it is possible to identify if the content is for installing a rule or for removing it. Therefore, two counters are implemented one for installed rules and the other for removed rules. With this procedure we know the amount of rules installed and removed in the switches.

A second module called Network Profile accesses to the aforementioned counters, read them and print the information into a screen. This method offers outcomes in real time while the SDN network is running.

C. Model Checker

Model Checking is an automatic verification technique for finite state concurrent systems. It automatically provides complete proofs of correctness. A model checker systematically explores all possible ways to execute a program (as opposed to testing, which only executes one path depending on your input data).

The process for model-checking a software consists on three main steps:

- Modelling: converts the system into a formalism. For example, a state diagram. It is necessary to identify all the system states and the transitions from one to another.
- Specification: Determine the correctness properties to be checked by the tool. For instance: no loops in a SDN network or the existence of black holes.
- Verification: validates the correctness properties in the modelled system and offers the outcomes obtained (whether or not the property is fulfilled in the system).

The main advantage of this tool is that all possible cases are contemplated, so the outcomes obtained are absolute, which means that if the property *no loops in the system* has been validated, the network will never have a loop. Therefore, we consider it as a tool needed for a SDN network, due to it allows to validate important properties of a network before deploying it.

The most pronounced disadvantage in this technique is the *State Space Explosion* problem. It consists on having a huge state space, therefore the validation of a property would suppose a high complexity and computationally would be inordinately costly.

This issue always appears in SDN networks, because the behavior of a SDN network application depends on many factors: the end-host applications sending and receiving traffic, the switches handling the incoming packets, installing rules and generating events (which are sent to the controller). All of them affect the program running on top of the controller leading to endless possibilities (states with transitions). Consequently, we consider that develop a Model Checker for validating properties in SDN Networks from scratch is not feasible.

The solution adopted was to study in depth a Model Checker already created and integrate the tool to this project. Very few such tools have been developed. Nevertheless, there is an interesting open source project called NICE which provides a model checker for old NOX controller [12].

NICE addresses the State Space Explosion issue by mean of the *Symbolic Execution Engine*. They consider that the event handlers of the SDN application are the key to explore the whole state space. These handlers must be triggered in order to exercise all different code paths. Hence, the first option is explore all possible inputs for triggering them (which it does not scale) and the second option is to identify the packets that trigger these events (Symbolic packets) and feed the network with them. This task is performed by the *Symbolic Execution Engine*. The Fig.9 represents the structure of NICE tool.

![Fig. 9. NICE tool structure](image)

Given a SDN network application, a network topology and the correctness properties to be validated, ‘NICE performs a state space search and outputs traces of property violations’ [12].

A research is being carried out to adapt NICE to other controllers platforms. The accomplished work has been the creation of a simple SDN application for old NOX controller and the execution of NICE tool with this input. The future work is to create an UML diagram of the entire tool in order to understand properly how it works and find out which is the next step to migrate NICE to Ryu controller.

IV. RELATED WORK

One of the main purposes of this project is to allow the composition of SDN network applications with software modules implemented for different SDN controllers. In this context, few solutions have been proposed. Two of the most relevant contributions in this field are FlowVisor and OpenVirteX [13], [14] which are network hypervisors that split the traffic into slices. Each slice manages its traffic by mean of a network application. Both of them suppose an important advance, improving the security and flexibility of the network due to the integration of various controller operating at the same time. Moreover, these projects offer to multiple tenants to share the network. However, the network applications of each slice cannot cooperate to process the same traffic, so it is similar to have different networks manage each one by a unique controller.

Flowbricks [15], is another SDN hypervisor that integrates heterogeneous controllers using only the standardized controller to switch communication protocol. The most flagrant point of this project, is that it only runs on emulated environments and not over standard network hardware.

Debugging and troubleshooting have been important subjects in computing infrastructures, parallel and distributed systems, embedded systems, and desktop applications [1]. The two predominant strategies applied to debug and troubleshoot are runtime debugging (e. g., gdb-like tools) and post-mortem analysis (e. g., tracing, replay and visualization). Despite the constant evolution and the emergence of new techniques to improve debugging and troubleshooting, there are still several open avenues and research questions [16]. Debugging and troubleshooting in networking is at a very primitive stage. In traditional networks, engineers and developers have to use tools such as ping, traceroute, tcpdump, nmap, netflow, and SNMP [17] statistics for debugging and troubleshooting. Debugging a complex network with such primitive tools is very hard. SDN offers some hope in this respect. The hardware-agnostic software-based control capabilities and the use of open standards for control communication can potentially make debug and troubleshoot easier. The flexibility and programmability introduced by SDN is indeed opening new avenues for developing better tools to debug, troubleshoot, verify and test networks.
Therefore, other of the most important objectives in this project is the provision of a varied toolset for debugging, diagnosing and troubleshooting the SDN network. We briefly describe and analyse similar efforts in this area, such as NetSight [18] which includes different tools such as: an interactive network debugger, a live network monitor, a logger and a hierarchical network protocol. NetSight assembles packet histories using postcards, event records sent out whenever a packet traverses a switch. This approach decouples the fate of the postcard from the original packet, helping to troubleshoot packets lost down the road, unlike approaches that append to the original packet. Each postcard contains the packet header, switch ID, output port, and current version of the switch state. Combining topology information with the postcards generated by a packet, NetSight can reconstruct the complete packet history: the exact path taken by the packet along with the state and header modifications encountered by it at each hop along the path. NetSight is developed in C++, it focuses on the following controller frameworks: NOX [7], POX [6] and RipL-POX [19], but it does not specify the OF versions supported.

NICE [12] is a powerful model checker whose objective is uncover bugs in SDN network applications. To reach this goal, they propose a tool that combines model checking and symbolic execution. Given a SDN network application, a SDN network topology and the properties to be checked (e.g., assure no loops in a SDN network) performs a space search and shows traces of property violations.

[20] is a publication that proposes another Model Checker for SDN networks. The correctness properties they want to validate with this tool are: loop freedom (a packet does not loop back to a switch which it has already visited) and no invalid drop (a packet is not dropped due to an invalid controller update to some switch). They use the Abstraction Method to address the scalability problem. This technique keeps only one packet in the system, and provokes network updates, injecting packets (whose header values are arbitrary) directly into the ports of the switches.

V. Future Work

The preliminary tests performed have produced positive outcomes. It suggests that the proposed architecture in this paper can be a viable solution for tackling the issue of application portability across different SDN control platforms. Nevertheless, there is a great room for enhancement. One of the main problems found in the architecture is to guarantee the applications running on top of the controller do not collide one with other. A mechanism should avoid the installation of incompatible rules in the switches. To address this problem, a new architectural element has been proposed to be introduced within the overall architecture, the Core Layer. This layer is placed between the Backend and the Shim layer. Its main task is to provide the necessary logic for the coexistence of the SDN applications.

A SDN network deployed with our architecture could have a bigger delay in the communication between data plane and control plane than a conventional SDN network, due to the integration of heterogeneous controllers which must be interconnected. Therefore, performance tests are being performed to verify that this extra delay is insignificant and does not harm to the behavior of the SDN network. Other important drawback is the possible incompatibility between some Client Controllers and some Server Controller. If the client possesses a different version of the OpenFlow protocol that the version supported by the Server Controller exits a conflict and it is impossible to interconnect them. To solve this last problem, we are trying to find the commonalities between different versions of OpenFlow.

The impact of all the future modifications in the NetIDE Architecture will be analyzed and the tools will be adapted and integrated into the new architecture.

The provided toolset has some shortcomings that must be solved. For instance, the ‘Runtime’ tools detailed in depth: the Logger and the Profiler have been developed for Ryu controller platform. So, both tools have to be adapted for the rest of the controllers. With the Core element described previously, this adaptation would be much easier.

All the tools that are fed with messages exchanged between the two planes, should capture those messages from the Core layer instead of the Shim layer module of each Server Controller. So only one development would be needed instead of one development for each controller.

Regarding to the tools presented in subsections III-A, III-B and III-C some improvements have been proposed.

In the first case, a new filter will be implemented. This filter allows to the user, choose the type of openflow message wants to visualize (flow mod, packet out, port mod, etc.). The graphic that the Profiler generates to show the memory consumption is not updated while the tool is gathering the data. So the next step is provide a ‘real-time’ graphic that is updated at the same time the data is collected. The model checker suggests various open issues. This tool can guarantee that one SDN application does not provoke loops or black holes in a given network, but it can not assure that a set of applications fulfil that property. Another issue is the high execution time of model checker when the topology provided to the tool is composed of several elements. Besides, some controllers have mechanisms to avoid loops in the network, so one of the main utilities of model checker tool is satisfied.

VI. Conclusions

Based on fact that this architecture has been implemented in a real prototype, the work presented in this paper is a feasible solution to address the problem of supporting SDN applications written for different controllers within the same SDN network. Moreover, the set of tools provided allow to debug, diagnose and troubleshoot the network.

In summary, the presented architecture and the provided tools are valid for practical deployment and solve heterogeneity, migration and debugging issues of SDN networks.
for distributing those messages to the consumer application or intercepted by a central element that will be the responsible generated by an application and sent to another) will be changing messages between applications. This protocol determines the behavior of the server that provides the messages and the behavior of the client). It is based on a broker architecture, which means that all information (all messages generated by an application and sent to another) will be intercepted by a central element that will be the responsible for distributing those messages to the consumer application or delete these messages (broker). In other words, the communication between two applications will never be direct, without passing first through the middle element (described above).

On one hand, several advantages are found using the broker-architecture approach:

• Complete independence between applications. The client application does not have to know where the consuming application is, in fact there may be no message consumer. It only needs to know how to send messages to the broker, which will route the messages to the right applications based on business criteria (queue name, routing key, topic, message properties, etc.) rather than on physical topology (IP addresses, host names).

• It is not necessary that the lifetime of applications overlap, because the messages sent by the first application will be kept in the broker and the receiver application will be able to retrieve these messages any time later.

• Even if the application fails, messages sent to the broker will be buffered there.

On the other hand, there are some drawbacks, which should be taken into account:

• Since all messages must pass through the broker, there is some overhead in this element, causing a bottleneck, deteriorating the performance between applications (e.g. increasing latency).

• There is a single point of failure in the broker. Apart from all the advantages offered in a broker architecture, this software also provides other features that make it even more exible:

• Libraries are available for almost all programming languages.

• Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker in order to avoid queuing system fails. (One of the most agrant properties of the broker architecture).

APPENDIX

In this appendix, an overview of RabbitMQ technology is exposed, citing its operating principles and its main advantages.

RabbitMQ is one of the technologies that implements AMQP (an open standard application layer protocol for exchanging messages between applications). This protocol determines the behavior of the server that provides the messages and the behavior of the client). It is based on a broker architecture, which means that all information (all messages generated by an application and sent to another) will be intercepted by a central element that will be the responsible for distributing those messages to the consumer application or