SERVAL: an Internet software *VLAN* switch developed in Erlang *

Alejandro García Castro, Francisco Javier Morán Rúa
Igalia Software Engineering
Gutenberg, 34B 2º, Polígono de A Grela – 15008 A Corunha
e-mail: {acastro,jmoran}@igalia.com

Juan José Sánchez Penas
University of Corunha, Computer Science Department
Campus de Elvinha – 15071 A Corunha
e-mail: juanjo@dc.fi.udc.es

Abstract

There are situations in which it is very interesting to connect a machine to a different *Local Area Network* from the one its network card is actually connected to. Some network applications require our local host to be virtually connected to a remote *Local Area Network*. This article describes a proposal to develop a software system that emulates the operation of a switch, allowing to create *Virtual Local Area Networks* over the Internet, that completes the current similar solutions. We have created a prototype developed with Erlang/OTP using a client/server architecture and we are working on the integration with the Operating Systems using virtual network interfaces. Erlang is very suitable to face the main issues of this system: performance, communications and fault tolerance. We have accomplished performance and functional tests to assess the suitability of the designed system using the prototype. The paper will explain the current results of the research and describe future work.

1 Introduction

SERVAL is a research and development project whose aim is to assess the feasibility of a system for creating *Virtual Local Area Networks* (*VLANs*) using a software server. The main goal of this software is to provide a way to set up *VLANs* between computers, no matter their location or the connection they use to access the network. For this purpose, we have designed a system that emulates the operation of a hardware switch. The Operating System in the client does not have a regular network interface, but a special program that, acting as a virtual interface, communicates with the SERVAL server. The clients can connect to *VLANs* defined inside the server, which works as a software switch forwarding the messages between the clients in the same *Virtual Local Area Network* (*VLAN*).

Nowadays the solutions available to link two *Local Area Networks* (*LANs*) do not provide some features that would be desirable in some cases. The main technologies for connecting remote *LANs* currently are: *Virtual Private Networks* (*VPNs*) and *VLANs*. The applications we have in mind as interesting examples to be implemented on top of this technologies range from mobility solutions to file sharing.

The main issues of this project that we have to face are:

- Client/server architecture: we have to support this kind of architecture because the connection from local range IP addresses is a requirement, and therefore P2P solutions would not satisfy our needs.
- Performance: the emulation would have no sense if we do not have suitable latency and throughput. The system should be able to overcome stress situations. Scalability is also an important feature.

*Partially supported by Xunta de Galicia - PGIDITSIN0313E

- Operating System integration: the interfaces to our switch in the client side must be virtual network interfaces. User space applications would use these interfaces as the regular ones. Our target Operating Systems are GNU/Linux and Microsoft Windows.

- Communications security: this kind of systems should assure their communications, because the traffic goes through an unsafe medium.

We have designed a system following these requirements: a client/server architecture aimed to solve the main risks we have detected. We have decided to use Erlang/OTP [1] as the development environment because its features fit very well with the project goals.

In this research we want to face and measure the main risks we can see to develop a system like the one we describe. The paper will explain the current results of the research and describe future work.

In Section 2, the current alternative solutions for connecting remote *LANs* are described, and their advantages and disadvantages discussed; the motivations for the project are also presented. In Section 3, the main goals and system requirements for the research and development project are explained. After that, Section 4 introduces the system architecture, leaving for Section 5 the details related with the use of Erlang inside the system. Current status of the project, including some performance tests, is presented in Section 6, before concluding in Section 7.

2 State of the art and motivation

Nowadays technology brings us some options to connect remote *LANs*, but these applications do not provide us features that are very interesting in some environments.

We could use *Virtual Private Networks* (*VPNs*) to create a virtual connection between remote *LANs* communicating over possible untrusted networks. With this technology we can communicate remote networks emulating neighbor networks with a router in the middle. But using this kind of technology we can not transmit non-routed traffic between the network (therefore, local area protocols cannot be used). Besides, we need to configure a router to send out the packages. An example of this kind of software is FreeS/Wan [2], which implements IPSec [3], an standard protocol for encrypting IP traffic between two networks connected by two IPSec gateways.

Another option would be the use of *Virtual Local Area Networks* (*VLANs*) connecting *Local Area Networks* that are physically separated, enabling non-routed traffic between networks (and therefore local area protocols). *Virtual Local Area Networks* are normally implemented using the 802.1Q [4] protocol, which sends layer two traffic with *Virtual Local Area Network* information to define *Local Area Networks* using ports of different switches. The fact of being able to communicate any kind of traffic between the networks, would simplify some regular tasks when we share resources and will also enable the use of applications that communicate with each other using protocols like Rendezvous or SMB. The main limitation of this kind of solutions is that, nowadays, we can only deploy a system like that if we have control over all the physical switches placed between the host and the network we want to connect it to; besides, all the intermediate machines should have that feature implemented. We cannot forget that a telecommunications company, even controlling all the hardware of the network, can not touch the configuration of their machines dynamically in a safe way, because any mistake would spoil all the traffic of the network. We also have to remark that changing the configuration of the machine is not simple, and a trained technical assistance would be needed.

The system we propose in this paper can emulate this *VLANs* behavior using a scalable, distributed, TCP/IP server that acts as a software switch. The clients would run programs that simulate logical network cards connected to the software switch.

We have taken some features of both *VPN* and *VLAN* systems to define the main goals of the project. The security of *VPNs* is a very important feature, because communications are performed through an unsafe medium. Authentication and authorization are a important issues that needs to be solved properly. Hardware switches are designed to handle a great flow of packages, therefore the system has to be ready to manage heavy stress conditions. The performance is another main issue we have to face and specifically the system scala-

bility. We should consider a group of thousands of clients that define a group of *VLANs* trying to send their discovery messages through the switch, the latency has to be correctly handled. In any case, the use of this kind of systems must be well designed because the amount of traffic that the local protocols produce can be very large.

Some of the standards and well-known technologies we are using or considering for the project are:

- *Local Area Network* technologies: Ethernet, Token Ring, etc. We also should review systems to manage this kind of traffic, congestion management.

- *Virtual Local Area Network* technologies: 802.1Q [4], 802.1D [5] and current hardware that supports it.

- TCP/IP transport protocols: UDP and TCP. We want to do research about which one should be more appropriate.

- Application level protocols and their encryption systems: SSL and TSL [6].

- Network interface emulation, both for GNU/Linux and Microsoft Windows.

We have discussed and proposed some applications of the SERVAL technology together with *R, Cable e Telecomunicacións de Galicia, S.A.*, to learn more about possible use cases and increase the knowledge about the system requirements. The idea was to find out some applications in which the advantages of the technology would make them specially interesting for our project:

- Virtual corporative *LANs* among several physical networks in a company, with a simple and flexible configuration. Enterprises could define communications between their headquarters easily, they could even work from home, using their personal Internet connection, but accessing the network and resources of the company. This is very interesting for mobility: if a worker is in a different place than the rest of the company, he can still connect to the network and develop his tasks or access to a document that he has in his account.

- File sharing: users could easily create a private network between them to transmit information. They just have to use standard local area network protocols and they could share resources and services.

- Games through Internet that could only be used before in a *LAN* environment. This is an interesting product for a telecommunications provider company like, because the easiest way to play in a network is using machine and software discovery protocols that only work in a *LAN*. The tool to manage the connections to the *VLANs* can be an easy interface that allows a regular user to connect to this networks and play network games the same way he/she is used to do in local environments. The entertainment industry is nowadays an important part of the telecommunication business.

3 Project goals

The main goal of this research is to assess the feasibility of the use of the current technologies for building a software system able to create and control *VLANs*. We have agreed some functional requirements that the developed system should fulfill:

- Client/server architecture: the basic architecture of the system should have these two layers. We need this kind of architectural design because of the current Internet connectivity, there are a lot of conditions where the clients are in a network with local IP address range. In these cases we need this kind of architecture to assure the connection between the hosts, because P2P technologies would not adapt correctly to our needs. The flexibility and control that the server application provide us seems to be an interesting feature for the system. Anyway, we have also considered the *peer-to-peer* (P2P) architecture, there are some conditions where the P2P could be a good solution. Therefore, our main goal is the client/server architecture but we will consider the way to adapt the system to a P2P deployment.

- GNU/Linux and Microsoft Windows link layer integration: it is an important point for the system usability. Due to these requirements

we need a multiplatform development environment, able to produce software to be run in both GNU/Linux and Microsoft Windows. The perfect integration of the client with the Operating System lets us use the regular facilities and programs with SERVAL transparently. The Operating System will detect a new network interface that will use to transmit traffic like any other interface of the system. The integration with current local area technologies like Ethernet will also be a very interesting target because of the facilities that include. This is one of the main risks of the project, especially in the Microsoft Windows environments. In GNU/Linux we already have a virtual Ethernet driver (TAP) that lets us redirect the traffic to a user space application. Microsoft Windows environments are closed and the assess of the solution is not so easy, we will have to develop a system virtual driver.

- Performance: it is also a very important risk of the research, because we have to accomplish some minimum results in order to be able to properly emulate a *LAN*. The latency of the server is an important issue to solve: we have to reach a minimum latency even under stress conditions. The throughput should also be optimized, because we could loose too much bandwidth using the system. Regarding this subject we have to consider the scalability of the system, which should be the best solution for being able to handle a lot of concurrent users at the same time. We also have to think about the transport layer protocol used for the transmission: UDP or TCP; it seems that depending on the concrete conditions, one alternative could be better than the other, so this feature should be configurable.

- Fault tolerance: the system should be designed to continue working in the presence of software or hardware failures. The server could be deployed in a cluster of computers, and the design of processes and protocols should take into account any possible problem derived from any kind of error.

- Security: one of the most important features that the system should accomplish is to secure the connections; even the performance could be penalized in some conditions. Authentication, authorization and encryption should be added to the system, and the connections should be assured to avoid security problems. There are applications of the system where security is not an important issue but when we want to transmit sensitive information we must assure the communication.

- Heterogeneous *LAN* protocol encapsulation: regarding the type of protocols we should support at least Rendezvous and SMB. Anyway, it is very interesting to support other network technologies, being the optimal solution that the virtual interface had no difference with the rest of the network interfaces of the system. If we completely implement the virtual Ethernet driver we can accomplish this goal.

4 Software and hardware architecture

When we thought about building a system that emulates *VLANs* in a *WAN* environment, we had to decide the global system architecture we were going to use. It should be noticed, before going on with the detailed description, that, by *global architecture*, we mean both the hardware and software skeletons of the system.

The first option we evaluated was whether a *peer-to-peer* approach would be interesting. With the use of a *peer-to-peer* a approach, the scenario would be the following one:

Each user of the system would have in his computer a client of the SERVAL system installed. This client would negotiate a connection with another user having also installed the SERVAL program. Once the connection between the clients was established, they would be connected at link layer level, being this medium Ethernet compatible. Therefore, the Operating System of each computer would see the other one as if it was in its *LAN* with all the advantages this fact has.

This scenario looks fairly attractive but has several serious drawbacks that made us reject it as a general target architecture. The main disadvantage is that if both clients are in private networks, with private network IP addresses, then the *peer-to-peer* connection would not be feasible.

Another drawback is that this skeleton is not good if we want to emulate a lot of machines in the same *LAN*. This is so because the program in each client would have to open a *peer-to-peer* connection with each computer belonging to the same *VLAN*. This would cause a serious network overhead. For example, if we wanted to have a hundred users in a *VLAN*, the number of connections which would be necessary in order to open in each computer is one hundred and, altogether, they would be up to ten thousand. Besides, this skeleton would not be persistent and each client needs to know the addresses of all the other clients we want to include in the same emulated *LAN*.

A second architectural possibility to solve the system scalability problem with the number of connections would be a bus structure. This can be viewed as an improvement of the *peer-to-peer* approach. In the bus architecture approach, the programs would be joined forming a line. Hence, in this one-dimensional structure, clients would propagate messages in order they reach their final destination.

The bus architecture, as we can see, reduces the number of opened connections because each SERVAL program only has to be linked with two more users and the configuration complexity is lower than the *peer-to-peer*. However, it still has two important drawbacks: the connection among private IP clients, which is not possible, and how a client can belong to several *VLANs*. Another problem is the different latency of the messages depending on the position of the clients involved in the communication. Therefore, the bus architecture is not suitable to fulfill our requirements either.

After having discarded the previous system architectures, we came to the conclusion that the one we were looking for, more adapted to the project needs, was the client/server model.

In the client/server model the entities taking part are the following ones:

Server The server will be a program running as a daemon in an Internet accessible host. This program, the SERVAL server, will listen for connections coming from the users of the system. Its function is similar to the function a hardware switch has in a *LAN*. It will be able to group users in different independent sets and will be the mediator among the clients.

When a user wants to send a message to another client, he has to send it first to the server and, then, the server forwards it to its final destination. The fact that the server can group users in sets can be compared to the *VLANs* existing in some hardware switches.

Client The clients in this architecture will be the users who want to connect to our system. They will have to install in their computers the client program to access the virtual switch and through it, the rest of clients.

The user Operating System, accessing the SERVAL client, will be able to see the other clients in his groups as if they were in the same *Local Area Network*.

With the client/server model, the drawbacks and limitations existing for a P2P or bus architecture are overcome. In order to communicate the client and the server, operations and messages similar to the ones used in the link layer protocols were defined.

The messages interchanged between the client and the server are shown in Figure 1. The client can connect and disconnect to the server, ask for the list of available *VLANs*, and join or leave one of them. Other messages allow the client to send a message to a given *VLAN* (and therefore to all the users connected to it).

Other important messages are the `addressOfClientRequest` and its answer from the server side: the `addressOfClientResponse`. They were created to emulate the ARP Ethernet link layer protocol and are used when at network level a client wants to talk to another.

At network level clients communicate with each other knowing their network level address. For instance, if we are using TCP/IP knowing their IP. However, at link level it is not enough with the IP address to contact with the destination but it is necessary to know the link layer address of the next hop towards the destination as well. Therefore, in SERVAL we use the two former messages, `addresssOfClientRequest` and `addressOfClientResponse`, to find out the SERVAL link layer address of a client knowing its network address.

Finally, the global architecture chosen has to be both fault tolerant and scalable. By fault tolerance

we mean the system should be resistant to a partial system crash, being able to overcome the situation and continue the normal operation. By scalability we mean that if the system requirements grow, and the number of clients connected is higher than the initially expected, new resources can still be added to the server in order to increase its performance and fulfil the new requirements.

With this goals in mind, we have extended the client/server model to be a distributed system. So, in the final architecture, instead of having only a node of the SERVAL program running in a host, we have several nodes which collaborate with each other: they will detect a node failure and restart it if possible, the clients of a crashed node will be moved to another node of the cluster, and so on.

5 Implementation of the system using Erlang

As explained in the previous section, we have to build three software artefacts:

- SERVAL server. It is the program which plays the role of a hardware switch in a real *VLAN* environment. It has to be run in an IP accessible from all clients. Its function is to manage all the operations related to both the *VLANs* and client management. For instance, it accepts input connections from clients, creates *VLANs*, routes messages among clients etc.

- SERVAL client. It is the software which clients must use to access the server. It has two parts:

  - User land adapter It is the program which receives Ethernet frames from the virtual Ethernet driver and maps them to messages of the communications protocol. Next, these messages are sent to server.

    It has also an interface to receive requests directly from user. For example, requests to join *VLANs*, to abandon them etc.

  - Virtual Ethernet driver. It is an Ethernet driver which implements a virtual network card. It controls the communications with the Ethernet Operating System kernel API and the user land adapter.

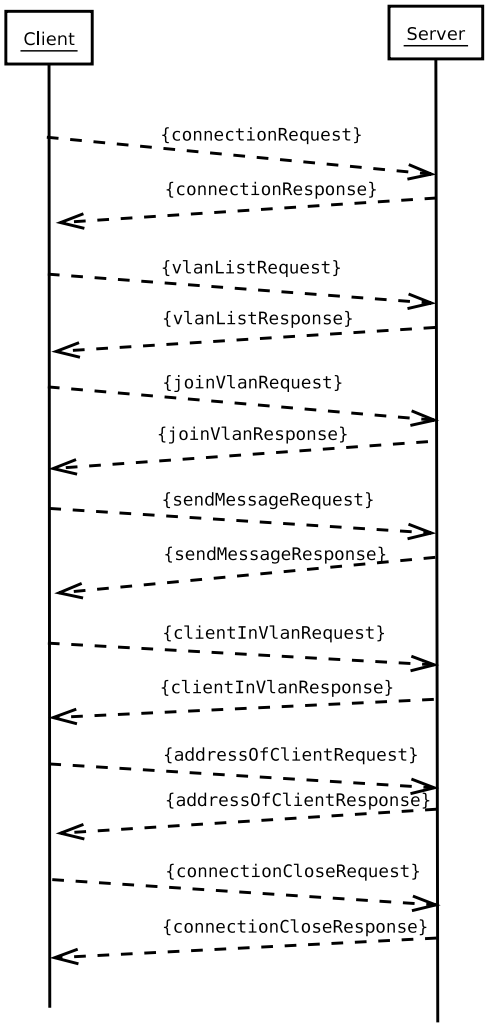

Figure 1: Messages interchanged between client and server

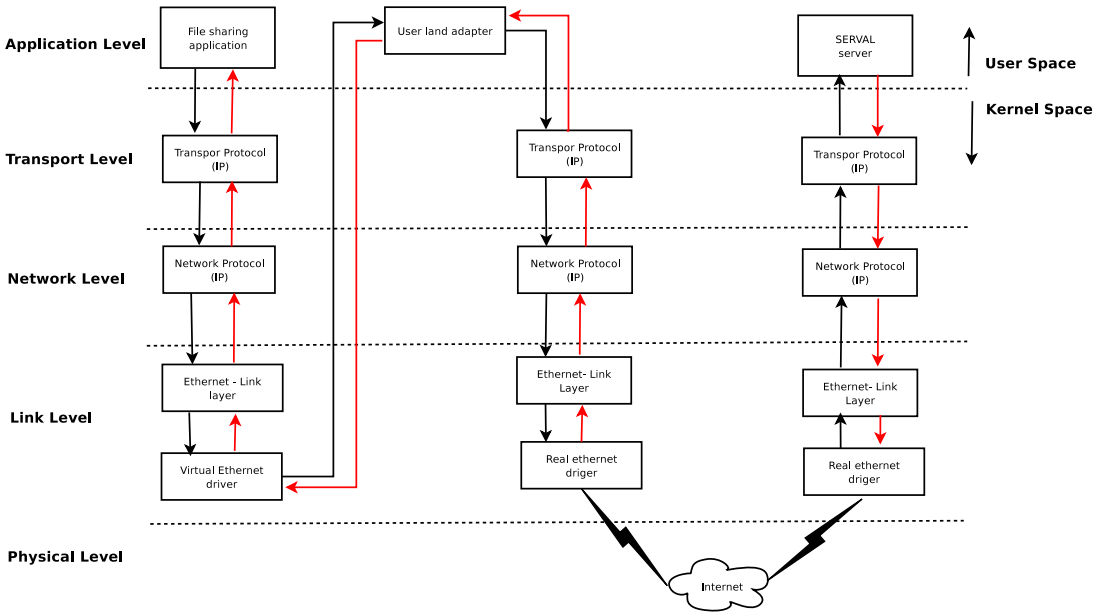

Figure 2: Relation between the client and Ethernet driver

A diagram which summarizes the relation between the client agent developed and the virtual Ethernet driver is shown in Figure 2. It shows how an end user application running in a computer connected to a server would use the client and the communication protocol stack. The messages sent by the application level program are encapsulated into a transport level protocol. Then, the transport protocol is encapsulated into a network level one. After this, the network level datagrams are converted into Ethernet frames which, next, are sent through the virtual Ethernet interface. This virtual interface delivers the frames to the user land adapter which, finally, maps them to the protocol messages used to communicate witch the server. They are encapsulated through the protocol stack again and are sent by a real network interface.

The reverse path is followed by the messages delivered to the user agent by the real network interface after the demultiplexing which happens on ascending the communication protocol stack.

In the rest of this section, we detail the Erlang implementation of both the server and the client agent.

5.1 Server implementation in Erlang

The process structure of the multinode switch can be observed in Figure 3. They are represented the process classes and the relations existing among them. The relations we emphasize are two:

- *Creation link relation.* A process class A has a creation link relation with a process class B when processes of class A create processes of class B. This relation type is shown by the continuous lines.

- *State link relation.* A process class A has a state link relation with a process of class B when in the state of processes of class A is stored the process identifier or the registered name of processes of class B. This relation type is represented as dotted lines.

The task each process class carries out is explained to understand the Erlang implementation of the virtual *VLAN* switch accurately.

serval_app This process class is an Erlang application behavior. It has been created to start/stop the server.

serval_sup This process class is an Erlang supervisor behavior. It supervises the *serval_server_logger*, *serval_tcp_port_manager* and *serval_udp_port_manager* process classes.

serval_server_logger It is an Erlang generic server behavior. It has the mission of logging all the information sent by the other processes existing in a node.

serval_tcp_port_manager This one is another generic server. It listens on a TCP port waiting for incoming connections. Every time it receives a connection request it spawns a new process called *serval_connection_manager*.

serval_udp_port_manager It is also implemented as a generic server and its function is to listen on an UDP port waiting for incoming messages. When it receives the first message coming from a source socket (source IP, source port) spawns another process, **serval_connection_manager**, for that connection.

serval_connection_manager It is a supervisor and is entrusted with the task of coordinating and supervising the *serval_connection_communications_tcp*, *serval_connection_communications_udp* and *serval_connection_operation* process classes.

serval_connection_operation It is a generic server. This process carries out the operations associated with the messages sent by the client.

serval_connection_communications_udp This process class does the sending of messages from the virtual switch to the SERVAL user agents. The messages which arrive from the clients to the server are received in UDP communications by the **serval_udp_port_manager**.

serval_connection_communication_tcp This process class does both the receiving of the messages which arrive to the server from clients and the delivery of the messages sent by the virtual switch to the user agents.

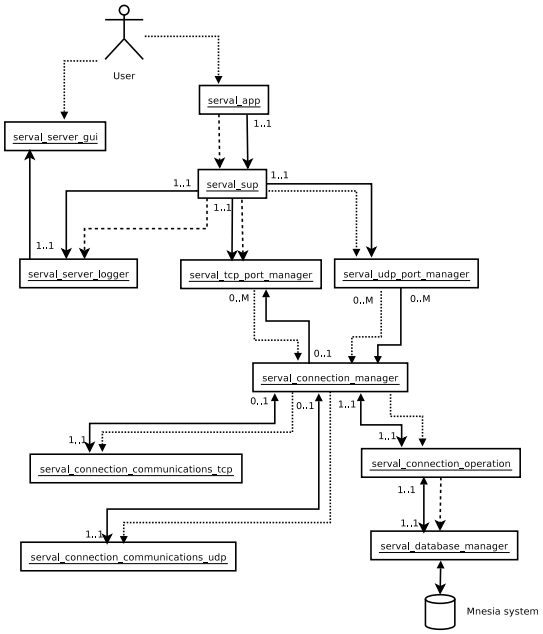

Figure 3: Link process class diagram

5.2 Client implementation

Regarding the user agent implemented in Erlang, the process diagram of the design can be observed in the Figure 4.

The process classes which take part in the user agent are:

serval_test_client_gui This process control the interface which is used to make requests to the server and to send and receive messages from *VLANs*.

It is the process which talks to the graphics system and receives all the events from it when users click in the interface widgets.

serval_test_client Processes of this process are created by the *serval_test_client_gui* when the user requests a connection with the server.

It is the process entrusted with the task of sending messages to the SERVAL server and receiving from the Internet all the information sent to the client.

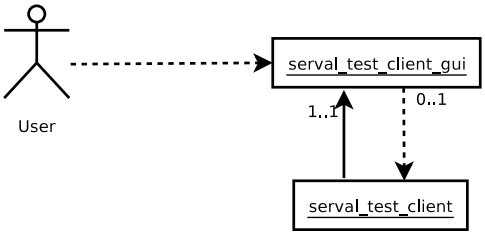

Figure 4: SERVAL user agent link process class diagram

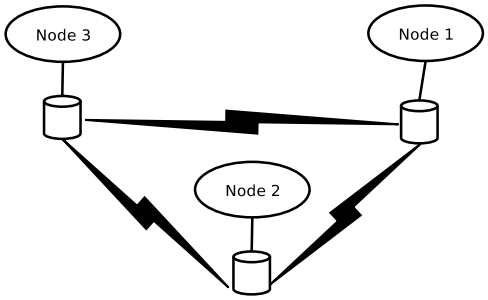

Figure 5: SERVAL server cluster with 3 nodes

5.3 Storing *VLANs* information in Mnesia

In Section 4, it was stated that the project goal was to build a scalable, concurrent and fault tolerant system.

To achieve this target, we decided to make our virtual switch a multiple node software. We mean that in our architecture we have several instances of the SERVAL application which can attend client requests. Therefore, clients can connect to any of the nodes of the server, and independently of the node they access they see each other and the same *Virtual Local Area Networks*. In other words, this means that a client connected to a node A has to be able to send a message to another client connected to a node B if both belong to the same *VLAN*.

In this multinode architecture, then, it is necessary to have communication among nodes to share *VLANs* information. We decided to use Mnesia, a distributed database management system included in Erlang/OTP. With this distributed database we can access information of the existing *VLANs* and clients connected to each of them from any of the nodes of the system.

The use of Mnesia has big advantages for this project. Thanks to its distribution capabilities, synchronization among nodes is done automatically by this DBMS and it is transparent to SERVAL. In this way, as we rely on Mnesia, synchronization of the virtual switch nodes is done without overloading the code with synchronization tasks. This idea is represented in Figure 5, which shows the situation in which there are three nodes running belonging to the SERVAL cluster. It can be observed how each of the nodes has a local copy of the distributed Mnesia tables and how the communication among nodes is done using the database.

5.4 ASN1 Erlang compiler for communication protocol implementation

We decided to use the *Abstract System Notation One - ASN1* for the definition, transmission and encapsulation of our internal, client/server, communications protocol.

ASN1 is formal language for abstract description of the messages interchanged in communications protocols, with independence of the programming language chosen and the data memory representation. ASN1 is a standard since 1984 and, consequently, its codification framework is mature and it has been used successfully in a lot of different scenarios.

In the project we have used the Erlang ASN1 compiler. This compiler in very useful because it generates coding and decoding functions which can be directly used by Erlang programs.

5.5 Process collaboration scenarios

In this section we describe several collaboration scenarios with the aim to understand more accurately how the SERVAL system works.

5.5.1 SERVAL server starting

When the SERVAL server is started, it is launched a process of class *serval_app*. Next, the *serval_app* spawns three more process of class *serval_server_logger*, *serval_tcp_port_manager* and *serval_udp_port_manager* respectively.

Our SERVAL server has support to maintain communications with the clients using as transport protocol TCP or UDP. This is the reason why, when the SERVAL server is launched, the processes *serval_tcp_port_manager* and *serval_udp_port_manager* are created. The first listens for input connections in a TCP port and the second listens for packets in an UDP port.

5.5.2 SERVAL client connection request to SERVAL server

We are going to explain what happens when a client requests a TCP connection with the SERVAL server.

First an *open_port* TCP message is received by the process *serval_tcp_port_manager*. Second, this process creates a process *serval_connection_manager* and, then, this last one spawns two more processes, a *serval_connection_communications_tcp* and a *serval_connection_operation*.

With these three processes we have the structure to manage all the operations related to the client has requested the connection.

5.5.3 Message interchange between processes belonging to the same *VLAN*

When a client A wants to send a message to another B, first, A has to find out B client identifier knowing its network level address. In order to get B client identifier, it sends the SERVAL server a `adressOfClientRequest`.

All clients in the same *VLANs* that A, receive this `addressOfClientRequest`. The one whose network level address matches the one included in the message, that's to say B, sends back to A an `addressOfClientResponse` with its client identifier.

Next, A sends the SERVAL server a `sendMessageRequest` with the data and the B client identifier as destination address. The SERVAL server checks that B is in the same *VLANs* and, if this condition is true, delivers B the message. After this, sends A a `sendMessageResponse`.

We can observe all this behaviour in the figure 6.

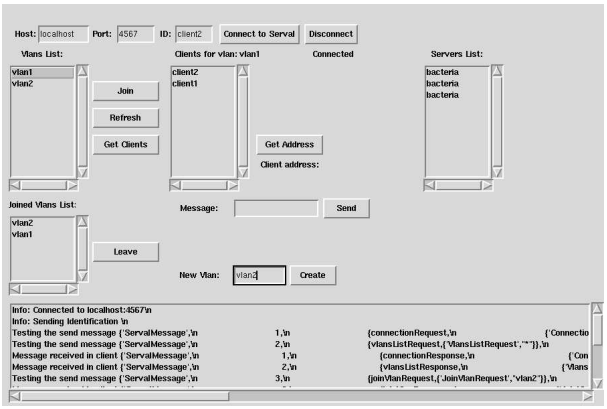

Figure 7: Client screenshot

6 Current status of the research

6.1 The prototype GUI

In Figure 7, the graphical user interface developed for the prototype user agent can be seen.

In the GUI window of the example, the client is connected to a SERVAL server which is listening for input connections in port 4567. The user is connected to the virtual switch through a client with address *client1*. We can see that there are two *VLANs* created, *vlan1* and *vlan2*, and that *Client1* joined both. Finally, we can observe that *vlan1* contains two clients with the address *client1* and *client2*.

6.2 Testing the system

Two key features of the system, as already explained, are performance and fault tolerance. Every design and new characteristic we add to the system is developed thinking of what impact it will have over these two variables. But we are not only think in the impact of the modifications but trying to measure and to check the system with the new additions. In this subsection, some performance measurements and fault tolerance tests that we have carried out are described.

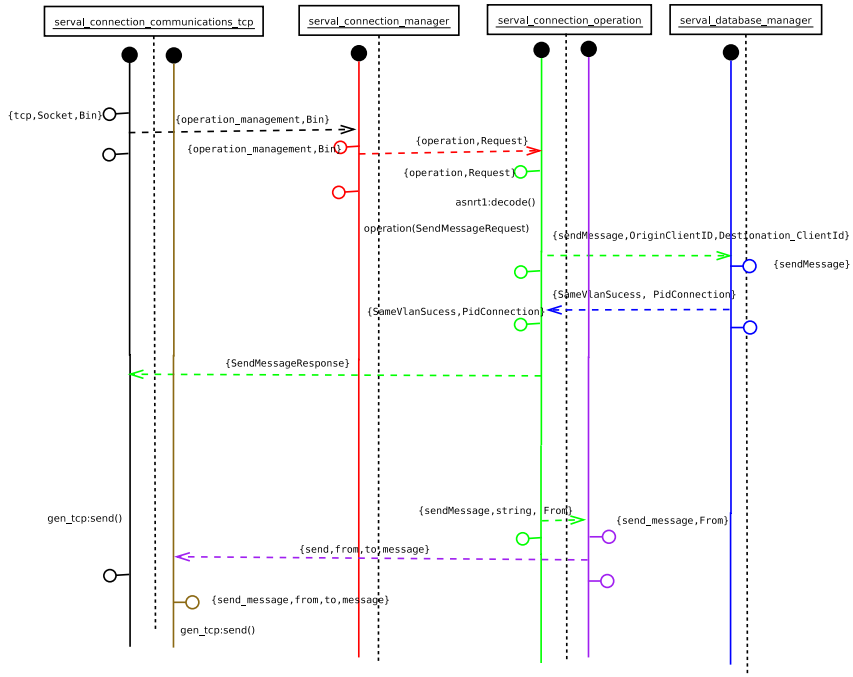

Figure 6: Message interchange between processes belonging to the same *VLANs*

6.2.1 Performance tests

Due to the real time nature of the system, and the amount of concurrent users that should potentially be able to handle, a good performance is an essential requirement for the system.

We have concluded that the performance in SERVAL can be measured by the two following variables:

- **Operation latency**. The operation latency is the time it takes an operation to be completed, since it is ordered until it is finished. The lower the operation latency, the higher the system performance. As final goal, we are specially interested in reducing the message latency of a message sent from a client to another through the server.

- **Throughput**. The throughput of a system is the number of operation requests it can handle in a period of time. A higher throughput means that the system is able to carry out a lot of operations concurrently. We should also consider the bandwidth reduction caused by use of our internal protocol. We increase the size of the packages internally, because we need to store some extra information, we should keep them small to enhance the use of the medium. Currently we do not consider this increment of the size of the packages a problem.

In order to make these measurements we have developed a special module for creating performance graphics. It is call *serval_gnuplot* and uses the GPL application *gnuplot*.

With this module carried out performance tests in both the client and server side. We will describe two examples in which we have obtained performance improvements after analyzing the operation latency and the system throughput.

The `get_vlans` message This protocol message has a latency which grows with the number of *VLANs*. So, the high latency of this message blocks another messages in the server queue, which cannot be delivered by the server quickly.

After some performance tests, the Mnesia query

for getting all the *VLANs* created in the server, was pointed out as the system bottleneck for this operation.

In figure 8 we can see the time process the *get_vlans* with different number of *VLANs* created in the server.

We have represented also the *VLANs* number against the *get_vlans* latency. The graphichs obtained can be seen in figure 9.

The impact in the latency of the messages that the number of *VLANs* created in the system has, can be easily observed.

The solution we chose to solve this problem was the use of a memory cache to speed up the request operation. So that, with this cache we were able to rise up the throughput of the virtual switch and, besides, we could drop the latency of the *get_vlans* message.

The `adressOfClientRequest` message After analyzing the system operations performed during the message interchange between the clients of the system, we detected that the number of *addressOfClientRequest* sent by the client agents was too high.

Many of these *addressOfClientsRequests* messages, however, were asking for the address of the same client. Hence we implemented a cache in client side to store the mapping between network addresses and the SERVAL identifiers.

Doing this we got a double improvement. First, we reduced the number of *addressOfClientsRequests* messages sent to the SERVAL switch dropping its average load; and second, we decreased the latency of this message. In Figure 10, we can observe how with the client address cache the number of *get_addr* messages received in the server is less than the number of the other message types.

6.2.2 Fault tolerance

To have fault tolerance features was already stated in Section 3 as an essential goal for the project.

There are different strategies which can be followed to create a fault tolerant system. In our case, the concurrency and distribution properties of Erlang have allowed us to build a robust recovery system based on the multiple node server described in Section 4.

The idea is that if a node crashes the system has mechanisms for letting the rest of nodes continue the work that was being carried out by the crashing one. This can be easily explained introducing some situations and how they are overcome:

One node crash If a node crashes, the recovery mechanism consists mainly in the reconnection engine implemented in the clients.

The description of this engine is as follows: each client receives from the server an address and port list with all the nodes belonging to the cluster as answer to the connection message. Therefore, when a client detects the node is connected to is unreachable, then it requests the connection with another of the nodes. This second node is obtained from the node list that, as mentioned, is stored in the client state.

When the server receives a connection request, it uses Mnesia to check if the client was connected through another node before. We consult Mnesia because we use a distributed table which registers each client connected to SERVAL. The information we record for each client is:

- The link layer SERVAL address. We call it *client identifier* as well.
- The process identifier of the SERVAL process we use to manage the client.

Taking this into account, we query the former Mnesia table to find out if there is a row with the same link layer address that the one of client is requesting the connection:

- If we get zero rows this means that the client is not doing a reconnection because the node it was acceding has crashed. We record the information for the the new client in a new row.
- If we get one row this fact means that the client is doing a reconnection. Hence, we have to update the row obtained with the new process identifier of the process created in the new access node to manage the connection.

Client crash This scenario describes which recovery actions are performed when a unsuitable client disconnection takes place, and the message

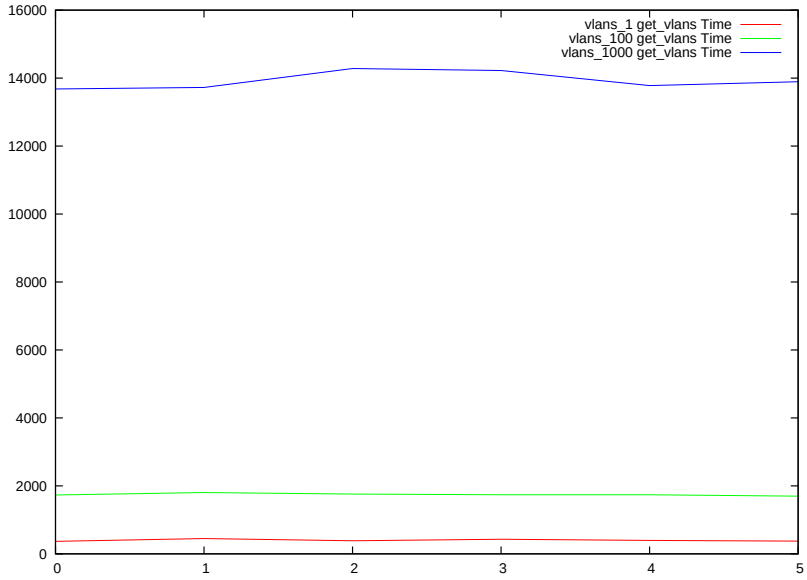

Figure 8: Time to process the get_vlan message depending on the *VLANs* existing in the server

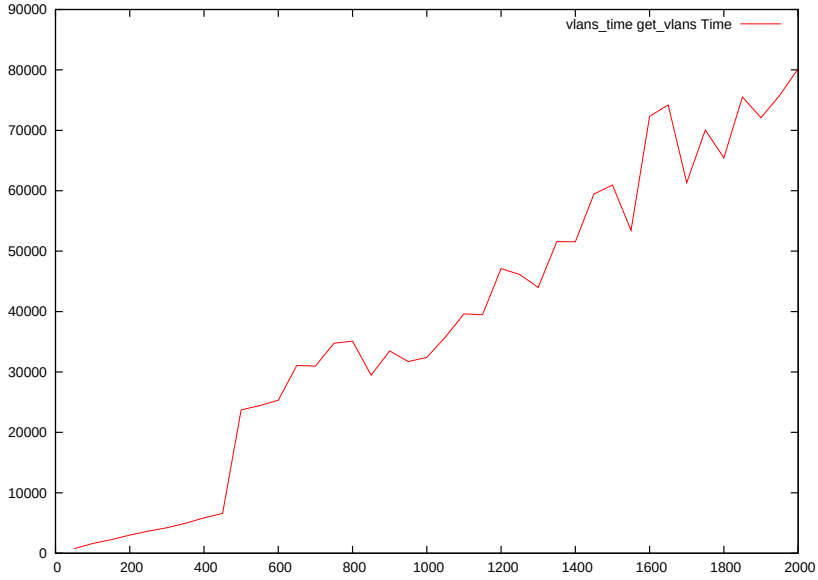

Figure 9: *VLANs* number against *get_vlans* latency

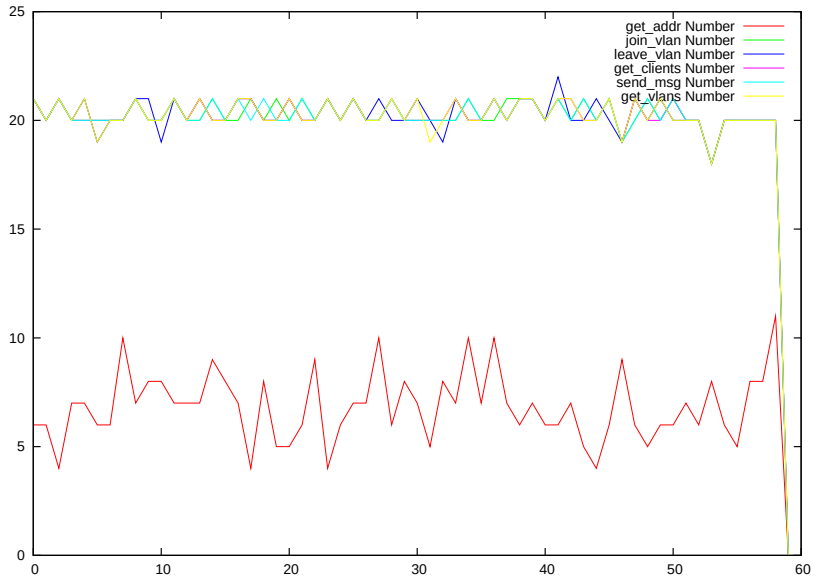

Figure 10: Number of each message type received in the server

`connectionCloseRequest` is not sent by the client to the server.

The client crash can be caused by a software failure, but the message could be also lost due to some network problem. When this happens, the processes responsible for the client management are kept alive in the node the client was connected to. Therefore, in this situation, if another client sends a message to this crashed one, in the switch cluster the client is detected as connected. Because of this, the switch sends the message, though it will never arrive to its final destination, because it is down (we will explain why we do not matter this lost of messages).

Next, we are going to describe the strategy we decided to implement when the former situation happens. Each client incorporates a *keep alive* mechanism which sends the server a message at regular periods of time. Each server node has a process which monitors the receiving of the keep alive messages coming from clients. If this monitoring process detects that a client is not sending the keep alive messages and it has not requested the connection close, then it kills the management processes for this client. Besides, the monitoring process deletes from Mnesia the information related to the *VLANs* the crashed client was connected to.

As we can observe, there is a period of time between a client crashes until this fact is detected in the server. If a message is sent to the crashed client during this period of time, the messages do not reach the destination. We could implement acknowledgement messages to guarantee they always arrive to its final destination. However, we do not wish a too heavy protocol because we are emulating a link layer scenario. Therefore, these failures rarely happen and will be detected by upper layers in the communication protocol suite used.

6.2.3 Future work

Nowadays, we have in mind to work on several things, some of which are:

Congestion control protocol We want the virtual server switch nodes to monitor its internal work load. So, to succeed in this task we are implementing the processes necessary to measure this magnitude. Besides, it is necessary to extend the link layer protocol in order it incorporates the messages needed for congestion control.

SERVAL clients are being also improved with the ability to process the congestion control

information sent by the server. So, if they detect that the work load of the node they are accessing to is too high, they can decide to ask another node with lower load lodge them.

Access Control List For many real environments it is very important to be able to give and revoke permissions to clients to do certain operations or to access certain resources in the virtual switch.

For instance, we can wish to let users with a certain profile create *VLANs* in our server and forbid this operation to other users groups. Another example, is the possibility to join a *VLAN*. A server administrator may wish to have a predefined set of *VLANs* created in the virtual switch. He may wish to have the control to authorize or deny the access to each *VLAN* following a per user or per group policy. All this can be done implementing the ACL engine and we are hands on.

Ethernet driver In the section (4) we saw that one the components of the architecture is the interface with the link layer of the communication protocol kernel stack.

We are working in the strategy to implement this virtual Ethernet driver. He are also assessing the possibility to use the TAP device driver. The TAP is a Virtual Ethernet network device. It was designed as low level kernel support for Ethernet tunnelling. It provides to user-land application two interfaces:

- `/dev/tapX` character device
- `tapX` virtual Ethernet interface.

The user-land SERVAL Erlang client could use this device `/dev/tapX` to write Ethernet frames which will be received by the kernel. On the other hand, each Ethernet frame wrote by the kernel to the `tapX` interface will be received by the SERVAL client by reading the `/dev/tapX` device file.

Secure communications We are emulating a virtual switch, so our virtual operation environment is a *VLAN*. In a *LAN* the Ethernet frames don't leave the network limits so the security policy can be relaxed if we rely on our *VLAN* users.

However, although the SERVAL server virtual environment is a *VLAN*, the real environment is a *WAN*. As a consequence, all our link layer traffic is going to cross through the Internet and will be exposed to be sniffed by everybody. So we are implementing SSL support in client and server side to cipher communications.

7 Conclusions and future work

In this paper we have explained the main motivations and goals of the SERVAL project, the designed solution proposed to achieve them and the current results and work that we are developing. Through the paper we have established that Erlang is a suitable technology for this project, where network communications, performance and fault tolerance are the main requirements.

We think that our project can be interesting and applicable in a lot of real and diverse scenarios - pointed out in the paper - in which emulating *Virtual Local Area Networks* over Internet is a good solution. SERVAL *Virtual Local Area Networks* management will allow a more flexible way of designing network topologies.

We can transmit that nowadays we are satisfied with the results we are obtaining and with the future of the project. In fact, we are encouraged with having a real and almost complete SERVAL prototype in the near future that will be the first step to build an actual system.

8 Acknowledgements

The authors want to thank Alberto García González, José Juán González Alonso, Iago Toral Quiroga, Adrian Otero Vila and Ángel Vidal, and in general the whole Igalia company, for their collaboration in the development of the SERVAL project.

References

[1] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming*

in Erlang, 2nd edition. Prentice Hall International, 1996.

[2] The FreeS/WAN Project. Ipsec gnu/linux implementation, 2004. http://www.freeswan.org/.

[3] The Internet Engineering Task Force (IETF). Ip security protocol (ipsec), 2004. http://www.ietf.org/html.charters/ipsec-charter.html.

[4] IEEE Project 802.2 Working Group. IEEE standard for local and metropolitan area networks: Virtual bridged local area networks. Technical report, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 1998.

[5] IEEE Project 802.2 Working Group. IEEE standard for information technology–telecommunications and information exchange between systems–ieee standard for local and metropolitan area networks–common specifications–media access control (mac) bridges. Technical Report ISO/IEC 15802-3:1998, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 1998.

[6] T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, IETF - (Internet Engineering Task Force), January 1999.

[7] W. Stallings. *Local Networks*. Macmillan Publishing Company, New York, 3d edition, 1990.

[8] IEEE Project 802.2 Working Group. IEEE standard for local and metropolitan area networks: Overview and architecture. Technical Report IEEE 802-2001, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 2001.

[9] K. Pitt, D.; Sy. Address-based and non-address-based routing schemes for interconnected local area networks. *Computer Science Press*, 1986.