IP4JVM: A Didactic Native Implementation of the IPv6 Suite for OpenJDK and its Application to IPv6 Testing

Ariel Sabiguero Yawelak
Instituto de Computación - Facultad de Ingeniería
Universidad de la República - Montevideo, Uruguay
e-mail: asabigue@{fing.edu.uy|ieee.org}

***Abstract*—Java Virtual Machines do not provide communication services other than those offered by the underlying Operating System (OS). Java sockets are just wrappers for the ones provided by the OS. In this work, we present how we replaced standard OS wrappers with a Java native implementation of most part of the IPv6 protocol stack, building an operational IPv6 implementation.**

The modified Java Virtual Machine has proved useful for education, protocol development. We motivate its application to protocol testing and expect to produce results on the testing field too.

***Keywords*—Java, IPv6, JVM, NAT66, DHCPv6, TTCN-3**

I. Introduction

Java [1] is an overloaded word used to refer a programming language, a virtual machine and a platform. The Java Language is a *state of the art*, object-oriented language with a syntax similar to that of C. The way Java provides access to networking is through a set of classes provided by the Java Platform, without providing any particular language primitive for it. To achieve portability across different platforms, network services offered to a Java developer are those available to all officially supported platforms, not being able to exploit particular platform capabilities in a standard way, as it cannot be standardized. Java connectivity services only offers basic TCP and UDP sockets and clumsy network interface handling. This fact per-se turns Java into a bad choice for low level, network protocol development. With this limitations in mind we started working on Java native networking capabilities, while working on Internet Protocol testing.

A. IPv6 Protocol Testing requirements

Testing a complex protocol suite as IPv6 [2] requires low level manipulation capabilities that are beyond Java networking capabilities. This fact is not new, but it imposes some constraints for Java usage on low level networking. IPv6 testing is standardized by the Internet Engineering Task Force through the IPv6 Ready Logo [3]. When a test system is built for v6RL certification, it must be capable of performing operations like enabling or disabling an interface, assign an IPv6 address and more operations that cannot be performed using Java language. We decided to work in order to overcome these limitations.

When we initially addressed this challenge, we decided that having a working IPv6 implementation would ensure that we have all building blocks required to model, describe, generate and analyze IPv6 traffic. With that naïve idea in mind, we addressed two problems at the same time: to provide Java with something that is not designed for and to generate all elements required for IPv6 protocol testing.

Internet Protocol for Java Virtual Machine (IP4JVM) is a set of collaborative, individual projects, that addresses the problem of providing Java a native implementation of the IPv6 protocol that is suitable both for IP communication of Java applications and for testing IPv6 devices.

This document is organized as follows. On Section II the evolution of the tool to current state is presented. Section III presents some thoughts on networking education through protocol implementation. Section IV describes some highlights of the implementation, presenting the main architectural elements. On Section V the intended usage for testing is presented. Present and future possibilities are suggested, together with TTCN-3 perspectives. Section VI summarizes an presents some of the short term objectives for this project. The work concludes on Section VII.

II. Evolution

This tool is the combined result of successive pieces of work done by grade students from the Software Engineering Career at Universidad de la República. Some of the tasks were done in cooperation with IRISA, in particular, with ARMOR team. Each of the building stages had to be designed as a standalone, self contained, yet meaningful task, addressable in a short period of time by students. We were fortunate enough to get excellent students every time.

A. Initial stack

The first part of the work represented a big challenge, addressed by L. Rodríguez. Nothing but ideas existed then. The objective was to build the foundations for protocol stack handling and to implement a minimal subset of the IPv6 suite that would be able to handle UDP traffic. To achieve this goal, all the layers of the protocol stack had to be, at least, minimally implemented. All objects and behavior were built using a regular JDK, but to be able to replace standard IPv6 handling, we had to modify the implementation of the JVM. Back in 2006, there was no commercial implementation of JVM whose source code was available for experimentation. By that time, SableVM [4] was selected. The result of this stage was a set of classes and a patch to SableVM that allows

Java applications run unmodified using UDP IPv6 services implemented in Java. This initial prototype was tested with custom UDP applications and also, some of v6RL tests were run against it successfully.

B. TCP/IPv6

The next evolution of the stack had the main objective of being to run a standard Java, network-oriented application on top of it. The selected application was Apache Tomcat JEE web server container. Even though it might sound a simple scenario, the implications are serious: real web applications could use the implemented stack. This task was addressed by R. Abelenda and I. Corrales.

Several important things changed in Java during 2007. Maybe the most relevant one was the release of OpenJDK [5] under a GPL license. As soon as possible, we ported our work from SableVM into OpenJDK. TCP implementation was addressed too and seamlessly integrated into the stack. Some missing features needed to be added in underlying layers of the stack, but by the end of this stage, most of Java IPv6 networking features were available to the user. Our modified JVM was able to run standard applications without modifications, achieving the goal of having standard bytecode using our stack.

C. Mobile IP

Afterward R. Abelenda made a stage in IRISA, France, where he implemented the foundations for Mobile IPv6 support. Several things had to be added to the tool in order to be able to tunnel traffic. Initial routing structures and algorithms were added to the stack. IPv6 protocol encapsulation, different Security Association options and Security Association Databases had to be defined too. Routing headers were added to IPv6 implementation.

D. Routing, DHCPv6 and NAT66

The last evolution of the stack was addressed by L. Scasso and M. Techera. The stack was mature enough to be able to start doing innovative things.

DHCPv6 was addressed too. The implementation was verified against Dibbler DHCPv6 server and also, v6RL test cases from a TAHI tool were run against it. Fun and debate popped up when Network Address Translation implementation was suggested. It was an excellent timing issue that IETF published two NAT66 [6] drafts during this stage and they both were implemented and tested. We showed possible to implement state of the art protocols while they are being designed and standardized. A web application was developed in order to interact with the stack and configure it dynamically, on runtime. Sure, only the application had to be developed, as the Tomcat JEE web container was already capable of running on top of our modified stack. This web application allows us to interact with the stack by adding or removing addresses, enabling or disabling routing. It is also possible to enable or disable natting through an interface.

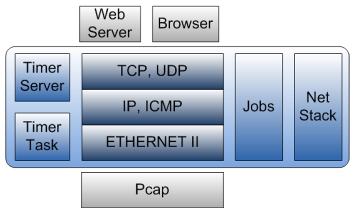

Fig. 1. IP4JVM architecture

III. Results in Education

It is always a challenge to teach networking. Right from the beginning, we were faced with the question of deciding if it was possible or not to build a complete protocol stack by grade students. Fortunately, the answer was positive. Splitting the task into smaller work units, addressable and meaningful was difficult too. It would have been for anybody to get committed into something that does not produce any concrete result. A lesson learned thought this years is that there is a big gap between understanding a protocol and implementing even some parts of it. Students handling all the concepts required for the task, face a big problem when they have to get into RFCs details and decide how to make an implementation. We underestimated this part of the work and it took more than we expected every time.

One of the reasons why IP4JVM is an interesting teaching tool is that it is simpler to understand than the internals of an OS kernel. IP4JVM is smaller and tries to be simple to understand in its design, even if that has bad performance consequences. We tend to think that it has become more difficult for a student to understand the tool. On the other hand, thanks to Object Orientation features present in the Java language, it is possible to have more control on the modifications and impact of changes.

It is possible to show a student that innovative and state of the art work is still within their reach. During all this work it was really nice to feel people motivated and sometimes wondering if it was possible that they did what they did. Making them dare to implement protocols, make them capable of questioning them, to experiment with them and to enhance them. Implementing NAT66 while the draft is being discussed by the IETF also mean that this project enables state of the art experimentation on protocols with few resources.

IV. Implementation

IP4JVM is a Java native implementation of the IPv6 protocol suite. As interfacing with the OS has to be done in C language, there must be some moment where Java objects have to be transformed into their C representation and transmitted through the wire. The decision taken was to do as much as possible in Java and minimize C handling. TCP, UDP, IPv6, ICMPv6 and even EthernetII codification and de-codification is implemented in Java and only transmission and reception are done at C level. Figure 1 depicts conceptual building blocks of the architecture.

A. The Stack

The core of the implementation is a generic protocol management framework, centered on the class `NetStack`. This class models the main network protocol manager, being responsible for processing all network information. The network process is designed using a stack of network protocols. The stack uses an abstract definition of a layered protocol, to be independent of the implementation, and uses a representation that orders them by its level in the network process model.

Each protocol implementation, specializations of the abstract class `Layer`. The protocol-suite implementations are grouped in a list of levels, each of this levels depends on the network design model, and contains the protocols that could decode and process the information being handled. The lowest part of the stack is a C bridge that just copy all the packets that are received by the physical Ethernet card into the NetStack and backward. When a packet arrives, it is copied into Java and inserted into the stack, marked as **Incoming**. CRC verification is one of the first tasks (Applications) associated to the stack.

B. IPv6 implementation

With the architecture described in Subsection IV-A, it is straightforward to note that IPv6 implementation itself is a set of classes that meets stack's API, model IPv6 messages and behaves like it. Things like a MAC address had to be modeled in order to provide a full stack implementation. Java class `ip4jvm.net.addresses.MACAddress` specializes `ip4jvm.javafwrk.Address`, making it something that the NetStack can handle in an abstract way, independently from the particular implementation, The `MACAddress` class knows particular handling of IPv6 address mapping into MAC addresses, like broadcast addresses and multicast ones.

IPv6 is added to the stack by specializing the abstract class `ip4jvm.javafwrk.Protocol`. Indeed public class IPv6Protocol provides a centralized point for different aspects of the protocol, ranging from implementation constants to routing and redirect messages processing.

The complete description of IPv6 implementation is beyond the scope of this article, but we will just sketch how it was used to implement NAT66. Figure 2 shows the core hierarchy of NAT66 set of classes. The second NAT66 draft defined two techniques: Two Way Algorithmic and Topology Hiding Option, being the first one mandatory. The implementation consists on a new class, called `IPv6NatProtocol` that specializes the class `IPv6Protocol`. NAT processing is done in an abstract way. Two different implementations of the abstract class `IPv6NatProtocol` implement each option: `IPv6NatTwoWayAlgorithmicProtocol` and `IPv6NatTopologyHidingProtocol`. It is possible to implement as many NAT techniques as required with minimal modification of the overall structure.

Before moving to the next subsection, it is worth mentioning that it becomes difficult to see the boundaries of protocol development and software engineering practices on the way problems are solved using object orientation features provided in Java. Maybe a protocol engineer, who has worked using C, would have never thought of inheritance as an implementation alternative.

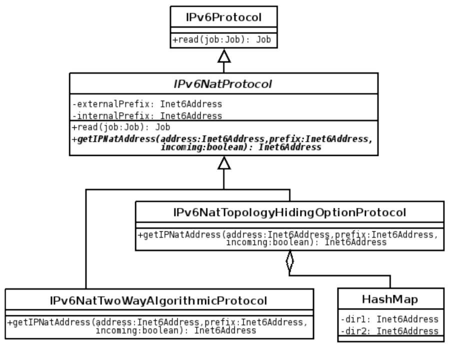
IPv6Protocol
+read(job:Job): Job
IPv6NatProtocol
-externalPrefix: Inet6Address
-internalPrefix: Inet6Address
+read(job:Job): Job
+getIPNatAddress(address:Inet6Address,prefix:Inet6Address, incoming:boolean): Inet6Address
IPv6NatTopologyHidingOptionProtocol
+getIPNatAddress(address:Inet6Address,prefix:Inet6Address, incoming:boolean): Inet6Address
IPv6NatTwoWayAlgorithmicProtocol
+getIPNatAddress(address:Inet6Address,prefix:Inet6Address, incoming:boolean): Inet6Address
HashMap
-dir1: Inet6Address
-dir2: Inet6Address

Fig. 2. NAT66 hierarchy

V. Application to Testing

IP4JVM was build with testing in mind. With current maturity and coverage of the implementation, there is enough availability of building blocks to address testing of devices based on IP4JVM objects and features. The following paragraphs present what could be current application scenarios and near future ones.

A. Current possibilities

IP4JVM is ready to be deployed and it is capable of performing probes that are beyond the scope of a regular JVM. This facts enables reuse of existing tools from the Java platform to be used for network protocol testing. Even though this seems somehow not natural, it presents certain benefits. For small companies, it is simpler to find experts on Java that could understand how to code testcases than specialists on a proprietary platform or on a testing language. The whole JUnit Testing Framework can be used to automate execution and gather results also on network protocol testing. This fact directly enables standardized -for the Java world, not yet for the protocol testing community- frameworks, tools and resources.

With the standard usage of IP4JVM it is possible to completely automate the behavior of a node, coding a Java application that runs on it. That IP4JVM would be configured with required MAC address, IPv6 addresses and so on, and could implement as many interfaces as required. Conformant behavior is taken from granted, as the implementation seeks compliance with the standards, making it simple to engineer testcases: just code the expected behavior. When it is required to test the response of a system under non conformant messages, it is only required to register the faulty implementation of the protocol that would produce the invalid message.

B. Next steps focused on networking

The next step on the development of the tool will be targeted to testing. It will allow a Java developer to access from his code and create `NetStack` objects. After this a single program running on a single virtual machine would be able to instantiate as many protocol stacks as required. Each of this stacks would be able to behave as a router or a host, and be connected independently to required physical interfaces. The behavior, internal structures and configuration will be isolated and independent. After a socket is obtained from a stack, it will present the same usage to Java, making regular programs run on top of these stacks.

Each of this stacks could provide the same services as those offered through standard network operations. It would be possible to open a socket in one `NetStack` instance and afterwards, another socket on a different `NetStack` instance. Both sockets coordinated inside the virtual machine, making each of them behave in a particular way. A single virtual machine could implement as many stacks as required, all of them, cooperating on a given test purpose. We could achieve a similar level of expressiveness as the one described before using several, distinct, virtual machines, being more efficient on resource usage.

Once again, all the possibilities could be exploited and combined with available Java tools for testing. The limit of what can be done with these elements seem to be able to implement v6RL test case specifications and maybe more.

C. Application to TTCN-3

When we introduced TTCN-3 in Section I, we mentioned that there were two standard mappings, one to C and the other to Java. Java mapping could only rely on JNI to be able to implement executable test cases. There is a new set of options now available for TTCN-3 testing with Java. Now it could be possible to code TRI and TCI with the same language as the actual protocol message description and interchange is done. This would enable faster development of the components required to turn an abstract test specification into an executable one. Network protocol testing with IP4JVM and TTCN-3 would enable test developers to fully exploit Java features and the new capability of expressing IPv6 protocol structures in Java.

VI. Future Work

There is much to be done in order to have a complete and useful implementation. The first and relevant thing to be done is to certify the implementation as IPv6 Ready. This is not a minor goal, but a really important one if we want to transmit confidence to eventual users that the stack is functionally conformant. Moreover, if testing is to be done based on IP4JVM, then we believe it is mandatory to start from a good quality implementation.

Different link layer transports should be implemented. Currently we support Ethernet bridging, but we find it interesting to implement IEEE 802.11 b/c/g/n/s data link layers.

Network oriented applications should be addressed to. It would be adequate to also have full Java resolver implementation. DHCPv6 server should be implemented in Java too. According to v6RL, a router must be able to send Router Advertisements.

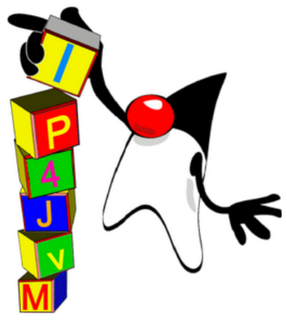

Fig. 3. IP4JVM logo

There is also much research to be done on the testing applications of IP4JVM, we already got there. IP4JVM should be applied directly to IPv6 protocol testing and combined with TTCN-3 language, so as to learn how to get the best results from both.

VII. Conclusions

It is not simple to draw some conclusions out of an ongoing project. It is possible to do native networking with Java. Even though somebody may say that nothing is impossible, we found strange that after a decade of IPv6 and even a little bit more of Java, we were not able to find other groups working on the same direction. It made us wonder if we would succeed or not.

It is possible to apply different development software engineering strategies to protocol development. We showed that incremental delivery of functionality, agile programming techniques and object orientation can be applied to protocol development.

We produced a platform for research, protocol testing and education in the field of networking. Being able to develop NAT66 while it is being drafted allows us to claim that similar results may be achieved on other fields, making IP4JVM an adequate option for protocol fast prototyping and testing.

We contributed with the education of several students during this journey. It makes us believe -one more time- that serious projects can be done with local professionals and low budget. We would like to think that their insight of network protocols would have never been the same if they had not get the chance to implement a state of the art protocol as sexy as IPv6.

Acknowledgment

The author would like to thank C. Viho and A. Baire for supporting this idea from the beginning.

References

[1] Sun Microsystems, “Java,” http://java.sun.com.
[2] S. Deering and R.Hinden, “RFC 2460 - Internet Protocol, Version 6 (IPv6) Specification,” http://www.rfc-editor.org/rfc/rfc2460.txt, 1998.
[3] IETF, “IPv6 Ready Logo,” http://www.ipv6ready.org.
[4] Etienne M. Gagnon et al., “The SableVM Project,” http://www.sablevm.org.
[5] “OpenJDK,” http://www.openjdk.org/.
[6] M. Wasserman and F. Baker, “IPv6-to-IPv6 Network Address Translation (NAT66) - draft-mrw-behave-nat66-02.txt,” http://www.ietf.org/internet-drafts/draft-mrw-behave-nat66-02.txt, 2008.