jLISP: An Open, Modular and Extensible Java-Based LISP Implementation

Andreas Stockmayer*, Mark Schmidt*, Michael Menth*

* Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany

Abstract—**LISP is a standardized overlay protocol implementing the locator/identifier split for the Internet. Multiple extensions exist, e.g., for NAT traversal, traffic engineering, etc. Existing LISP implementations are mostly platform-specific, hard to extend or they are closed source. In this work we present a Java-based implementation for LISP which is open source, modular, and features a plugin mechanism that enables simple integration of new functionality. It provides behavior for LISP nodes, infrastructure nodes, and various extensions, it is easily portable, can be run on various operating systems and platforms, in particular on Android smartphones. The demo illustrates LISP-based communication including the extensions mentioned above. In addition, the extensibility is demonstrated by a plugin for a statistics application.**

I. INTRODUCTION

The locator/identifier (Loc/ID) split separates the name of a host, its identifier, from the address of its location, the locator. The locator/identifier binding can be provided by a distributed database, the mapping system, so that traffic to a specific node can be tunneled or address-translated to its locator after lookup of its identifier. Around 2006 this concept was suggested to solve the problem of quickly increasing BGP [1] routing tables in the Internet [2] and a working group in IETF was set up to standardize the Locator/Identifier Separation Protocol (LISP) [3]. LISP by itself cannot solve this scaling issue as large-scale adoption is prerequisite. However, LISP provides an overlay network which is attractive, e.g., for traffic engineering (TE). LISP differentiates from other routing overlays through its control plane which automatically maps locators to identifiers, and the associated mapping system which supports service-specific mapping. LISP may support software-defined networking (SDN) [4], datacenter networking, service function chaining (SFC), NAT traversal, mobile networking, and others.

There are several closed-source and open-source LISP implementations. We started extending them to integrate novel functionality but discovered that they rather focus on performance than extensibility. Some of them are platform-dependent, implement only a subset of standardized features, or are no longer supported. As we feel the need for a LISP software base for research purposes, we provide jLISP as an open-source, easy to extend, and platform-independent LISP implementation that also runs on smartphones. In this paper, we give an introduction to LISP, review other implementations, explain the functionality and software architecture of jLISP and how new features can be integrated, and present a demo that illustrates jLISP in different application scenarios.

The rest of the paper is structured as follows: In Section II we give a brief overview of the LISP protocol. Section III reviews other LISP implementations. jLISP is presented in Section IV and Section V presents the demo. Section VI concludes this work.

II. LISP

The LISP protocol implements the Loc/ID split. Nodes have Endpoint Identifiers (EID) as names and Routing Locators (RLOCs) as globally routable addresses. Within a LISP domain, EIDs may be used for forwarding. Tunnel routers (xTRs) [3] encapsulate LISP traffic to other LISP domains after retrieval of RLOC/EID mappings from the mapping system. To make EIDs of a LISP domain reachable over the Internet, the xTR registers them at the mapping system. Figure 1 illustrates communication with LISP. A source node with EID *10.0.0.1* in LISP domain *10.0/16* sends a packet to a destination node with EID *20.0.0.1* in LISP domain *20.0/16*. The packet is forwarded to its default gateway which is xTR with RLOC *172.10.0.1*. The xTR sends a Map Request to the Map Resolver, the interface of the mapping system, and receives a Map Reply containing the RLOC *172.20.0.1* for EID *20.0.0.1*. The xTR encapsulates the packet with a LISP header containing control information, a UDP header to port *4342*, and IP header to the destination RLOC. Upon reception of the packet, the destination xTR strips these headers and forwards the packet to the destination node.

The xTR can enable interworking with the non-LISP Internet by address translation [5]. That means, it acts like a NAT for traffic leaving the LISP domain for the classic Internet. With this approach, a LISP domain is treated like a private network. In contrast, proxy xTRs (PxTR) make nodes in LISP domains globally reachable via BGP.

A LISP Mobile Node (MN) [3] is a mobile device with an EID and xTR functionality. If the MN becomes connected to a host network, it registers that address as RLOC for its EID at the mapping system to ensure global reachability.

LISP-TE allows forwarding LISP traffic over a sequence of so-called re-encapsulating tunnel routers (RTRs). To that end, the EID is mapped to the RLOC sequence of these RTRs. Upon reception of a packet, the RTR decapsulates the packet, looks up the mapping, and re-encapsulates the packet to the next hop in the LISP overlay.

LISP NAT traversal [6] makes xTRs behind a NAT reachable in the Internet, which is especially useful for MNs. A NAT traversal router (NTR) facilitates that function. If an xTR behind a NAT attempts to register at the mapping system, it receives a response with a list of NTRs. It then registers at an
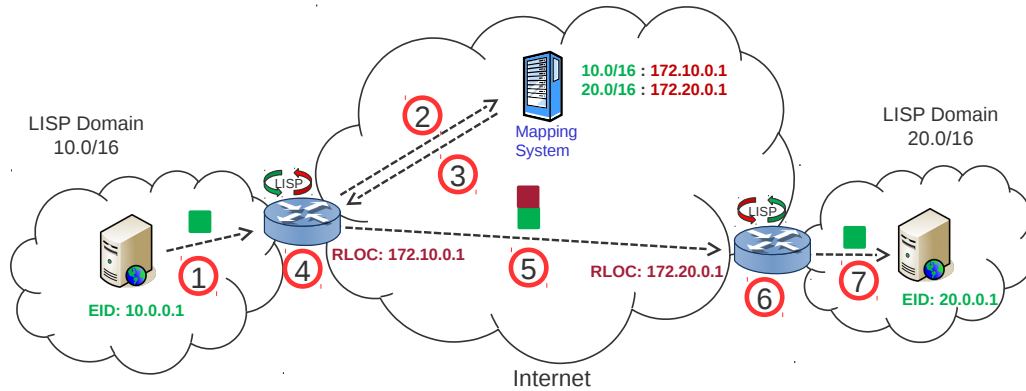
Fig. 1: Basic communication with LISP.

NTR. The NTR adds the RLOC of the xTR to its NAT table. Furthermore, it registers the xTR's EIDs with its own RLOC at the mapping system.

The LISP canonical address format (LCAF) enables more advanced TE by extending mappings with context information. E.g., the RLOC of an EID may depend on the specific service or traffic class of the packet so that voice and data traffic is tunneled by xTRs over different paths.

LISP is currently extended with features for security and hybrid access, there are multiple use cases for the MN concept in the context of datacenters and multihoming, and the LISP control plane may be applied to other data planes.

III. RELATED WORK

OpenLISP [7] is an early open source implementation whose data plane is implemented in the kernel of FreeBSD and its control plane in the user space of FreeBSD and Linux. The limitation of the data plane to FreeBSD [8] makes OpenLISP hard to use. Code for the kernel stack is neither easy to read nor a suitable base for fast prototyping of extensions. The control plane code for Linux is written in C, also hard to read, and documentation exists only for the usage of the program. Thus, also the control plane is hard to extend.

Open Overlay Router [9] (OOR), the successor of lisp-Mob [10], is an open source implementation that uses LISP or VXLAN [11]/GRE [12] as an overlay for SDN. Its focus is support for network function virtualization (NFV), e.g., by SFC, and integration with OpenDaylight [13]. OOR is available for Linux, Android (only on rooted devices) and OpenWRT. The extended feature set makes OOR attractive for application, but bloats the code which is written in C. This is an obstacle for developers who want to implement their own features in the code base.

Lispers.net [14] is a Python implementation aiming to implement the complete feature set of LISP. Additional non-LISP-specific functions and behaviors are added which makes lispers.net a general overlay controller that focuses on bleeding edge LISP technology. As the source code of lispers.net is proprietary, it cannot be used for own extensions.

There are several other proprietary LISP implementations. The implementations on Cisco routers and on AMV's FRITZ!Box home routers are most widely spread. Both can be used to test LISP and to connect with the beta network. However, they are not open source and cannot extended befor own purposes.

IV. jLISP ARCHITECTURE

jLISP is an easily extensible and portable open source implementation of LISP. It is implemented in Java and runs in the user space. We chose Java as programming language for platform independence. The code is object-oriented and modular which facilitates readability, reusability, and extensibility. We describe the architecture of jLISP, comment on supported features and a plugin mechanism, and report some performance results.

A. Architecture

jLISP is split into three different, independent modules which are realized as standalone Java packages. Figure 2 depicts these modules: the data plane, the control plane, and general networking. They are used for the implementation of xTRs, RTRs, NTRs, the mapping system, and in particular for novel LISP-based applications.
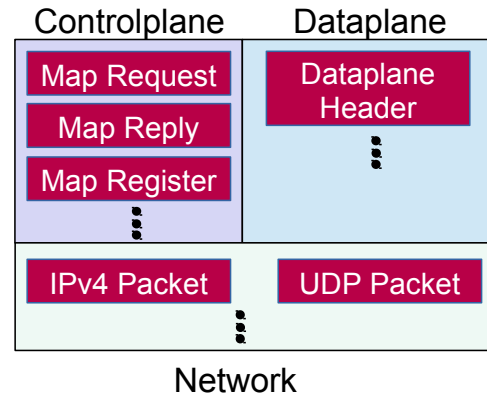

Fig. 2: jLISP modules

The control plane module contains classes for all LISP control messages, e.g., Map-Register, Map-Request, etc., which allow building and parsing control messages. A mapping system implementation consists only this control plane module, a datagram socket, and some additional logic.

The data plane module contains classes to encapsulate payload with a LISP header and decapsulate it while extracting header information. An RTR implementation leverages the control plane, the data plane module, a datagram socket, and some additional logic. An RTR receives LISP-encapsulated traffic, reads one header field, performs Map Requests, and sends LISP-encapsulated traffic to the next overlay hop.

The general networking module contains classes to build and parse transport protocol headers as well as IPv4 and IPv6 headers. he xTR uses this module for interpreting non-LISP traffic to retrieve destination EIDs from IP packets. LCAF implements conditional forwarding and requires Layer 4 information, e.g., port numbers which can be obtained from packets with classes from this module.

A central feature of jLISP is the simple construction of objects for control plane, data plane, or general networking packets. They may be either constructed from parameters or parsed from a byte stream of a packet yielding the packet data. This serialization of packet objects into packet byte streams and deserialization of packet byte streams into packet objects enables a simple handling of packets for application programmers without dealing with low-level programming. This is a significant advantage compared to existing implementations and important for extensibility and rapid prototyping. Therefore, this framework allows to write lightweight tools and LISP applications with only little knowledge of the full source code of jLISP. The control plane module is already used in our practical networking course [15] to let students write a simple variant of the LIG [16].

jLISP is a user space program and does not require modifications of the operating system. Therefore, jLISP can be run on devices without special privileges and thereby avoids some security concerns. Technically, jLISP provides a tun device over which all raw IP traffic with EID source addresses is forwarded to the xTR application. The xTR application receives raw IP traffic from the tun device, encapsulates it, and forwards it to a datagram socket,or receives encapsulated LISP traffic from a datagram socket,decapsulates it, and forwards it to the tun device. In fact, one datagram socket is used for LISP data traffic and another for LISP control traffic. As a potential modification, the tun interface may be swapped by a tap interface which offers control of Layer 2 traffic in the network. This facilitates, e.g., a simple extension of xTRs to ARP proxies to connect remote LISP domains to one VPN. The drawback is the need for parsing and constructing Layer 2 headers of the traffic which requires more computing effort. This architectural base enables porting jLISP to any platform that supports Java and offers a tun/tap driver or a VPN API like Android which may be used as substitute for a tun/tap device.

B. Features

jLISP is compatible with the LISP RFCs.. Based on the presented modules, jLISP provides explicit programs for all LISP components: xTRs, RTRs, NTRs, and the mapping system.

The xTR component is split into an ingress tunnel router and egress tunnel router (ITR/ETR). The ITR receives raw traffic and LISP-encapsulates it before forwarding. The ETR re-ceives LISP-encapsulated traffic and decapuslates it. The xTR may also re-encapsulate packets and provide RTR functionality by calling the encapsulation routine after decapsulation instead of forwarding the raw traffic.

A MN is built on the base of the xTR. The xTR is equipped with an EID on its LISP tun interface. This address is the sole prefix this xTR is responsible for and the xTR registers that EID with the mapping system whenever it receives a new RLOC which is the address of the external network interface.

The mapping system uses hash maps to store register messages for EID prefixes. They are retrieved with an algorithm for longest prefix match. Since the entire register message is saved, this structure supports new LISP register formats by design as the information is stored as opaque data. Therefore, the mapping system can store both normal EID-to-RLOC mappings and more complex EID-to-LISP-TE-path mappings containing as list of RTRs, and return them on request. The storage backend of the mapping system can be replaced by another that provides a class with a store and request method for mappings. Normally, the mapping system is filled with Map Register messages from an xTR. We also provided a non-standardized interface to allow third-party controllers to fill the mapping system with mappings and program a network. This can be used for TE experiments and integration with third-party controllers. Potential use cases are NFV and SFC applications.

jLISP also provides an NTR which is a modified xTR with a NAT and some additional logic. We provide two components: the one implementing the current Internet draft [6] and a modified version. This modification reduces the computation load on the NTR. In the current draft, the NTR interprets the Message Register message from the registering node and sends a new one to the mapping system. We proposed that a registering nodes sends an encapsulated Map Register message to the NTR, which decapsulates and forwards it to the mapping system. Furthermore, our modification especially ensures that communication behind a provider NAT still works if a flow sends packets infrequently. To that end, the client sends empty keep-alive messages to the NTR which prevents that the connection between client and NTR is deleted from the NAT table.

C. Plugins

jLISP offers a plugin mechanism to improve extensibility. For control plane traffic, jLISP provides hooks to intercept control messages before they are sent and after they are received which allows plugins to modify them. For control plane traffic, raw traffic may be intercepted before being received by the ITR or sent by the ETR, and encapsulated traffic before being sent by the ITR or received by the ETR. This enables a developer with only little knowledge of jLISP to intercept packets at any stage of the normal LISP pipeline.

D. Throughput and Load Measurements

We tested the implementation on commodity laptops on a 100 Mb/s LAN and could use the full bandwidth. The load was handled by a single core of a mobile CPU, but jLISP is able to spread it across up to 100 CPU cores if needed. This is achieved with two thread pools that are filled with up to 50 worker threads to process incoming and outgoing traffic.

V. Demo

The demonstration illustrates LISP communication between a MN behind a NAT and a node in a LISP domain using jLISP components.

We use a semi-virtualized testbed which is visualized in Figure 3. A testbed server hosts two LISP domains with nodes and xTRs realized as virtual machines (VMs). It further hosts a mapping system and an NTR. The MN runs on an Android machine. An OpenWrt [17] router with integrated NAT is used as access point (AP) and connects the smartphone with the server.
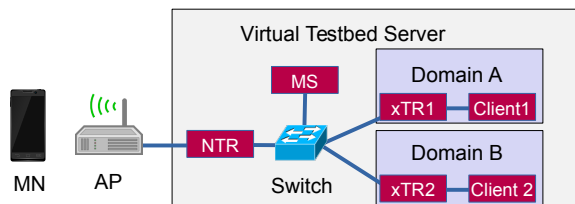

Fig. 3: Testbed setup

To be able to use hardware-accelerated virtualization on the Intel x86 platform used in the testbed server, some extensions are needed as the architecture itself is not virtualizable. Intel VT-x [18] enables basic hardware acceleration on that platform. To actually use these hardware features, the hypervisor that runs the VMs has to support them. We use KVM [19] as hypervisor, which is part of the Linux kernel, with QEMU [20] as virtualizer. The server itself is set up with an Ubuntu 15.10 [21] operating system. The VMs are managed with the libvirt [22] framework as frontend. As a result, we run multiple VMs per host with a performance close to a dedicated physical machine. The connection between the VMs and the physical interface of the testbed connected to the access point is realized by an Open vSwitch [23]. The demo shows a MN exchanging traffic with a node in one of the LISP domains. The MN first attempts to register itself at the mapping system but does not receive a Map Notify message confirming it registration. Instead, the MN receives a notification that it is located behind a NAT including a list of available NTRs. Then, the alternative NAT traversal proposed in Section IV is applied. The MN sends an encapsulated Map Register message to the NTR. The NTR decapsulates it, adds an entry to its NAT table, and forwards the Map Register to the mapping system. After the MN receives a Map Notify from the mapping system, it starts sending packets to the node in the LISP domain. The first packet is sent to the NTR. The NTR faces a cache miss, requests the RLOC for the destination EID from the mapping system, and forwards the LISP-encapsulated packet to the xTR of the LISP domain. The xTR delivers the packet to the destination. The destination responds, the packet is relayed to the xTR which also faces a cache miss, requests the RLOC for the EID of the MN from the mapping system, receives the RLOC of the NTR, and forwards the packet to the NTR. After reception of the packet, the NTR consults its NAT table for the MN's EID, and forwards the packet to the MN.

We demonstrate the extensibility, we provide a simple statistics plugin running on any node. It reports the number of sent packets to the master node while packets are exchanged. The master node aggregates and presents the data.

VI. Conclusion

We presented jLISP as a novel implementation of LISP which excels by a highly extensible architecture (modularity, object-orientation, plugin mechanism), and offers itself for rapid prototyping. jLISP is sufficiently fast and offers parallel processing if needed. It is platform-independent, runs in the user space, and supports all features of the currently standardized LISP protocol. The demo leverages jLISP and runs on a semi-virtualized testbed. It illustrates how a Mobile Node behind a NAT communicates with the help of NAT traversal with a node in a LISP domain. The NAT traversal implements an improvement of the current Internet draft. A simple statistics application illustrates jLISP's plugin mechanism.

References

[1] Y. Rekhter, T. Li, and S. Hares, "RFC4271: A Border Gateway Protocol 4 (BGP-4)," Jan. 2006.

[2] M. Menth, M. Hartmann, D. Klein, and P. Tran-Gia, "Future Internet Routing: Motivation and Design Issues," *it - Information Technology*, vol. 5, no. 6, Dec. 2008.

[3] D. Farinacci, V. Fuller, D. Meyer and D. Lewis, "RFC6830: The Locator/ID Seperation Protocol (LISP)," Jan. 2013.

[4] A. Rodriguez-Natal, M. Portoles-Comeras, V. Ermagan, D. Lewis, D. Farinacci, F. Maino, and A. Cabellos-Aparicio, "LISP: A Southbound SDN Protocol?" vol. 53, no. 7, pp. 201–207, Jul. 2015.

[5] D. Lewis, D. Meyer, D.Farinacci and V. Fuller, "RFC6832: Interworking between Locator/ID Seperation Protocol (LISP) and Non-LISP Sites," Jan. 2013.

[6] V. Ermagan, D. Farinacci, D. Lewis, J. Skriver, F.Maino, C. White , "NAT traversal for LISP," https://tools.ietf.org/html/draft-ermagan-lisp-nat-traversal-10.

[7] Luigi Iannone, "The OpenLISP Project," http://www.openlisp.org.

[8] The FreeBSD Team, "FreeBSD," http://freebsd.org/.

[9] Barcelona Tech University, "Open Overlay Router," http://www.openoverlayrouter.org/.

[10] ——, "LISPMob: LISP Mobile Node," http://www.lispmob.org.

[11] M.Mahalingam, D.Dutt, K.Duda, P.Agarwal, L.Kreeger, T.Sridhar, M.Bursell, C.Wright, "RFC 7348 Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," Aug. 2014.

[12] D.Farinacci, T.Li, S.Hanks, D.Meyer, P.Traina, "RFC2784 Generic Routing Encapsulation (GRE)," Mar. 2000.

[13] The OpenDaylight Team, "OpenDaylight," https://www.opendaylight.org/, 2011.

[14] D. Farinacci, "lispers.net," http://www.lispers.net, 2014.

[15] M. Schmidt, A.Stockmayer, "Internet Lab MSc. Course University of Tuebingen," https://thallo.informatik.uni-tuebingen.de.

[16] D.Farinacci, D.Meyer, "RFC6835: The Locator/ID Seperation Protocol Internet Groper (LIG)," Jan. 2013.

[17] OpenWrt team, "Openwrt," https://openwrt.org/.

[18] Intel Corp, "Intel Virtualization Technology(VT-x)," http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2006.

[19] A. Kivity *et al.*, "kvm: the Linux virtual machine monitor," in *Linux Symposium*, 2007.

[20] QEMU team, "QEMU 2," http://wiki.qemu.org/ChangeLog/2.0, 2014.

[21] Canonical Ltd, "Ubuntu 15.10 (Wily Werewolf)," http://releases.ubuntu.com/15.10/, 2015.

[22] Red Hat, "libvirt: The Virtualization API," http://libvirt.org, 2012.

[23] Open vSwitch team, "Open vswitch," http://openvswitch.org/.