A Self-Configuring Communication Virtual Machine[1]

S. Masoud Sadjadi[2], Selim Kalayci[3], and Yi Deng[4], *Member, IEEE*
School of Computing and Information Sciences
Florida International University, Miami, FL, U.S.A
{sadjadi,skala001,deng}@cs.fiu.edu

***Abstract*— Today's communication-based applications are mostly crafted in a stovepipe development paradigm, which is inflexible to be used by various domain-specific applications and costly in the development phase. In a previous paper [1], we proposed a new design called CVM (Communication Virtual Machine) to overcome these problems by having a high-level API which can be reused and extended easily for user-centric applications in any domain. Within CVM framework, we came across a practical issue, which is actually the case for any end-to-end multimedia communication, namely the NAT-traversal (network address translation) problem that limits the reliability and availability of CVM and variants of CVM. In this paper, we explain about the necessity of self-configuration for the NAT-traversal problem in end-to-end communications, and propose a solution within the core CVM framework.**

Keywords: Communication virtual machine, network address resolution, autonomic computing, self-configuration.

I. INTRODUCTION

Communication-based software which mainly provides end-to-end transfer of data, such as audio, video, etc. developed and used today, are mostly being developed in a stovepipe style and gives a general-purpose solution for whatever type of domain (purpose) it is going to be used. But, what really needed is to have a user-centric application which can be tailored easily from a generic framework by a domain expert, without needing any software development phase. This led us to a prior and ongoing work [1], in which, this generic-framework is called CVM (Communication Virtual Machine).

However, this generic framework needs to have a solution for a very common practical problem, namely the NAT-traversal problem, in its core, so that any variants of CVM would be ready for use without requiring any additional effort. We propose a self-configuring solution within our CVM framework, which is able to adapt itself to different NAT configurations, so that the establishment of a communication session and media transfer within the session can proceed without requiring any user-intervention.

There are four basic types of NAT devices which act differently when dealing with network traffic coming-in/going-out the private network they are responsible for. We need two phases to overcome the NAT problem. First, each participant of the communication session needs to detect the NAT type and the public address information, it is using while communicating with outside world. Second, we need a mechanism to traverse the NAT devices in one or two ends of the communication and reach to the intended user. For the first phase, STUN (Simple Traversal of UDP Datagrams through NATs) [2] protocol is a very common and effective method used today. For the second phase, either a well-known relay node can be used to forward the packets both ways, or UDP hole punching technique [3] can be used for a direct end-to-end communication. Most of the communication applications today need to go through the same phases in development to incorporate some variation of the above techniques in their applications to handle the NAT traversal problem. Our self-configuring CVM framework eases the task of application developers, by providing tools that can be used to generate a domain-specific communication application, and resolving the NAT-traversal issue transparently.

By embedding a self-configuring NAT-traversal solution within our layered CVM architecture, we are offering a generic framework for communications-related applications in which the NAT-traversal is accomplished transparently as one core functionality provided within CVM.

To validate our results, we conducted several demonstrations with our CVM prototype for almost all kind of communication tasks that people use today, such as, instant messaging, audio/video transfer, file transfer, etc. We run our prototype on different machines and simulated all different combinations of NAT configurations (except symmetric NAT) between users. We will talk more about details of our testbed in Section 4.

The results we got from these simulations validated our design, and CVM prototype was able to self-configure itself to the NAT configuration it is bound, which results in successful communication sessions between participants. This makes our CVM framework more reliable and robust to move it towards the idea behind our CVM approach, providing a generic framework that can be re-used easily for domain-specific communication applications.

[1] This work was supported in part by the National Science Foundation (grants HRD-0317692, OCI-0636031, and REU-0552555) and in part by IBM (SUR and Student Support awards in 2005, 2006 and 2007).

[2] Asst. Prof. S. Masoud Sadjadi is with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (phone: 305-348-1835; fax: 305-348-2336; e-mail: sadjadi@cs.fiu.edu).

[3] Selim Kalayci is with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (e-mail: skala001@ cis.fiu.edu).

[4] Prof. Yi Deng is with the School of Computing and Information Sciences Florida International University, Miami, FL 33199 USA (e-mail: deng@cis.fiu.edu).

II. BACKGROUND

Convergence of data, voice and multimedia communication over IP networks has enabled a wide-range of communication-based applications. But, the stovepipe approach followed to develop these applications does not clearly separate the concerns of application logic from network-level communication logic and media delivery. This causes the end-product to have fixed set of functionalities and it is difficult to adapt it to various user needs. Also, most of these applications are not able to interoperate without extra work. Moreover, domain-specific applications like telemedicine and disaster management, needs to go through the whole application development cycle which is costly, lengthy and error-prone. This kind of problems led us to our prior work [1], in which we proposed a user-centric, model-driven approach for designing and developing such communication-based solutions across application domains. We called this framework **CVM** (Communication Virtual Machine).

First of all, in CVM framework, general purpose or domain specific communication needs are specified in a so-called *communication schema*, which is independent from network-level issues and actual media communication. This communication schema, which can be formed by the user intuitively or can be provided as certain templates, constitutes the base of communication instance logic. This schema is then instantiated, negotiated, synthesized and executed, in order, to meet the needs of the user.

This model-driven communication is supported in CVM by four-layer architecture as shown in Figure 1. This architecture lets us to separate and encapsulate major concerns of communication modeling, synthesis, coordination, and the actual delivery of communication elements. Next, we will briefly talk about each layer and their responsibilities.

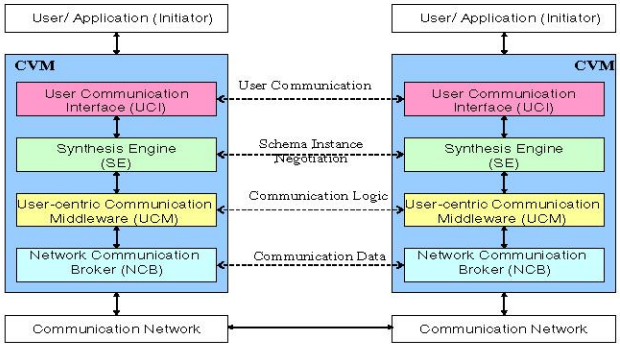

Fig. 1. CVM Architecture

UCI (User Communication Interface) layer is responsible for providing means to define and manage *communication schema*. For this purpose, we defined both an XML-based and a graphical *communication modeling language*, for more details you can refer to [1, 4]. **SE** (Synthesis Engine) layer first manages the negotiation of communication schema between participants, and then transforms this schema to an executable *communication control script*. This script represents the network-independent control logic for the communication session specified in the communication schema. **UCM** (User-centric Communication Middleware) layer is the execution engine for communication control scripts. It manages this task by invoking the proper services provided by the NCB layer. **NCB** (Network Communication Broker) layer provides a uniform API of high-level and network-independent services to UCM layer to manage the actual communication tasks needed by the application. We will talk more about this layer in Section 3.

NAT (Network Address Translation)[5] method is used to map an end-node's internal network address to a globally accessible network address, when that end-node wants to communicate with another end-node that doesn't belong to the same private network. This is done basically for two reasons, to hide the internal address within a private network from outside world, or to be able to accommodate more end-hosts than the number of public addresses available within a private network. NAT devices placed at the edge of a private network does the mapping of an internal address to a public address, and also keeps the state information so that the packets belonging to the same session can reach to the correct internal node.

While enabling the communication between internal and external nodes transparently, NAT has several drawbacks for end-to-end communications. In direct end-to-end communications, the peers need end-point address of the peer they want to communicate. For this reason, applications at each peer forward the address information they can be reached, to the other peers. However, if a peer is behind a NAT device, the information it forwarded to other peers is its private address information and doesn't match with the mapping at the NAT device it is behind. So, the receiving peers will not be able to reach to the correct peer, because the address information they got is either invalid or points to a wrong end-point.

Besides the negative effect caused by NAT devices by altering end-point address information, NAT devices treat communication through them in four different ways. In literature NAT devices are categorized into four categories according to their treatment for the communication going through it. First one is called *Full Cone NAT*, which maps the requests coming from the same internal IP address and port to the same global IP address and port. Second one, *Restricted Cone NAT*, does the mapping like Full Cone NAT, but an external node can send a packet to an inside node only if the inside node had sent a packet to that external node before. Third, *Port Restricted Cone NAT* is the stricter version of Restricted Cone NAT, where a request from an external IP address and port can reach to an internal IP address and port only if the internal node had sent a packet to that external IP address and port before. Fourth, *Symmetric NAT* maps the same internal IP address and port to some global IP address and port for reaching a specific destination IP address and port. But, if the same internal IP address and port wants to reach to a different destination IP address and port, another mapping is used. Also, an external

node can send a packet, if the internal node had previously sent a packet to it.

To be able to provide direct end-to-end communication between peers, first, we need to detect the NAT type and the mapped address information at each peer. Then, we need to incorporate this information within a mechanism to be run by peers collaboratively. We incorporate a widely used method for the first phase in our framework, which is called **STUN** (Simple Traversal of User Datagram Protocol (UDP) Through NATs) [2]**.** In this simple client-server protocol, the end-host as a client interacts with two separate STUN servers in a certain manner, which ends up revealing the aforementioned information about the end-host and the NAT device it is behind. We need to note one important fact here, that, this protocol does not help for retrieving address information where there is a symmetric NAT. The reason for that comes from the nature of symmetric NAT, which maps the internal IP address and port onto a different external IP address and port when contacting with different destination peers.

Before the direct connection is established between peers, end-point address information is exchanged between peers through some centralized server or a relay node. But, this is not enough for peers to deliver direct messages with each other. Because, Restricted cone and Port Restricted cone NAT requires, an application at an internal node to send a packet to an application at an external node to allow the packets pass through in the reverse direction. Coming over this problem needs both peers to work collaboratively which is formalized in a method called **UDP hole punching** [3]. In UDP hole punching, basically, both peers send a dummy packet to the address they received from the other side, making their NAT devices to allow for further incoming packets to be received from the other side. After this step, it is expected to have a direct end-to-end communication between two peers behind any NAT devices, except for symmetrical NAT.

III. SELF-CONFIGURATION IN CVM

Our CVM framework makes use of self-configuration approach to overcome NAT-traversal problem. Since NAT-traversal problem is related with network address management and delivery of actual data, it is handled by the NCB layer of CVM. So, first we will explain some details about NCB abstraction and design and then we will describe how self-configuration is handled by NCB.

A. *Network Communication Broker (NCB)*

NCB serves as a unified abstraction within CVM framework, to provide basic user-centric communication services reusable by different domain-specific communication applications. Applications built on NCB are transparent to the details of underlying network protocols and infrastructure.

We identified the core abstractions to satisfy the needs of user-centric multimedia communication applications. First, NCB abstraction provides registration and presence mechanism, to allow users to be involved in a communication environment and a way for the users to identify the presence of other users in the system. Second, NCB abstraction provides a user session, which defines a certain communication session involving two or more participants exchanging different kinds of media with each other. Third, it encapsulates the details of end-host networking infrastructure which makes it portable over heterogeneous networking infrastructures. Fourth, its interface is able to notify or queried about the underlying network and system events. Finally, it allows self-management policies to be defined and applied to control how media is exchanged and delivered.

Based on these abstractions, NCB architecture is shown in Figure 2. *NCB manager* handles the initialization and basic configurations of NCB. Also, it creates a new session manager for each session and oversees all of the active user sessions. *Session manager* deals with an individual session, specifically, the participants involved in a session, call processing and negotiating and preparing the groundwork for media delivery before delegating the control to *Media Processing&Transmission* module to actually deliver the media. *Signaling* module handles basic signaling operations at the network level, such as registration, inviting a user, media negotiation, etc. *QoS and Self-management* module assists the Media Delivery module by automatically adapting media transmission parameters based on the policies defined and the status of the current network environment. We will talk about *NetworkAddressManager* in the next section.

For the implementation of NCB, we chose the SIP (Session Initiation Protocol) [6] as the signaling protocol, since SIP is accepted as a standard protocol for today's multimedia communication applications and is interoperable with most communication infrastructures through various kinds of gateways. Media negotiation is done through SDP (Session Description Protocol) [7] message exchanges, which is sent along with SIP messages. Real-time multimedia transmission is achieved using RTP (Real-time Transport Protocol) [8]. More specific details about implementation can be found in [9].

B. *Self-Configuration in NCB*

Self-configuration in NCB, to allow end-hosts behind different types of NAT, and communicate directly with each other is managed by the help of *NetworkAddressManager* module. As seen in Figure 2, *NetworkAddressManager* intercepts all the calls going for the Signaling module. Signaling module handles basic signaling operations such as registration, user invite, media parameters negotiation. This module inserts and forwards address information to presence server, which in turn may forward it to other end-hosts. Hence, the address information used in the packets going out of Signaling module must include correct information for a successful communication.

Before going forward to how this self-configuration is done by *NetworkAddressManager,* we need to mention about two variables our solution keeps track of, within *NetworkAddressManager* module. First variable is, *localAddress*, which includes the local IP address of the end-

host, and checked periodically to decide whether a NAT detection procedure is needed or not. Second variable, *globalAddress*, includes the external IP address acquired as the result of the NAT detection procedure through STUN

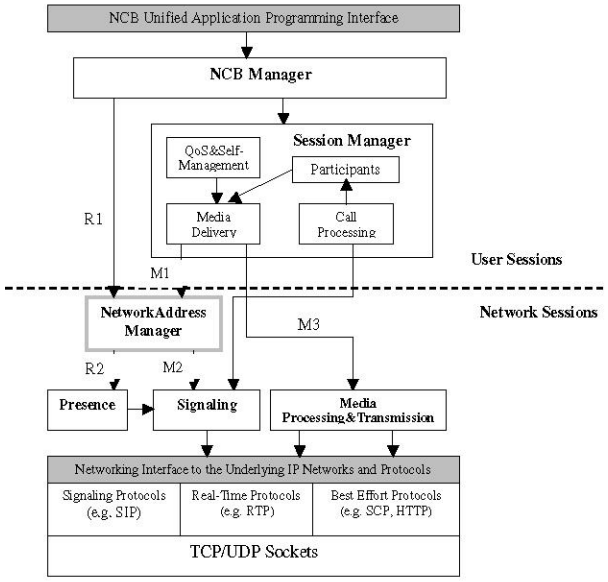

Fig. 2. Architecture of NCB

protocol. These variables are initialized at the startup of the application, and if the end-host is not behind any NAT device, they will include the same information.

Now, let's explain the interception labeled as R1 in Figure 2. One of the tasks of Presence module is to register the end-host with the presence server periodically. Presence server keeps track of location of each end-host registered with it. The interception made at this point makes the end-host to verify/update its localAddress and globalAddress information periodically. This saves us some time during the media session negotiation (M1, M2). If there was a change in these variables before a media session negotiation, it can be detected in one of the previous periodic registration procedures. In this case, media session negotiation can proceed without needing any NAT detection procedure, because it had already been performed.

Verify/Update process for the localAddress and globalAddress mentioned above is as simple as this: If the localAddress information did not change during the last registration interval, no action is necessary; we just verify localAddress and globalAddress. But, if the localAddress is different from the previous value, it means that the network configuration and address information has changed since, and we need to update localAddress information, and also globalAddress information through running the NAT detection procedure.

Actually, network configuration change does not happen very frequently, unless end-host handovers between different wireless networks. Hence, interception done periodically through Registration module will just verify the current address information most of the time. Even though we intercept each registration message, this is not crucial for a successful direct connection between end-hosts. Because even though we do periodic verify/update operation through registration messages, we also intercept the messages sent for media session negotiation (M1). This is done to make sure, correct information is put into SDP messages, especially if the time interval between each registration period is long. During the negotiation phase of the Media Delivery process, labeled as M1and M2, end-hosts forward their address information within SDP messages to each other via the SIP server. Each end-host processes the received SDP message to retrieve the address information and other negotiation related information before establishing a direct communication link between them. For this reason, just before exchanging SDP messages, end-hosts verify/update their address information. Each end-host sets the connection parameter within their SDP message as the globalAddress. This phase accomplishes the task of exchanging the correct address information, so that end-hosts can use this information to directly connect to each other.

After the negotiation is complete, end-hosts theoretically can start communicating directly with each other, because they have the address information needed to reach each other. But, if there is a restricted cone or port restricted cone NAT intercepting the packets destined to an end-host, end-host behind this type of NAT can not receive any packet from an external node, unless it first sends a packet destined to that external end-host. Thus, we need to use UDP hole punching in such a case, to traverse this type of NAT and reach to the end-host behind it. In our system, after the successful media session negotiation phase, we use UDP hole punching mechanism at all times, to start a media communication between end-hosts. The reason for using UDP hole punching all the time is its simplicity and cost-effectiveness. Because even though in some cases it may be unnecessary to use UDP hole punching, it requires only one random packet to be sent towards the remote end-point, which doesn't cause any overhead in implementation and doesn't consume time or other resources. If we didn't choose to use UDP hole punching all the time, we would need to have some extra steps. First, we would need to pass the type of NAT to the remote side along with address information. Then we would need to process NAT type information and decide whether to use UDP hole punching or not. As a result, using UDP hole punching at all NAT type cases serves out our needs with almost no overhead.

Self-configuration mechanism we described above fulfills the task of transparent NAT traversal, to achieve direct end-to-end communication between end-hosts behind any type of NAT except the symmetric NAT. The end-hosts involved in a communication session can exchange different types of media with each other directly, even when some of the end-hosts are behind a NAT device. But, if the global network address for one of the end-hosts changes during the media transmission, that end-host becomes unreachable from other end-hosts involved in the communication session. This problem may be addressed through re-establishing the session between the users through passing the updated address information. But, this naïve method needs to be

refined to minimally effect the ongoing communication session; especially when multiple parties are involved and multiple media sessions are established. Coming up with a low overhead solution to this problem remains as a future study, which would provide self-healing property to our system.

IV. EXPERIMENTAL RESULTS

We have implemented a CVM prototype including self-configurable NCB, using Java technologies. We incorporated the open source JAIN SIP [10] implementation by NIST and also open source STUN implementation from Sip-communicator [11] project. More details of our prototype implementation can be found in [1, 9].

We are using an in-home server as the registration server and SIP server. All SIP messages between peers are forwarded through this server, while media transmission is done directly between peers. We deployed CVM prototype implementation on several different machines to demonstrate its self-configuration capability. These machines were either on Open internet, or behind Full cone, restricted or port restricted cone NAT. We tried to have a direct media communication session, and transmit live audio-video streams between all possible combinations of these machines. Results approved the self-configurability of our system without adding too much extra overhead to the end-hosts. We were able to establish a direct media session and transmit audio-video messages successfully in all demonstrations. We also tested the self-configuration after the application has been started up and logged in to the system, by switching between different wireless networks with different NAT types. This demonstration also went through without any problem.

V. RELATED WORK

Multimedia communication applications widely used today, such as MSN Messenger, Yahoo Messenger and Google Talk provides generic platforms which can not be extended to satisfy the needs of specific user requirements. Also, these applications do not interoperate with each other, requiring users to switch between these tools.

Communication middleware has been worked on quite extensively. One of these works is presented by Schmidt [12] talks about the usage of patterns and frameworks to reduce the complexity associated with the large and growing number of multimedia data types, traffic patterns and end-to-end QoS requirements. He also pointed out the difficulties of developing communication applications caused by the limitations of low-level native APIs and the limitations of higher-level middleware.

Stiller et al. [13] also described the design and implementation of a communication middleware called Da CaPo++, which is adaptable to different application needs. Authors claimed that Da CaPo++ automatically configuring suitable communication protocols, and offering an easy to use object-oriented API.

Java Telephony API [14] is a high-level API specific to the traditional telephony applications, so it does not support multimedia communication applications. Parlay API [15] enables the rapid creation of telecommunication services. These frameworks mostly address the concerns for server-side architectures, whereas NCB layer in our framework addresses the concerns of the client-side middleware.

Commercial software, such as Skype [16], MSN Messenger [17] resolves the NAT problem, but their solution is proprietary or uses intermediaries or customized entities to forward the media traffic. Open source SIP Communicator [11] implements NAT detection techniques but doesn't provide a reliable NAT traversal solution. Another open source implementation YATE [18] provides a partial solution to the NAT problem via the usage of YATE servers to forward the RTP streams to the correct address.

VI. CONCLUSION

CVM provides a flexible and easy-to-use framework for developing domain-specific applications. CVM is composed of a layered architecture, separating the concerns of application logic and network-level communication and media delivery issues. In this paper, we have presented our self-configuration solution for the NAT-traversal problem within the CVM architecture. This is done by intercepting the outgoing media negotiation messages with our NetworkAddressManager module in NCB layer. This module, using several mechanisms like STUN and UDP hole punching in a well-defined systematic way, enables end-hosts to communicate with each other directly. Transparent NAT-traversal solution provided by NCB layer follows our separation of concerns methodology, and makes NCB easy to deploy in other systems.

Experiments we have conducted with three different types of NAT configurations gave us successful results. NAT-traversal for another type of NAT, namely the symmetric NAT has not been addressed in this paper. But, it is also the fact that the communication between end-hosts with at least one of them behind a symmetric NAT always can be realized by using some relay hosts in between.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation (grants HRD-0317692, OCI-0636031, and REU-0552555) and in part by IBM (SUR and Student Support awards in 2005, 2006 and 2007). The authors would like to thank Chi Zhang, Raju Rangaswami, Peter Clarke, Vagelis Hristidis, Nagarajan Parbakar, and Andrew Allen for their contributions to this work. Also, we would like to thank Yingbo Wang, Yali Wu, Mario Lorenzo, Marylurdys Hernandez, Eric Johnson, Eduardo Monteiro, Weixiang Sun, Eric Sanchez, Yingbo Wang, Robert Redway, Farid Hosseini, Yasmary Hernandez, Jonathan Corrales, and Onyeka Ezenwoye for their participation in CVM prototype implementation.

REFERENCES

[1] Y. Deng, S. M. Sadjadi, P. Clarke, C. Zhang, V. Hristidis, R. Rangaswami, and N. Prabakar, "A communication virtual machine", the 30th International Computer Software and Applications Conference, September 2006.

[2] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", RFC 3489, March 2003.

[3] Bryan Ford, Pyda Srisuresh, and Dan Kegel , "Peer-to-Peer Communication Across Network Address Translators", USENIX, April 2005.

[4] Peter J. Clarke, Vagelis Hristidis, Yingbo Wang, Nagarajan Prabakar and Yi Deng. "A Declarative Approach for Specifying User-Centric Communication", Symposium on Collaborative Technologies and Systems (CTS), 2006.

[5] P. Srisuresh, and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, August 1999.

[6] M. Handley, H. Schulzrinne, E. Schooler and J. Rosenberg, "SIP: Session Initiation Protocol", RFC 2543, March 1999.

[7] M. Handley, and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

[8] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 3550. July 2003.

[9] C. Zhang, S. M. Sadjadi, W. Sun, R. Rangaswami, and Y. Deng, "A User-Centric Network Communication Broker for Multimedia Collaborative Computing", International Conference on Collaborative Computing (CollaborateCom), November 2006.

[10] JAIN SIP: https://jain-sip.dev.java.net/, 2006.

[11] SIP Communicator**:** https://sip-communicator.dev.java.net/, Network Research Team
Louis Pasteur University - Strasbourg, France

[12] D. C. Schmidt. "Applying Patterns and Frameworks to Develop Object-Oriented Communication Software", volume 1 of Handbook of Programming Languages, MacMillan Computer Publishing, 1997.

[13] Burkhard Stiller and Christina Class and Marcel Waldvogel and Germano Caronni and Daniel Bauer, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience", IEEE Journal on Selected Areas in Communications, vol. 17 no. 9 Sept. 1999, pages 1580 – 1598.

[14] Java Telephony API (JTAPI): http://java.sun.com/products/jtapi/.

[15] Parlay API: http://www.parlay.org

[16] Skype: http://www.skype.com.

[17] MSN Messenger: http://messenger.msn.com.

[18] Yet Another Telephony Engine (YATE): http://yate.null.ro/pmwiki/.