Container NATs and Session-Oriented Standards: Friends or Foe?

Alessandro Amirante
Meetecho S.r.l.

Simon Pietro Romano
University of Napoli Federico II

***Abstract*—This article highlights issues that arise when deploying network address translation middle-boxes through containers. We focus on Docker as the container technology of choice and present a thorough analysis of its networking model, with special attention to the default *bridge* network driver that is used to implement network address translation functionality. We discuss some unexpected shortcomings we identified and elaborate on the suitability of containers for deploying services based on the Interactive Connectivity Establishment standard protocol. To support our findings, we present experiments we conducted in a real-world operational environment, namely a WebRTC service based on the Janus media server.**

CONTEXT AND MOTIVATION

■ **Internet-based distributed applications** rely more and more on the microservices paradigm for what concerns their design. Microservices indeed allow us to embrace well-known design patterns like the separation of concerns, thanks to the presence of a number of independently operating components, whose scalability can easily be guaranteed through dynamic deployment and orchestration techniques. When it comes to the implementation, containers currently represent the most natural choice, since they allow for an easy mapping between the designed microservices and their respective real-world operational counterparts.

In this article, we tackle a specific issue that one has to face when adopting the aforementioned approach. Namely, we take distributed

Digital Object Identifier 10.1109/MIC.2019.2952064
Date of publication 8 November 2019; date of current version 17 January 2020.

WebRTC[1] applications as an outstanding example of systems taking advantage of the (container-based) microservices paradigm. For such applications, we make the consideration that real-world deployments (which typically happen in the cloud) necessarily call for the adoption of the Interactive Connectivity Establishment[2] (ICE) protocol, which has been specifically devised in order to solve network node reachability issues in the presence of network address translators (NATs).

In the depicted scenario, based on our experience with deploying scalable real-time multimedia services in the wild through Docker containers, we identify a specific issue that arises when relying on server-side Docker-based components, which get deployed behind NATs and are hence not configured with publicly reachable IP addresses. Such an issue indeed caused us lots of troubles in the past and forced us to deeply investigate the way Docker implements the NAT concept. The article hence presents the lessons we learnt and provides some suggestions as to how best deploy one's own containers for real-world operational environments.

The article is organized as follows. "Interactive Connectivity Establishment" introduces the basic concepts behind the Interactive Connectivity Establishment protocol and its utilization to optimize the NAT traversal in the Internet. "Docker Networking Model" presents Docker's networking architecture, by delving into the details of the so-called container networking Model. This paves the way for the discussion in "Docker NAT Functionality," where we specifically focus on how Docker implements the previously discussed NAT traversal techniques. "Real-World Example: Janus WebRTC Services" analyzes a real-world example associated with container-based WebRTC applications, with special reference to the deployment of one or more Janus WebRTC media servers behind NATs. "Final Considerations" discusses lessons learnt and proposes a few deployment guidelines for services like the one analyzed in the article. Finally, "Conclusion" concludes the paper by summarizing its main contributions.

INTERACTIVE CONNECTIVITY ESTABLISHMENT

NATs have been considered enemies of real-time applications since the session initiation protocol (SIP)[3] has become widely adopted in the early 2000s as the signaling protocol to initiate Voice over IP or multimedia teleconferencing calls. This was the main reason which drove to the definition of the ICE technique as a means to provide network address translation (NAT) traversal functionality to any session-oriented protocol. It leverages both STUN[4] and TURN,[5] the former being a protocol for address discovery and connectivity check, the latter being a protocol for involving media relays in the communication. ICE proves effective in the presence of all types of NAT, as it will be explained in more detail in "Docker NAT Functionality." It starts from the assumption that for a single host multiple IP addresses might be associated

- *host* addresses: these are the IP addresses that have been assigned to the interfaces the host in question is equipped with, plus the ports that have been picked up for the network communication to happen;
- *server-reflexive* address: this is a specific NAT binding (i.e., an <*IPaddress, port*> pair) that is allocated on a NAT when the host sends a packet through the NAT to a STUN server deployed in the Internet;
- *relayed* address: this is a specific <*IPaddress, port*> pair that is reserved on a publicly available TURN server when the host sends a TURN Allocate request to the TURN server itself;
- *peer-reflexive* address: similar to a server-reflexive address, but discovered by the peer itself from a direct response received, as briefly explained later in this section.

All of the above (transport) addresses are called *ICE candidates* and can indeed exist both in their UDP and TCP versions.

An ICE transaction takes place in six steps: *gathering*, *prioritizing*, *encoding*, *offering/answering*, *checking*, *completing*. During the *gathering* phase, all potential ICE candidate addresses are collected through inspection of the local interfaces configuration, as well as through transactions with the configured (if any) STUN and TURN servers. A priority is then assigned to each and every collected candidate (*prioritizing* phase). Typically, host candidates have a higher priority than peer-reflexive candidates, which in turn come before server-reflexive candidates. Relayed addresses have the lowest priority and are usually left as a last resort. During the *encoding* phase, candidates, together with their assigned priorities, are then properly encoded as session description protocol (SDP)[6] attributes and sent (either individually, as so-called trickle candidates,[7] or as a bundle, within the payload of a signaling protocol) to the remote party, which will answer back with its own collected candidates. This is the *offering/answering* phase. Once all of the ICE candidates are available to both parties, they can get properly paired (in descending order of combined priority). For each such candidate pair, reachability checks are conducted by issuing STUN requests toward the remote party. If the *checking* phase successfully completes, the *completing* phase begins, as we have found a "working" pair of addresses, which can then be used for the communication.

It is worth noticing that during the checking phase, the received "binding response" message might contain an address which does not match any of the already collected candidates. Such an address, also known as *peer-reflexive candidate*, indeed contains the same IP address as the checked candidate, but a different port number. Peer-reflexive addresses form new candidate pairs to be tested through the default STUN-based checking approach.

From the above description, it is clear that ICE has been conceived at the outset as a solution to effectively cope with the presence of NATs in the network. In this article, we will focus on few important shortcomings that we identified while deploying session-oriented real-time multimedia services in operational environments involving container-oriented, Docker-based, NAT middle boxes. In order to do that, we will have to dig a bit deeper into the details of how the networking stack is realized in Docker. This is the main subject of the following section.

DOCKER NETWORKING MODEL

The Linux kernel features an extremely mature and performant implementation of the network stack. Docker networking uses such kernel's networking stack as low-level primitives to create higher level network drivers. To some extent, Docker networking *"is"* Linux networking. Networking features in Docker are implemented in the libnetwork library.[8]

The Docker networking architecture is built on top of a set of interfaces called the *Container Networking Model* (CNM), whose components are depicted in Figure 1 and described below.

Network Sandbox

A *Network Sandbox* contains the configuration of a network stack that is used by a container. It includes the routing table, names resolution service, virtual interface card, etc. In Linux, Docker's Network Sandboxes are implemented through *Network Namespaces*, which allows us to have different and separate instances of network interfaces and routing tables that operate independent of each other. By default, in fact, the set of network interfaces and routing entries is shared across the entire operating system. Namespaces provide resource isolation by wrapping a global system resource into an abstraction, which is only bound to processes within the same namespace.

Endpoint

An *Endpoint* is the means by which a container joins a network. Containers joining multiple networks have multiple endpoints within the same sandbox.

Network

The CNM defines a *Network* as a simple collection of endpoints that have connectivity between them. A network can be implemented through a Linux bridge, a VLAN, etc.

Network Driver

A Docker *Network Driver* provides the means through which Docker networks can actually work. Network drivers are pluggable modules that can be used simultaneously and concurrently by sandboxes. Each Docker network is instantiated through a single network driver. The Docker engine provides an out-of-the-box set of native network drivers, namely *host*, *bridge*, *overlay*, and *macvlan*, which are briefly introduced in the following sections. New

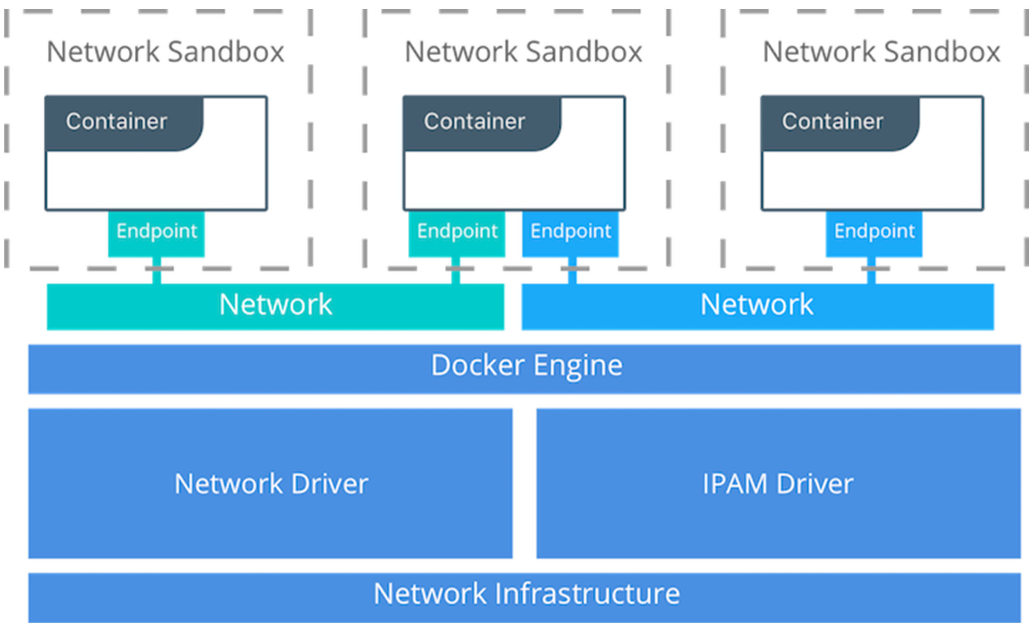


Figure 1. Docker's Container Networking Model.

network drivers can also be implemented and loaded into the *Docker engine*, e.g., to provide integration with new hardware products.

1) *Host driver:* With the *host* driver, a container uses the networking stack of the host. There is no namespace separation, and all interfaces on the host can be used directly by the container.
2) *Bridge driver:* The *bridge* driver creates a Linux bridge on the host that is managed by Docker. By default, containers on the same bridge can communicate with each other. External access to containers can also be configured through the bridge driver. This driver is used to implement NAT functionality in Docker, i.e., networks realized via the bridge driver are NATted networks. We will further analyze the behavior of this module in "Docker NAT Functionality."
3) *Overlay driver:* The *overlay* driver creates an overlay network that supports multihost networks.
4) *Macvlan driver:* The *macvlan* driver is an underlay driver that exposes host network interfaces directly to containers running on the host itself. Macvlan allows a single physical interface to have multiple MAC and IP addresses; as such, it can be used to provide IP addresses to containers that are exposed directly in the underlay network and are routable on it.

During the network and endpoints lifecycle, the CNM model controls the IP address assignment for network and endpoint interfaces via the IPAM driver(s). Libnetwork has a default, built-in IPAM driver and allows third party IPAM drivers to be dynamically plugged. On network creation, the user can specify which IPAM driver libnetwork needs to use for the network's IP address management.

DOCKER NAT FUNCTIONALITY

We conducted a thorough analysis of the built-in NAT functionality provided by Docker, namely by its *bridge* network driver, and elaborate on its suitability for ICE-based applications. The bridge driver is used by default when a container is created if no other network mode is specified. As already anticipated in "Docker Networking Model," every time a container uses the bridge driver to join a network, it ends up being behind a NAT. Such NAT is implemented by Docker by leveraging *Netfilter,*[9] a framework provided by the Linux kernel that enables various networking-related operations. The observations we make in this section, then, are

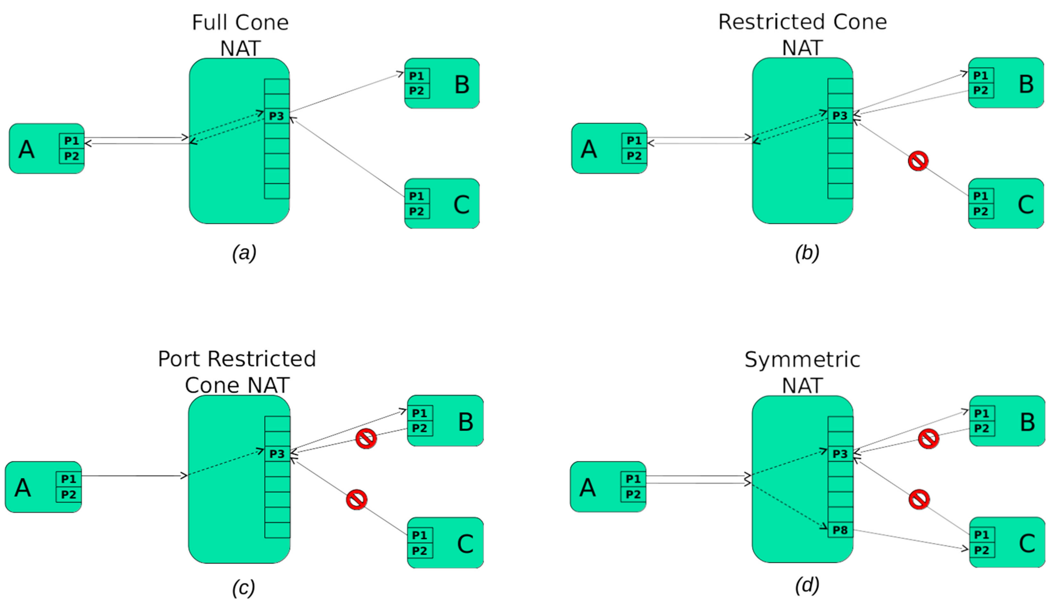
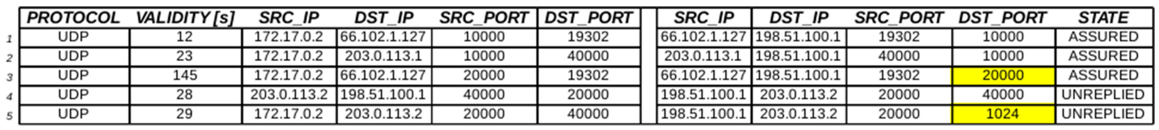


Figure 2. Types of NATs. (a) Full cone. (b) Restricted cone. (c) Port restricted cone. (d) Symmetric.

not Docker-specific but can be applied to any NAT implementation based on Netfilter. Our first goal was to classify the NAT provided by Docker among the four different types defined in RFC3489,[3] namely *Full Cone*, *Restricted Cone*, *Port Restricted Cone*, *Symmetric*. Even though this is not the most fine-grained classification of NAT variations, as it has been extended in RFC4787,[10] it is accurate enough for the sake of this article. Figure 2 sketches the behavior of these four types of NATs. A full cone NAT forwards any packet addressed to its public-facing transport address, as depicted in Figure 2(a). A restricted cone only allows forwarding of packets coming from the IP address of the host that has been previously contacted [see Figure 2(b)]. A port restricted cone only forwards packets coming from the transport address of the host that has been previously contacted [see Figure 3(c)]. For these first three cases, outgoing packets sent from the same NAT-ted transport address are mapped onto the same public-facing transport address regardless of the destination. A symmetric NAT behaves like a port restricted cone with the only exception that outgoing packets addressed to different hosts are mapped onto different transport addresses [see Figure 2(d)].

To determine the type of NAT, we made use of the *Netcat* tool[11] to send and receive UDP packets from/to a container, and the Netfilter's *conntrack* command to display the NAT table of the host machine on which the container is running. An example of NAT table is depicted in Figure 3. A new entry is created as soon as the kernel sees an incoming/outgoing packet that belongs to a new "connection." The left-hand side of the table reports the source and destination transport addresses of such packet, plus a *Validity* field that sets the expiration time of that entry. The validity is refreshed for each packet of the same connection that is detected. The right-hand side of the table, instead, is related to packets expected on the backward path (i.e., the so-called "expectations table"). Finally, the *State* column reports the state of the entry. For UDP flows, as soon as the entry is created (i.e., when the first packet is sent/received), the state is set to UNREPLIED and the validity to 15 s; the state



	PROTOCOL	VALIDITY [s]	SRC_IP	DST_IP	SRC_PORT	DST_PORT	SRC_IP	DST_IP	SRC_PORT	DST_PORT	STATE
1	UDP	12	172.17.0.2	66.102.1.127	10000	19302	66.102.1.127	198.51.100.1	19302	10000	ASSURED
2	UDP	23	172.17.0.2	203.0.113.1	10000	40000	203.0.113.1	198.51.100.1	40000	10000	ASSURED
3	UDP	145	172.17.0.2	66.102.1.127	20000	19302	66.102.1.127	198.51.100.1	19302	20000	ASSURED
4	UDP	28	203.0.113.2	198.51.100.1	40000	20000	198.51.100.1	203.0.113.2	20000	40000	UNREPLIED
5	UDP	29	172.17.0.2	203.0.113.2	20000	40000	198.51.100.1	203.0.113.2	20000	1024	UNREPLIED

Figure 3. Docker host NAT table.

changes to ASSURED when a packet is detected in the backward direction, and the validity is set to 180 s. The experiments we made showed that at a first glance the Docker bridge network driver seems to behave like a port restricted cone NAT, given that packets with the same source port and addressed to different hosts are mapped onto the same public-facing transport address (see Figure 3 rows 1 and 2). A more accurate analysis, though, revealed that there are cases where it acts as a Symmetric NAT instead, i.e., packets with the same source address and port addressed to different destinations are mapped to different external transport addresses. This is the case when an incoming packet addressed to a given port is received by the Docker host prior to detecting any outgoing connection having the same source port. Rows 3–5 in Figure 3 show how packets sent by the container from the same port number 20 000 have been mapped to different public ports because of the incoming packet previously detected and listed at row 4. This behavior is also depicted in Figure 3(d). As it will be clarified in "Real-World Example: Janus WebRTC Services," such behavior is very unfortunate and may easily cause problems when deploying containerized ICE-based services, as symmetric NATs are difficult to deal with.

REAL-WORLD EXAMPLE: JANUS WEBRTC SERVICES

Deploying services behind a NAT is not uncommon. For example, all cloud providers, like Amazon AWS or Microsoft Azure, allow people to easily create virtual machine (VM) instances that live in their datacenters. Such VMs are behind a NAT that can be configured to enable port forwarding.

As we saw in "Docker Networking Model", all microservices deployed within Docker containers by default also use the bridge network driver and, as such, end up being behind a NAT.

Deploying session-based services behind a NAT is usually not recommended. In fact, even though ICE has been conceived to cope with NAT traversal in the first place, set-up times usually take longer when there are NATs involved, and relay servers (i.e., TURN servers) may be needed to ensure connectivity in all network environments. However, as long as the NAT types involved are not very restrictive at both sides of the communication, ICE can still converge quickly.

As anticipated in "Docker NAT Functionality," the NAT functionality provided by the Docker bridge network can be assimilated to either a port restricted cone or symmetric behavior. Having a variable behavior which cannot be predicted beforehand is far from ideal. To provide a real-world example, we designed a few experiments to deploy a Janus.[12] WebRTC server instance in a Docker container. Before delving into the details of our experiments, we briefly introduce the WebRTC architecture.

WebRTC Architecture

WebRTC extends the classic web architecture semantics by introducing a peer-to-peer communication paradigm between browsers. The WebRTC architectural model draws its inspiration from the SIP architecture. Signaling messages are used to set up and terminate communications. They are transported by the HTTP or WebSocket protocol via the web server, which can modify, translate, or manage them as needed. It is worth noting that the signaling between browser and server is not standardized in WebRTC, as it is considered to be part of the application. As to the data path, WebRTC defines the *PeerConnection* abstraction which allows media to flow directly between browsers without any intervening servers.

A WebRTC web application is typically written as a mix of HTML and JavaScript. It interacts with web browsers through the standardized WebRTC API, as well as other standard APIs, allowing to properly exploit and control the real-time browser function, both proactively (e.g., to query browser capabilities) and reactively (e.g., to receive browser-generated notifications). The WebRTC API must hence provide a wide set of functions, like connection management (in a peer-to-peer fashion), encoding/decoding capabilities negotiation, selection and control, media control, firewall, and NAT element traversal. Session description represents an important piece of information that needs to be exchanged. It specifies the transport information, as well as the media type, format, and all

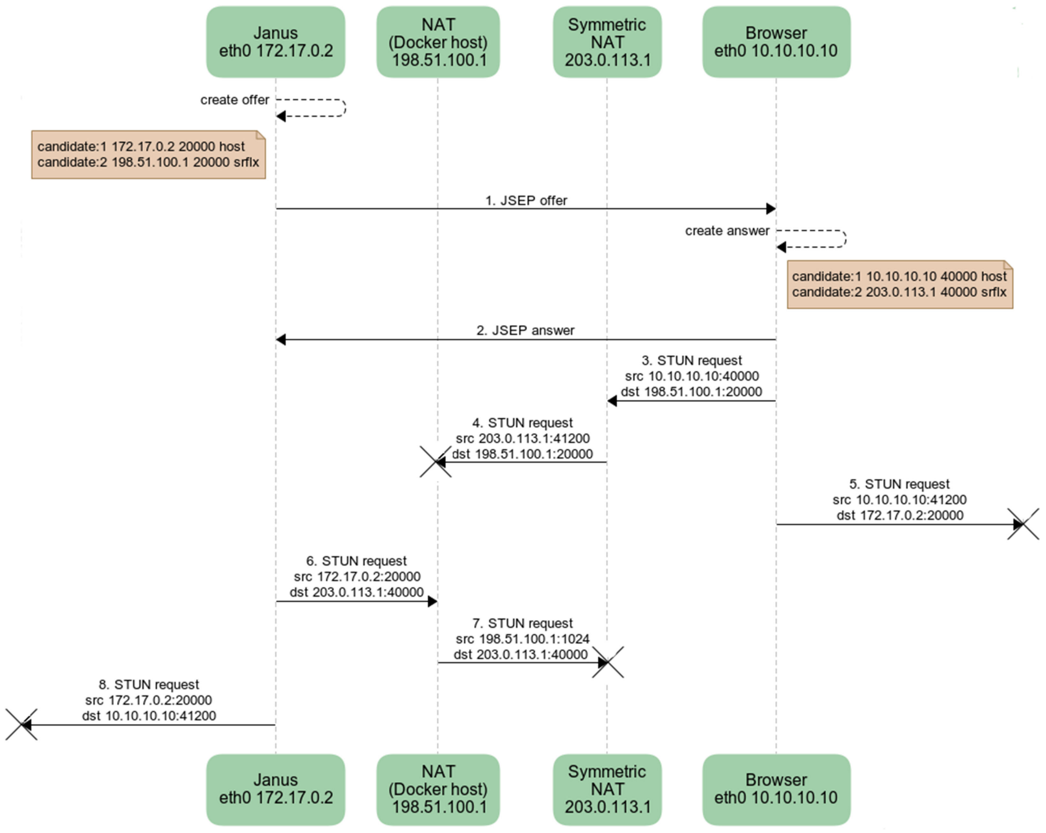

Janus eth0 172.17.0.2
NAT (Docker host) 198.51.100.1
Symmetric NAT 203.0.113.1
Browser eth0 10.10.10.10
create offer
candidate:1 172.17.0.2 20000 host
candidate:2 198.51.100.1 20000 srflx
1. JSEP offer
create answer
candidate:1 10.10.10.10 40000 host
candidate:2 203.0.113.1 40000 srflx
2. JSEP answer
3. STUN request src 10.10.10.10:40000 dst 198.51.100.1:20000
4. STUN request src 203.0.113.1:41200 dst 198.51.100.1:20000
5. STUN request src 10.10.10.10:41200 dst 172.17.0.2:20000
6. STUN request src 172.17.0.2:20000 dst 203.0.113.1:40000
7. STUN request src 198.51.100.1:1024 dst 203.0.113.1:40000
8. STUN request src 172.17.0.2:20000 dst 10.10.10.10:41200

Figure 4. ICE failure.

associated media configuration parameters needed to establish the media path. The Internet Engineering Task Force is now standardizing the JavaScript Session Establishment Protocol (JSEP).[13] JSEP provides the interface needed by an application to deal with the negotiated local and remote session descriptions (with the negotiation carried out through whatever signaling mechanism might be desired), together with a standardized way of interacting with the ICE state machine.

The JSEP approach delegates entirely to the application the responsibility for driving the signaling state machine: the application must call the right APIs at the right times and convert the session descriptions and related ICE information into the defined messages of its chosen signaling protocol.

It is worth mentioning that, even though WebRTC allows for direct browser-to-browser communication, more complex application scenarios like, e.g., conferencing and real-time group-based multimedia communication, definitely call for the introduction of functionality (e.g., mixing, transcoding, forwarding of the media involved in a group-shared session) that cannot be implemented at the end-systems in an effective way. In these cases, in-network (i.e., server-side) components (media servers, selective forwarding units, multipoint control units, etc.) come to the fore. Such components obviously need to adhere to the WebRTC standards in order to be capable of seamlessly interacting with WebRTC-enabled browsers on the end-user's side.

Testing Docker's NAT Functionality

WebRTC strongly depends on ICE for its operations and WebRTC applications represent an important use case.

On the server side of our testbed, we used the Docker bridge network and the Google public STUN service (stun.l.google.com:19302) for address discovery. We did not involve a TURN server in these experiments.

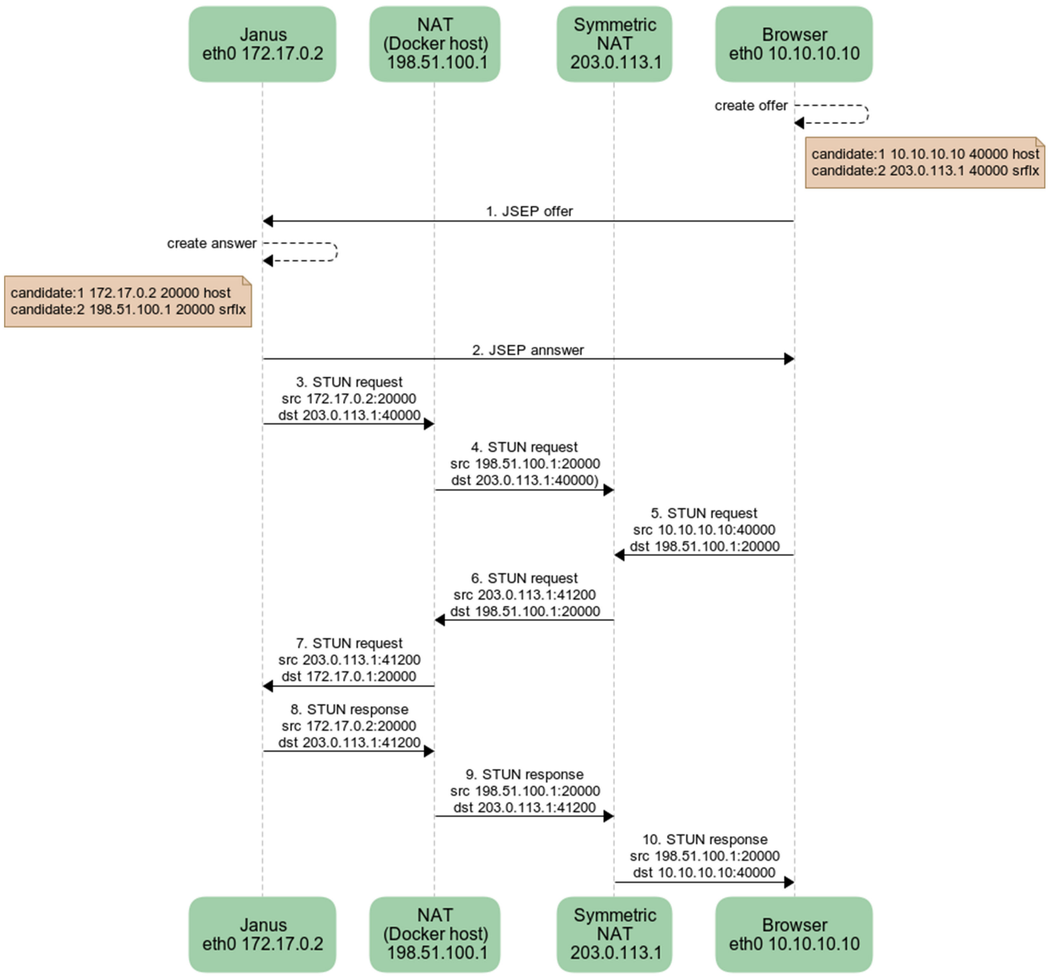


Figure 5. ICE success.

We leveraged the Janus *Streaming* plugin to set up a WebRTC *PeerConnection* between the browser and Janus in order to receive a media stream. The browser client was behind a symmetric NAT. This last hypothesis is not unrealistic. As an example, the Firefox browser, since version 57, has a built-in packet filter, which makes it behave like it were behind a Symmetric NAT. Google Chrome will likely provide the same feature in the near future.

In this scenario, Janus first gathers its ICE candidates and sends them over to the browser within a JSEP *offer* containing the SDP local session description. The browser then gathers its own ICE candidates and sends them to Janus within a JSEP *answer*. Right after that, the browser starts sending STUN check messages toward the candidates announced by Janus, which in turn also starts the connectivity check phase as soon as it receives the ICE candidates from the browser. This flow is depicted in Figure 4. The communication set up eventually fails because the STUN request sent by the browser through the Symmetric NAT (message #4) arrives at the Docker host before message #6. In this case, as explained in "Docker NAT Functionality," the Docker NAT uses a different public port than the one advertised by Janus in its SDP. If instead Janus had sent its connectivity check first, the communication would have been successful. This is more likely to happen when the browser sends the JSEP offer and Janus answers (e.g., with the Janus *EchoTest* plugin), as illustrated in Figure 5.

FINAL CONSIDERATIONS

In "Docker NAT Functionality," we analyzed the operation of the Docker bridge network driver, which implements NAT functionality, and demonstrated how it behaves either as a port restricted cone or a symmetric NAT, depending on the arrival timing of packets. This makes its usage for ICE-based applications not recommended, as it can easily make the connection set up fail as we saw in "A Real-World Example: Janus WebRTC Services." In order to effectively deploy ICE-based services within Docker containers, two solutions come to our mind. The former is to leverage the Docker *Host* network driver. Special attention should be paid in this case when deploying multiple containers on the same host, as port conflicts can easily occur. The latter is to assign a dedicated IP address to containers. This can be done by using the Macvlan network driver, as introduced in Section Network Driver, or by leveraging a tool called *Pipework,*[14] that we have been using for a long time and which proved to be very effective.

CONCLUSION

This article has analyzed in some detail how Docker currently implements NAT functionality. We have highlighted how containers get usually deployed on the host as independent networking nodes lying behind a host-provided NAT. Such a NAT typically shows either a portrestricted or a symmetric behavior. In the latter case, it severely hinders the correct functioning of the hosted containers, especially in those situations where such containers are actually providing some server-side functionality to remote third parties.

We took WebRTC-enabled distributed multimedia applications as an interesting use case and demonstrated how such applications are best supported by working around the need for address translation, namely through assigning publicly reachable independent IP addresses to the containers deployed on the host and offering server-side services.

When such an approach is not practicable (e.g., because of the lack of publicly assignable IP addresses), one alternative is to rely upon the Docker-provided *host* network driver, with the caveat that port conflicts should be avoided in case of the co-existence of multiple container instances on the same host. The only remaining alternative would be to let the server-side containers stay behind the Docker NAT and configure them in such a way as to leverage TURN servers deployed in the public Internet. This is obviously a solution that is far from optimal, since it highly depends on the presence of a relay node that actually acts "on behalf" of the container and then forwards to it (back and forth) the traffic that it is managing.

REFERENCES

1. S. Loreto and S. P. Romano, "How far are we from WebRTC-1.0? An update on standards and a look at what's next," *IEEE Commun. Mag.*, vol. 23, no. 60, pp. 200–207, Jul. 2017.
2. A. Keranen, C. Holmberg, and J. Rosenberg, "Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal," RFC8445, Jul. 2018.
3. J. Rosenberg *et al.*, "SIP: Session initiation protocol," RFC3261, Jun. 2002.
4. J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN—simple traversal of user datagram protocol (UDP) through network address translators (NATs)," RFC3489, Mar. 2003.
5. R. Mahy, P. Matthews, and J. Rosenberg, "Traversal using relays around NAT (TURN): Relay extensions to session traversal utilities for NAT (STUN)," RFC 5766, Apr. 2010.
6. M. Handley, V. Jacobson, and C. Perkins, SDP: Session Description Protocol RFC4566, Jul. 2006.
7. E. Ivov, E. Rescorla, J. Uberti, and P. Saint-Andre, "Trickle ICE: incremental provisioning of candidates for the interactive connectivity establishment (ICE) protocol ddraft-ietf-ice-trickle-21.txt," Apr. 2018.
8. 2015. [Online]. Available: https://github.com/docker/libnetwork
9. 1999. [Online]. Available: https://netfilter.org/
10. F. Audet and C. Jennings, "Network address translation (NAT) behavioral requirements for unicast UDP," RFC4787, Jan. 2007.
11. 1995. [Online]. Available: https://sectools.org/tool/netcat/
12. A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Janus, a general purpose WebRTC gateway," in *Proc. Conf. Principles, Syst. Appl. IP Telecommun.*, 2014, Art. no. 7.
13. J. Uberti and C. Jennings, E. Rescorla JavaScript Session Establishment Protocol draft-ietf-rtcweb-jsep-25, Oct. 2018.
14. [Online]. Available: https://github.com/jpetazzo/pipework/

Alessandro Amirante is currently CTO at Meetecho s.r.l. His research interests include the field of networking and systems, with special focus on next generation network architectures, multimedia services over the Internet, virtualization techniques, and software containers. He actively participates in Internet Engineering Task Force standardization activities, mainly in the Applications and Real Time area. He received the M. Sc. degree in telecommunications engineering in 2007 and the Ph.D. degree in computer engineering and systems from the University of Napoli "Federico II," Italy, in 2010. He is also a member of the IETF Network Operations Center team. With the rest of the Meetecho team, he provides remote participation services for all of the IETF meetings. Contact him at alex@meetecho.com.

Simon Pietro Romano is an associate professor with the Department of Electrical Engineering and Information Technology, University of Napoli Federico II. He teaches computer networks, computer architectures, network security, and telematics applications. He is also the co-founder of Meetecho, a startup and university spin-off dealing with scalable video streaming and WebRTC-based unified collaboration, as well as of SECurity Solutions for Innovation (SECSI), that focuses on network security. He actively participates in Internet Engineering Task Force (IETF) standardization activities, mainly in the Applications and Real Time area. He has conducted research activities in the field of networking since 1998. He has been working on the definition of advanced networking architectures capable to provide end-users with optimized quality of experience, thanks to a closed-loop approach going from SLA-based management down to policy-based configuration of the devices and autonomic monitoring. He has contributed to the field of real-time multimedia applications, with special regard to the design, implementation, and standardization of scalable, distributed architectures for conferencing, media control, and telepresence. He has carried out research in the field of network security, mainly focusing on distributed intrusion detection and critical infrastructure protection. Finally, he has started investigating issues associated with distribution of control in 5G-compliant, container-based, virtualized architectures. He has a long track of participations in R&D projects, both at the national and at the international level. He is currently leading the Secure Hybrid in Network caching Environment project funded by the European Space Agency. Contact him at spromano@unina.it.