Building Blocks of UDP

NETWORKING 101, CHAPTER 3

User Datagram Protocol, or UDP, was added to the core network protocol suite in August of 1980 by Jon Postel, well after the original introduction of TCP/IP, but right at the time when the TCP and IP specifications were being split to become two separate RFCs. This timing is important because, as we will see, the primary feature and appeal of UDP is not in what it introduces, but rather in all the features it chooses to omit. UDP is colloquially referred to as a *null protocol*, and RFC 768, which describes its operation, could indeed fit on a napkin.

Datagram

> A self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination nodes without reliance on earlier exchanges between the nodes and the transporting network.

The words datagram and packet are often used interchangeably, but there are some nuances. While the term "packet" applies to any formatted block of data, the term "datagram" is often reserved for packets delivered via an unreliable service—no delivery guarantees, no failure notifications. Because of this, you will frequently find the more descriptive term "Unreliable" substituted for the official term "User" in the UDP acronym, to form "Unreliable Datagram Protocol." That is also why UDP packets are generally, and more correctly, referred to as datagrams.

Perhaps the most well-known use of UDP, and one that every browser and Internet application depends on, is the Domain Name System (DNS): given a human-friendly computer hostname, we need to discover its IP address before any data exchange can occur. However, even though the browser itself is dependent on UDP, historically the protocol has never been exposed as a first-class transport for pages and applications running within it. That is, until WebRTC entered into the picture.

The new Web Real-Time Communication (WebRTC) standards, jointly developed by the IETF and W3C working groups, are enabling real-time communication, such as voice and video calling and other forms of peer-to-peer (P2P) communication, natively within the browser via UDP. With WebRTC, UDP is now a first-class browser transport with a client-side API! We will investigate WebRTC in-depth in WebRTC, but before we get there, let's first explore the inner workings of the UDP protocol to understand why and where we may want to use it.

Null Protocol Services §

To understand UDP and why it is commonly referred to as a "null protocol," we first need to look at the Internet Protocol (IP), which is located one layer below both TCP and UDP protocols.

The IP layer has the primary task of delivering datagrams from the source to the destination host based on their addresses. To do so, the messages are encapsulated within an IP packet (Figure 3-1) which identifies the source and the destination addresses, as well as a number of other routing parameters .

Once again, the word "datagram" is an important distinction: the IP layer provides no guarantees about message delivery or notifications of failure and hence directly exposes the unreliability of the underlying network to the layers above it. If a routing node along the way drops the IP packet due to congestion, high load, or for other reasons, then it is the responsibility of a protocol above IP to detect it, recover, and retransmit the data—that is, if that is the desired behavior!

<table>
<tr><th>Bit</th><th colspan="2">+0..7</th><th colspan="2">+8..15</th><th colspan="2">+16..23</th><th>+24..31</th></tr>
<tr><td>0</td><td>Version</td><td>Header Length</td><td>DSCP</td><td>ECN</td><td colspan="3">Total Length</td></tr>
<tr><td>32</td><td colspan="4">Identification</td><td>Flags</td><td colspan="2">Fragment Offset</td></tr>
<tr><td>64</td><td>Time To Live</td><td colspan="2">Protocol</td><td colspan="3">Header Checksum</td></tr>
<tr><td>96</td><td colspan="6">Source IP Address</td></tr>
<tr><td>128</td><td colspan="6">Destination IP Address</td></tr>
<tr><td>160</td><td colspan="6">Options (if present)</td></tr>
<tr><td>...</td><td colspan="6">Payload</td></tr>
</table>

Figure 3-1. IPv4 header (20 bytes)

The UDP protocol encapsulates user messages into its own packet structure (Figure 3-2), which adds only four additional fields: source port, destination port, length of packet, and checksum. Thus, when IP delivers the packet to the destination host, the host is able to unwrap the UDP packet, identify the target application by the destination port, and deliver the message. Nothing more, nothing less.

Bit	+0..7	+8..15	+16..23	+24..31
0	Source Port		Destination Port	
32	Length		Checksum	
...	*Payload*			

Figure 3-2. UDP header (8 bytes)

In fact, both the source port and the checksum fields are optional fields in UDP datagrams. The IP packet contains its own header checksum, and the application can choose to omit the UDP checksum, which means that all the error detection and error correction can be delegated to the applications above them. At its core, UDP simply provides "application multiplexing" on top of IP by embedding the source and the target application ports of the communicating hosts. With that in mind, we can now summarize all the UDP non-services:

No guarantee of message delivery
: No acknowledgments, retransmissions, or timeouts

No guarantee of order of delivery
: No packet sequence numbers, no reordering, no head-of-line blocking

No connection state tracking
: No connection establishment or teardown state machines

No congestion control
: No built-in client or network feedback mechanisms

TCP is a byte-stream oriented protocol capable of transmitting application messages spread across multiple packets without any explicit message boundaries within the packets themselves. To achieve this, connection state is allocated on both ends of the connection, and each packet is sequenced, retransmitted when lost, and delivered in order. UDP datagrams, on the other hand, have definitive boundaries: each datagram is carried in a single IP packet, and each application read yields the full message; datagrams cannot be fragmented.

UDP is a simple, stateless protocol, suitable for bootstrapping other application protocols on top: virtually all of the protocol design decisions are left to the application above it. However, before you run away to implement your own protocol to replace TCP, you should think carefully about complications such as UDP interaction with the many layers of deployed middleboxes (NAT traversal), as well as general network protocol design best practices. Without careful engineering and planning, it is not uncommon to start with a bright idea for a new protocol but end up with a poorly implemented version of TCP. The algorithms and the state machines in TCP have been

honed and improved over decades and have taken into account dozens of mechanisms that are anything but easy to replicate well.

UDP and Network Address Translators §

Unfortunately, IPv4 addresses are only 32 bits long, which provides a maximum of 4.29 billion unique IP addresses. The IP Network Address Translator (NAT) specification was introduced in mid-1994 (RFC 1631) as an interim solution to resolve the looming IPv4 address depletion problem—as the number of hosts on the Internet began to grow exponentially in the early '90s, we could not expect to allocate a unique IP to every host.

The proposed IP reuse solution was to introduce NAT devices at the edge of the network, each of which would be responsible for maintaining a table mapping of local IP and port tuples to one or more globally unique (public) IP and port tuples (Figure 3-3). The local IP address space behind the translator could then be reused among many different networks, thus solving the address depletion problem.

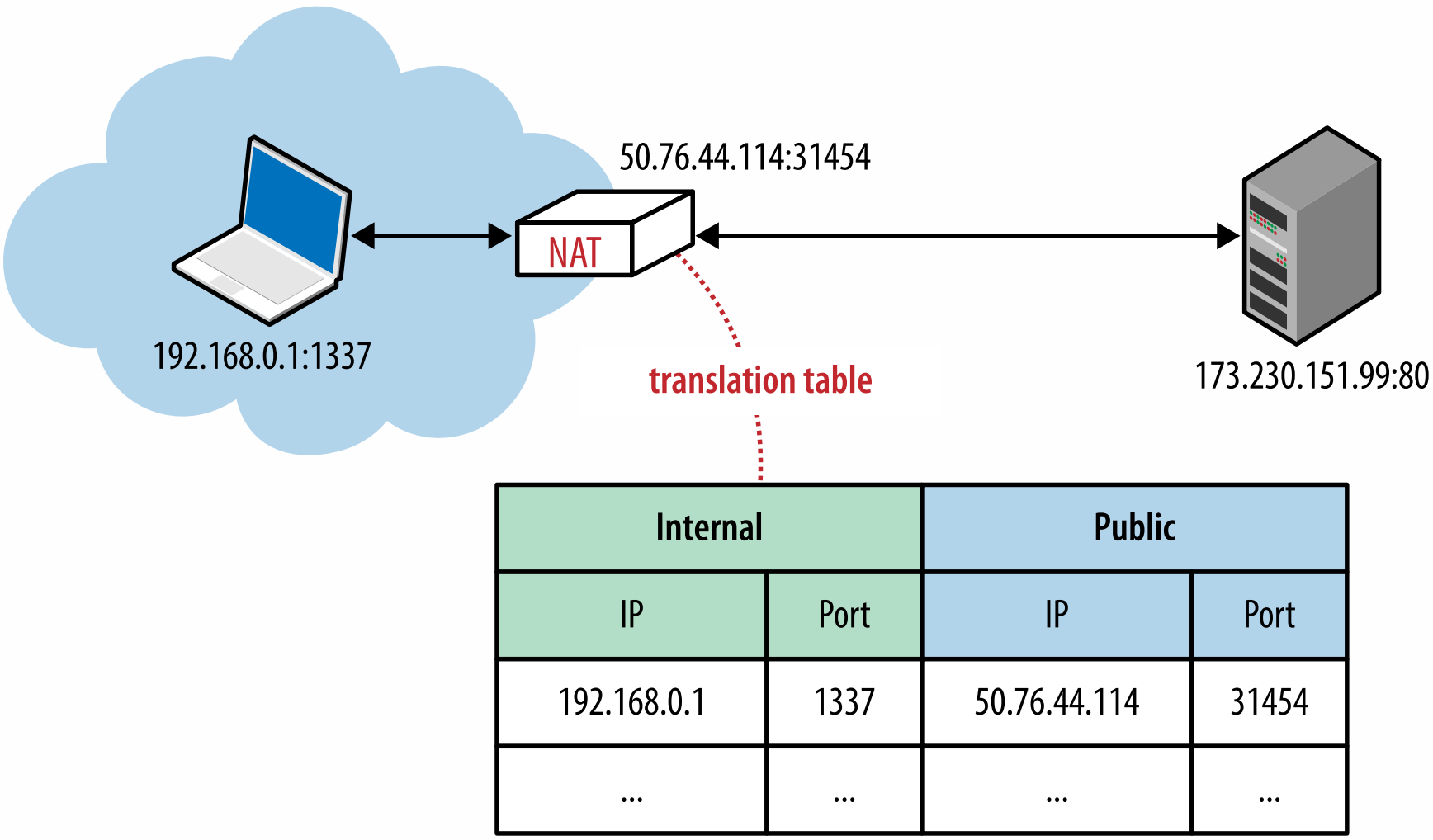


Figure 3-3. IP Network Address Translator

Unfortunately, as it often happens, there is nothing more permanent than a temporary solution. Not only did the NAT devices resolve the immediate problem, but they also quickly became a

ubiquitous component of many corporate and home proxies and routers, security appliances, firewalls, and dozens of other hardware and software devices. NAT middleboxes are no longer a temporary solution; rather, they have become an ossified part of the Internet infrastructure.

Reserved Private Network Ranges §

Internet Assigned Numbers Authority (IANA), which is an entity that oversees global IP address allocation, reserved three well-known ranges (Table 3-1) for private networks, most often residing behind a NAT device.

IP address range	Number of addresses
10.0.0.0–10.255.255.255	16,777,216
172.16.0.0–172.31.255.255	1,048,576
192.168.0.0–192.168.255.255	65,536

Table 3-1. Reserved IP ranges

One or all of the preceding ranges should look familiar. Chances are, your local router has assigned your computer an IP address from one of those ranges. That's your private IP address on the internal network, which is then translated by the NAT device when communicating with an outside network.

To avoid routing errors and confusion, no public computer is allowed to be assigned an IP address from any of these reserved private network ranges.

Connection-State Timeouts §

The issue with NAT translation, at least as far as UDP is concerned, is precisely the routing table that it must maintain to deliver the datagrams. NAT middleboxes rely on connection state, whereas UDP has none. This is a fundamental mismatch and a source of many problems for delivering UDP datagrams. Further, it is now not uncommon for a client to be behind many layers of NATs, which only complicates matters further.

Each TCP connection has a well-defined protocol state machine, which begins with a handshake, followed by application data transfer, and a well-defined exchange to close the connection. Given this flow, each middlebox can observe the state of the connection and create and remove the routing entries as needed. With UDP, there is no handshake or connection termination, and hence

there is no connection state machine to monitor.

Delivering outbound UDP traffic does not require any extra work, but routing a reply requires that we have an entry in the translation table, which will tell us the IP and port of the local destination host. Thus, translators have to keep state about each UDP flow, which itself is stateless.

Even worse, the translator is also tasked with figuring out when to drop the translation record, but since UDP has no connection termination sequence, either peer could just stop transmitting datagrams at any point without notice. To address this, UDP routing records are expired on a timer. How often? There is no definitive answer; instead the timeout depends on the vendor, make, version, and configuration of the translator. Consequently, one of the de facto best practices for long-running sessions over UDP is to introduce bidirectional keepalive packets to periodically reset the timers for the translation records in all the NAT devices along the path.

TCP Timeouts and NATs §

Technically, there is no need for additional TCP timeouts on NAT devices. The TCP protocol follows a well-defined handshake and termination sequence, which signals when the appropriate translation records can be added and removed.

Unfortunately, in practice, many NAT devices apply similar timeout logic both to TCP and UDP sessions. As a result, in some cases bidirectional keepalive packets are also required for TCP. If your TCP connections are getting dropped, then there is a good chance that an intermediate NAT timeout is to blame.

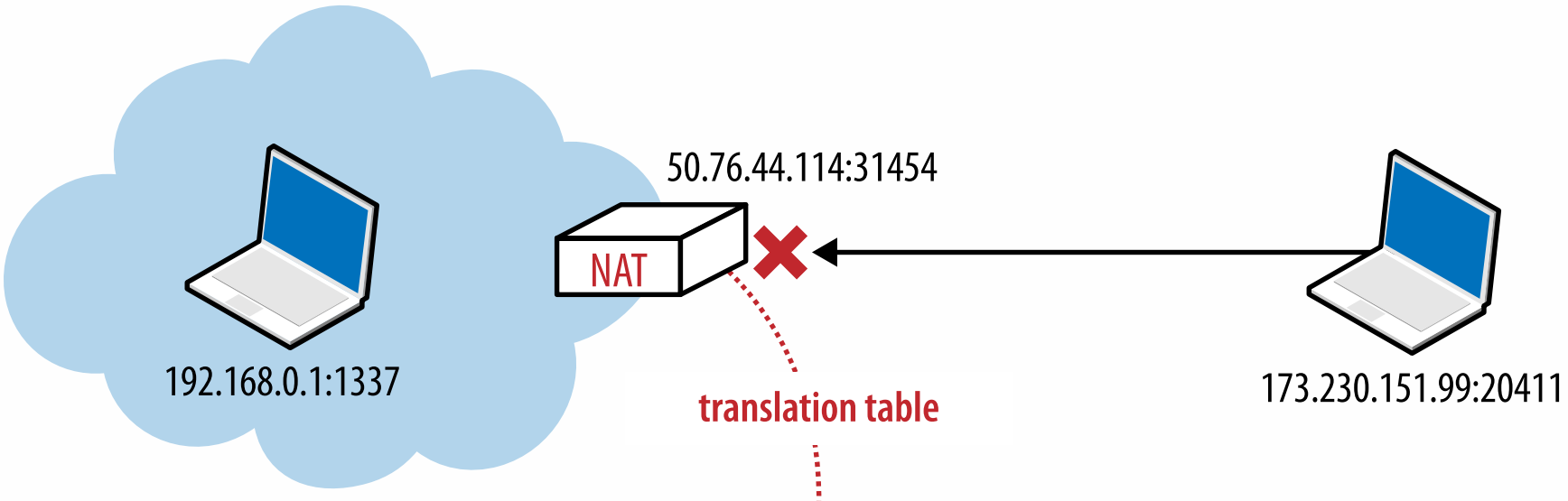
NAT Traversal §

Unpredictable connection state handling is a serious issue created by NATs, but an even larger problem for many applications is the inability to establish a UDP connection at all. This is especially true for P2P applications, such as VoIP, games, and file sharing, which often need to act as both client and server to enable two-way direct communication between the peers.

The first issue is that in the presence of a NAT, the internal client is unaware of its public IP: it knows its internal IP address, and the NAT devices perform the rewriting of the source port and address in every UDP packet, as well as the originating IP address within the IP packet. However, if the client communicates its private IP address as part of its application data with a peer outside of its private network, then the connection will inevitably fail. Hence, the promise of "transparent" translation is no longer true, and the application must first discover its public IP

address if it needs to share it with a peer outside its private network.

However, knowing the public IP is also not sufficient to successfully transmit with UDP. Any packet that arrives at the public IP of a NAT device must also have a destination port and an entry in the NAT table that can translate it to an internal destination host IP and port tuple. If this entry does not exist, which is the most likely case if someone simply tries to transmit data from the public network, then the packet is simply dropped (Figure 3-4). The NAT device acts as a simple packet filter since it has no way to automatically determine the internal route, unless explicitly configured by the user through a port-forwarding or similar mechanism.



Internal		Public	
IP	Port	IP	Port
?	?	50.76.44.114	31454

Figure 3-4. Dropped inbound packet due to missing mapping

It is important to note that the preceding behavior is not an issue for client applications, which begin their interaction from the internal network and in the process establish the necessary translation records along the path. However, handling inbound connections (acting as a server) from P2P applications such as VoIP, game consoles, file sharing, and so on, in the presence of a NAT, is where we will immediately run into this problem.

To work around this mismatch in UDP and NATs, various traversal techniques (TURN, STUN, ICE) have to be used to establish end-to-end connectivity between the UDP peers on both sides.

STUN, TURN, and ICE §

Session Traversal Utilities for NAT (STUN) is a protocol (RFC 5389) that allows the host application to discover the presence of a network address translator on the network, and when present to obtain the allocated public IP and port tuple for the current connection (Figure 3-5). To do so, the protocol requires assistance from a well-known, third-party STUN server that must reside on the public network.

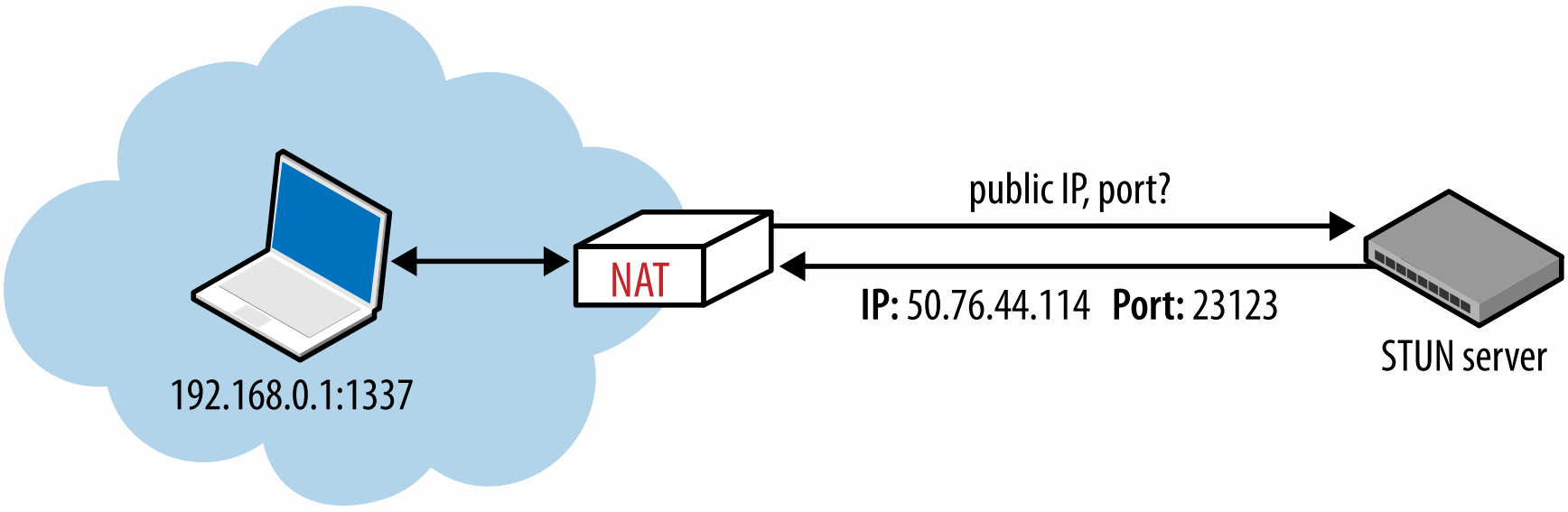


Figure 3-5. STUN query for public IP and port

Assuming the IP address of the STUN server is known (through DNS discovery, or through a manually specified address), the application first sends a binding request to the STUN server. In turn, the STUN server replies with a response that contains the public IP address and port of the client as seen from the public network. This simple workflow addresses several problems we encountered in our earlier discussion:

- The application discovers its public IP and port tuple and is then able to use this information as part of its application data when communicating with its peers.
- The outbound binding request to the STUN server establishes NAT routing entries along the path, such that the inbound packets arriving at the public IP and port tuple can now find their way back to the host application on the internal network.
- The STUN protocol defines a simple mechanism for keepalive pings to keep the NAT routing entries from timing out.

With this mechanism in place, whenever two peers want to talk to each other over UDP, they will first send binding requests to their respective STUN servers, and following a successful response on both sides, they can then use the established public IP and port tuples to exchange data.

However, in practice, STUN is not sufficient to deal with all NAT topologies and network configurations. Further, unfortunately, in some cases UDP may be blocked altogether by a firewall or some other network appliance—not an uncommon scenario for many enterprise networks. To

address this issue, whenever STUN fails, we can use the Traversal Using Relays around NAT (TURN) protocol (RFC 5766) as a fallback, which can run over UDP and switch to TCP if all else fails.

The key word in TURN is, of course, "relays." The protocol relies on the presence and availability of a public relay (Figure 3-6) to shuttle the data between the peers.

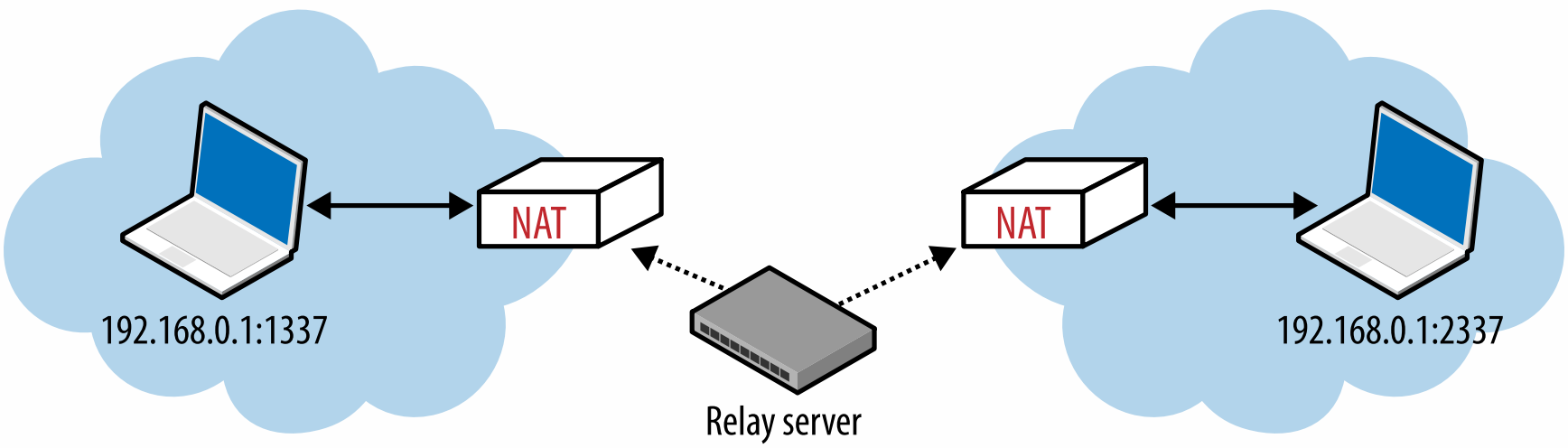


Figure 3-6. TURN relay server

- Both clients begin their connections by sending an allocate request to the same TURN server, followed by permissions negotiation.
- Once the negotiation is complete, both peers communicate by sending their data to the TURN server, which then relays it to the other peer.

Of course, the obvious downside in this exchange is that it is no longer peer-to-peer! TURN is the most reliable way to provide connectivity between any two peers on any networks, but it carries a very high cost of operating the TURN server—at the very least, the relay must have enough capacity to service all the data flows. As a result, TURN is best used as a last resort fallback for cases where direct connectivity fails.

STUN and TURN in Practice §

Google's libjingle is an open-source C++ library for building peer-to-peer applications, which takes care of STUN, TURN, and ICE negotiations under the hood. The library is used to power the Google Talk chat application, and the documentation provides a valuable reference point for performance of STUN vs. TURN in the real world:

- 92% of the time the connection can take place directly (STUN).
- 8% of the time the connection requires a relay (TURN).

Unfortunately, even with STUN, a significant fraction of users are unable to establish a direct P2P tunnel. To provide a reliable service, we also need TURN relays, which can acts as a fallback for cases where

direct P2P communication is not an option.

Building an effective NAT traversal solution is not for the faint of heart. Thankfully, we can lean on Interactive Connectivity Establishment (ICE) protocol (RFC 5245) to help with this task. ICE is a protocol, and a set of methods, that seek to establish the most efficient tunnel between the participants (Figure 3-7): direct connection where possible, leveraging STUN negotiation where needed, and finally fallback to TURN if all else fails.

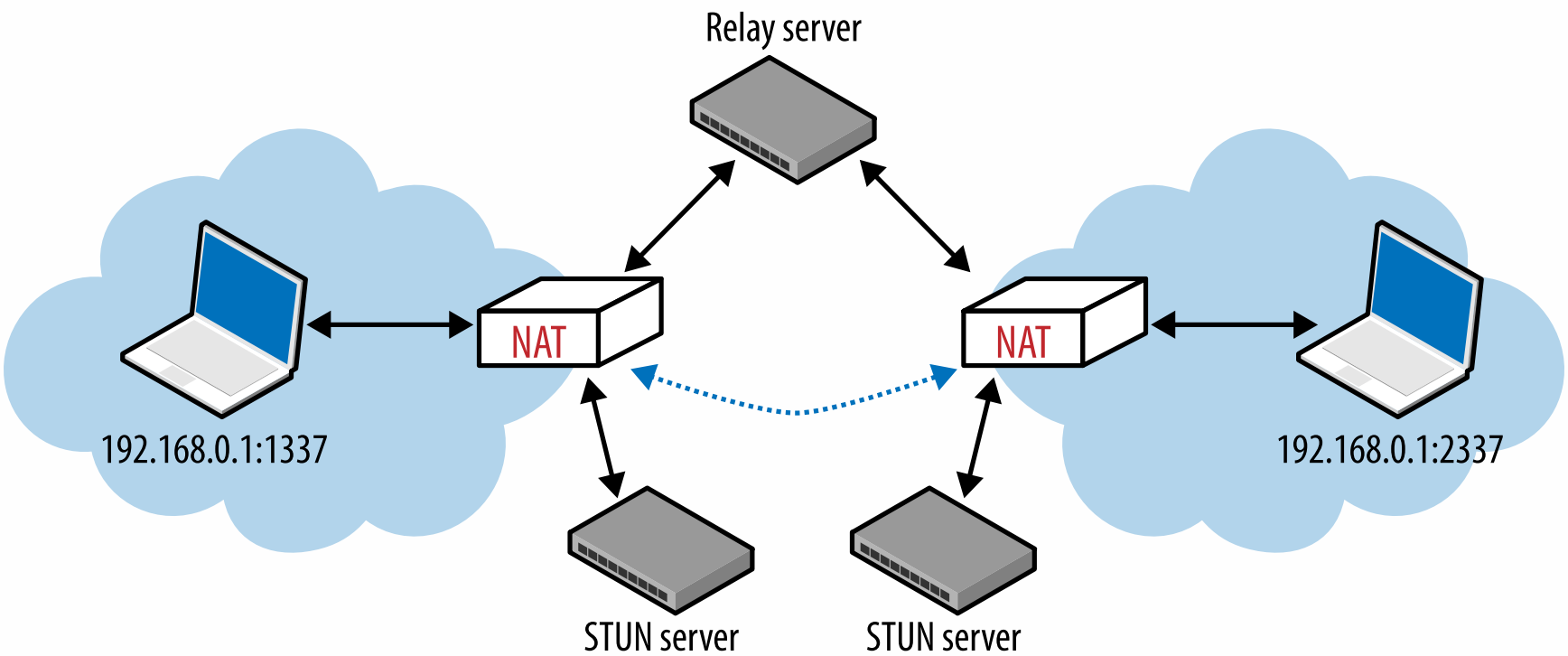


Figure 3-7. ICE attempts direct, STUN, and TURN connectivity options

In practice, if you are building a P2P application over UDP, then you most definitely want to leverage an existing platform API, or a third-party library that implements ICE, STUN, and TURN for you. And now that you are familiar with what each of these protocols does, you can navigate your way through the required setup and configuration!

Optimizing for UDP §

UDP is a simple and a commonly used protocol for bootstrapping new transport protocols. In fact, the primary feature of UDP is all the features it omits: no connection state, handshakes, retransmissions, reassembly, reordering, congestion control, congestion avoidance, flow control, or even optional error checking. However, the flexibility that this minimal message-oriented transport layer affords is also a liability for the implementer. Your application will likely have to reimplement some, or many, of these features from scratch, and each must be designed to play well with other peers and protocols on the network.

Unlike TCP, which ships with built-in flow and congestion control and congestion avoidance, UDP

applications must implement these mechanisms on their own. Congestion insensitive UDP applications can easily overwhelm the network, which can lead to degraded network performance and, in severe cases, to network congestion collapse.

If you want to leverage UDP for your own application, make sure to research and read the current best practices and recommendations. One such document is the RFC 5405, which specifically focuses on design guidelines for applications delivered via unicast UDP. Here is a short sample of the recommendations:

- Application *must* tolerate a wide range of Internet path conditions.
- Application *should* control rate of transmission.
- Application *should* perform congestion control over all traffic.
- Application *should* use bandwidth similar to TCP.
- Application *should* back off retransmission counters following loss.
- Application *should not* send datagrams that exceed path MTU.
- Application *should* handle datagram loss, duplication, and reordering.
- Application *should* be robust to delivery delays up to 2 minutes.
- Application *should* enable IPv4 UDP checksum, and *must* enable IPv6 checksum.
- Application *may* use keepalives when needed (minimum interval 15 seconds).

Designing a new transport protocol requires a lot of careful thought, planning, and research—do your due diligence. Where possible, leverage an existing library or a framework that has already taken into account NAT traversal, and is able to establish some degree of fairness with other sources of concurrent network traffic.

On that note, good news: WebRTC (WebRTC) is just such a framework!

« *Back to the Table of Contents*