Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

Secure Sessions for Ad Hoc Multiparty Computation in MPyC

Master's thesis

Emil Nikolov

Id nr: 0972305
emil.e.nikolov@gmail.com

Supervisor : Dr. ir. L.A.M. (Berry) Schoenmakers

March 11, 2024

Contents

Todo list

Glossary

E^3 Extensible Evaluation Environment 27

ACL Access Control List 20

AOT Ahead-Of-Time 38, 41, 44, 47

API Application Programming Interface 20

CA Certificate Authority 10

CGNAT Carrier-Grade NAT 15

CIDR Classless Inter-Domain Routing vii, 12

DERP Designated Encrypted Relay for Packets 14, 20

DNS Domain Name System 9

DTLS Datagram Transport Layer Security 10

ECDH Eliptic Curve Diffie-Hellman 12

FTP File Transfer Protocol 7

HTML HyperText Markup Language 7

HTTP HyperText Transfer Protocol 9, 10

HTTPS HyperText Transfer Protocol Secure 10

ICE Interactive Connectivity Establishment 15

IP Internet Protocol 8, 11, 19, 23

IPSec Internet Protocol Security vii, 11

ISP Internet Service Provider 7, 11, 15

JIT Just-In-Time 38, 41, 44, 47

JS JavaScript 37, 41, 43, 47

LAN Local Area Network 8, 19

MAC Media Access Control 23

MPC Secure Multiparty Computation 37, 43

NAT Network Address Translation 13

NAT-PMP NAT Port Mapping Protocol 14

P2P Peer to Peer 19, 20

PCP Port Control Protocol 14

PDU Protocol Data Unit 7

PKI Public Key Infrastructure 10

PSK Pre-Shared Key 11

QUIC Quick UDP Internet Connections 9, 10

SMTP Simple Mail Transfer Protocol 7

SSH Secure Shell Protocol 7, 11

SSI Self-Sovereign Identity 23

SSL Secure Sockets Layer 10

STUN Session Traversal Utilities for NAT 14

TAP Network Tap 19

TCP Transmission Control Protocol 9, 10

TLS Transport Layer Security 9, 10, 11, 17

TUN Network Tunnel 19

TURN Traversal Using Relays around NAT 14

UDP User Datagram Protocol 8, 10, 14, 17

UPnP Universal Plug and Play 14

URL Universal Resource Locator 9

VPN Virtual Private Network vii, 10, 11, 12, 17, 19, 20

WASM WebAssembly 38, 41, 44, 47

WebRTC Web Real-Time Communication 15

WWW World Wide Web 7

List of Figures

Chapter 1

Introduction

Part I

The State of Multiparty Communications over the Internet

include a section on MPC?

Chapter 2

Internet Communications

This chapter provides important background information on secure Internet communications, the challenges to multiparty communications and the approaches to overcome them.

Is there a more fitting section name than "The Internet"

The Internet is a global network that consists of numerous interconnected computer networks spanning billions of host devices owned by diverse parties from around the world. Key components of the Internet include the Internet Protocol Suite (known as TCP/IP) and the physical infrastructure that connects the individual networks. Sections of the infrastructure are deployed and managed by different tiers of Internet Service Providers (**ISPs**) who also maintain links between each other. To ensure efficient utilization of the hardware, the Internet relies on packet-switching techniques that divide the data traffic into smaller individually processed **packets**.

Communication protocols are usually organized into abstraction layers based on the scope of their functionality with higher-layer protocols relying on the services provided by the lower-layer protocols. Several reference models define different layering schemes, for example, the OSI model recognizes 7 layers, while TCP/IP itself combines some of the layers and recognizes 4. Figure 2.1 shows how the two models relate to each other and describes the responsibilities of the various layers. Throughout this thesis, we will refer to the 7 layer numbers of the OSI model as they are more widely used in the literature.

The packets that are transmitted over the network contain information from multiple protocols organized in Protocol Data Units (**PDUs**) with the lower-layer PDUs **encapsulating** the higher-layer PDUs as a (usually opaque) **payload**. Additionally, the PDU of a protocol contains control information for the protocol itself in the form of a **header** or **footer**.

Services that are implemented as Application layer (L7) protocols on top of TCP/IP include the World Wide Web (**WWW**), file transfer (FTP), email (SMTP), instant messaging, remote access (SSH [LY06]) and others. The Web is a collection of interconnected documents that use Web technologies such as HTML and JavaScript. It is typically accessed via a user-agent software such as a **Web Browser**.

The following sub-sections will briefly cover the main protocols of the Internet Protocol Suite, the issues for multiparty communications and some of the low-level mitigation techniques.

add a paragraph that describes what the rest of the section will contain

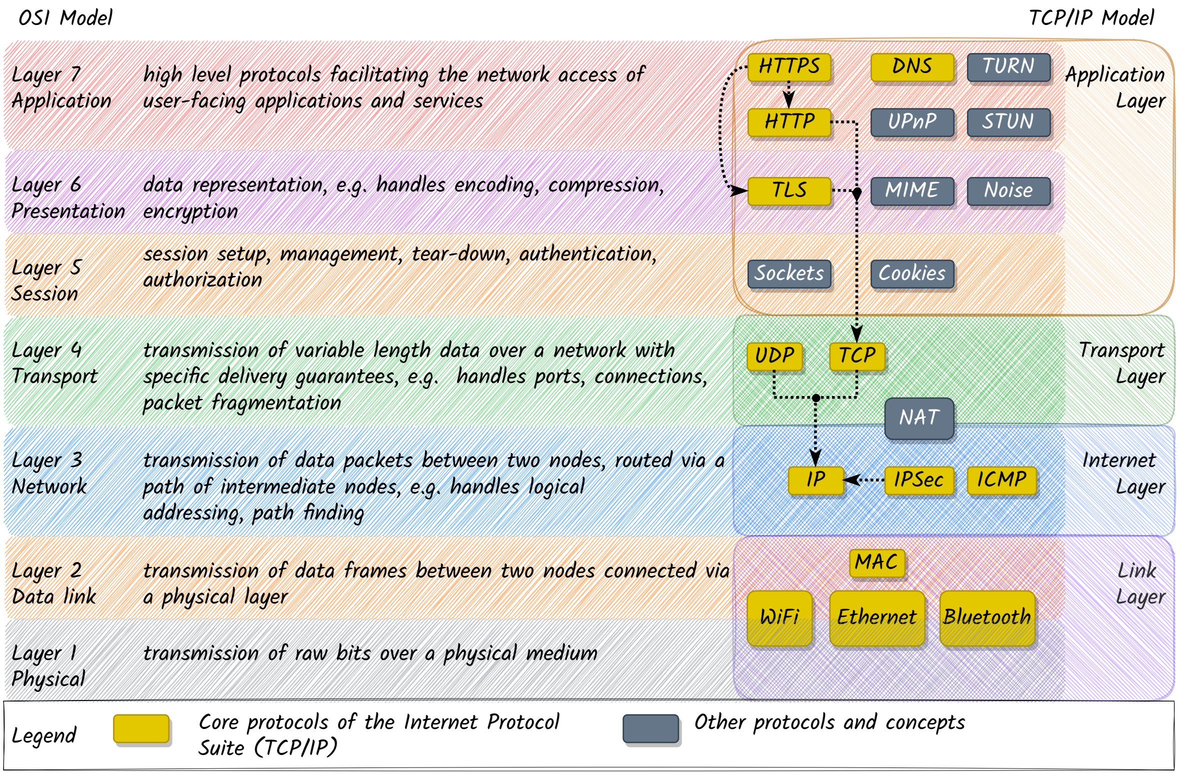
OSI Model
TCP/IP Model
Layer 7
Application
high level protocols facilitating the network access of user-facing applications and services
Layer 6
Presentation
data representation, e.g. handles encoding, compression, encryption
Layer 5
Session
session setup, management, tear-down, authentication, authorization
Layer 4
Transport
transmission of variable length data over a network with specific delivery guarantees, e.g. handles ports, connections, packet fragmentation
Layer 3
Network
transmission of data packets between two nodes, routed via a path of intermediate nodes, e.g. handles logical addressing, path finding
Layer 2
Data link
transmission of data frames between two nodes connected via a physical layer
Layer 1
Physical
transmission of raw bits over a physical medium
HTTPS
DNS
TURN
HTTP
UPnP
STUN
TLS
MIME
Noise
Sockets
Cookies
UDP
TCP
NAT
IP
IPSec
ICMP
MAC
WiFi
Ethernet
Bluetooth
Application Layer
Transport Layer
Internet Layer
Link Layer
Legend
Core protocols of the Internet Protocol Suite (TCP/IP)
Other protocols and concepts

Figure 2.1: OSI model mapping of the Internet Protocol Suite

2.1 Communication Protocols

2.1.1 Network Layer (L3)

The IP [81] is a Network layer (L3) protocol of the Internet Protocol Suite that is responsible for transferring datagrams between devices across the boundaries of their Local Area Networks (**LANs**) by possibly routing them via multiple intermediate devices (e.g. routers). A datagram is a self-contained unit of data, typically associated with connectionless protocols that provide no guarantees for delivery or ordering (e.g. IP, UDP). IP datagrams have a header that contains fields such as the **IP addresses** of its source and destination, and a payload that encapsulates the data from the Transport Layer (L4) protocols. A **router** is a device that is part of multiple networks and relays datagrams between them based on a routing table that maps IP address ranges to networks.

packet is the physical envelope of the IP datagram

show a diagram of the internet with multiple local networks

2.1.2 Transport Layer (L4)

User Datagram Protocol (**UDP**) is a very thin Transport layer (L4) protocol that only provides port multiplexing and checksumming on top of IP. - Port multiplexing - uses 16-bit numbers to allow multiple processes behind the same IP address to establish their own communication channels - Checksumming - used to detect errors in the datagram header and payload

As with IP, UDP packets are referred to as datagrams because they are not delivered reliably and if such features are required, they must be implemented by the consumer of the protocol.

Transmission Control Protocol (**TCP**) is another Transport layer (L4) protocol. Like UDP, it provides port multiplexing and checksumming, but it offers stronger delivery guarantees. Some of the features it offers are listed below:

- Connection management - TCP establishes reliable connections between the communicating hosts and can gracefully terminate them when required
- Segmentation - TCP splits variable-length data streams into segments that fit inside IP datagrams and transmits them individually
- ordering - segments have sequence numbers to ensure that they are reassembled in the correct order at the receiving host
- Error detection and correction - TCP retransmits a segment if its checksum fails

Both TCP and UDP are useful in different scenarios. UDP is faster and is used for applications that can tolerate packet loss, e.g. video streaming, VoIP, or in cases where it is preferable for an application to implement its own reliable delivery. TCP has a higher overhead than UDP but its reliable delivery is a good default for most applications on the Internet.

Quick UDP Internet Connections (**QUIC**) is a more recent Transport layer (L4) protocol that is built on top of UDP and offers similar features to TCP, but with lower latency in some scenarios. It provides an additional level of multiplexing within a process, which allows multiple streams of data to be sent over the same connection asynchronously. To better optimize the establishment of secure connections, QUIC is tightly coupled with TLS version 1.3. One of the main uses of QUIC is inside HTTP version 3.

2.1.3 Application Layer (L7)

HyperText Transfer Protocol (**HTTP**) is an Application layer (L7) protocol that enables stateless request/response interactions on the Web between web servers and clients (e.g. browsers). Similar to other L7 protocols, it uses Universal Resource Locators (**URLs**) for locating resources.

URL format: `scheme://host:port/path?query=value#fragment`

URL example: `http://www.example.com:80/path/to/file.html`.

HTTP up to version 2 uses TCP as a transport protocol and since version 3 uses QUIC. HTTP provides several features such as:

- Request Methods - used by the client to specify the action to perform on the resource behind the given URL, e.g. GET, POST, PUT, DELETE, etc.
- Headers - used to provide additional information about a request or response, e.g. Content-Type, Authorization, Cache-Control
- Status codes - used to indicate the result of a request, e.g. if it was successful (200), or if the resource is missing (404)
- Cookies - used to include stateful information about the user kept on the client-side
- Caching - used to specify that the result of a request can be cached for a certain time to avoid repeating the request's action.

The Domain Name System (**DNS**) operates at the Application Layer (L7) and allows the conversion of human-readable domains to IP addresses, e.g. `google.com` to `142.250.179.142`.

2.2 Secure Communication Protocols

The communication protocols from the previous section do not encrypt the payloads of their packets, which allows intruders and routers between the communicating hosts to see the data. This section provides an overview of protocols that provide security to those communication protocols.

2.2.1 Transport Layer (L4) to Application Layer (L7)

Transport Layer Security (**TLS**) [Res18] and its precursor Secure Sockets Layer (**SSL**) are protocols that provide secure communications to Application layer (L7) protocols on top of a reliable Transport layer (L4) protocol like TCP. Datagram Transport Layer Security (**DTLS**) is a related protocol that works with connectionless transport protocols like UDP. TLS is usually placed somewhere between the Presentation layer (L6) and the Transport layer (L4) because Application layer (L7) protocols use it as a transport protocol while having to manage the TLS connections.

TLS relies on digital certificates and Public Key Infrastructure (**PKI**) to establish trust between the communicating parties and to prevent man-in-the-middle attacks. A certificate includes information such as:

- Subject - an identifiable name for the certificate's owner. Depending on the use case it can be a domain name, an IP address, an email address or others.
- Subject's public key - an asymmetric public key that is used by other parties to verify that they are communicating with the subject who is expected to be in control of the corresponding private key. The public key can also be used to encrypt messages that only the subject can decrypt.
- Issuer - an entity that is responsible for validating the identity of the subject. It is usually a Certificate Authority (**CA**) that is trusted by the consumer of the certificate, but in the case of a self-signed certificate, it can be the subject itself
- Issuer's signature - a signature of the certificate's contents using the issuer's private key

Trusted CAs that serve as the root of trust are usually included in the operating system or a Web browser. This allows applications to verify the certificates of the servers they communicate with. Web browsers use HyperText Transfer Protocol Secure (**HTTPS**) [Res00] - a variant of HTTP that uses TLS to secure the underlying TCP or QUIC connections of Web applications. The one-way approach where only the server has to authenticate itself to the clients is usually sufficient for most web interactions. TLS can also be used for mutual authentication, where both of the communicating parties have to present a valid certificate to each other, but this requires additional infrastructure to manage the client-side certificates. This mode of operations is sometimes used in Zero Trust networking, in microservice architectures and TLS-based VPN applications.

Due to asymmetric cryptography being computationally expensive, TLS uses a hybrid approach. The communicating parties use the asymmetric key/s and Diffie–Hellman key exchange to agree on a set of symmetric session keys for authentication and encryption/decryption of their data.

TLS is rather complex because it needs to support many possible use cases while remaining backward compatible. It allows the negotiation of security parameters like cipher suits.

TLS operates at the Transport layer (L4) and above so it encrypts the application traffic, but not the IP datagrams. While an ISP or an intruder with access to the network cannot decrypt the traffic to see what is being communicated, it could see which IP addresses are communicating with each other.

Secure Shell Protocol (**SSH**) is an Application layer (L7) protocol that allows a client to securely log in and execute commands on a remote server. It does not rely on TLS but uses similar cryptographic primitives. Both the client and the server must authenticate to each other using public-key cryptography or a password. The cryptographic material must be distributed via a side channel either manually or via automation tools.

2.2.2 Network Layer (L3)

Network layer (L3) security protocols provide host-to-host security in Internet Protocol communications. Their implementations are usually configured in the operating systems of the hosts and are invisible to the higher-layer protocols that depend on the Internet Protocol.

IPSec

Internet Protocol Security (**IPSec**) is a Network layer (L3) protocol suite that provides host-to-host secure communications between IP hosts. It has two modes of operation:

- Transport mode - encrypts the payload of an IP datagram, but not the header. This mode secures the traffic between two network hosts, but similarly to TLS, the routers between the hosts can still see the IP addresses of the hosts and the ports they are communicating on
- Tunneling mode - encrypts the entire IP datagram and encapsulates it in a new IP datagram with an unencrypted header. This mode is used for secure comm allows a host to decrypt the encapsulated both the traffic connection between a client and a server that can decrypt the encapsulated IP datagram and forward it to its destination.is often used to securely connect a client to a server that can decrypt the encapsulated IP datagrams and forward them to their destination. The client

IPSec always requires mutual authentication, unlike TLS, where it is optional. Authentication can be achieved either via digital certificates, Pre-Shared Keys (**PSKs**) or others.

VPN This way the VPN client can hide the IP addresses of the final destinations of the traffic from hosts between itself and the VPN server. Depending on the use case, the VPN server could either facilitate the client's secure access to other resources on

OpenVPN Protocol

-
-
-
-

WireGuard

WireGuard [Don17] is a more recent protocol with a design informed by lessons learned from IPSec and OpenVPN and a key management approach inspired by SSH. It is a lower-level protocol that focuses on configuration simplicity while network topology, peer discovery and key distribution are left as a responsibility of higher-level systems that use it as a building block. Wireguard is implemented as a Layer 3 overlay over UDP tunnels. WireGuard has both user-space implementations that use a TUN driver and also has direct support built into the Linux Kernel since version 5.6 (May 2020). The kernel implementation allows for better performance because it does not need to copy packets between the kernel and user-space memory.

WireGuard's cryptography is based on the **Noise Protocol Framework**[Per18]. Noise is another recent effort that applies the ideas of TLS in a simplified way. It serves as a blueprint for designing use-case-specific protocols for establishing secure communication channels based on Eliptic Curve Diffie-Hellman (**ECDH**) handshake patterns. It powers the end-to-end encryption in messaging applications such as WhatsApp and Signal, and Virtual Private Network (**VPN**) software such as WireGuard and Nebula.

elaborate

noise is transport agnostic

noise has limited cipher suites

The snippets below show a minimal set of configuration options that need to be provided for two peers to be able to form secure tunnels with each other.

```
# peer1.conf
[Interface]
Address = 101.0.0.1/32
ListenPort = 53063
PrivateKey = ePTiXXhHjvAHdWUr8Bimk30n0gh3m241RAzsN0JZDW0=

[Peer]
PublicKey = BSn0ejd1Y3bKuD+Xpg0ZZe0f+Ies/oql0NZxw+S0mkc=
AllowedIPs = 101.0.0.2/32
Endpoint = peer1.example.com:38133
```

```
# peer2.conf
[Interface]
Address = 101.0.0.2/32
ListenPort = 38133
PrivateKey = sN/d6XUPEVPGSziVgCC0n0ivDK+qAoYC3nxnssQ5Rls=

[Peer]
PublicKey = e/TxvPmrgcc1G4cSH2bHv5J0PRHXKjYxTFoU8r+G93E=
AllowedIPs = 101.0.0.1/32
```

Each peer has a public/private key pair that is used for authentication and encryption based on the Noise Protocol Framework [Per18]. The `Address` field specifies the virtual IP address that the local network interface will use, while the `AllowedIPs` field specifies what virtual IP addresses are associated with a peer's public key. A peer's `Endpoint` field specifies the URL at which it can be reached. Only one of the peers must be configured with a reachable endpoint for the other one. In the above example once `peer1` initiates communication with `peer2`, `peer2` will learn the current endpoint of `peer1` and will be able to communicate back with it.

CIDR is a common notation for describing IP address ranges, e.g. '192.168.0.1/16', where the number after the slash describes the bit-length of the fixed prefix for a subnet.

2.3 Challenges and Solutions for Multiparty Communications

The version of the Internet Protocol, that was originally deployed globally (IPv4), uses 32-bit numbers as IP addresses, allowing for around 4 billion unique addresses. Due to the popularity of the Internet, there are many more devices than available IPv4 addresses, which has caused challenges. IPv6 is a newer version of the protocol that uses a larger 128-bit address space which is sufficient for assigning 100 addresses for each atom on Earth. However, its adoption has been slow, as according to Google[Goo23] as of 2023 around 41% of their users access their services over IPv6. Additionally, despite that IPv6 allows for all devices to be addressable on the Internet, for security reasons, most of them would use firewalls to block incoming remote traffic that is not associated with outgoing connections.

A widespread solution to the addressing problem is Network Address Translation (**NAT**). It allows many devices without globally unique IP addresses to initiate connections to publicly addressable devices on the Internet via a limited number of gateways that must have globally unique IP addresses. A NAT gateway replaces the local source IP address of each outgoing IP datagram with its own public IP address before passing it on to the next link on the way to the destination while maintaining a mapping between the source and destination IPs in a translation table. The destination host can then address its responses back to the NAT gateway's public IP address, which in turn replaces its own IP from the incoming datagrams with the IP of the local device and forwards them to it. If the IP datagrams encapsulate TCP/UDP packets, the gateway additionally rewrites the source and destination ports, which means that NAT techniques can be placed somewhere between Layers 3 and 4 of the OSI model.

The effect of NAT on connectivity is similar to an IPv6 firewall as they both allow devices on a local network to initiate bidirectional communication to remote devices with public IP addresses, but connections cannot be natively initiated by the remote devices. As Figure 2.2 shows, it follows that when two devices are behind separate NATs, neither can contact the other first. **Client/Server** communication is less affected by this limitation because Servers are usually deployed to a public IP address that can be contacted by Clients with local IP addresses. **Peer-to-Peer** communication, however, is more challenging because the peers are often devices in separate residential networks behind different NATs. Several **NAT traversal** techniques try to solve this with different performance tradeoffs and success that varies depending on the NAT [JA07] and its behavior when mapping ports and IP addresses.

describe some of the nat behaviors, e.g. if the source IP address/port vary per destination are changed depending on the destination/port mapping algorithms, if it maps ports, IPs, whether the mapped IPs are different per destination and others.

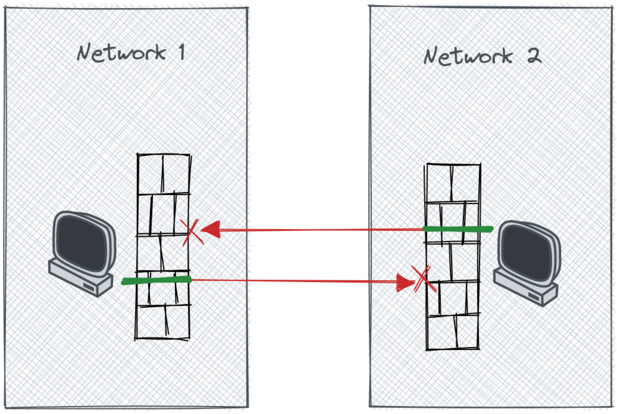

Figure 2.2: Two parties behind separate NATs

One approach based on the Client/Server model is to use a publicly addressable **relay** server that is contacted by the NATed devices and then forwards the Peer-to-Peer traffic to the intended recipient. Compared to direct communication, relaying results in a higher network latency due to the longer path that each packet must travel. Maintaining a relay server requires some technical expertise and may be costly depending on the expected throughput. Despite the drawbacks, relaying works under most networking scenarios and is therefore often used as a fallback in case all other approaches fail to find a direct path. Protocols such as Traversal Using Relays around NAT (**TURN**) [Red+20] and Designated Encrypted Relay for Packets (**DERP**) [Tai22] can be used to securely implement relaying.

The NAT gateway in many residential networks is a Router device under the customer's control that has a statically or dynamically assigned public IP address. Most routers can be manually configured through their admin page to forward all traffic that arrives at a given port to a specific device on the local network. Remote applications can then initiate a connection to the local device if they know the IP address of the router and the forwarded port. The manual configuration, however, can be inconvenient and many users may be unaware of that setting because it is not necessary for the more straightforward Client/Server communications. Some routers also support programmatic configuration of port forwarding via a Layer 7 protocol like Universal Plug and Play (**UPnP**) or its successors NAT Port Mapping Protocol (**NAT-PMP**) and Port Control Protocol (**PCP**). However, these protocols are not always supported and are often disabled by the local network administrators due to security concerns related to bugs in their implementation, vulnerable IOT devices on the local network or malicious programs being able to expose local devices to the internet.

connection reversal

An efficient NAT traversal approach that works with some types of NATs is to use Session Traversal Utilities for NAT (**STUN**) [Pet+20] in combination with UDP hole punching (Figure 2.3). STUN is a protocol operating at Layer 7 that allows a client application to detect the presence of NAT gateways on the network path to a public STUN server, and identify their types and the public IP address that they map to externally. The process usually involves the following steps:

- An application sends UDP datagrams to the STUN server (1, 3 in fig. 2.3)
- The STUN server responds with the source IP address and port specified inside the datagrams (2, 4 in fig. 2.3)
- The application compares its own endpoint with the source endpoint observed by the STUN server and if the values differ, it can be inferred that they were rewritten by a NAT. Additional STUN servers are contacted to determine if the NAT maps IPs and ports in a predictable fashion.

UDP hole punching is a related technique that, depending on the NAT types, can allow direct communication between two applications behind separate NATs. The applications must discover each other's externally mapped endpoints, perhaps via the STUN server. If the NATs use the same external port regardless of the remote destination:

If, else?

- The two applications simultaneously send UDP packets to each other's external endpoints (5 in fig. 2.3)
- Their respective NATs will process the outgoing packets to the other peer and create a port mapping for the reverse traffic - a "punched hole"
- When the incoming traffic from a peer arrives at the other peer's NAT, it will be forwarded correctly due to the port mapping that was created earlier

NATs that map different ports per remote destination sometimes allocate port numbers predictably, which can be used by the peers to try to guess the port that will be opened by the opposing side's NAT.

Talk about traversal friendly NATs and unfriendly NATs

add a bullet list with

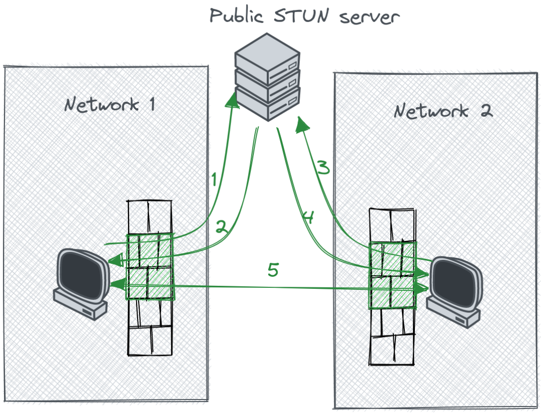

Figure 2.3: NAT traversal via STUN

In mobile networks like 4G and 5G, the ISP often utilizes a Carrier-Grade NAT (**CGNAT**) as part of their infrastructure, while all devices under the user's control, including the router, only have local IP addresses. STUN techniques would fail to discover a direct path between two parties behind separate CGNATs or other unpredictable NAT algorithms. The only remaining possibility is to relay the traffic via a publicly reachable third-party host using a protocol similar to TURN.

Interactive Connectivity Establishment (**ICE**) is a protocol that describes a standard way for peers to gather candidate addresses for direct communication via STUN and TURN and then exchange them via a signaling server. The protocol continuously checks which candidates provide the best connection and adjusts them.

Web Real-Time Communication (**WebRTC**) is a framework that allows peer-to-peer communications between Web applications in Web browsers. Web applications are normally limited to HTTP connections and cannot use raw UDP or TCP connections. WebRTC implements the ICE functionality in Web browsers and provides an API to Web applications.

only 65000 ports per IP address means that CGNATs that provide more than 65000 connections from client devices require more than one public IP address

hairpinning - Hairpinning, also known as NAT loopback or NAT reflection, is a technique used by NAT devices to allow hosts on a private network to access a public server using

Chapter 3

Overlay Networks

An **overlay network** is a higher-order solution that provides additional networking functionality on top of an existing underlay network like the Internet. From the point of view of its consumers, an overlay network may appear at a lower OSI layer, despite being implemented using protocols from higher layers. For example VPNs can provide virtual interfaces to the Operating System at the Link layer (L2) or Network layer (L3) while being implemented on top of a Transport layer (L4) protocol like User Datagram Protocol (**UDP**) or a Presentation layer (L6) protocol like Transport Layer Security (**TLS**). Virtual IP addresses can be assigned to the hosts and applications that are already designed to work with TCP/IP can directly use the virtual network interfaces via the regular TCP/IP mechanisms provided by the operating system.

Other overlay networks are both implemented and used at the Application layer (L7). To communicate via such an overlay network, applications often have to implement specific functionality in their software by utilizing a framework or a library.

Figure 11.1 shows an approximate OSI model mapping of several protocols and network overlay solutions from the point of view of the systems that use them and the arrows show dependency relations between them.

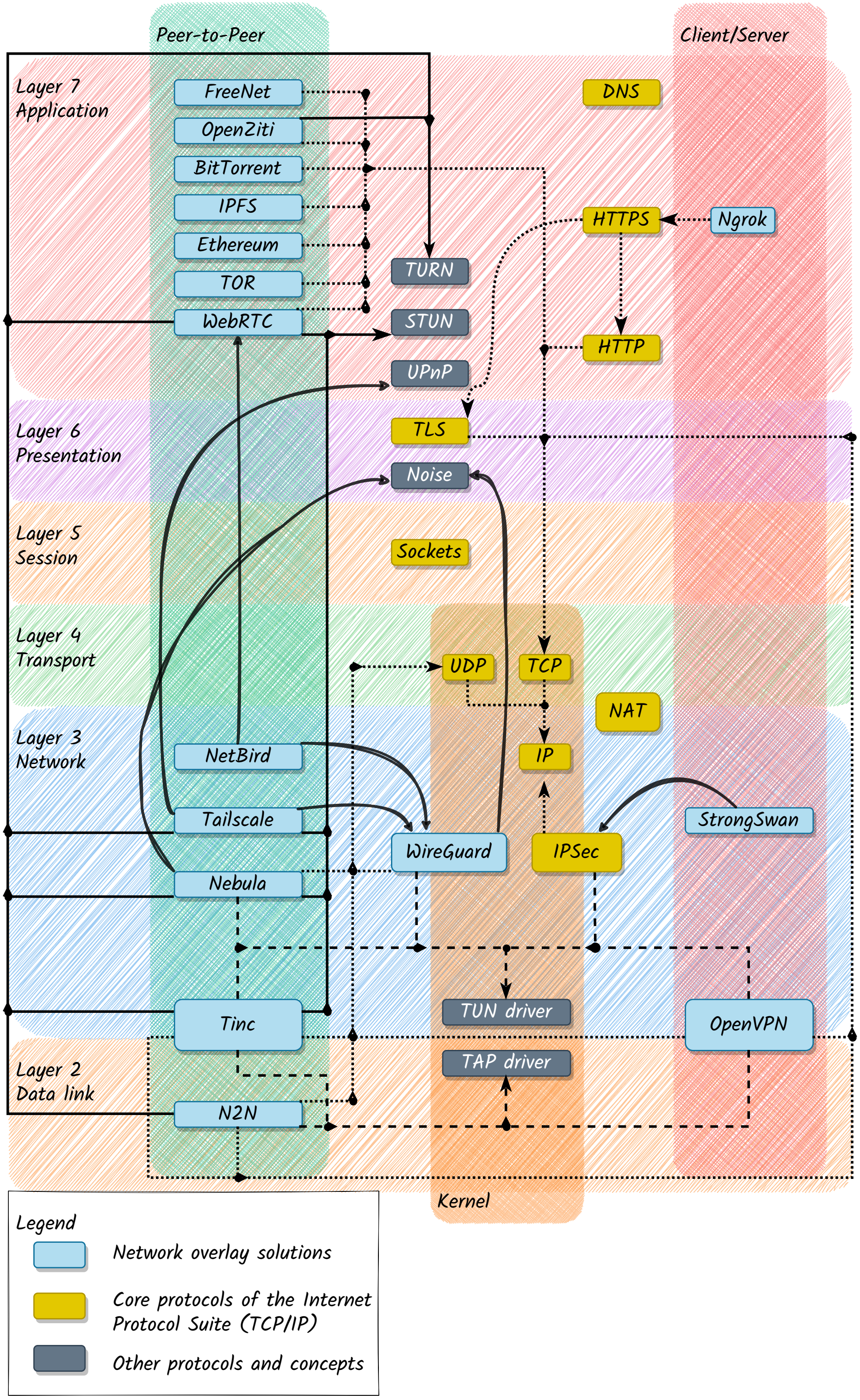

Figure 3.1: OSI model mapping of various protocols

3.1 Data Link Layer (L2) and Network Layer (L3)

3.1.1 Client/Server VPNs

Layer 2 virtual networks provide a virtual network switch that allows remote machines to be on the same virtual LAN and share the same IP address range. Layer 3 virtual networks provide a virtual network router that allows remote machines to be on separate LANs. Depending on the specific implementation, the overlay network can either be implemented directly in the Operating System's kernel, or on top of a driver like TUN or TAP.

- Layer 2 vs Layer 3 Networks
  - Layer 2 overlay networks bridge other networks
    - virtual network switch
    - remote machines are on the same virtual LAN and can share the same IP address range
    - allows broadcast/multicast
    - TAP driver
  - Layer 3 overlays route traffic between separate local networks
    - virtual network router
    - remote machines are on separate LANs
    - simpler to configure
    - TUN driver
    - the low-level solutions from the previous section are complex to set up.
    - overlay networks package some of those solutions for a specific use case Most overlay networks use a combination of the NAT traversal techniques mentioned previously. They can be placed in Layers 2, 3 or 7. Layer 2 overlays act as a virtual network switch, while Layer 3 overlays act as a virtual network router. Layer 7 overlays are implemented in user-space as libraries or applications that run on top of the network stack of the host operating system. Layer 2 and 3 overlays can either be implemented as kernel modules or as user-space applications that use a **TUN/TAP** driver to interface with the kernel.
    - The term "VPN" is somewhat overloaded as it can refer to different related concepts.

    VPNs are implemented as Layer 2 or 3 network overlays. They are commonly used for securely connecting machines from different LANs. They provide software emulation of a network interface controller via a TUN/TAP driver on the operating system level and allow other software to transparently use the functionality of the Internet Protocol (**IP**) suite without requiring extra changes. Traditional VPNs such as IPSec [FK11] and OpenVPN [Ope22] use a centralized service that all (encrypted) client communications must pass through. This introduces a single point of failure and a potential bottleneck that might negatively impact the performance of the multiparty computations due to their Peer to Peer (**P2P**) nature.

explain the names

tls vs ipsec vpns. TLS vpns offer a virtual network interface at layer 3, but run over L7 TLS

OpenVPN Client/Server

We already discussed the VPN protocol behind OpenVPN. Here we will go into the client and server software and how they are implemented and discuss the typical network topologies.

3.1.2 Mesh VPNs

- Tinc
- N2N
- Tailscale
- Nebula
- ZeroTier

Mesh VPNs such as Tinc [Sli22], Tailscale [Tai] and Nebula [Def22] utilize NAT Traversal techniques to create direct P2P links between the clients for the data traffic. Authentication, authorization and traffic encryption are performed using certificates based on public key cryptography.

All three are open-source, except Tailscale's coordination service which handles peer discovery and identity management. Headscale [Fon22] is a community-driven open-source alternative for that component. Tinc is the oldest of the three but has a relatively small community. It is mainly developed by a single author and appears to be more academic than industry motivated. Nebula and Tailscale are both business driven. Tailscale was started by some high-profile ex-googlers and is the most end-user-focused of the three, providing a service that allows people to sign up using identity providers such as Google, Microsoft, GitHub and others. They also provide an Admin console that allows a user to easily add their personal devices to a network or share them with others. It also has support for automation tools like Terraform for creating authorization keys and managing an Access Control List (**ACL**) based firewall. Nebula was originally developed at the instant messaging company Slack to create overlay networks for their cross-region cloud infrastructure, but the authors later started a new company and are currently developing a user-centric platform similar to Tailscale's. Nebula is more customizable than Tailscale and since it is completely open-source it can be adapted to different use cases, but it is also more involved to set up. A certificate authority needs to be configured for issuing the identities of the participating hosts. Furthermore, publicly accessible coordination servers need to be deployed to facilitate the host discovery. Tailscale employs a distributed relay network of Designated Encrypted Relay for Packets (**DERP**) servers, while Nebula can be configured to route via one of the other peers in the VPN.

3.2 Application Layer (L7)

3.2.1 OpenZiti

OpenZiti is a commercial product that provides an SDK for creating P2P overlay networks. It is developed by NetFoundry, a company that provides a Software as a Service platform for creating and managing P2P overlay networks. The SDK is open-source and can be used to create custom overlay networks. The SDK is written in Go and provides a high-level Application Programming Interface (**API**) for creating and managing the overlay network. It uses a centralized service for peer discovery and identity management. The data traffic is routed via a P2P network of relays. The relays are hosted by NetFoundry and are not open-source. The SDK is available for Go, Java, Python, C# and C++.

edit

- uses relays

3.2.2 LibP2P

- libP2P
- ngrok
- TOR
- BitTorrent
- IPFS
- Ethereum
- Teleport
- Freenet

Chapter 4

Digital Identities for Multiparty Computations

Digital identity mechanisms provide standardized ways for referring to entities in the context of digital interactions. We have already seen some examples of digital identities in the previous chapter:

- MAC addresses - host identities at the Data link layer (L2)
- IP addresses - host identities at the Network layer (L3)
- Digital Certificates - identities that can be cryptographically verified

Key concepts related to digital identity mechanisms include:

- Issuance - the process of assigning a digital identity to an entity. Often this involves a trusted third party like a government or a digital identity provider like Google or GitHub. Another option is SSI where a user manages their own identity.
- Authentication - the process of verifying the identity of an entity
- Credentials - use case-specific attributes associated with an identity
- Authorization - the process of verifying if an identity's owner is allowed to perform an action on a resource

Other notable examples include:

- Email addresses - identities for entities participating in email communications
- User names - user identities in various digital platforms that can be verified via a password
- ID cards - government-issued identity documents are increasingly able to support digital interactions
- Biometrics - used to prove an identity via sensor data

- IP addresses
  - Cryptographically Generated Addresses - rfc3972
  - Privacy Extensions - rfc4941
  - DNS
- MAC addresses
- Government
  - ID Cards / Passports

        - NFC
    - Estonia's Digital Government
    - DigiD in NL
    -
- Certificates
- TLS
- IPSec
- WireGuard
- OpenID Connect
- Crypto currency addresses
    - Hierarchical Deterministic Keys
    - ENS
- Self-Sovereign Identity
    - Decentralized Identifiers
    - Verifiable Credentials
    - Identity Overlay Network (ION) from Microsoft
        - DPKI
- Social Networks
    - Centralized
        - Facebook
        - Twitter
        - Reddit
    - Decentralized
        - Mastodon - Mastodon is an open-source, decentralized social networking platform that provides an alternative to traditional social media giants like Twitter. It operates on a federated model, which means that instead of being a single, centralized platform, it is a network of independently operated servers, known as "instances". Each instance hosts its own community with its own rules, but users on any instance can interact with users on other instances thanks to a common protocol, creating a connected, distributed social network.
        - Bluesky - An ex-twitter initiative to move away from centralized social media platforms, where a single organization controls the network. Instead, Bluesky aims to develop a standard protocol for social media that would enable the development of a variety of interoperable, decentralized platforms.
        - Nostr
- Identity Privacy

Part II

Proof of Concepts and Evaluation

Chapter 5

Testing methodology

In the following chapters, we will design and implement several solutions for ad hoc MPC sessions based on a subset of the previously discussed related works:

- Internet protocol
- Wireguard
- Tailscale
- Headscale
- ? Headscale with DID identity?
- ? WebRTC?
- A custom solution that automates the WireGuard configuration by visiting a web page

Additionally, we will analyze and compare them in terms of performance, security and usability

5.1 Measuring performance

During the preparation phase of the project, we developed the Extensible Evaluation Environment (E^3) framework which simplifies and automates the process of deploying machines in different geographical regions, connecting them via an overlay network and executing multiparty computations between them, where each machine represents a different party.

To summarize, E^3 is a set of scripts that use several automation tools:

- Terraform - declarative provisioning
- NixOS - declarative Linux distribution
- Colmena - declarative deployment for NixOS
- PSSH - parallel execution of remote scripts over ssh
- DigitalOcean - a cloud provider

It allows us to quickly provision cloud virtual machines in multiple regions and reproducibly deploy all necessary software for running a multiparty computation over a chosen network overlay solution. The source code of E^3 can be found on GitHub

Each solution will be deployed using the E^3 framework and the performance will be quantitatively measured in terms of the time it takes to execute several MPyC demos. The selected demos have different complexities in terms of communication rounds and message sizes which will allow us to observe their impact on the overall performance.

1. Secret Santa - high round complexity with small messages
2. Convolutional Neural Network (CNN) MNIST classifier - low round complexity with large messages

The demos will be configured at three different input size levels

- Low,
- Medium
- High

Furthermore, the demos will be executed in several networking scenarios:

1. 1-10 parties in the same geographic region
2. 1-10 parties evenly distributed across two nearby regions
3. 1-10 parties evenly distributed across two distant regions
4. 1-10 parties distributed across multiple distant regions

5.2 Security

We will analyze aspects such as

- key distribution
- trust model - are there any trusted third parties and what would be the consequences if they are corrupted or breached
- traffic encryption
- identity strength

5.3 Usability

For each solution, we will describe the steps that the parties need to perform to execute a joint multiparty computation. Those steps will be analyzed in terms of:

- Complexity - how much technical expertise is expected from the parties to be able to execute the steps
- Initial effort - how much effort is each party expected to put in preparing for their first joint computation
- Repeated effort - after the initial setup, how much effort is required to perform another computation
  - with the same set of parties
  - with another set of parties
- Finalization effort - how much effort is required to finalize the MPC session once it is complete and clean up any left-over artifacts or resources so that the machine of each party is in its original state

Chapter 6

Internet Protocol based solution

This solution focuses on directly using the internet protocol without involving an overlay network. Our goal is to analyze the implications of using only the functionalities that MPyC directly supports to serve as the reference for our later experiments.

6.1 Implementation details

We will manually set up the multiparty computations via the public IP addresses of the machines and DNS.

6.2 Performance analysis

6.3 Security analysis

6.4 Usability analysis

Chapter 7

WireGuard based solution

This solution creates an overlay network by manually configuring WireGuard on each machine.

7.1 Implementation details

7.2 Performance analysis

7.3 Security analysis

7.4 Usability analysis

Chapter 8

Tailscale based solution

Tailscale is a VPN solution that configures a mesh of direct Wireguard tunnels between the peers.

8.1 Implementation details

8.2 Performance analysis

8.3 Security analysis

8.3.1 Trust model

There is a centralized service that deals with the key distribution, which needs to be trusted to provide the correct public keys for the correct parties

8.3.2 Identity

Identity is based on third party identity providers such as Microsoft and GitHub

- Magic DNS
-

8.4 Usability analysis

With tailscale each party needs to

- register a Tailscale account
- Download and install tailscale on the machine they want to run a multiparty computation
- Run tailscale on their machine and logs into their account in order to link it to their own Tailnet
- Share their Tailscale machine with the Tailnets of each of the other parties
- Download the demo they want to run
- Form the flags for running the chosen demo

  - add -P $HOST:$PORT for each party using their Tailscale hostname/virtual IP
- Run the demo

Chapter 9

Headscale based solution

This solution is similar to the Tailscale one, but it uses Headscale - a self-hosted open-source alternative to the closed-source Tailscale coordination service.

9.1 Implementation details

9.2 Performance analysis

9.3 Security analysis

9.3.1 Trust model

There still is a centralized service like in the Tailscale solution, but here it is self-deployed.

9.3.2 Identity

9.4 Usability analysis

Chapter 10

MPyC Web

This chapter presents the design of MPyC Web - an MPC connectivity solution based on web technologies (L7) that can run in web browsers.

Design goals:

- Runs in browsers (client-side) on both PCs and smartphones
- Uses the MPyC Python framework with as few modifications as possible
- Peer-to-peer with a fallback to relaying
- Secure
- Simple to use

10.1 Python Runtime Selection

10.1.1 Selection Criteria

While web browsers natively support only JavaScript (**JS**), there are approaches with different tradeoffs that can enable other languages, including Python. In this section we will investigate several web-based Python runtimes and choose the most appropriate one for MPyC Web.

MPyC depends on several Python packages that are implemented in C, such as numpy, gmpy2, scipy and relies heavily on asyncio and TCP sockets from Python's standard library. The runtime we choose should support as many of those dependencies as possible to avoid having to find alternatives or reimplement them.

Another important aspect is the runtime performance due to the heavy operations involved in MPCs.

Fast startup time, low build and deployment complexity are less crucial but nice to have properties of the runtime, as they offer a better experience for both users and developers.

To summarize, our criteria for runtime selection are:

- Python ecosystem/package availability
- Runtime performance
- Startup, build and deployment time

10.1.2 Available Solutions

There are a number of available solutions for using Python in web browsers, falling under three main categories[Moz19] [Anv19]:

1. Transcrypt[Tra24] - Ahead-Of-Time (**AOT**) JavaScript transpiler - the Python code is transpiled to JavaScript and the resulting code is executed by the browser at runtime. is an example project
2. Skulpt[Sku], Brython[Bry24] - Just-In-Time (**JIT**) JavaScript interpreters - the Python code is converted to JavaScript in the browser at runtime
3. Pyodide[Pyo], MicroPython[Mic], RustPython[Rus] - WebAssembly (**WASM**)[WAS17] interpreters - a Python interpreter is implemented in a compiled language like C/C++/Rust/Go that can target WASM - a secure and efficient binary instruction format for a virtual machine that can run in browsers with close to native performance.

10.1.3 Ecosystem Availability

From the listed

10.1.4 Preliminary Runtime Benchmarks

We will measure the time it takes each Python runtime to execute a given set of benchmark functions to have a preliminary indication of how they might perform for MPyC. To put that data in context, we will also include the results for the native performance of Python's default implementation (CPython) as well as JavaScript running in a browser.

The results will be presented in terms of operations per second, for an input size of 100 000, e.g. if a script benchmarks integer additions, then a loop of 100 000 additions is considered 1 operation. Similarly if we are copying a list or generating a sequence of numbers, then a single operation will be a sequence of 100 000 numbers. This is done to keep the relative number of function calls much lower than the operation being tested, as function calls are generally more computationally expensive and can skew the results.

Additionally our setup will make sure to run each benchmark for at least 0.2 seconds in order to reduce the volatility of the results. It will start with a single iteration of the 100 000-sized operation and then exponentially increase the iterations until the time limit is reached.

The set of benchmark functions was chosen based on implementation simplicity and portability across the tested runtimes. We will only show a few code examples below for illustration purposes, but the full benchmark suite can be found on GitHub:

- **assign** - assigning a number to a variable

```python
def assign(iters=100_000):
    for _ in itertools.repeat(None, iters):
        x = 1
```

- **multiply** - multiplying two numbers

```python
def multiply(iters=100_000):
    for _ in itertools.repeat(None, iters):
        17 * 41
```

- **bigint** - large number arithmetics
- **randlist** - generates a list of 100 000 random integers
- **cpylist** - copies a pre-generated list of 100 000 integers
- **sortlist** - sorts a list of 100 000 integers
- **fibonacci** - generating the fibonacci sequence mod 100 000

```python
def fibonacci(n=100_000):
    if n < 2:
        return n
    a, b = 1, 2
    for _ in itertools.repeat(None, iters):
        a, b = b, (a + b) % 100_000
    return a
```

- **primes** - generates the prime numbers from 0 to 100 000

The following machine was used for conducting the tests:

- OS - Microsoft Windows
- CPU - AMD Ryzen 7 2700X
- RAM - 32 GB
- Browser: Google Chrome v123.0.6312.28

Table 11.1 shows the benchmark results. A few interesting observations:

- JavaScript in a browser is a lot faster than native Python in most of the tests
- Transcrypt is faster than the other browser-based runtimes, probably because it is able to optimize the JavaScript code as it generates it ahead of time and it is also less focused on parity with CPython and more on interoperability with the JavaScript ecosystem
- The results vary quite a bit, e.g. Pyodide is faster than Brython at generating the fibonacci sequence, but slower at big int arithmetics and multiplications
- Pyodide appears to be faster on average than the other WASM-based and JS-based interpreters

Table 10.1: Benchmark results of the Python runtimes measured in operations per second for inputs of size 100 000

	Benchmark		**Results**		**(ops/sec)**			
Project	**assign**	**multiply**	**bigint**	**randlist**	**cpylist**	**sortlist**	**fibonacci**	**primes**
JavaScript	19,327	19,413	19,351	713	4,227	35	3,271	1,029
CPython	775	380	69	21	4,280	81	168	200
Transcrypt	1,289	1,285	1,282	176	6,135	35	337	1124
Pyodide	400	111	26	5	4,425	45	53	73
Brython	292	166	45	0.7	5,076	67	9	45
MicroPython	69	36	6	16	403	41	22	10

	Benchmark		**Results**		**(ops/sec)**			
Skulpt	40	27	1	16	6,232	5.2	8	18
RustPython	60	59	8	0.8	525	0.1	6	6

10.1.5 Startup

10.1.6 Conclusions

Out of the discussed runtime options, Pyodide offers the widest support for packages from the standard library and third-party packages. None of the options support the sockets package due to browser limitations, but an alternative implementation can be built on top of the WebRTC API.

Out of those projects, Py

Table 10.2: Summary of Python runtimes for browsers

Project	Approach	Startup	Performance	Ecosystem
Transcrypt	transpiler (AOT)	0s	fast	(−) n/a
Skulpt	JS-based interpreter (JIT)	0s	slow	(−) n/a
Brython	JS-based interpreter (JIT)	0s	medium	(−) no numpy
Pyodide	WASM-based interpreter (JIT)	5s	medium+	(+) asyncio (+) numpy (+) gmpy2 (+) scipy (+) python wheels (+) filesystem (−) no sockets
MicroPython	WASM-based interpreter (JIT)	1s	slow	(−) n/a
RustPython	WASM-based interpreter (JIT)	5s	slow	(−) n/a

10.2 Design of MPyC Web

Web based solution for running MPC in browsers on the client side via WebAssembly and WebRTC for peer-to-peer connectivity.

The lifecycle of a message between the workers of 2 peers, “Peer A” and “Peer B” looks like this:

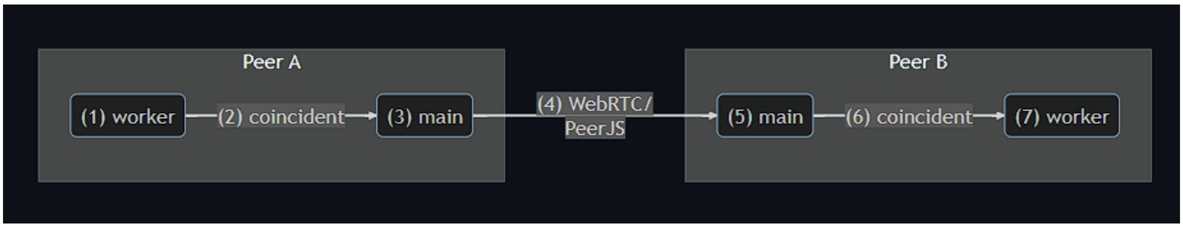

Figure 10.1: MPyC Web

1. Python Worker on Peer A
   - creates a message that contains a secret share computed via the mpyc, which serializes it to binary via struct.pack(). The resulting message is non-ascii, but when escaped can look like this:

```
b'\x06\x00\x00\x00\xc9\x93\\\xc0\xff\xff\xff\xff\x04\x00\x80\x99\x1b\x01'
```

   - worker calls a main thread function (sendRuntimeMessage) via a coincident proxy and passes the serialized value, as well as the destination Peer’s ID (Peer B).

The worker does not need to await the result of this call, but it does because of xworker.sync.

2. Coincident handles the message by:
   - (Worker) JSON.stringify() -> Atomics.wait()
   - (Main) JSON.parse() -> Run the Main Thread function -> Atomics.notify()
3. Main thread on Peer A fires an event to notify PeerJS that it needs to transport the message and immediately returns in order to not block the Worker
4. PeerJS serializes the message with some added fields via MessagePack and transmits it to Peer B
5. Main thread on Peer B receives the message and forwards it to the appropriate worker callback via coincident
6. Coincident does the JSON.stringify() -> Atomics.wait() stuff (?) from main to worker on Peer B
7. Worker on Peer B handles the message by passing it to an asyncio coroutine that is waiting for it or stores it for later use

10.3 Implementation details

10.4 Performance analysis

10.5 Security analysis

10.5.1 Trust model

There still is a centralized service like in the Tailscale solution, but here it is self-deployed.

10.5.2 Identity

10.6 Usability analysis

Chapter 11

MPyC Web

This chapter presents the design of MPyC Web - an MPC connectivity solution based on web technologies (L7) that can run in web browsers.

Design goals:

- Runs in browsers (client-side) on both PCs and smartphones
- Uses the MPyC Python framework with as few modifications as possible
- Peer-to-peer with a fallback to relaying
- Secure
- Simple to use

11.1 Python Runtime Selection

11.1.1 Selection Criteria

While web browsers natively support only JS, there are approaches with different tradeoffs that can enable other languages, including Python. In this section we will investigate several web-based Python runtimes and choose the most appropriate one for MPyC Web.

MPyC depends on several Python packages that are implemented in C, such as numpy, gmpy2, scipy and relies heavily on asyncio and TCP sockets from Python's standard library. The runtime we choose should support as many of those dependencies as possible to avoid having to find alternatives or reimplement them.

Another important aspect is the runtime performance due to the heavy operations involved in MPCs.

Fast startup time, low build and deployment complexity are less crucial but nice to have properties of the runtime, as they offer a better experience for both users and developers.

To summarize, our criteria for runtime selection are:

- Python ecosystem/package availability
- Runtime performance
- Startup, build and deployment time

11.1.2 Available Solutions

There are a number of available solutions for using Python in web browsers, falling under three main categories[Moz19] [Anv19]:

1. Transcrypt[Tra24] - AOT JavaScript transpiler - the Python code is transpiled to JavaScript and the resulting code is executed by the browser at runtime. is an example project
2. Skulpt[Sku], Brython[Bry24] - JIT JavaScript interpreters - the Python code is converted to JavaScript in the browser at runtime
3. Pyodide[Pyo], MicroPython[Mic], RustPython[Rus] - WASM[WAS17] interpreters - a Python interpreter is implemented in a compiled language like C/C++/Rust/Go that can target WASM - a secure and efficient binary instruction format for a virtual machine that can run in browsers with close to native performance.

11.1.3 Ecosystem Availability

From the listed

11.1.4 Preliminary Runtime Benchmarks

We will measure the time it takes each Python runtime to execute a given set of benchmark functions to have a preliminary indication of how they might perform for MPyC. To put that data in context, we will also include the results for the native performance of Python's default implementation (CPython) as well as JavaScript running in a browser.

The results will be presented in terms of operations per second, for an input size of 100 000, e.g. if a script benchmarks integer additions, then a loop of 100 000 additions is considered 1 operation. Similarly if we are copying a list or generating a sequence of numbers, then a single operation will be a sequence of 100 000 numbers. This is done to keep the relative number of function calls much lower than the operation being tested, as function calls are generally more computationally expensive and can skew the results.

Additionally our setup will make sure to run each benchmark for at least 0.2 seconds in order to reduce the volatility of the results. It will start with a single iteration of the 100 000-sized operation and then exponentially increase the iterations until the time limit is reached.

The set of benchmark functions was chosen based on implementation simplicity and portability across the tested runtimes. We will only show a few code examples below for illustration purposes, but the full benchmark suite can be found on GitHub:

- **assign** - assigning a number to a variable

```
def assign(iters=100_000):
    for _ in itertools.repeat(None, iters):
        x = 1
```

- **multiply** - multiplying two numbers

```
def multiply(iters=100_000):
    for _ in itertools.repeat(None, iters):
        17 * 41
```

- **bigint** - large number arithmetics
- **randlist** - generates a list of 100 000 random integers
- **cpylist** - copies a pre-generated list of 100 000 integers
- **sortlist** - sorts a list of 100 000 integers
- **fibonacci** - generating the fibonacci sequence mod 100 000

```
def fibonacci(n=100_000):
    if n < 2:
        return n
    a, b = 1, 2
    for _ in itertools.repeat(None, iters):
        a, b = b, (a + b) % 100_000
    return a
```

- **primes** - generates the prime numbers from 0 to 100 000

The following machine was used for conducting the tests:

- OS - Microsoft Windows
- CPU - AMD Ryzen 7 2700X
- RAM - 32 GB
- Browser: Google Chrome v123.0.6312.28

Table 11.1 shows the benchmark results. A few interesting observations:

- JavaScript in a browser is a lot faster than native Python in most of the tests
- Transcrypt is faster than the other browser-based runtimes, probably because it is able to optimize the JavaScript code as it generates it ahead of time and it is also less focused on parity with CPython and more on interoperability with the JavaScript ecosystem
- The results vary quite a bit, e.g. Pyodide is faster than Brython at generating the fibonacci sequence, but slower at big int arithmetics and multiplications
- Pyodide appears to be faster on average than the other WASM-based and JS-based interpreters

Table 11.1: Benchmark results of the Python runtimes measured in operations per second for inputs of size 100 000

	Benchmark		**Results**		**(ops/sec)**			
Project	**assign**	**multiply**	**bigint**	**randlist**	**cpylist**	**sortlist**	**fibonacci**	**primes**
JavaScript	19,327	19,413	19,351	713	4,227	35	3,271	1,029
CPython	775	380	69	21	4,280	81	168	200
Transcrypt	1,289	1,285	1,282	176	6,135	35	337	1124
Pyodide	400	111	26	5	4,425	45	53	73
Brython	292	166	45	0.7	5,076	67	9	45
MicroPython	69	36	6	16	403	41	22	10

	Benchmark		**Results**		**(ops/sec)**			
Skulpt	40	27	1	16	6,232	5.2	8	18
RustPython	60	59	8	0.8	525	0.1	6	6

11.1.5 Startup

11.1.6 Conclusions

Out of the discussed runtime options, Pyodide offers the widest support for packages from the standard library and third-party packages. None of the options support the sockets package due to browser limitations, but an alternative implementation can be built on top of the WebRTC API.

Out of those projects, Py

Table 11.2: Summary of Python runtimes for browsers

Project	Approach	Startup	Performance	Ecosystem
Transcrypt	transpiler (AOT)	0s	fast	(−) n/a
Skulpt	JS-based interpreter (JIT)	0s	slow	(−) n/a
Brython	JS-based interpreter (JIT)	0s	medium	(−) no numpy
Pyodide	WASM-based interpreter (JIT)	5s	medium+	(+) asyncio (+) numpy (+) gmpy2 (+) scipy (+) python wheels (+) filesystem (−) no sockets
MicroPython	WASM-based interpreter (JIT)	1s	slow	(−) n/a
RustPython	WASM-based interpreter (JIT)	5s	slow	(−) n/a

11.2 Design of MPyC Web

Web based solution for running MPC in browsers on the client side via WebAssembly and WebRTC for peer-to-peer connectivity.

The lifecycle of a message between the workers of 2 peers, "Peer A" and "Peer B" looks like this:

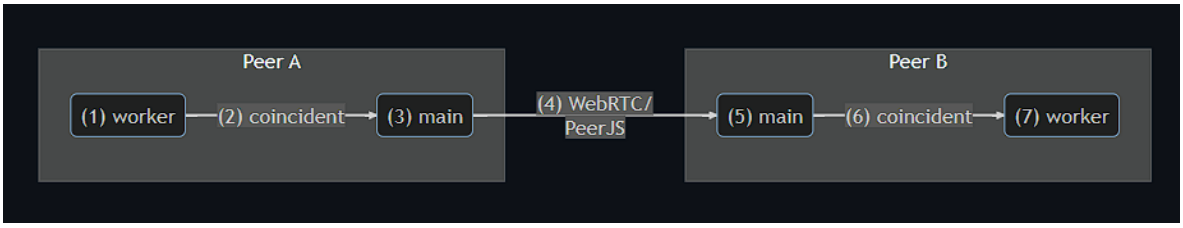

Figure 11.1: MPyC Web

1. Python Worker on Peer A

   - creates a message that contains a secret share computed via the mpyc, which serializes it to binary via struct.pack(). The resulting message is non-ascii, but when escaped can look like this:

```
b'\x06\x00\x00\x00\xc9\x93\\\xc0\xff\xff\xff\xff\x04\x00\x80\x99\x1b\x01'
```

   - worker calls a main thread function (sendRuntimeMessage) via a coincident proxy and passes the serialized value, as well as the destination Peer's ID (Peer B).

The worker does not need to await the result of this call, but it does because of xworker.sync.

2. Coincident handles the message by:
    - (Worker) JSON.stringify() -> Atomics.wait()
    - (Main) JSON.parse() -> Run the Main Thread function -> Atomics.notify()
3. Main thread on Peer A fires an event to notify PeerJS that it needs to transport the message and immediately returns in order to not block the Worker
4. PeerJS serializes the message with some added fields via MessagePack and transmits it to Peer B
5. Main thread on Peer B receives the message and forwards it to the appropriate worker callback via coincident
6. Coincident does the JSON.stringify() -> Atomics.wait() stuff (?) from main to worker on Peer B
7. Worker on Peer B handles the message by passing it to an asyncio coroutine that is waiting for it or stores it for later use

11.3 Implementation details

11.4 Performance analysis

11.5 Security analysis

11.5.1 Trust model

There still is a centralized service like in the Tailscale solution, but here it is self-deployed.

11.5.2 Identity

11.6 Usability analysis

Bibliography

[81] *Internet Protocol*. Request for Comments RFC 791. Internet Engineering Task Force, Sept. 1981. 51 pp. DOI: 10.17487/RFC0791. URL: https://datatracker.ietf.org/doc/rfc791 (visited on 05/14/2023) (cit. on p. 8).

[Anv19] Anvil. *Running Python in the Web Browser*. Anvil. 2019. URL: https://anvil.works/blog/python-in-the-browser-talk (visited on 01/30/2024) (cit. on pp. 38, 44).

[Bry24] Brython. *Brython Documentation*. Jan. 29, 2024. URL: https://brython.info/static_doc/3.12/en/intro.html (visited on 01/29/2024) (cit. on pp. 38, 44).

[Def22] Defined. *Nebula: Open Source Overlay Networking | Nebula Docs*. 2022. URL: https://docs.defined.net/docs/ (visited on 12/01/2022) (cit. on p. 20).

[Don17] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23160. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/ (visited on 09/28/2022) (cit. on p. 11).

[FK11] Sheila Frankel and Suresh Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. Request for Comments RFC 6071. Internet Engineering Task Force, Feb. 2011. 63 pp. DOI: 10.17487/RFC6071. URL: https://datatracker.ietf.org/doc/rfc6071 (visited on 05/28/2023) (cit. on p. 19).

[Fon22] Juan Font. *Juanfont/Headscale*. Dec. 6, 2022. URL: https://github.com/juanfont/headscale (visited on 12/06/2022) (cit. on p. 20).

[Goo23] Google. *IPv6 – Google*. May 21, 2023. URL: https://www.google.com/intl/en/ipv6/statistics.html (visited on 05/21/2023) (cit. on p. 13).

[JA07] Cullen Fluffy Jennings and Francois Audet. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. Request for Comments RFC 4787. Internet Engineering Task Force, Jan. 2007. 29 pp. DOI: 10.17487/RFC4787. URL: https://datatracker.ietf.org/doc/rfc4787 (visited on 05/26/2023) (cit. on p. 13).

[LY06] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Protocol Architecture*. Request for Comments RFC 4251. Internet Engineering Task Force, Jan. 2006. 30 pp. DOI: 10.17487/RFC4251. URL: https://datatracker.ietf.org/doc/rfc4251 (visited on 05/28/2023) (cit. on p. 7).

[Mic] MicroPython. *Overview — MicroPython Latest Documentation*. URL: https://docs.micropython.org/en/latest/ (visited on 01/30/2024) (cit. on pp. 38, 44).

[Moz19] Mozilla. *Pyodide: Bringing the Scientific Python Stack to the Browser – Mozilla Hacks - the Web Developer Blog.* Mozilla Hacks – the Web developer blog. Apr. 16, 2019. URL: https://hacks.mozilla.org/2019/04/pyodide-bringing-the-scientific-python-stack-to-the-browser (visited on 01/29/2024) (cit. on pp. 38, 44).

[Ope22] OpenVPN. *Community Resources.* OpenVPN. 2022. URL: https://openvpn.net/community-resources/ (visited on 11/30/2022) (cit. on p. 19).

[Per18] Trevor Perrin. "The Noise Protocol Framework". In: (July 1, 2018) (cit. on p. 12).

[Pet+20] Marc Petit-Huguenin et al. *Session Traversal Utilities for NAT (STUN).* Request for Comments RFC 8489. Internet Engineering Task Force, Feb. 2020. 67 pp. DOI: 10.17487/RFC8489. URL: https://datatracker.ietf.org/doc/rfc8489 (visited on 05/26/2023) (cit. on p. 14).

[Pyo] Pyodide. *Pyodide — Version 0.25.0.* URL: https://pyodide.org/en/stable/ (visited on 01/29/2024) (cit. on pp. 38, 44).

[Red+20] Tirumaleswar Reddy.K et al. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN).* Request for Comments RFC 8656. Internet Engineering Task Force, Feb. 2020. 79 pp. DOI: 10.17487/RFC8656. URL: https://datatracker.ietf.org/doc/rfc8656 (visited on 04/24/2023) (cit. on p. 14).

[Res00] Eric Rescorla. *HTTP Over TLS.* Request for Comments RFC 2818. Internet Engineering Task Force, May 2000. 7 pp. DOI: 10.17487/RFC2818. URL: https://datatracker.ietf.org/doc/rfc2818 (visited on 05/26/2023) (cit. on p. 10).

[Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3.* Request for Comments RFC 8446. Internet Engineering Task Force, Aug. 2018. 160 pp. DOI: 10.17487/RFC8446. URL: https://datatracker.ietf.org/doc/rfc8446 (visited on 05/26/2023) (cit. on p. 10).

[Rus] RustPython. *RustPython.* RustPython. URL: https://rustpython.github.io/ (visited on 01/30/2024) (cit. on pp. 38, 44).

[Sku] Skulpt. *Skulpt Docs.* URL: https://skulpt.org/docs/index.html (visited on 01/29/2024) (cit. on pp. 38, 44).

[Sli22] Guus Sliepen. *Tinc Docs.* Nov. 30, 2022. URL: https://www.tinc-vpn.org/docs/ (visited on 11/30/2022) (cit. on p. 20).

[Tai] Tailscale. *Tailscale.* Tailscale. URL: https://tailscale.com/kb/ (visited on 11/30/2022) (cit. on p. 20).

[Tai22] Tailscale. *DERP Servers.* Tailscale. Dec. 21, 2022. URL: https://tailscale.com/kb/1232/derp-servers/ (visited on 05/28/2023) (cit. on p. 14).

[Tra24] Transcrypt. *TranscryptOrg/Transcrypt.* Transcrypt - Python in the Browser, Jan. 28, 2024. URL: https://github.com/TranscryptOrg/Transcrypt (visited on 01/29/2024) (cit. on pp. 38, 44).

[WAS17] WASM. *WebAssembly.* 2017. URL: https://webassembly.org/ (visited on 01/29/2024) (cit. on pp. 38, 44).