Universal Getter - uget - Written in Oberon! Table of contents

Universal Getter - uget - Written in Oberon!

Libraries for: Gemini, Nex, Spartan, HTTP, HTTPS

Introduction

The goal of this project is to create a universal downloader capable of downloading—and in the future, uploading—files using the protocols: Nex, Spartan, Gemini, HTTP, and HTTPS. This tool is designed to resemble the well-known Unix command-line tool, *`wget`*, but with added support for a wider variety of protocols. To better understand the purpose of this project, let's explore the concept of communication protocols.

A communication protocol is a set of rules that defines how data is transmitted between devices or systems. These protocols govern how data is formatted, sent, and received, ensuring effective communication despite differences in hardware, software, or network infrastructures. Some of the most common protocols are HTTP, HTTPS, and TCP/IP (for internet connectivity).

The importance of this project lies in helping individuals who need to download files from less common communication protocols. While accessing and downloading files from HTTPS is easy with various browsers, it is not always the case with less popular protocols. By extending the project into a fully functional Gemini browser, I aim to provide people with the tools to enjoy the same ease of access, whether they're working with HTTPS or other protocols.

What Do We Need for a Universal Downloader?

To develop such a downloader, we first need to understand communication protocols, how they work, and how to handle them effectively. I will do my best to share the knowledge I've gained with the reader.

Client-Server Model

To establish a connection, we need entities between which the connection can exist. Typically, we don't use a connection without specifying the endpoints involved. This leads us to the concepts of *client* and *server*.

Client: A device or program that requests services or resources from a server.
Server: A device or program that provides services or resources to clients.

Clients initiate communication by sending requests, while servers respond with the requested resources. Together, clients and servers form the backbone of client-server architecture, enabling scalable communication between devices over a network.

In the context of this project, we will need both a client and a server to facilitate downloading (or, as I prefer to call it, "reading bytes"). We decided to use Michael Lazar's Spartan server-client model written in Python, and implement both the server and the client in our preferred language, Oberon. More details on why Oberon is a great choice will be provided in section #####.

Data Structures for Downloading Files

Next, we need an efficient data structure to read bytes from a server. Some protocols don't provide the content length of the file upfront, so we decided to use a dynamic array, where I spent a week developing a module specifically for dynamic arrays. This module allows us to dynamically allocate space, enabling the reading and writing of bytes from servers.

In the case of HTTPS, we sometimes know the content length in advance, meaning dynamic reallocation isn't always necessary. However, where content length is unknown, dynamic arrays prove useful for handling varying file sizes.

At this point, we are able to read bytes from servers and, eventually, download files.

How Does Downloading Work?

Downloading files involves building requests for each protocol and successfully reading files by dynamically allocating space. For HTTPS, where content length is known, we avoid dynamic reallocation, but for other protocols, like Spartan and Gemini, dynamic arrays are necessary.

All the tools, modules, and implementations developed in this project are open-source and designed to be useful for others. My long-term plan is to extend this project and eventually create a Gemini client browser, providing users with a simple yet effective way to interact with a wide range of protocols.

Let us dive into the implementation of the dynamic array module.

Dynamic array module implementation details

As mentioned, the project has a *`dynamicarray`* module that defines the main structure and functions for managing a dynamic array. The array is a flexible data structure that can grow as needed to accommodate more elements.

Let us look into the structure:

Types and Variables

- `dynamicarray`: This is a pointer to the `DynamicarrayDesc` record, which represents the array.
- `DynamicarrayDesc`: This record has the following fields:
  - `size`: The current number of elements in the array.
  - `capacity`: The maximum number of elements the array can hold before needing to resize.
  - `content`: A pointer to an array of characters (the dynamic content of the array).
  - `appender`: A procedure for appending data to the array.

Procedures

- `Init`: Initializes a dynamic array to a given size with null values.
- `Create`: Creates and initializes an empty dynamic array.
- `Resize`: Doubles the capacity of the dynamic array.
- `Append`: Appends a string to the array. If the array is empty, it allocates memory first; if not, it calls `ReallocAndAppend`.
- `ReallocAndAppend`: Handles memory reallocation and appends a string to the array.
- `writetofile`: Writes the content of the array to a file.

Dynamic Array Reallocation and Append

One of the key functions is `ReallocAndAppend`, which is responsible for reallocating memory for the dynamic array when it needs more space to append a new string. Let us take a look into some of the key parts of the code, which is the main Algorithm for reallocation.

```
PROCEDURE ReallocAndAppend(arr: dynamicarray; str: ARRAY OF CHAR);

VAR

  tmp: POINTER TO ARRAY OF CHAR; (as we can see we use a temporary pointer to an array)

BEGIN
```

```
NEW(tmp, LEN(arr^.content^));
```

Step 1: Create a temporary array with the same size as the original

```
copy(arr^.content^, tmp^);
```

Step 2: Copy content of the array to the temporary array

```
NEW(arr^.content, LEN(tmp^) + LEN(str) + 1);
```

Step 3: Allocate more space for the new content

^ the one extra length is for the null terminator.

```
arr^.size := LEN(str);
```

Step 4: Update the size to the length of the new string

```
arr^.capacity := arr^.capacity + LEN(str) + 1;
```

Step 5: Increase the capacity

```
copy(tmp^, arr^.content^);
```

Step 6: Copy the content back into the original array

```
copy(str, arr^.content^);
```

Step 7: Copy the new string into the array content

```
arr^.content^[LEN(tmp^) + LEN(str)] := 0X;
```

Step 8: Null-terminate the string to prevent data corruption.

```
END ReallocAndAppendinky;
```

Summary of the Code

1. **Temporary Array Creation**: A temporary pointer to an array `tmp` is created to hold the content of the current array. This is done to preserve the existing data before reallocating memory for the new content.
2. **Copy Existing Content**: The content of the original array is copied into the `tmp` array. This ensures that data is not lost during reallocation.
3. **Memory Reallocation**: The memory for the original array is reallocated with enough space to hold both the old content and the new string. The new capacity is updated.
4. **Copy Back the Old Content**: After reallocation, the old content from `tmp` is copied back into the newly allocated memory space.
5. **Append the New String**: The new string is appended to the array, and the array is null-terminated (with `0X`).

This method ensures that the dynamic array can grow dynamically to hold more data as needed.

Other Key Functions

`Append`: Decides whether to initialize a new array or call `ReallocAndAppend` based on whether the array is currently `NIL` or not.

`Init`: Initializes the array with null values.

`writetofile`: Writes the array content to a file by iterating through the array and writing each character.

The Spartan client module

Now let us dive into the implementation details of the Spartan client module. The `SPARTAN` module is a part of a system that provides functionality for creating a simple HTTP client that can connect to a server, send requests, and retrieve responses. It is structured using object-oriented programming (OOP) principles in Oberon, with a focus on low-level socket communication. Here's a breakdown of the key components and functions:

Data Structures

1. **`transport` and `transportDesc`**: These types represent the transport layer (network communication). The `transport` type is a pointer to a `transportDesc`, which acts as the base type for different transport implementations.
2. **`bsdSocket` and `bsdSocketDesc`**: The `bsdSocket` type extends `transportDesc` and is used to handle BSD-style sockets (network communication) for the client.
3. **`Client` and `ClientDesc`**: The `Client` type is a pointer to a `ClientDesc` record, which represents the client connection to a server. It contains various fields such as:
   - `host`, `port`, and `path`: The target server details.
   - `connectionFlag`: A boolean indicating whether the connection was successfully established.
   - *`rspnBody`*: The body of the server's response.
   - `header`: The headers of the response.
   - `statusCode`: The HTTP status code from the server response.
   - `eol` and `null`: Represent end-of-line characters and null characters.
   - `dyno`: A dynamic array that stores data.

Key Procedures

- **`clearState`**: Resets the state of a `Client` object, clearing response body, headers, status code, and other connection-related information.
- **`dynomaker`**: Initializes a new dynamic array. This is used to manage response data from the server.
- **`saver`**: Saves the content from the `dyno` array to a file. This allows the client to persist the response data.
- **`connector` and `disconnector`**: These procedures handle establishing and closing the connection to the server using the BSD socket.
- **`writer` and `reader`**: The `writer` sends the HTTP request to the server using the `Internet.Write` function. The `reader` reads the response from the server.
- **`buildRequest`**: Constructs the HTTP request string based on the `host`, `path`, and other necessary fields. This is sent to the server as part of the GET request.
- **`parseHeader`**: Parses the response header to extract useful information, such as the status code.

`get`: This is the main procedure for performing the HTTP GET request. It:

Clears the client state.
Establishes a connection to the server.
Builds and sends the request.
Reads the response and stores it in the *`rspnBody`*.
Disconnects from the server.

`Create`: A constructor for creating a new `Client` object. It initializes the necessary fields (e.g., `host`, `port`, `path`) and assigns the methods like `Connect`, `Disconnect`, `Write`, and `Read`.

Workflow Example

1. You create a `Client` instance by calling `Create` with server details (host, port, path).
2. The `get` procedure is called to send an HTTP request and retrieve the server's response.
3. The response data is stored in the *`rspnBody`* field of the `Client` object.
4. The connection is closed after the operation is complete.

This module is designed to be an efficient and lightweight HTTP client that operates at a low level, directly handling network communication using BSD sockets. It could be expanded further with more advanced features like handling other HTTP methods (POST, PUT, etc.) or managing timeouts and retries. Let us take a look at an important procedure. **`readTillCRLF` (Read until CRLF)**

Purpose: This procedure reads bytes from the socket until it encounters the CRLF (Carriage Return Line Feed) sequence (0x0D 0x0A). It does so byte by byte.

The buffer `buf` holds the bytes read.
The loop continues reading bytes until it encounters `0x0D` (CR, the first part of CRLF).
After reading a `0x0D`, it expects the next byte to be `0x0A` (LF), confirming the CRLF sequence.
Once CRLF is detected, the procedure stores the read header into `spartan^.header`.
It also calls `parseHeader` to process the header, which typically includes status codes and other metadata.

```
PROCEDURE readTillCRLF(spartan: Client); Without reading until CRLF is found we wouldn’t
be able to parse the header of the page, because we wouldn’t be able to know how many bytes
we need to read.

VAR
 buf: ARRAY 2 OF CHAR; For reading one character
 hdr: ARRAY 256 OF CHAR; For header
 b: BOOLEAN;
 i: INTEGER;

BEGIN

  COPY("", hdr); i := 0; First we initialize the header array
   REPEAT
        b := Internet.ReadBytes(spartan^.trn(bsdSocket).socket, buf, 1);
        IF b THEN
        IF buf[0] # 0DX THEN
        hdr[i] := buf[0];
        INC(i); We read until the character read was null terminator.
        END;
   ELSE

        Out.String("failed to read"); Out.Ln; HALT(2);
        END;
  UNTIL buf[0] = 0DX; (* Until CR (0x0D) is found *)
  hdr[i] := 0X;
  b := Internet.ReadBytes(spartan^.trn(bsdSocket).socket, buf, 1);

IF b THEN

IF buf[0] # 0AX THEN (* Expect LF (0x0A) next *)

  Out.String("0AX expected, got: "); Out.Int(ORD(buf[0]), 0);

  Out.Ln; HALT(1);
        END;
  ELSE
        Out.String("failed to read"); Out.Ln; HALT(3)
  END;

  NEW(spartan^.header, Strings.Length(hdr)+1);

  COPY(hdr, spartan^.header^);

  parseHeader(spartan, spartan^.header^); After we read, we put it all together. Refer to source
  code for more implementation details.

END readTillCRLF;
```

`get` (Main Get Request Handling)

Purpose: This is the main function, which is responsible for sending an HTTP request and receiving the response. It manages connection setup, request building, and response reading.

Explanation:

It first clears any previous state using `clearState`.

Establishes a connection using `spartan^.Connect`.

Sends an HTTP request using `writer`.

Then reads the response using `readTillCRLF` (to handle headers) and `readbuffer` (to handle the body).

The response body is stored in `spartan^.rspnBody`.

Please refer to the source code for the implementation details.

Summary of the SPARTAN module

Process:

Establish connection.

Send the request.

Read the response header (using `readTillCRLF`).

Read the body in chunks and appends it to `spartan^.dyno`.

After reading the full body, stores it in `spartan^.rspnBody`.

Connection Setup:

The `Create` procedure initializes the `Client` object, setting up the host, port, and path, and preparing socket connections.

The `Connect` and `Disconnect` procedures handle establishing and tearing down the network connection.

Request and Response Handling:

The `get` procedure constructs a request and sends it via `writer`.

The `readTillCRLF` function handles reading the headers from the response. ○

The `readbuffer` function reads the body of the response in chunks and stores it in `spartan^.dyno`.

Reading and Writing:

`readTillCRLF` ensures that the entire header is read before moving to the body, checking the CRLF sequence for proper formatting.

Once the header is processed, the body is read in chunks using `readbuffer`.

This Spartan module is a simple client capable of sending requests and receiving responses over a network socket, implementing basic HTTP-like interactions.

Breakdown of GEMINI module

As Gemini module is based off on Spartan module, let me explain their key differences. The main differences between the *Spartan* module and the *Gemini* module in the Oberon code lie in the functionality and the protocol handling for each. Here's a breakdown:

Protocol Handling

Gemini Module: The *Gemini* module is specifically designed for the Gemini protocol (`gemini://`). It includes functionality to create, connect, send requests, and handle responses over Gemini, which is a text-based protocol similar to HTTP but designed to be simpler and more focused on security.

Socket and TLS Handling

Spartan Module: Since the connection with spartan protocol is not encrypted, we don't implement encryption and certification.

Gemini Module: The *Gemini* module makes extensive use of **TLS (Transport Layer Security)** through the `mbedtls` library, which is used for secure communication. It defines a `TLSSocket` type and has procedures like `connect`, `disconnect`, `read`, and `write` specifically designed to manage the secure socket connection. The client also handles entropy, certificates, and personalization data for secure communication.

Status and Header Parsing:

Spartan Module: In the parseHeader procedure, since the structure is a lot simpler than gemini we don't need to obtain and sort the status codes, and the mimetype.

Gemini Module: The *Gemini* module includes specific procedures like `parseHeader` to parse the response headers, manage status codes, and extract the mimetype. This is critical for processing the response according to the Gemini protocol's rules.

Personalization and Certificate Management:

Spartan Module: No personalization and encryption.

Gemini Module: The *Gemini* module includes fields and procedures for handling personalization data (device-specific identifiers) and setting the certificate path (`setcertpath`, `setpers`). It manages these data fields in the `ClientDesc` structure to configure secure TLS connections.

Certificate Management in the Gemini Module

The module manages certificates primarily for the TLS connection setup, and the code for handling certificates includes functions like setting the certificate path, initializing the client with certificate details, and setting personalization data (like device-specific identifiers). Let us summarize.

setcertpath: This procedure takes a path to a certificate file and copies it into the `crtpath` field of the `Client` record.

setpers: This procedure allows setting personalized data, which can include device-specific identifiers. The data is copied into the `pers` field of the `Client`.

init: In the initialization procedure, the `mbedtls.init` function is called to set up the TLS context using the entropy, certificate data, and other details stored in the `Client`.

connector: This procedure sets up a connection, specifically handling the TLS connection initialization using the provided certificate details (like entropy, certificates, etc.).

disconnector: This procedure handles disconnecting the TLS connection and cleans up any resources used.

These procedures allow for configuring the necessary certificates, personalization data, and ensuring secure connections with the server using TLS.

Core Gemini Code Section (File Handling and Data Saving Logic)

This part of the Gemini protocol implementation handles receiving data from a Gemini server, giving the downloaded file the correct extension, and writing the data to a file.
1. ensureFileExtension

This procedure makes sure the file has the correct extension based on the MIME type. It loops over the filename to see if there is already a . character (dotPos):

FOR i := 0 TO len-1 DO IF filename[i] = '.' THEN dotPos := i; END; END;

If the extension exists but is wrong, or doesn't exist at all, it appends the correct one:
IF len + Strings.Length(extension) < LEN(filename) THEN Strings.Append(extension, filename);

For example, a PDF file would get a .pdf extension added.
2. copyFile

This makes a second copy of the file using a simple read/write loop. It reads each character from one file and writes it to another:

WHILE ~r1.eof DO Files.Read(r1, ch); IF ~r1.eof THEN Files.Write(r2, ch); END; END;

This is used to make a backup or fallback file like defaultname.gmi.

3. Get

This is the main download procedure. First, it builds the Gemini request and sends it:

req := buildRequest(gemini);

Then it writes the request to the server:

b := gemini.Write(gemini, req^);

After receiving the response, it checks for a successful status and extracts the MIME type:

IF Strings.Pos("application/pdf", gemini^.mimetype^, 0) >= 0 THEN ensureFileExtension(saveFilename, ".pdf");

Now it starts reading the response body using mbedtls.read. It reads data into a buffer:

r := mbedtls.read(gemini^.trn(TLSSocket).sslContext, readBuf, LEN(readBuf) - 1);

Then it writes the bytes to a file one by one:

FOR i := 0 TO readLen - 1 DO Files.Write(rider, readBuf[i]); END;

This loop runs until there is no more data or it fails too many times. Then, if the main filename is different from the default one, it makes a fallback copy:

IF saveFilename # defName THEN copyFile(saveFilename, defName);

Summary

This code is a complete and clean implementation of downloading and saving Gemini protocol data. It does not depend on external libraries (except for mbedTLS), and it writes files directly. It supports changing file extensions based on MIME type, making it easy to open files in appropriate apps.

It shows how a low-level, protocol-focused program works: connecting, reading bytes, handling MIME types, and writing output. This kind of code is useful when building tools like minimal browsers, command-line clients, or protocol testers.

uget: Main procedure logic

The uget.Mod module is a program designed to interact with different protocols such as Spartan, Nex, Gemini, HTTP, and HTTPS to retrieve files from remote servers. The program uses command-line arguments to determine the protocol, host, path, and additional options like user authentication for HTTPS connections.

Key Data Structures and Variables:

ARG Record: Holds details about the URL, including the protocol, host, path, and port.

```
TYPE
  ARG = RECORD
  protocol: ARRAY 64 OF CHAR;
  host: ARRAY 64 OF CHAR;
  path: ARRAY 246 OF CHAR;
  port: ARRAY 5 OF CHAR;
  prothost: strTypes.pstring;
END;
```

  arg: An instance of the ARG record holds the parsed command line arguments for the protocol, host, path, and port.

Connect Procedures: These procedures handle the connection to the respective protocol (Spartan, Gemini, Nex, HTTP, HTTPS) and retrieve/save the file.

ConnectSpartan, ConnectGemini, ConnectNex, ConnectHttp, ConnectHttps, and ConnectHttpsAuth are specialized to handle their respective protocols and support file retrieval. Let us look at an example of the ConnectGemini procedure:

```
PROCEDURE ConnectGemini;
VAR
  g: GEMINI.Client;
  name, answer: strTypes.pstring; pstring is a pointer to an array of char.
  Dynamicarray.dynamicarray; Here, we create the dynamic array structure from the
dynamicarray module.
BEGIN
  g := GEMINI.Create(arg.host, arg.port, arg.path); Out.String("trying to initialize
socket"); Out.Ln; g.Init(g);
Out.String("left init in connect gemini"); Out.Ln; answer := g.Get(g);
Saving is handled as a separate procedure in the gemini module, which will later be
imported to uget as well.
END ConnectGemini;
```

parseCommand: Parses the command line arguments into the ARG record, identifying the protocol, host, and path, and assigns default ports for each protocol.

The Main procedure in the uget.Mod module is the central logic of the program. It handles the execution flow based on the command-line arguments and invokes the appropriate connection procedure based on the protocol specified by the user. Let's break down its components:

```
PROCEDURE Main;
VAR
cmd: strTypes.pstring; The command that we input as a user
BEGIN
  IF Args.argc = 2 THEN
    cmd := getCommand(1);
     parseCommand(cmd^);
  IF arg.protocol = "spartan" THEN
  ConnectSpartan;
  ELSIF arg.protocol = "gemini" THEN
ConnectGemini;
……..
ELSE
  Out.String("unknown protocol: '"); Out.String(arg.protocol); Out.Char("'"); Out.Ln;
END;
ELSE
          Out.String("Usage: protocol://host/path [-u username -p password]" ); Out.Ln;
The following notation indicates that the input -u username -p password is an optional
implemented feature where it is handled  if authentication is required from the server.
END; END Main;
```

The program first checks if there are exactly two command-line arguments, meaning Args.argc = 2. This means the user has provided a protocol URL, for example something like https://example.com/file. If this condition is met, then the following line is used: cmd := getCommand(1). That means the first argument is retrieved, which is the URL or protocol with path.

Then the procedure parseCommand(cmd^) is called. This procedure is responsible for parsing the URL. It extracts things like the protocol (for example, http, https, gemini), the host, the path, and the port from the given URL. Depending on which protocol is extracted, a specific connection procedure is called. For example, if the protocol is "spartan", then ConnectSpartan is used. If it’s "nex", then ConnectNex is used. If it’s "gemini", then ConnectGemini is used, and so on. If the protocol is not recognized by the program, it prints an error message and stops.

If there are more than two arguments, meaning Args.argc > 2, then the program assumes that the user wants to provide authentication details like a username and

password. It checks if the second argument is "-u", which would mean the user is trying to enter a username. If so, then cmd := getCommand(4) is used to retrieve the URL or protocol. Then the program checks if cmd^ = "-p", which would mean the password is coming next. The username and password are then taken using usr := getCommand(3) and pwd := getCommand(5). If the password is an empty string, a warning message is shown to the user. After parsing the URL again, if the extracted protocol is HTTPS, the program will call ConnectHttpsAuth(usr^, pwd^) to handle the authentication. But if the protocol is not HTTPS, the program will print a message saying that authentication is not implemented for that protocol.
If the user gives fewer than two or more than five arguments, the program prints a usage message to help the user understand the correct way to use it.

Example usages of the program include
uget protocol://example.com/file.format
and with authentication:
uget protocol://example.com/file.format -u username -p password

Conclusion

In conclusion, the Universal Getter (uget) project was developed to offer users a unified, lightweight tool for downloading files from a wide range of internet protocols. At its core, the project demonstrates how minimal, protocol-aware design can be combined with efficient memory management using dynamic arrays. By supporting HTTP, HTTPS, Gemini, Spartan, and Nex, it proves that even under a constrained and minimalistic environment, one can build a modular system that can adapt to different communication models, whether verbose and modern like HTTP or minimal and strict like Spartan.

The tool not only handles basic file retrieval, but also implements authentication support for HTTPS, laying the groundwork for future security-aware extensions. Error handling and argument parsing were carefully structured to allow for flexible use cases while keeping the logic clean and easy to follow. More than just a downloader, uget reflects an architectural idea: that protocols can be treated equally under a single client abstraction, with only the connection procedures swapped as needed.

The project remains a work in progress, with ongoing ideas to evolve it into a full Gemini browser, and port it to Oberon Operating system. This would expand its capabilities beyond downloading, pushing it toward interactive, lightweight browsing experiences on low-resource systems. More broadly, uget serves as a proof of concept for building simple yet powerful tools outside the mainstream software ecosystem without relying on heavyweight libraries or bloated frameworks. I hope this tool—and the philosophy behind it, can inspire further experiments in clean, protocol-level computing, and be of practical use to anyone interested in alternative internet protocols or minimal client-side software.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Norayr Chillingaryan, whose unwavering support and guidance were invaluable throughout the entire course of this capstone project. His dedication went far beyond expectations, as he generously devoted countless hours to meet with me weekly after classes. His insightful feedback, patience, and encouragement significantly shaped the quality and direction of this work. Without his expert mentorship and continuous assistance, this project would not have been possible. I am profoundly grateful for his time, effort, and commitment.

I also wish to extend my sincere thanks to the Computer Science Department for providing a supportive academic environment and access to essential resources that facilitated my research and development process. The knowledge and skills imparted by the faculty have been instrumental in enabling me to undertake and complete this project successfully.

Finally, I am thankful to all my classmates and peers who provided encouragement and shared valuable discussions that enriched my learning experience. This project marks a significant milestone in my academic journey, and I appreciate everyone who contributed directly or indirectly to its completion.

References

Flounder. “Gemini Protocol FAQ.” Gemini Flounder Online, https://gemini.flounder.online/docs/faq.gmi. Accessed 15 Dec. 2024.

Geminiprotocol.net. “Software Using the Gemini Protocol.” Gemini Protocol, https://geminiprotocol.net/software/. Accessed 22 Feb. 2025.

Lazar, Michael. Spartan Protocol Implementation on GitHub. https://github.com/michael-lazar/spartan. Accessed 14 Jan. 2025.

LWN.net. “The Trouble with HTTP.” Linux Weekly News, 9 June 2021, https://lwn.net/Articles/845446/. Accessed 3 Mar. 2025.

MIT Center for Bits and Atoms. “Internet Protocols.” FAB Class Notes, https://fab.cba.mit.edu/classes/961.04/people/neil/ip.pdf. Accessed 12 Dec. 2024.

Mozz. “Smolnet Portal.” Portal.mozz.us, https://portal.mozz.us/about. Accessed 9 May. 2025.

Smolnet Portal. Gemini Page by Michael Mozz, gemini://mozz.us/. Accessed 9 May. 2025.

Stanford University. “Lecture 9: Networks and Protocols.” CS106E: Lecture Notes, https://web.stanford.edu/class/cs106e/lectureNotes/L09NNetworksProtocols.pdf. Accessed 22 Feb. 2025.

The Gemini Mailing List. Orbitalfox Archives, https://web.archive.org/web/20211020132800/https://lists.orbitalfox.eu/listinfo/gemini. Accessed 3 Mar. 2025.