SimpleX Chat: The Definitive Guide to Self-Hosted SMP and XFTP (and Making Nginx Digest Connections Without SNI)

SimpleX Chat: The Definitive Guide to Self-Hosted SMP and XFTP (and Making Nginx Digest Connections Without SNI)

There's no point in beating around the bush: most "secure" messaging apps (yes, we are talking about Signal or Session too) still have a structural Achilles' heel. Whether it's a phone number, a central user ID, or a directory server, there is always something that traces communications back to a specific identity.

Then came **SimpleX Chat**.

SimpleX is radically different: it has no user IDs, no accounts, and no centralized metadata. But how does it really work under the hood? And above all, how do you set up a serious self-hosted infrastructure with **SMP** (Simple Message Protocol) and **XFTP** (File Transfer Protocol) servers without losing your mind over the quirks of the protocol and Nginx?

In this article, we take a comprehensive look at the architecture of SimpleX, explaining how it works in theory and how to set it up in practice with Docker Compose and Nginx.

Anatomy and Philosophy of SimpleX: How Does the Network Work?

To understand why SimpleX's configuration has its own unique rules, we must understand its core philosophy: **the total elimination of network-level identities**.

Unidirectional Queues (Simplex Queues)

Unlike XMPP, Matrix, or Signal, where you have a fixed address or public key to which others send messages, in SimpleX nothing of the sort exists. Communication occurs via **temporary unidirectional queues** (hence the name *SimpleX*, which in telecommunications engineering refers to one-way transmission).

When you start a 1:1 chat with a contact, two separate queues are created on different servers:

1. **Your receiving queue**: This is created on **your** SMP server (e.g., `smp.example.org`). You tell your friend: *"If you want to write to me, deposit the messages in queue X on smp.example.org"*. Your client will connect to your server to download the messages.
2. **Their receiving queue**: This is created on **their** SMP server. When you write to them, your client connects to their server and deposits the message there.

This means that to talk to a person, **you use your server to receive and their server to send**. This structural asymmetry makes it impossible for a single SMP server to reconstruct the social graph (who is talking to whom).

When is your server used?

Understanding this schema makes it easy to identify when your machine (VPS/Server) is actually called into play:

- **1:1 Chat (Receiving)**: You receive messages from your contacts on `smp.example.org`.
- **Groups and Channels you create**: When you create a group or channel, the central queues of that group are hosted on your server. All participants will send and download messages from there.
- **File Exchange (XFTP)**: If you send images or files using your XFTP server (`xftp.example.org`), the encrypted files are temporarily uploaded there, and your contacts will download them from there.

If instead you join a group created by someone else or reply in a 1:1 chat, your client will write to third-party servers.

Public Channels and the Battery Drain Problem (Chat Relays)

In classic groups without intermediaries, if you send a message to a group of 100 people, your phone must encrypt the message 100 times and send it to 100 different queues on potentially different servers. This process (*client-side fan-out*), combined with the fact that SimpleX uses fixed **16 KiB** packets to hide text length, would destroy your smartphone's battery and bandwidth (you would have to upload 1.6 MB for a simple *"Hello"*).

For public channels and large groups, SimpleX introduced **Chat Relays** (or *channel relays*):

- Your client sends the post **only once** (16 KiB) to the Chat Relay.
- The Chat Relay (running on a server with unlimited bandwidth and CPU) handles the *server-side fan-out*, distributing the message to all

members.
- *Note on encryption*: Since channels are public spaces, the content is not encrypted end-to-end in the classic sense. The relay can read the messages it propagates (though they are digitally signed by the author to prevent tampering), but it still does not know the identity (IP or account) of either the publisher or the subscribers.

Clearnet / Tor (Darknet) Combinations in 1:1 and Group Chats

This is where people often get confused about what should run over Tor and what in the clear. Since in SimpleX there is no federation between SMP servers (each server is an island that doesn't talk to the others), interaction between Clearnet and Tor users follows very precise logical rules.

Here is how the various combinations work:

Case 1: 1:1 Chat between an Onion user and a Clearnet user

Suppose you receive messages on `my_server.onion` and your friend receives them on `their_server.com` (clearnet).

- **You write to them**: Your client (which has Tor active) connects to `their_server.com` passing through Tor exit nodes. This works without issues.
- **They write to you**: Their client must deposit messages on `my_server.onion`. To do this, **your friend must enable the Tor proxy (e.g., Orbot or SOCKS5) in their SimpleX app settings**. They don't need to change their receiving server (which remains in the clear), but they must give their client the ability to browse the Tor network to write to your queue. If they don't do this, they won't be able to send you messages.

Case 2: Joining an Onion Group

If you see an invite link to a group pointing to an onion server (e.g., `smp://...@xyz.onion/...`):

- All of the group's queues reside on that Tor server.
- To participate, **you just need to enable Tor (the SOCKS proxy) in your app settings**. Your client will use Tor to speak with the group server.
- **Your personal SMP server has nothing to do with this**: it does not need to be an onion service, and it will not receive any traffic from this group.

Case 3: Can I use my clearnet server to download messages from an onion group?

- **No.** Since SMP servers do not talk to each other (no federation), you cannot ask your server `smp.example.org` to bridge and download messages from the onion server. Your client must connect directly to the group's server via Tor.

The Real Technical Problem: SNI and Multiplexing on Port 443

Let's understand how traffic flows at the network level before touching the code.

1. **SMP Server**: Runs on native TLS. It uses port **5223** for direct connections from SimpleX clients (messages). This port must be exposed directly to the world, without Nginx in between. Additionally, SMP serves a courtesy static web page on port **443** (which we will map to local port **8444**). This page requires a valid Let's Encrypt certificate.
2. **XFTP Server**: Also runs on native TLS on port **443** (which we will map to local port **5443**). XFTP does not use commercial Let's Encrypt certificates; it generates a self-signed key pair on its first boot, and security is guaranteed by the client verifying the server's fingerprint included in the connection address.
3. **Nginx**: If we only have one public IP and want to serve other HTTPS sites on port **443** in addition to the SMP page and the XFTP server, we cannot use a classic HTTP reverse proxy (`proxy_pass` inside the `http` block). We must use Nginx's **stream** module to perform TCP multiplexing based on SNI (*Server Name Indication*).

And here is the trap.

To maximize privacy and prevent anyone analyzing network traffic (such as state censors or snooping ISPs) from knowing we are using SimpleX, **SimpleX clients do not send SNI when connecting to XFTP**.

When Nginx receives a TLS connection without SNI, the `$ssl_preread_server_name` variable remains empty (`""`). If Nginx's map doesn't explicitly handle the empty case, the connection falls to the default backend (which usually points to the standard HTTP block on `8443`). The HTTP web server receives unexpected raw TLS traffic, freaks out, throws a TLS error, and drops the connection. You won't see anything in Nginx's logs, and you will lose your mind trying to understand why file transfers aren't starting.

The solution? Map the empty string `""` directly to the XFTP server.

Step 1: Obtain an RSA 4096 Certificate (ECDSA Sucks for SMP)

Another undocumented headache: SimpleX's SMP server **does not support ECDSA certificates** (which are now the default generated by Certbot). If you pass it an ECDSA certificate, the container crashes silently or rejects connections. It strictly demands **4096-bit RSA keys**.

Force Certbot to generate the correct certificate for our SMP domain (e.g., `smp.example.org`):

```
sudo certbot certonly --nginx -d smp.example.org --key-type rsa --rsa-key-size 4096
```

Verify the certificate using this command:

```
openssl x509 -in /etc/letsencrypt/live/smp.example.org/fullchain.pem -text -noout | grep "Public Key Algorithm"
```

It must reply with `RSA-Encryption` and a key size of `4096 bit`.

Step 2: Docker Compose Configuration

Move to our deployment directory (e.g., `/opt/smp/`) and create the folder structure:

```
mkdir -p data/smp/config data/smp/logs data/xftp/config data/xftp/logs data/xftp/files
```

Now create the `docker-compose.yml` file:

```
services:
  smp-server:
    image: simplexchat/smp-server:latest
    container_name: smp-server
    restart: unless-stopped
    env_file: [.env.smp]
    ports:
      - "5223:5223"
      - "127.0.0.1:8444:443"
    volumes:
      - ./data/smp/config:/etc/opt/simplex:z
      - ./data/smp/logs:/var/opt/simplex:z
      - /etc/letsencrypt/live/smp.example.org/fullchain.pem:/certificates/smp.example.org.crt:ro
      - /etc/letsencrypt/live/smp.example.org/privkey.pem:/certificates/smp.example.org.key:ro
```

```
  xftp-server:
    image: simplexchat/xftp-server:latest
    container_name: xftp-server
    restart: unless-stopped
    env_file: [.env.xftp]
    ports:
      - "127.0.0.1:5443:443"
    volumes:
      - ./data/xftp/config:/etc/opt/simplex-xftp:z
      - ./data/xftp/logs:/var/opt/simplex-xftp:z
      - ./data/xftp/files:/srv/xftp:z
```

Create the environment files to pass configurations to the containers.

For `.env.smp`:

```
ADDR_PORT=smp.example.org:443
# You can add other optional variables to limit connections or enable logs
```

For `.env.xftp`:

```
ADDR_PORT=xftp.example.org:443
```

Step 3: Configure Nginx with the Empty SNI Trick

On Debian/Ubuntu distributions, Nginx's `stream` module might not be enabled by default. First, create a symbolic link to load it on Nginx startup:

```
sudo ln -s /usr/share/nginx/modules-available/mod-stream.conf /etc/nginx/modules-enabled/50-mod-stream.conf
```

Now open the main Nginx configuration file (`/etc/nginx/nginx.conf`) and add the `stream {}` block **outside** the existing `http {}` block:

```
stream {
    log_format stream_log '$remote_addr [$time_local] "$ssl_preread_server_name" -> $upstream_addr';
    access_log /var/log/nginx/stream.log stream_log;

    map $ssl_preread_server_name $backend {
        smp.example.org   127.0.0.1:8444;
        xftp.example.org  127.0.0.1:5443;
        ""                127.0.0.1:5443;   # <--- THE MAGIC: SimpleX clients without SNI land on XFTP
```

```
        default             127.0.0.1:8443;   # All other normal web sites on the
server
    }

    server {
        listen      443;
        listen      [::]:443;
        ssl_preread on;
        proxy_pass  $backend;
        proxy_connect_timeout 10s;
        proxy_timeout         600s;
    }
}
```

> *[!IMPORTANT] Since Nginx TCP Stream is bound to public port `443`, **you must modify all of your existing HTTP configuration files** (those in `/etc/nginx/sites-enabled/`). Replace any occurrence of `listen 443 ssl;` with `listen 8443 ssl;`. This way, when a browser requests one of your normal websites sending the correct SNI, Nginx Stream will read the hostname, find no match for SimpleX, apply the `default` rule, and forward the connection to `127.0.0.1:8443`, where the standard HTTP block will serve the web page normally.*

For port 80 management and Certbot certificate renewals, create a minimal configuration file for SMP in `/etc/nginx/sites-enabled/smp.example.org.conf`:

```
server {
    listen 80;
    listen [::]:80;
    server_name smp.example.org;

    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    location / {
        return 301 https://$host$request_uri;
    }
}
```

Restart Nginx to apply changes:

```
sudo nginx -t && sudo systemctl restart nginx
```

Step 4: Renewal Hook for Let's Encrypt

Since the SMP server mounts Let's Encrypt certificates directly from the filesystem and performs TLS handshake inside the container, it won't notice when a certificate is renewed until it is restarted.

Prevent connections from failing after 90 days by creating a Certbot deploy hook that restarts the container automatically only when the certificate is actually renewed:

```
sudo cat > /etc/letsencrypt/renewal-hooks/deploy/restart-smp.sh << 'EOF'
#!/bin/bash
docker restart smp-server
EOF

sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/restart-smp.sh
```

Step 5: Startup and Extracting SimpleX Addresses

Start the services:

```
docker compose up -d
```

Now we must extract the connection strings to pass to the clients.

SMP Address

Read SMP's logs to retrieve the public key fingerprint:

```
docker compose logs smp-server | grep Fingerprint
```

You should see output like:

```
smp-server   | Fingerprint: _Kv-q8JOE_Hfx5Nf_fymYBGqzF-3ToKJ2ac0KZLz-RA=
```

Compose the SMP address for the client: `smp://_Kv-q8JOE_Hfx5Nf_fymYBGqzF-3ToKJ2ac0KZLz-RA=@smp.example.org`

XFTP Address

Do the same with XFTP's logs:

```
docker compose logs xftp-server | grep -E "Fingerprint|address"
```

Look for the automatically generated public key. The final address to enter in SimpleX Chat for the file server will be like: `xftp://sUbQhuyLhyVcsMTgst0pkY8h75WcIIkP7ebuWO9IOsE=@xftp.example.org`

Verification and Connection Testing

We can test that Nginx is doing the routing correctly directly from an external machine using `openssl`.

To test if the routing with SNI (smp) responds correctly:

```
openssl s_client -connect smp.example.org:443 -servername smp.example.org
```

This must show the chain of your Let's Encrypt certificate.

To test the connection without SNI (which simulates the SimpleX client connecting to XFTP):

```
openssl s_client -connect xftp.example.org:443
```

This must show the chain of the self-signed certificate generated internally by XFTP (e.g., `CN = xftp.example.org`, signed by `SMP server CA` or similar).

If both test respond correctly, congratulations: you have a federated, autonomous, and extremely secure SimpleX Chat infrastructure, perfectly integrated with your web reverse proxy.