

BIP324 Proxy

Easy integration of v2 transport protocol for light clients



State of BIP324 adoption

- available since Bitcoin Core v26.0 (`-v2transport`, default-off)
- (very likely) default-on starting from Bitcoin Core v27.0 🍦
- more and more v2 peers available

- This is great, but...
 - ... what about other clients?

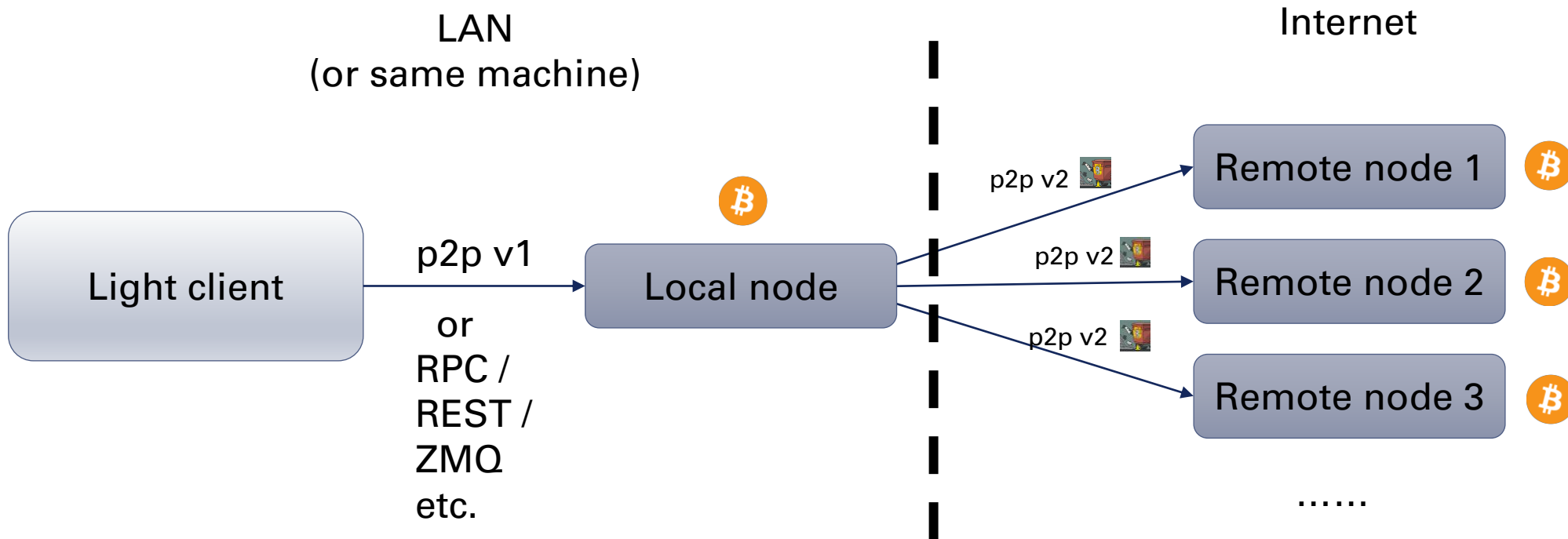
- Long-term vision: every Bitcoin P2P participant runs v2 transport

Implementing BIP324 – how hard could it be?

- Lots of new dependencies for cryptographical primitives:
 ElligatorSwift, ECDH (secp256k1-ellswift), HKDF-SHA256,
 ChaCha20, Poly1305, FSChaCha20, FSChaCha20Poly1305...
- Non-trivial v2 handshake phase (e.g. byte-by-byte reception loop needed for terminator detection due to non-deterministic size of garbage), v1 reconnection logic
- Realistically, this will still take some time to be available in alternative implementations, especially for light clients
- Can we enable that easier by offering an external language-independent module that talks v2 transport?

Light client – ideal scenario

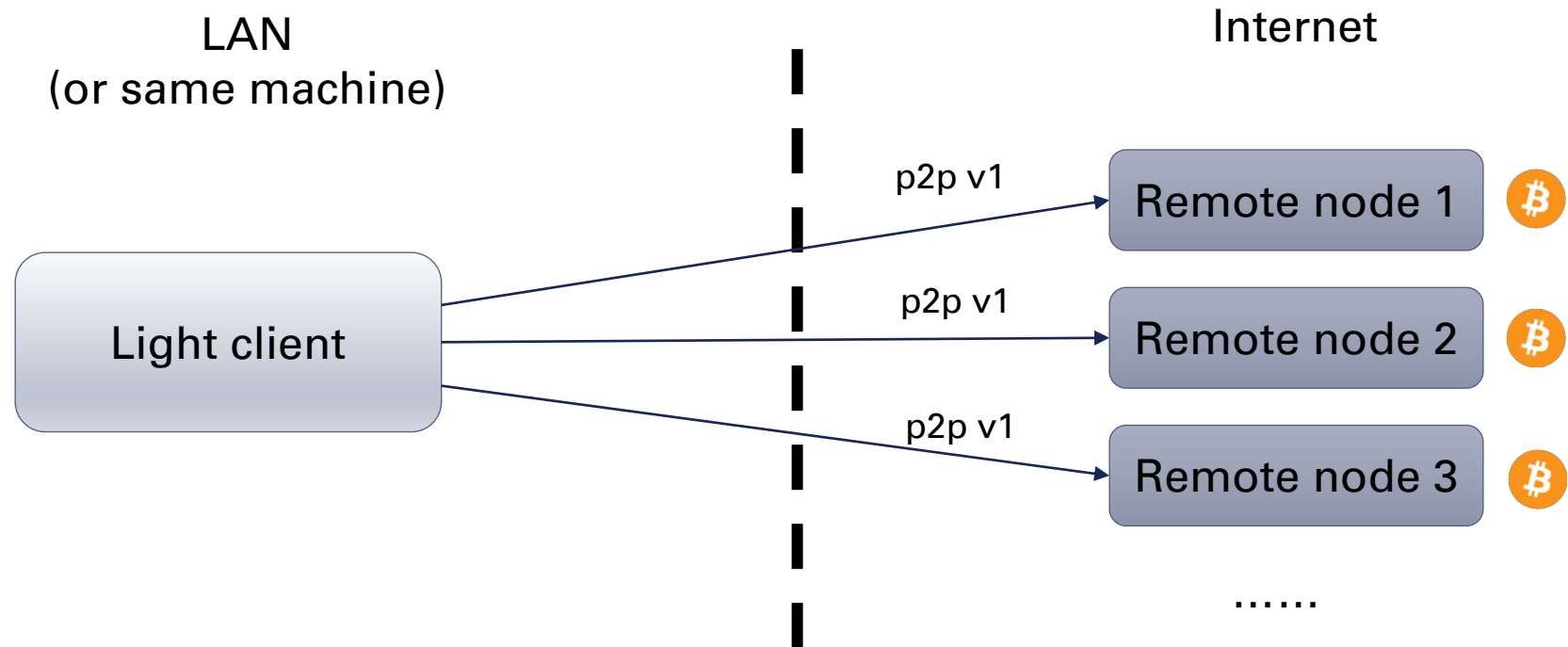
- Users run their own node, light clients only connect to that node



- Ideally, everyone runs their own node, but unrealistic (due to resource limits and increased setup effort)

Light client – typical scenario

- Light clients connect directly to remote nodes



 ISPs see P2P msgs in cleartext ✗

Introducing BIP324 Proxy 1/2

- Basic idea: run a local process with the sole purpose of translating between v1 and v2 transport protocols in each direction
- proxy listens on localhost (127.0.0.1:1324) for incoming v1 connections, establishes v2 connection to remote node
- remote peer destination is found in initial v1 VERSION message

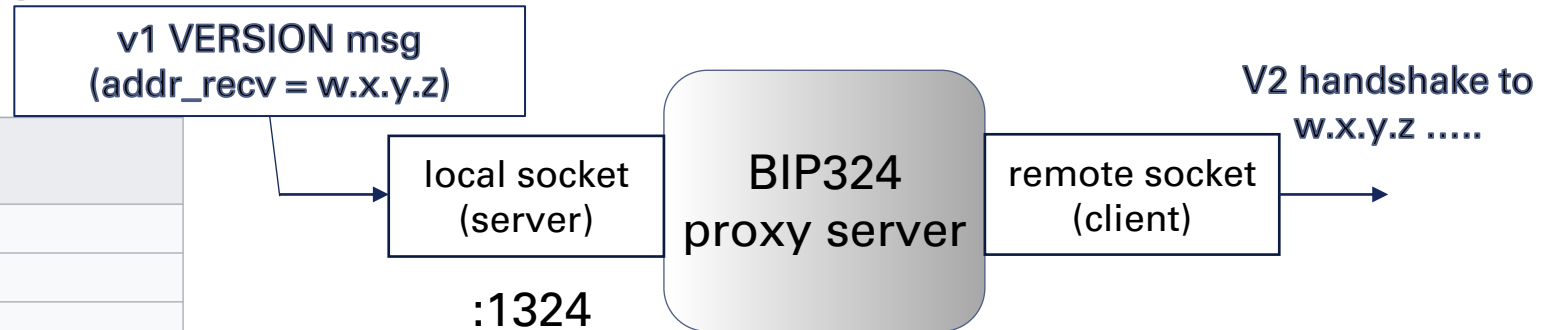
Message types

version

When a node creates an outgoing connection, it will immediately [advertise](#) its version. The remote node will respond with its version. No further communication is possible until both peers have exchanged their version.

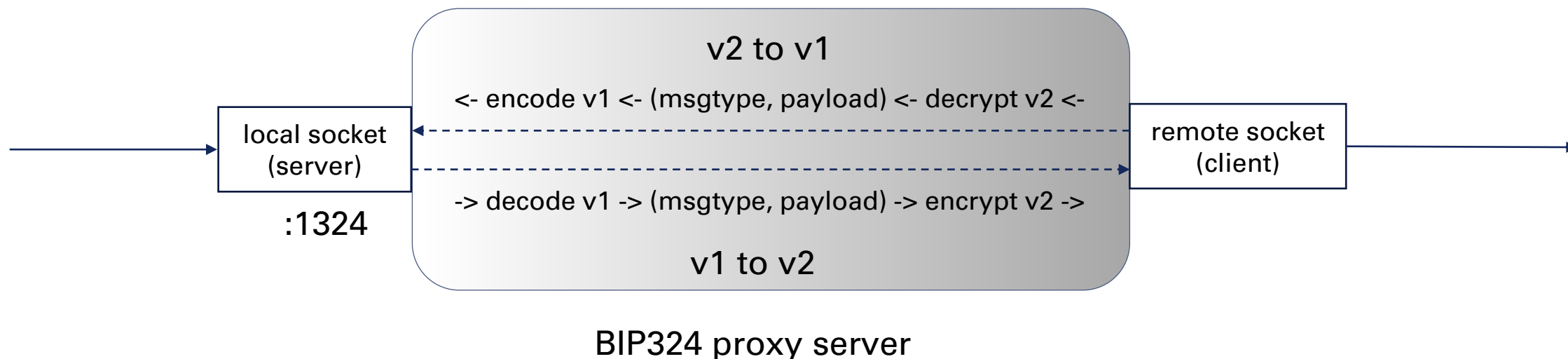
Payload:

Field Size	Description	Data type	Comments
4	version	int32_t	Identifies protocol version being used by the node
8	services	uint64_t	bitfield of features to be enabled for this connection
8	timestamp	int64_t	standard UNIX timestamp in seconds
26	addr_rcv	net_addr	The network address of the node receiving this message



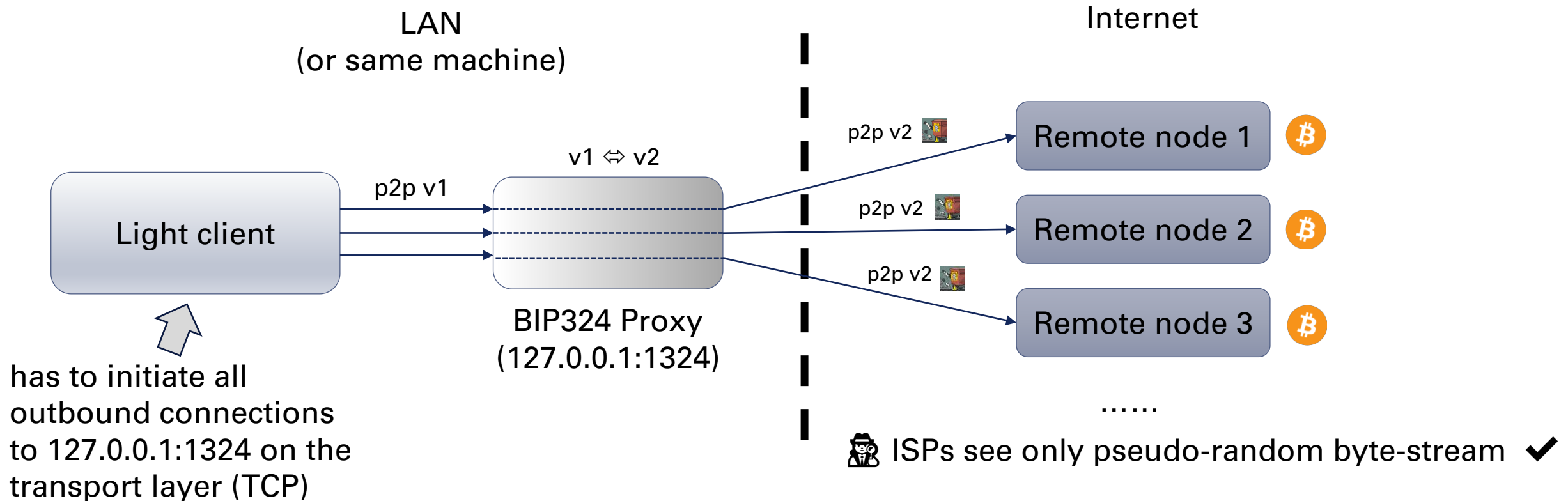
Introducing BIP324 Proxy 2/2

- light clients connects to the proxy instead of the actual remote node for every outbound connection
- important: modify the destination *only* on TCP level! (clients should still have the illusion that they are directly connected to the remote)
- no configuration of the proxy needed
- (almost) no changes needed in the client, see later slides



Light client – BIP324 proxy scenario

- Light clients connect to proxy, which transforms packets to v2



BIP324 Proxy – Implementation

- written in Python3, no external dependencies, ~750 LOC (mostly cryptographic primitives, taken from the Bitcoin Core test framework [PR #24748]; Kudos to stratospher and sipa)
- creates and listens on a server socket on :1324
- for each incoming connection on that socket:
 - create a new BIP324 proxy handler thread
 - receive v1 VERSION message
 - connect to remote address (contained in VERSION message)
 - perform v2 handshake
 - send earlier received VERSION message
 - main loop: translate between v1 <-> v2 (see next slide)
- reconnect with v1 is supported, but off by default (we want v2 connections exclusively)

BIP324 Proxy – Main loop (per thread)

```
while True:
    r, _, _ = select([local_socket, remote_socket], [], [])
    if local_socket in r: # [local] v1 ---> v2 [remote]
        msgtype, payload = recv_v1_message(local_socket)
        send_v2_message(remote_socket, send_l, send_p, msgtype, payload)
        log_recv('-->', msgtype, payload)
    if remote_socket in r: # [local] v1 <--- v2 [remote]
        msgtype, payload = recv_v2_message(remote_socket, recv_l, recv_p)
        send_v1_message(local_socket, msgtype, payload)
        log_recv('<--', msgtype, payload)
```

„on either side, unpack/decrypt the received message to (msgtype, payload) and forward it repacked/encrypted with the *other* transport version to the opposite side“

How to take use of BIP324 Proxy

In general:

1. Start proxy: `$./bip324-proxy.py`
2. Redirect all Bitcoin P2P connections to :1324

Two possible redirect solutions for step 2, each with their pros/cons:

1. user-space approach: patch the client
2. kernel-space approach: use a packet filter

Redirection – user space approach

- Patch the light client to initiate all outbound connections (TCP client socket `connect (...)` syscall) to `127.0.0.1:1324`
- Minimum-working patch is trivial, usually a single-line change
- Ideally, provide an option like `-bip324proxy=...`
(in theory, the proxy could also run on another machine)

- Pros:
 - Should be independent of the used operating system
 - No root privileges needed
- Cons:
 - Individual adaption needed for each client

Some examples of patches on the next slides...

Light client example #1: Nakamoto

- <https://github.com/cloudhead/nakamoto> 
- started in 2020 by Alexis Sellier (cloudhead)
- Language: Rust
- „Nakamoto is a privacy-preserving Bitcoin light-client implementation in Rust, with a focus on low resource utilization, modularity and security.“
- Commitment to implement v2 transport soon:



README



MIT license



Status

Peer-to-peer layer encryption (BIP 324), available in Bitcoin Core since v26.0, will also be implemented in Nakamoto soon.


Light client example #1: Nakamoto

The Patch:

```
diff --git a/net/poll/src/reactor.rs b/net/poll/src/reactor.rs
index e8eb547..ce9fdf7 100644
--- a/net/poll/src/reactor.rs
+++ b/net/poll/src/reactor.rs
@@ -465,7 +465,7 @@ fn dial(addr: &net::SocketAddr) -> Result<net::TcpStream, io::Error> {
     sock.set_write_timeout(Some(WRITE_TIMEOUT))?;
     sock.set_nonblocking(true)?;

-    match sock.connect(&(*addr).into()) {
+    match sock.connect(&net::SocketAddr::from(([127,0,0,1], 1324)).into()) {
         Ok(()) => {}
         Err(e) if e.raw_os_error() == Some(libc::EINPROGRESS) => {}
         Err(e) if e.raw_os_error() == Some(libc::EALREADY) => {
```

Light client example #2: Neutrino


- <https://github.com/lightninglabs/neutrino> 
- started in 2017 by Lightning Labs / Roasbeef
- First client implementation of BIP 157 / BIP 158
(„Client Side Block Filtering“ / „Compact Block Filters for Light Clients“)
- Language: Go
- „Neutrino is a Bitcoin light client written in Go and designed with mobile Lightning Network clients in mind. It uses a new proposal for compact block filters to minimize bandwidth and storage use on the client side, while attempting to preserve privacy and minimize processor load on full nodes serving light clients.“

Light client example #2: Neutrino

The Patch:

```
diff --git a/neutrino.go b/neutrino.go
index ef36f42..ab6c0cb 100644
--- a/neutrino.go
+++ b/neutrino.go
@@ -702,7 +702,7 @@ func NewChainService(cfg Config) (*ChainService, error) {
     dialer = cfg.Dialer
 } else {
     dialer = func(addr net.Addr) (net.Conn, error) {
-        return net.Dial(addr.Network(), addr.String())
+        return net.Dial(addr.Network(), "127.0.0.1:1324")
     }
 }
```


Light client example #3: bcoin

- <https://github.com/bcoin-org/bcoin> 
- started in 2014 by Fedor Indutny and Christopher Jeffrey, maintained now by Matthew Zipkin (pinheadmz)
- Language: JavaScript (and C/C++)
- Fastest alt client sync time after Bitcoin Core (according to Jameson Lopp's benchmarks)
- „Bcoin is an alternative implementation of the Bitcoin protocol, written in JavaScript and C/C++ for Node.js. Bcoin is well tested and aware of all known consensus rules. It is currently used in production as the consensus backend and wallet system for purse.io.“

Light client example #3: bcoin

The Patch:

```
diff --git a/lib/net/peer.js b/lib/net/peer.js
index 2271e789..918101c3 100644
--- a/lib/net/peer.js
+++ b/lib/net/peer.js
@@ -303,7 +303,7 @@ class Peer extends EventEmitter {
  connect(addr) {
    assert(!this.socket);

-   const socket = this.options.createSocket(addr.port, addr.host);
+   const socket = this.options.createSocket(1324, '127.0.0.1');

    this.address = addr;
    this.outbound = true;
```

„Light client“ example #4: Bitcoin Core 25.1 (latest release w/o BIP324 support)

The Patch:

```
diff --git a/src/net.cpp b/src/net.cpp
index 903fedb2fb..64d9b4d7d0 100644
--- a/src/net.cpp
+++ b/src/net.cpp
@@ -534,8 +534,8 @@ CNode* CConnman::ConnectNode(CAddress addrConnect, const char *pszDest, bool fCo
     if (!sock) {
         return nullptr;
     }
-    connected = ConnectSocketDirectly(addrConnect, *sock, nConnectTimeout,
-                                     conn_type == ConnectionType::MANUAL);
+    connected = ConnectSocketDirectly(CAddress{LookupNumeric("127.0.0.1", 1324), NODE_NONE},
+                                     *sock, nConnectTimeout, conn_type == ConnectionType::MANUAL);
 }
 if (!proxyConnectionFailed) {
     // If a connection to the node was attempted, and failure (if any) is not caused by a problem connecting to
```

Live demo

Let's try using BIP324 Proxy with one of the clients (using user-space redirection approach)

Redirection – kernel space approach

- Instead of touching the client, let the operating system handle the redirection directly on the network level
- Using a packet filter („firewall“) with modification rules

- Pros:
 - Light client doesn't need to be modified
- Cons:
 - Root privileges needed
 - OS / packet filter dependent
 - Potential conflict with other packet filter rules?
 - Need a criteria to only apply the rule to Bitcoin P2P connections (e.g. destination port 8333)

Redirection example using iptables

- iptables / Netfilter: Linux packet filter since version 2.4.x, started in 1998 (!) by Rusty Russell, author of c-lightning

```
# groupadd --system redirect_to_bip324-proxy
# iptables -t nat -A OUTPUT -p tcp --dport 8333 \
-m owner --gid-owner redirect_to_bip324-proxy \
-j DNAT --to-destination 127.0.0.1:1324

# sudo -g redirect_to_bip324-proxy <light-client-binary> ...
```

Drawbacks of BIP324 Proxy

- higher CPU usage / latency (have to talk v1+v2 in each direction, and switch between user-space and kernel-space twice per packet)
- more file descriptors used (3 sockets + 2 streams per connection)
- client has no awareness of the proxy; if a connection is closed, was that initiated by the remote node or the proxy? (e.g. due to a bug?)
- single point of failure: if the proxy crashes, all connections are lost
- divergence in TX/RX network statistics (v2 features a mild bandwidth reduction due to a more efficient msgtype encoding)
- can only pass in v1 messages and nothing else (due the missing mandatory VERSION message, we wouldn't know where to send them to), already-v2 connections are rejected

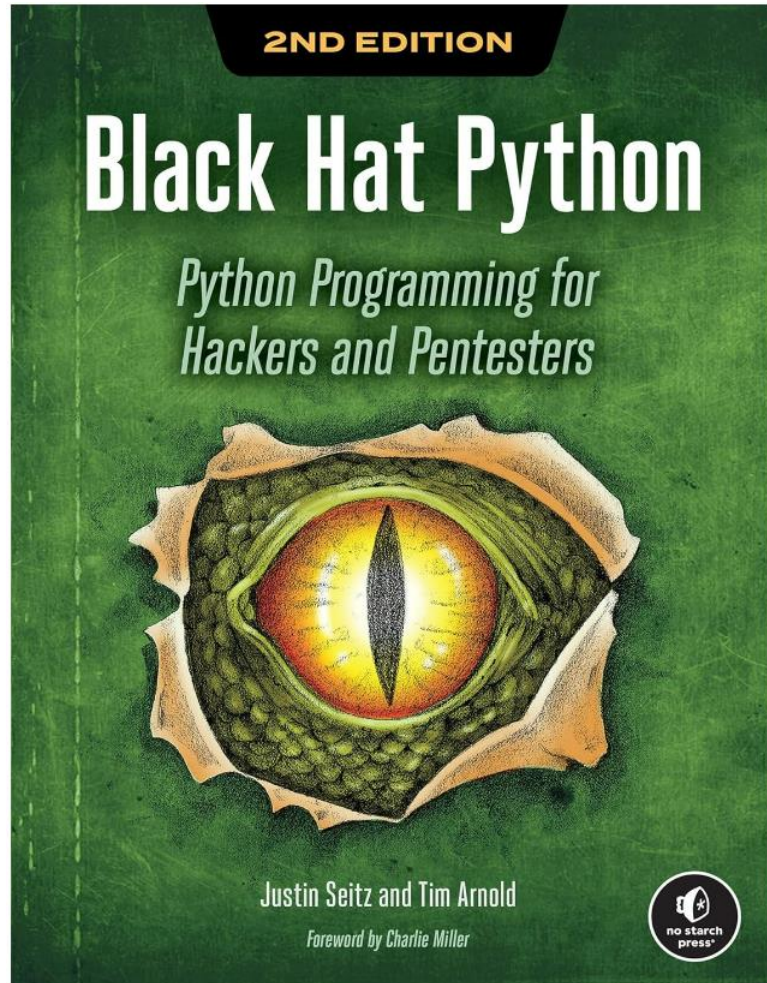
Benefits of BIP324 Proxy

- Easy v2 integration of transport protocol for light clients
- Compact reference implementation of BIP324 w/o distractions, executable version of the BIP pseudo-code
- Playground for new features on top of BIP324, e.g.:
 - Traffic shapability with decoy packets (ignored by receiver)
 - Bypass firewall rules by setting the EISwift-pubkey to a fixed prefix, to mimic other protocols (e.g. HTTP)
- Personal benefitts (to be done):
 - Opportunity to learn Rust
 - Opportunity to learn more about packet filters and related technologies (pf(Sense), iptables vs. nftables, eBPF, XDP, ...)

TODOs

- Complete rewrite in Rust
(the ChaCha20 implementation in Python is painfully slow)
- IPv6 support
- Automatic network detection (currently only mainnet supported)
- Add user agent postfix (e.g. „/nakamoto:0.3.0/**bip324proxy**/“)
- Collect TX/RX network statistics on each side, to prove „mild bandwidth reduction“ of BIP324 empirically
- Create small dashboard (show connection summary, network statistics, BIP324 session ids etc.)
- Investigate inbound connections support via reverse proxy
- Find a fancy name for the project (any ideas?)

Recommended reading, if you want to implement your own TCP proxy...



2	BASIC NETWORKING TOOLS	9
	Python Networking in a Paragraph	10
	TCP Client	10
	UDP Client	11
	TCP Server	12
	Replacing Netcat	13
	Kicking the Tires	17
	Building a TCP Proxy	19
	Kicking the Tires	24
	SSH with Paramiko	26
	Kicking the Tires	30
	SSH Tunneling	30
	Kicking the Tires	34

Find the project at

 <https://github.com/theStack/bip324-proxy>

Questions, comments?