chaincode

Deploying Rust-Lightning in the Wild

Antoine Riard, LN-Conf 2019, Berlin

History

A flexible Lightning implementation headed to be full-spec compliant.

A library, no built-in daemon, you decide your runtime.

Started by TheBlueMatt beginning of 2018.

Laid on top of the rust-bitcoin ecosystem.

Why Lightning ?

Reaching Bitcoin old promise (?) of instant payments ?

Enabling fancy financial contracts ?

Building streams of micro-transactions ?

Building a network of money pipes

But we don’t know yet the whole *how* of the pipes.

No more we know *what* are we going to flow through them.

And *where* they are going to be deployed and by *who*.

How ?

Multi-party channels ?

Channel factories ?

Eltoo-based update mechanism ?

Coinjoin-like splicing/multiple-party funding ?

What ?

Discreet Log Contracts ?

Hodl-invoices ?

Stuckless payments ?

Payment-point based escrows ?

Where ?

On broadband Internet ?

Nodes in a meshnet ?

Devices in an electricity grid ?

Hardware Security Modules ?

Who ?

Exchange/Merchants.

Consumers.

Routing nodes.

Lightning liquidity providers.

Rust-Lightning : a modular library

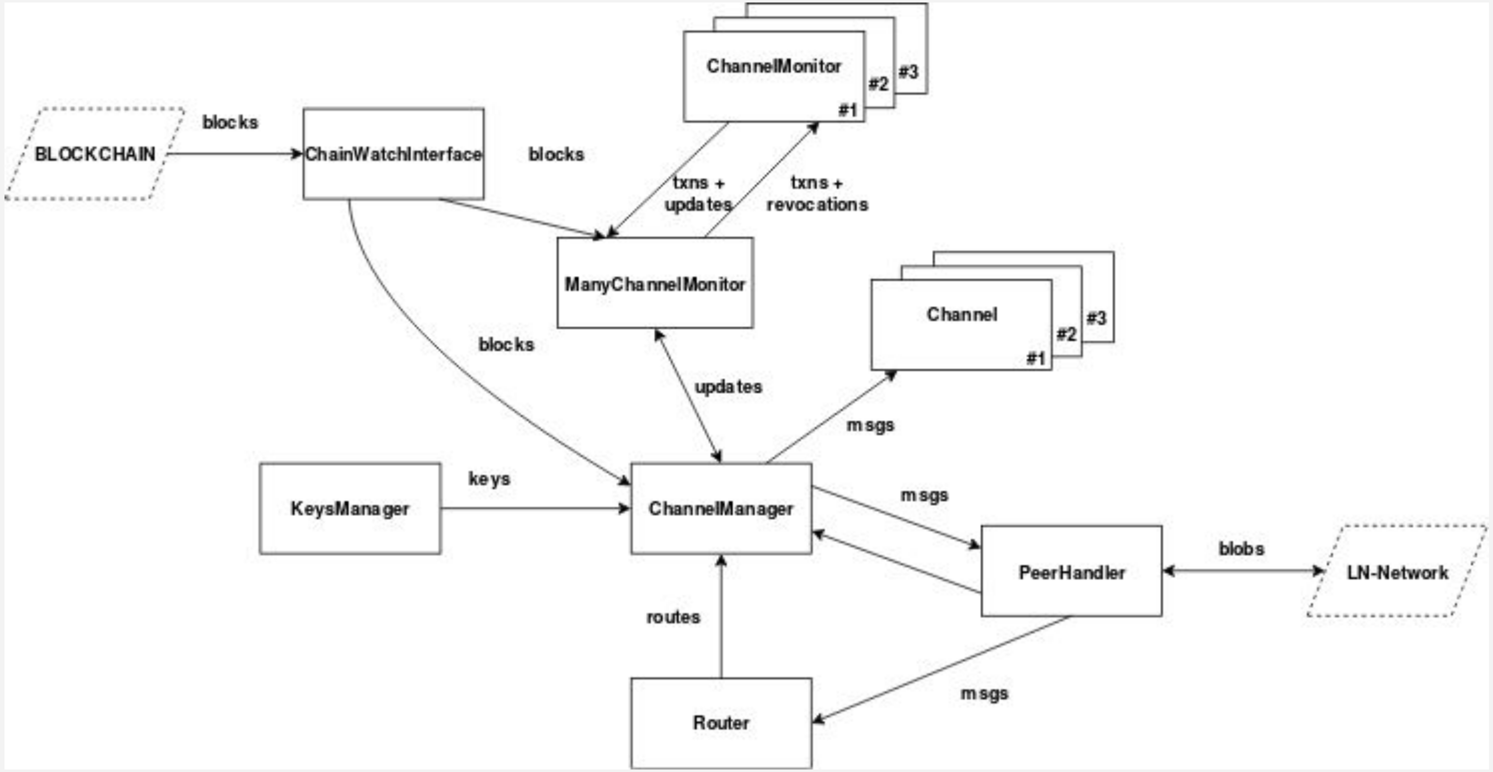
Simple, composable APIs to drive spec-compliant Lightning components.

Focus on fuzzing, testing and reviewing.

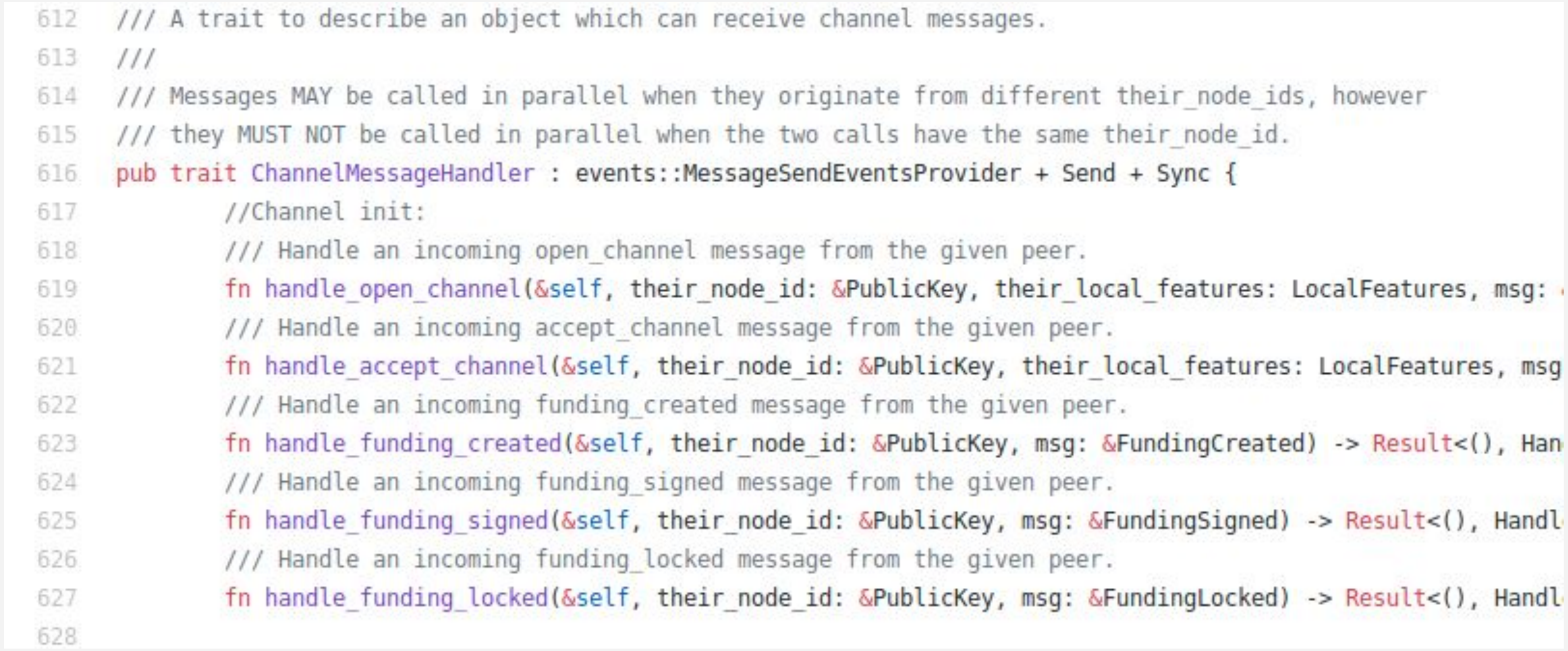
Should be easy to plug modules or integrate with existing architecture.

No dependency !

Anatomy of Rust-Lightning

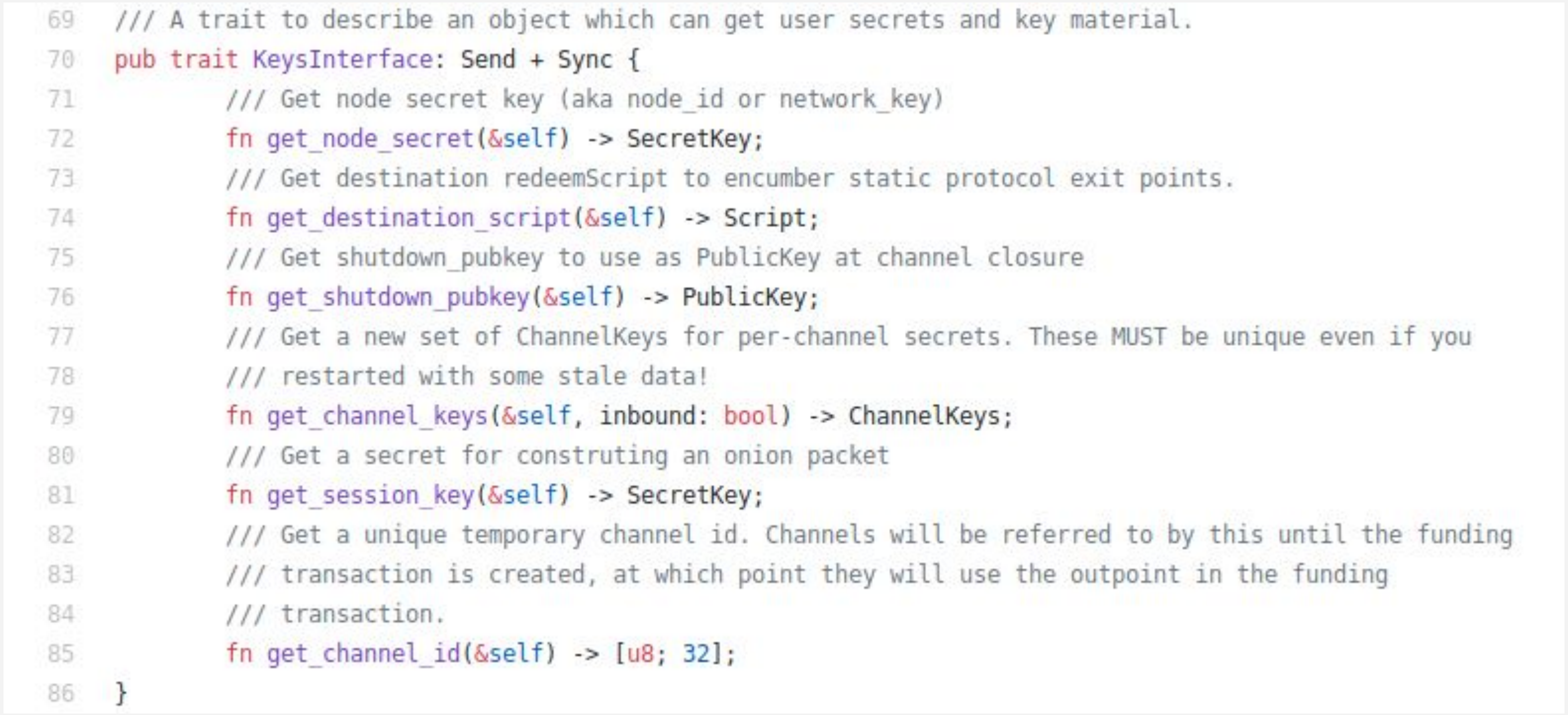


Anatomy of a LN node : ChannelManager



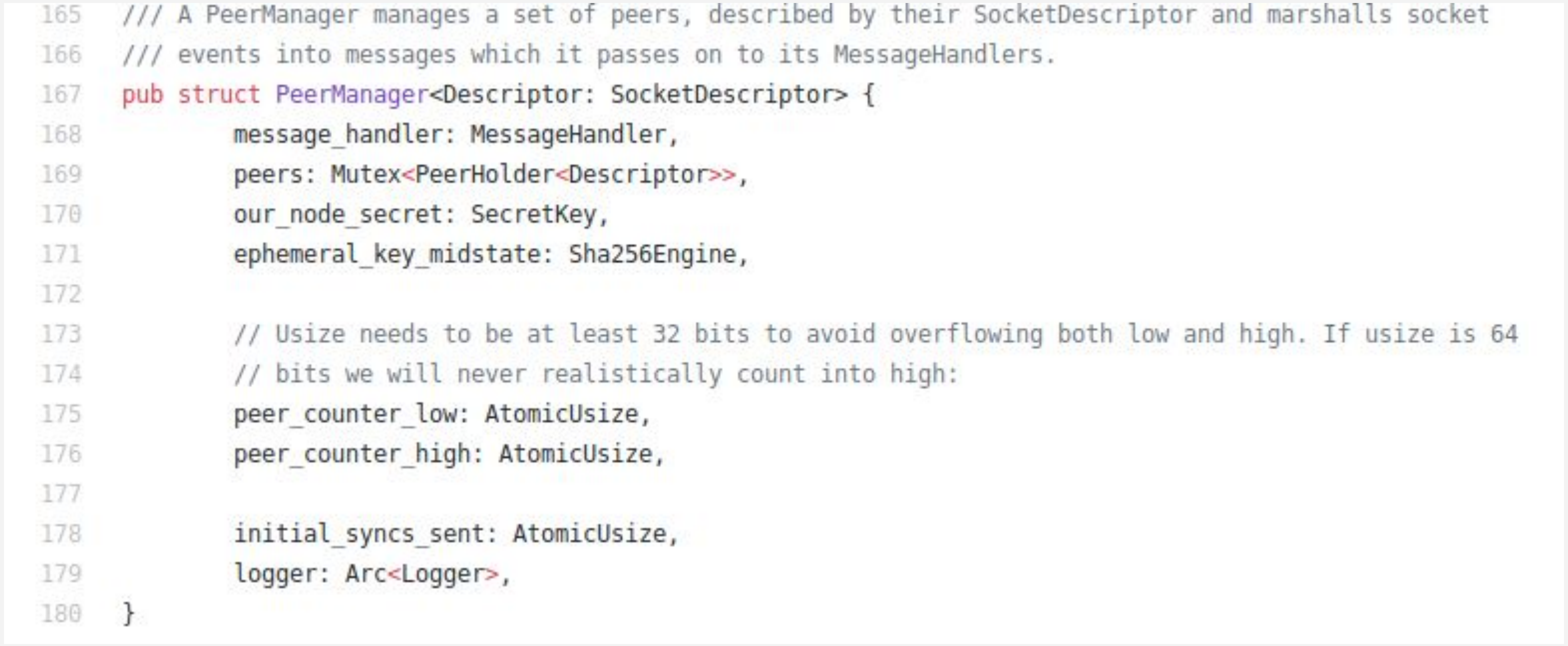
```
/// A trait to describe an object which can receive channel messages.
///
/// Messages MAY be called in parallel when they originate from different their_node_ids, however
/// they MUST NOT be called in parallel when the two calls have the same their_node_id.
pub trait ChannelMessageHandler : events::MessageSendEventsProvider + Send + Sync {
        //Channel init:
        /// Handle an incoming open_channel message from the given peer.
        fn handle_open_channel(&self, their_node_id: &PublicKey, their_local_features: LocalFeatures, msg:
        /// Handle an incoming accept_channel message from the given peer.
        fn handle_accept_channel(&self, their_node_id: &PublicKey, their_local_features: LocalFeatures, msg
        /// Handle an incoming funding_created message from the given peer.
        fn handle_funding_created(&self, their_node_id: &PublicKey, msg: &FundingCreated) -> Result<(), Han
        /// Handle an incoming funding_signed message from the given peer.
        fn handle_funding_signed(&self, their_node_id: &PublicKey, msg: &FundingSigned) -> Result<(), Handl
        /// Handle an incoming funding_locked message from the given peer.
        fn handle_funding_locked(&self, their_node_id: &PublicKey, msg: &FundingLocked) -> Result<(), Handl

```

Anatomy of a LN node : KeysManager



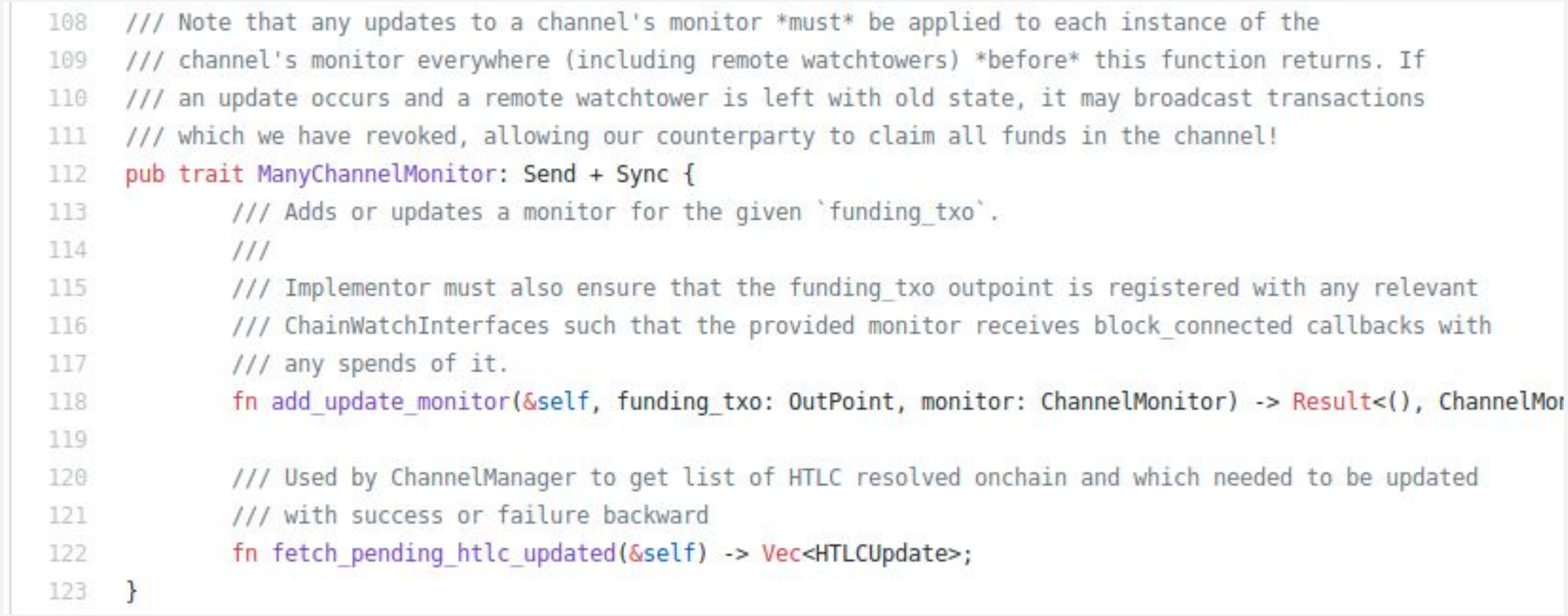
```
/// A trait to describe an object which can get user secrets and key material.
pub trait KeysInterface: Send + Sync {
        /// Get node secret key (aka node_id or network_key)
        fn get_node_secret(&self) -> SecretKey;
        /// Get destination redeemScript to encumber static protocol exit points.
        fn get_destination_script(&self) -> Script;
        /// Get shutdown_pubkey to use as PublicKey at channel closure
        fn get_shutdown_pubkey(&self) -> PublicKey;
        /// Get a new set of ChannelKeys for per-channel secrets. These MUST be unique even if you
        /// restarted with some stale data!
        fn get_channel_keys(&self, inbound: bool) -> ChannelKeys;
        /// Get a secret for construting an onion packet
        fn get_session_key(&self) -> SecretKey;
        /// Get a unique temporary channel id. Channels will be referred to by this until the funding
        /// transaction is created, at which point they will use the outpoint in the funding
        /// transaction.
        fn get_channel_id(&self) -> [u8; 32];
}
```

Anatomy of a LN node : PeerManager



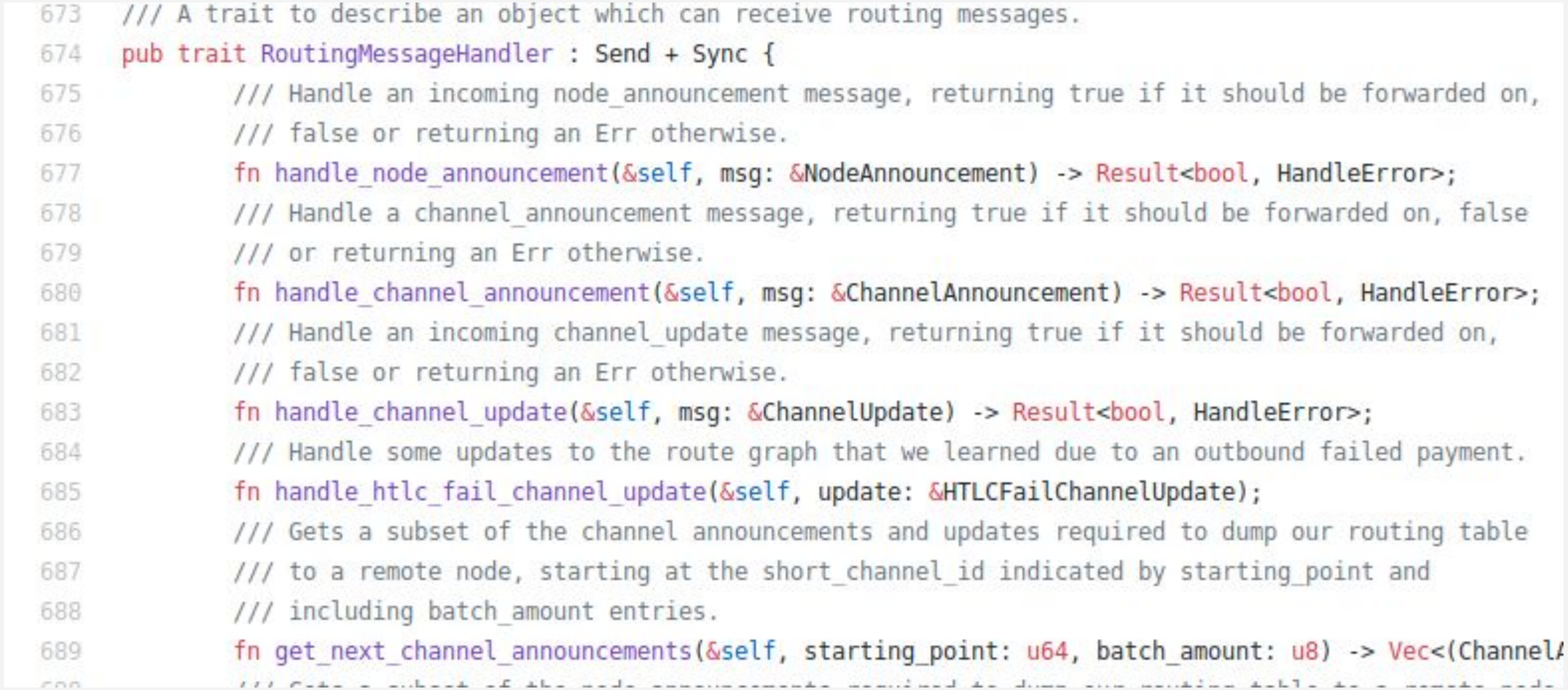
```rust
/// A PeerManager manages a set of peers, described by their SocketDescriptor and marshalls socket
/// events into messages which it passes on to its MessageHandlers.
pub struct PeerManager<Descriptor: SocketDescriptor> {
	message_handler: MessageHandler,
	peers: Mutex<PeerHolder<Descriptor>>,
	our_node_secret: SecretKey,
	ephemeral_key_midstate: Sha256Engine,

	// Usize needs to be at least 32 bits to avoid overflowing both low and high. If usize is 64
	// bits we will never realistically count into high:
	peer_counter_low: AtomicUsize,
	peer_counter_high: AtomicUsize,

	initial_syncs_sent: AtomicUsize,
	logger: Arc<Logger>,
}
```

Anatomy of a LN node : ManyChannelMonitor



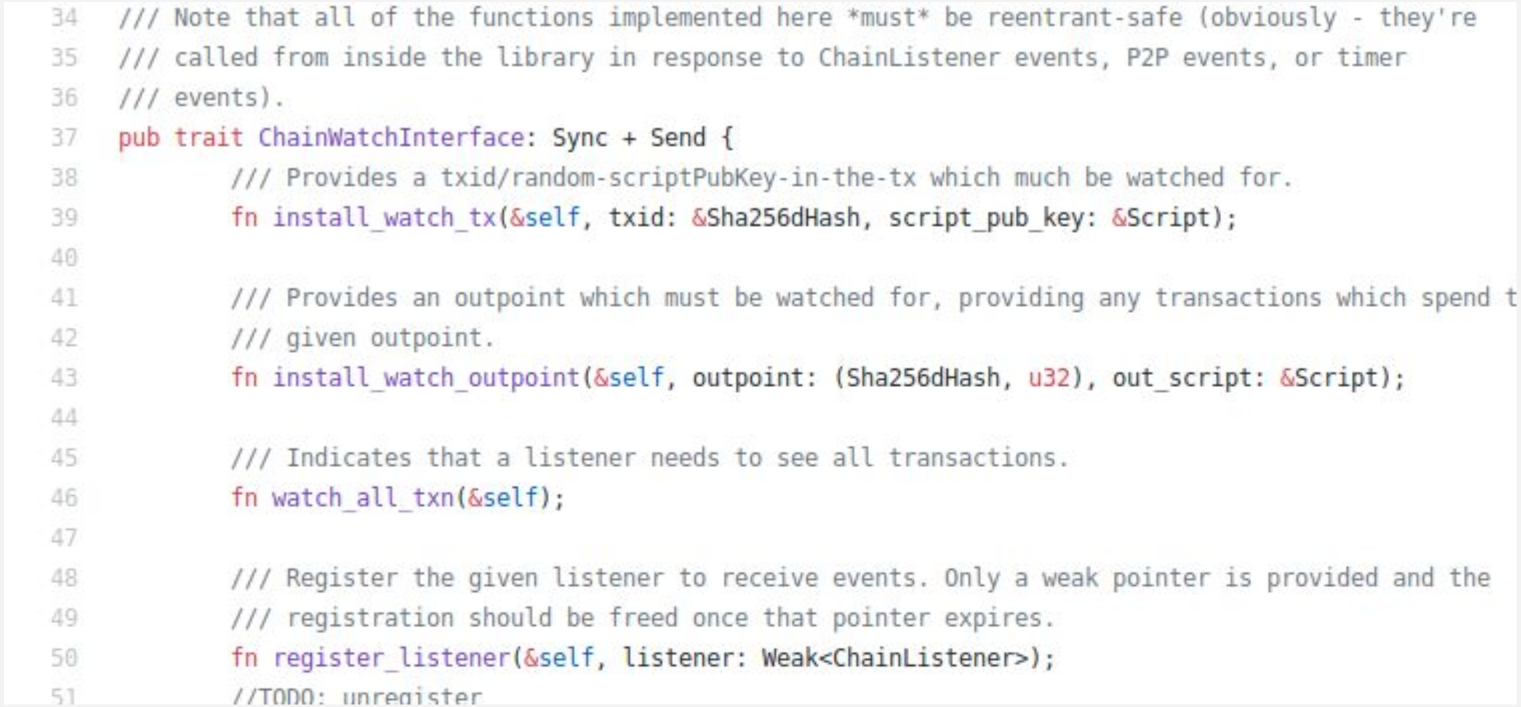
```rust
/// Note that any updates to a channel's monitor *must* be applied to each instance of the
/// channel's monitor everywhere (including remote watchtowers) *before* this function returns. If
/// an update occurs and a remote watchtower is left with old state, it may broadcast transactions
/// which we have revoked, allowing our counterparty to claim all funds in the channel!
pub trait ManyChannelMonitor: Send + Sync {
	/// Adds or updates a monitor for the given `funding_txo`.
	///
	/// Implementor must also ensure that the funding_txo outpoint is registered with any relevant
	/// ChainWatchInterfaces such that the provided monitor receives block_connected callbacks with
	/// any spends of it.
	fn add_update_monitor(&self, funding_txo: OutPoint, monitor: ChannelMonitor) -> Result<(), ChannelMon

	/// Used by ChannelManager to get list of HTLC resolved onchain and which needed to be updated
	/// with success or failure backward
	fn fetch_pending_htlc_updated(&self) -> Vec<HTLCUpdate>;
}
```

Anatomy of a LN node : Router



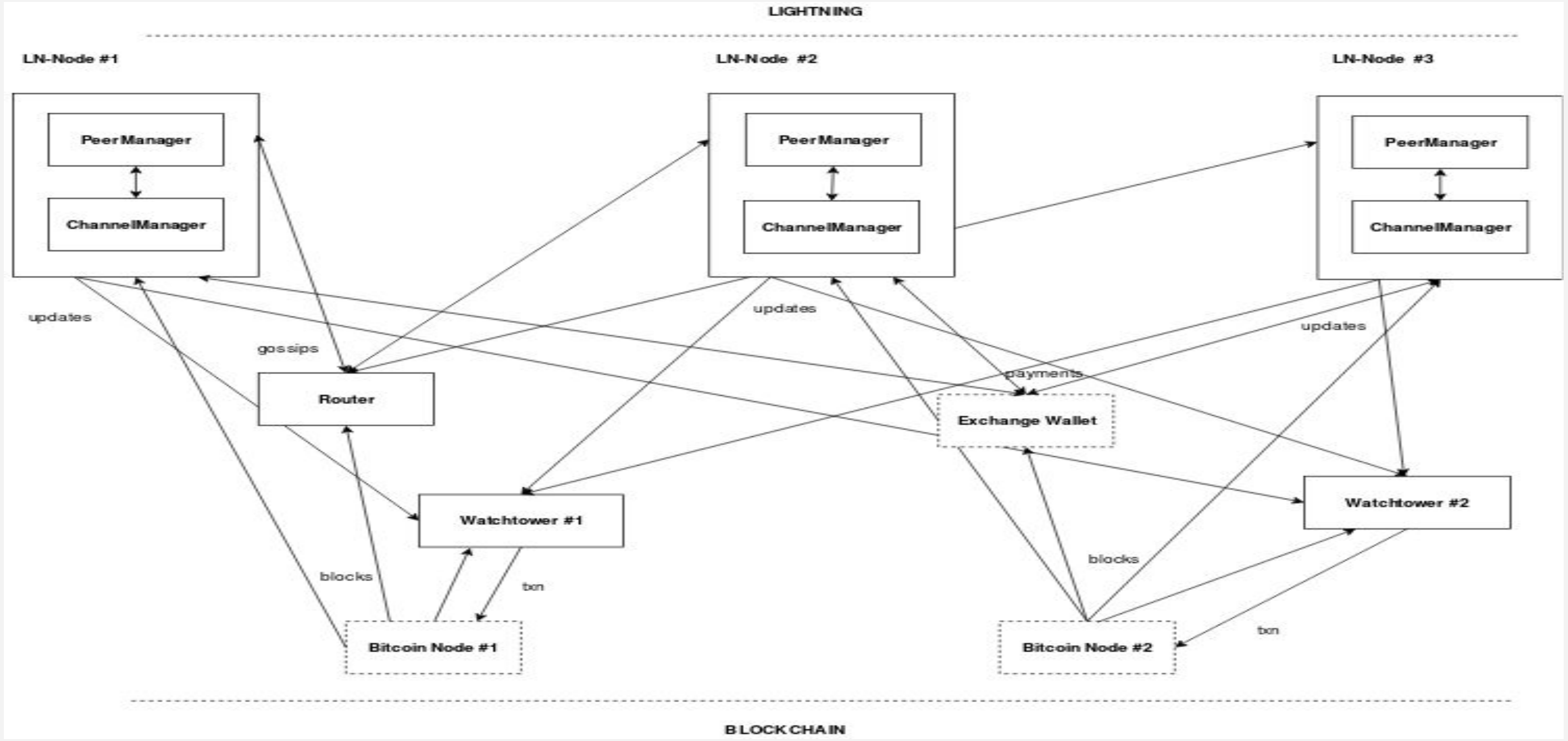
```
/// A trait to describe an object which can receive routing messages.
pub trait RoutingMessageHandler : Send + Sync {
	/// Handle an incoming node_announcement message, returning true if it should be forwarded on,
	/// false or returning an Err otherwise.
	fn handle_node_announcement(&self, msg: &NodeAnnouncement) -> Result<bool, HandleError>;
	/// Handle a channel_announcement message, returning true if it should be forwarded on, false
	/// or returning an Err otherwise.
	fn handle_channel_announcement(&self, msg: &ChannelAnnouncement) -> Result<bool, HandleError>;
	/// Handle an incoming channel_update message, returning true if it should be forwarded on,
	/// false or returning an Err otherwise.
	fn handle_channel_update(&self, msg: &ChannelUpdate) -> Result<bool, HandleError>;
	/// Handle some updates to the route graph that we learned due to an outbound failed payment.
	fn handle_htlc_fail_channel_update(&self, update: &HTLCFailChannelUpdate);
	/// Gets a subset of the channel announcements and updates required to dump our routing table
	/// to a remote node, starting at the short_channel_id indicated by starting_point and
	/// including batch_amount entries.
	fn get_next_channel_announcements(&self, starting_point: u64, batch_amount: u8) -> Vec<(ChannelA
```

Anatomy of a LN node : ChainWatchInterface

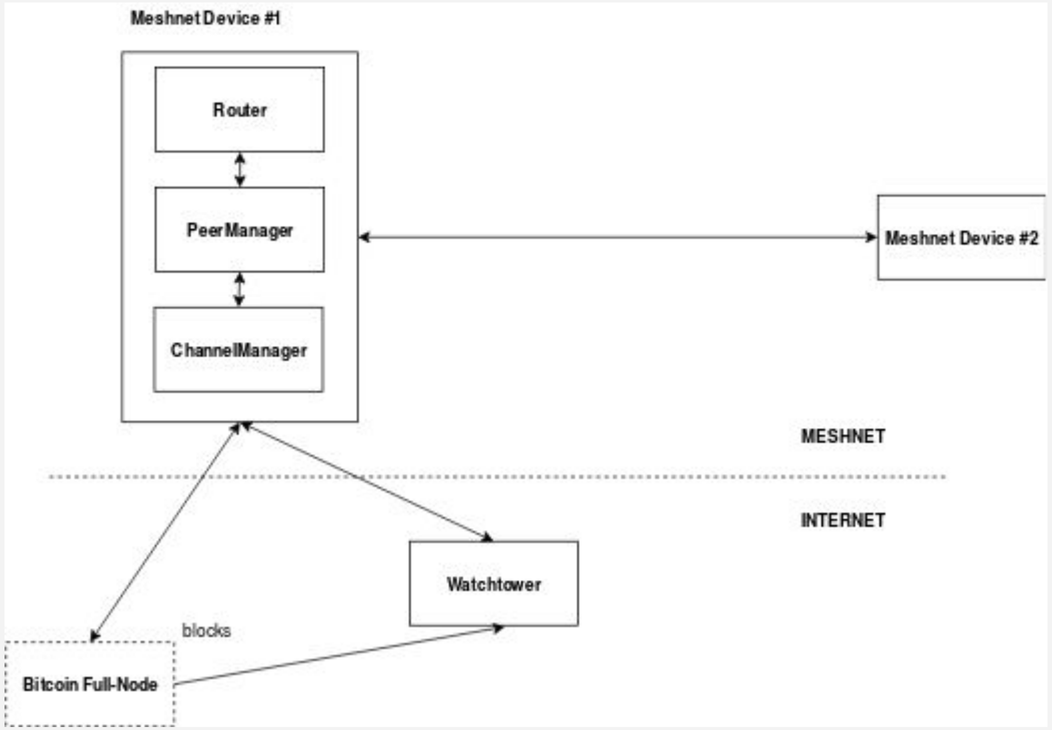


```
/// Note that all of the functions implemented here *must* be reentrant-safe (obviously - they're
/// called from inside the library in response to ChainListener events, P2P events, or timer
/// events).
pub trait ChainWatchInterface: Sync + Send {
        /// Provides a txid/random-scriptPubKey-in-the-tx which much be watched for.
        fn install_watch_tx(&self, txid: &Sha256dHash, script_pub_key: &Script);

        /// Provides an outpoint which must be watched for, providing any transactions which spend t
        /// given outpoint.
        fn install_watch_outpoint(&self, outpoint: (Sha256dHash, u32), out_script: &Script);

        /// Indicates that a listener needs to see all transactions.
        fn watch_all_txn(&self);

        /// Register the given listener to receive events. Only a weak pointer is provided and the
        /// registration should be freed once that pointer expires.
        fn register_listener(&self, listener: Weak<ChainListener>);
        //TODO: unregister
```

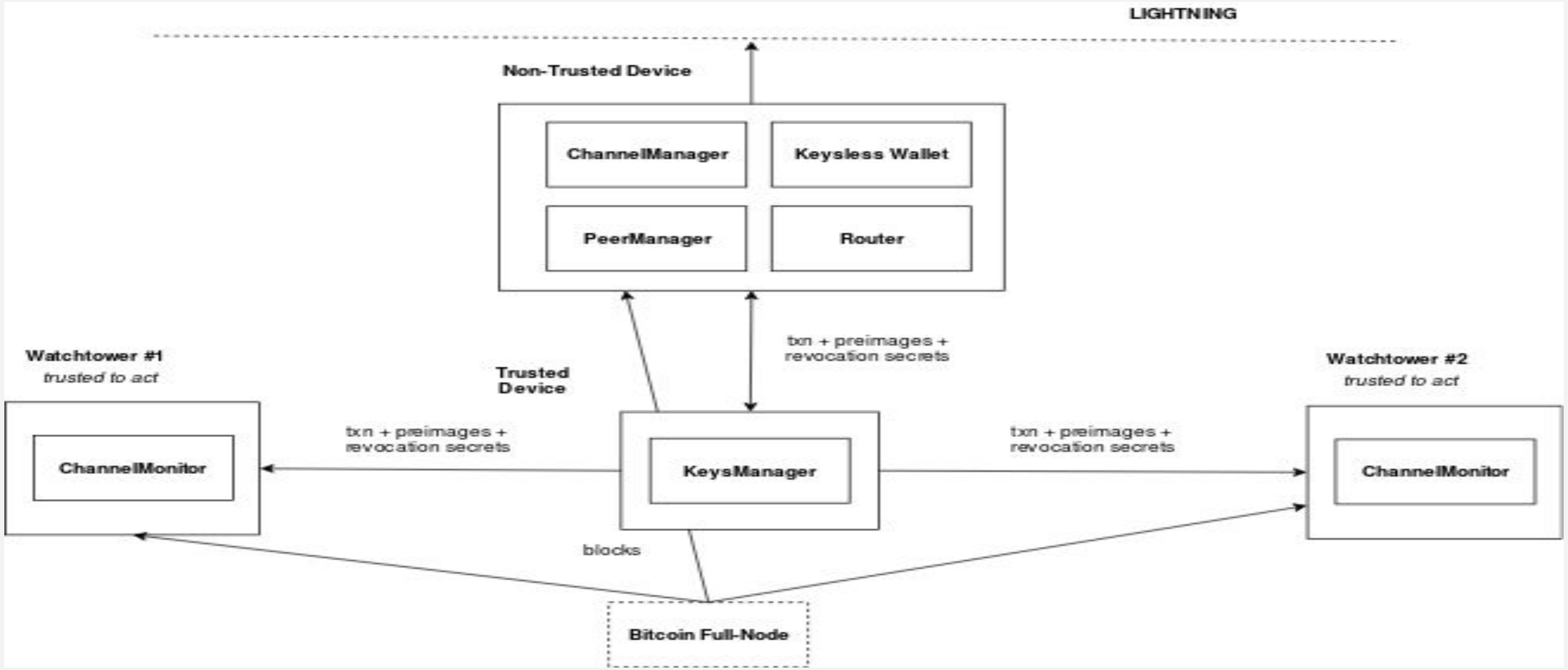
Scenario #1: an Exchange



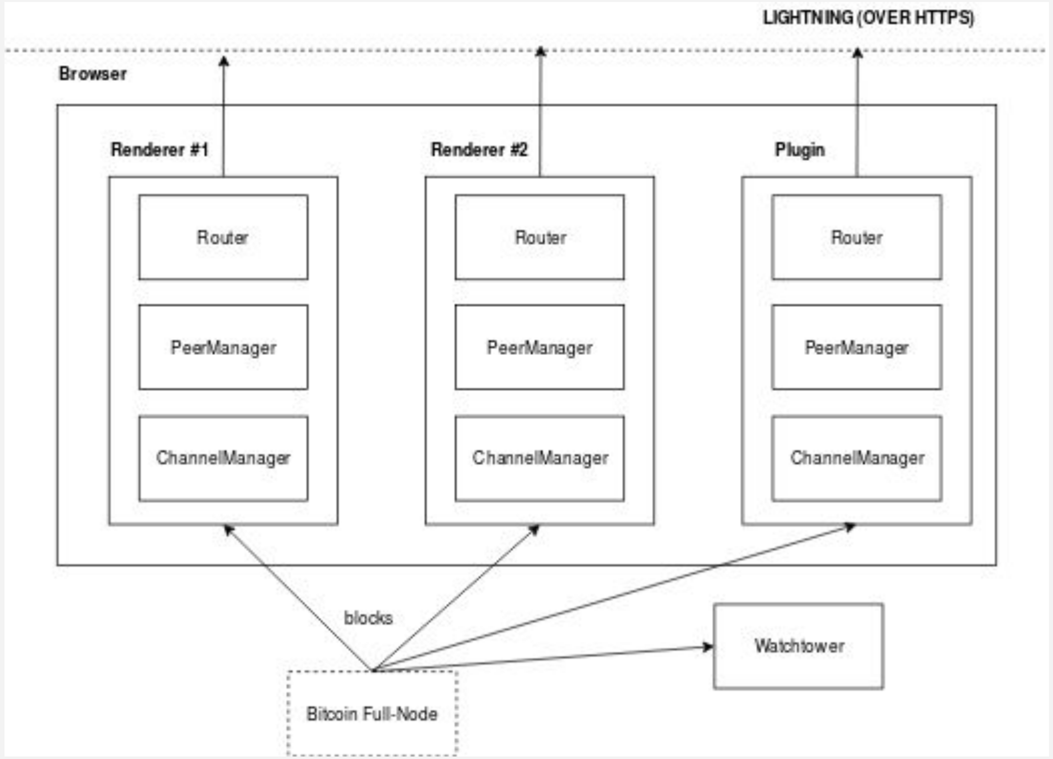
Scenario #2: a Meshnet



Scenario #3: a Hardware wallet



Scenario #4: a Browser



State of the project

Almost-ready for 1st release, doing more reviews and security improvements.

Works on testnet with other implementations with a sample daemon.

Waiting some object of 1.1 being spec’d out/deployed before to go mainnet.

New contributors welcome, get started in bitcoin protocol dev !

Thanks to Chaincode

chaincode

Thanks to LN-Conf

chaincode

Questions?

chaincode