# libchatter: Robust Networking for all

Adithya Bhat*, Colton Fike*, Aniket Kate*

contact: abhatk@purdue.edu

**Abstract**

We are observing an explosion in the number of distributed protocols such as Tendermint, Algorand, SBFT, HotStuff, Sync HotStuff, Opt Sync, SPURT, RandPiper, Drand to name a few. As these protocols start to get considered in practice, there is a definite need for an efficient networking library that can send and receive custom protocol messages while being efficient. Moreover, as most modern systems are multiprocessor systems, concurrency control is also an important aspect where the amount of processing performed by the network layer and the core application can be limited and tuned for optimum performance.

The libp2p library is the most prominent networking implementation, which is widely deployed in Ethereum and Ethereum based blockchains, Substrate, IPFS, Drand, Quorum, etc. It also has a rich API making it an almost perfect library for peer-to-peer networks used in the permissionless settings of thousands of nodes. However, when it comes to permissioned settings with up to few hundred nodes, we observe that there is a loss in performance (loss of messages despite running on the same machine and small loads, and high latency) for libp2p and that the libp2p API is hard to customize for new protocols.

In this work, we attempt to improve the networking for permissioned systems using libchatter: a Rust based networking library, that is more efficient than the state-of-the-art distributed networking libraries, inspired from libp2p and other network implementations in practice today. libchatter provides a WireReady trait interface with serialization, deserialization, and semantic checking and initializing functions. Users of libchatter can easily develop new protocols by implementing the trait and reacting to the messages created by the network layer. libchatter also allows fine-grained control of the network layer. Our performance benchmarks clearly show an improvement over libp2p by at least 4×.

## 1 Introduction

With the recent burst in distributed protocols such as random beacons [1, 2, 3, 4, 5, 6], distributed key generation [7, 8], and consensus protocols [9, 10, 11, 12, 13, 14, 15, 16], there is a clear need for abstractions for distributed networking libraries along with efficient implementations.

Several of the implementations [6, 7, 5] of these protocols assume broadcast channels or have implementations that are not reusable [16, 15, 14, 13]. Among all the network implementations, libp2p [17], Concord-BFT [16], Tendermint [11], and Diem [18] have networking implementations that are used in practical deployments. We will explore them in detail later.

The current distributed networking libraries lack some of the following functionalities:

(i) *Abstract Layer.* In order to generally implement Distributed protocols, the network layer must provide generic abstractions so that the end-user can implement any protocol of choice using the network layer.

(ii) *Concurrency Control.* In a distributed system, each node needs to take advantage of all available cores. Ideally, the networking layer must not interfere with or be a bottleneck for the core application using it. As such, the design used by libp2p and Golang based implementations, do not allow fine-grained control and thus tuning of concurrency for the network layer.

*Purdue University

(iii) *Extensibility.* The network layer should allow easy implementation of any custom protocol. With this respect, the related works [9, 10, 15, 17, 16] are very tightly integrated with their original use-case of core application.

(iv) *Efficiency.* The network layer should be efficient in terms of the latency of message sending/receiving, so that the core application does not time-out despite the nodes sending messages on time. The network layer must also be performant in terms of the number of messages output by the network layer.

**libchatter.** In this work, we present libchatter [19]: a Rust based networking library, that is more efficient than the state-of-the-art distributed networking libraries [17, 18]. libchatter provides a trait WireReady (see Figure 5) which defines serialization, deserialization, and semantic checking and initializing functions. Structs or Enums that implement this trait can be sent/received from the libchatter networks.

**Ease of Usage.** A protocol developer/researcher can take advantage of libchatter as follows (Examples are available [19].):

(i) Define (possibly an enum) containing the protocol messages. Implement WireReady trait for these messages. This can be done easily using serde [20] and bincode [21].

(ii) Create a runtime (with defaults or specify the number of worker threads) and create the network using this runtime. The network creation returns two objects: a sender and receiver ends of a channel.

(iii) Implement the protocol using the network sender and receiver. To send a message to a particular node, post $(senderId, msg)$ to the send channel end-point, or post $(n, msg)$ where $n$ is any number greater than the number of connected nodes to broadcast the message to all the nodes.

## Existing Solutions

**libp2p.** Libp2p [17] is a robust networking library designed for peer-to-peer networks. The library is available in Golang [22], Rust [23], C++ [24], Python [25], Javascript [26], and Java [27]. It is the most popular networking and is used by Ethereum [10], Drand [4], Quorum [28], SPURT [2], Substrate [29], and all Ethereum based blockchains. It is the state-of-the-art and the best solution available for permissionless peer-to-peer networks.

The architecture of libp2p is shown as follows:

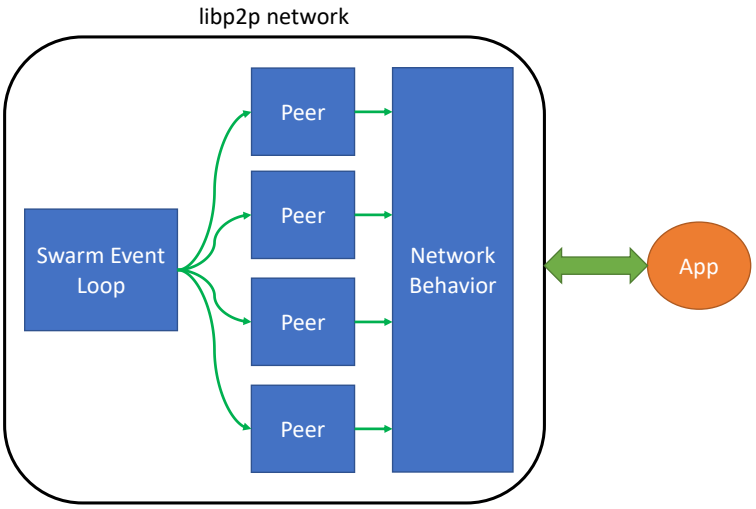


Figure 1: Architecture of networking in libp2p

In libp2p, a swarm is a network object managing the peers. The swarm is also given the network behavior, i.e., how the network should react to network messages. In this regard, libp2p is protocol-aware. The network behavior can forward the messages to the application or the entire application can be implemented in the network behavior (this is done for the protocols provided by libp2p such as MDNS or Kadamelia). Messages can be sent over the network by injecting events into the swarm.

While libp2p has rich high-level abstractions such as MultiAddr, Transports, protocol muxing, etc., in order to fully take advantage of libp2p, a protocol developer must implement futures and polling. This makes it less friendly to new developers. We also observed that despite running on localhost, messages would often be dropped, making this network unreliable. This is not a major problem for asynchronous and permissionless system, but is not okay in general. Hence, we look towards other solutions.

**Custom Solutions.** There are several custom network implementations that are designed specifically for the application. We describe some of the robust implementations here:

(i) **Tendermint.** Tendermint [11, 30] uses a custom networking solution that is very similar to libp2p [17]. It is also designed in Golang, which does not allow fine-grained concurrency control. The protocol is event driven, and the core application (Tendermint protocol) is driven by the events created by the network layer. This results in a tight coupling of the network and the core application and the two layers cannot make independent progress. They use protocol buffers [31] to build and serialize messages, and therefore new users are forced to use protocol buffers or rewrite the library significantly.

(ii) **Diem.** Diem [18, 15] implements the HotStuff protocol [15]. It builds a custom implementation of network whose architecture is shown in Figure 2.

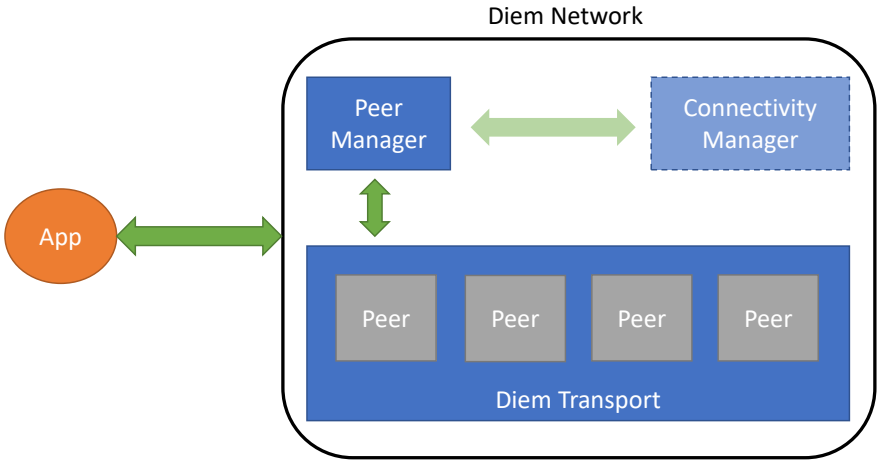


Figure 2: Architecture of networking in Diem

The core application receives a generic object called the `NetworkSender` and `NetworkEvents` that implements Stream (i.e., we can send messages to this object) and Sink (i.e., we can receive messages from this object).

Internally, the network layer contains a transport layer that describes how to connect and perform handshakes to connect to nodes in the system. The peer manager uses the transport to maintain connections with the other peers. It connects to the other nodes and listens to incoming connections. The peer manager can also be integrated with a connectivity manager that allows discovery of new nodes. The peer manager maintains peer objects that send and receive messages over the network. The peer objects return messages that are returned to the application through the peer manager. While sending messages, the peer manager forwards the messages from the application layer to the peer object.

A unique feature of this implementation is the use of custom diem channels that prioritizes connections in a round-robin manner instead of the standard random order used by tokio channels. They also implement their own serialization scheme called Diem Canonical Serialization (DCS) which is a deterministic serialization mechanism that can be derived on structs.

(iii) **Concord-BFT.** Concord-BFT [16, 32] is a C++ implementation of the SBFT [16] protocol. It uses asio [33] (asynchronous IO) library, that is highly performant. It has a simple architecture as shown below:

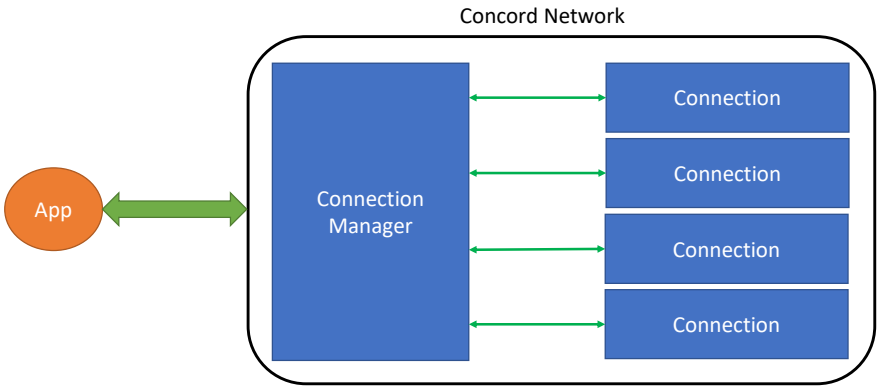


Figure 3: Architecture of networking in Concord-BFT

The connection manager maintains connections with all the peers (called connections) and messages to be sent are added to the write queues of the connection manager (using mutex locks). Message handlers are registered so that messages with a particular code are handled.

The Concord-BFT library allows concurrency control by specifying the number of workers used by ASIO. This allows fine-tuning of the core application. However, the networking library is tightly coupled with the core application and is thus not easily extensible.

# 2 libchatter

We present libchatter, a Rust networking library that is designed for distributed systems [19]. This library is designed and written while keeping performance as the main priority. Our library is inspired from the neat abstractions of libp2p [23], Diem [18], and Concord-BFT [32].

## 2.1 Architecture

The architecture of libchatter is described schematically in Figure 4. The network layer outputs an unbounded channel sender and receiver objects. Any application can use the sender/receiver object to send messages to/receive messages from the network.

The event loop of the networking layer (see Figure 4) uses select on two events:

(i) *New message from the application*: This event triggers the message to be routed to another *Peer* task that holds the socket over which the message can be sent to the corresponding node. This event is always triggered by the application.

(ii) *New message from the network*: This event triggers the message to be routed to the application after checking semantic consistencies (see Section 2.2). First, a Peer task receives protocol messages from the socket. Then the peer task notifies the stream map that it is ready. This triggers the network event loop with a new message from the network that is passed on to the application.

## 2.2 Messages and Semantic checks

The networking library supports any messages that satisfies the following constraints:

```
/// A wire trait tells us that the object can be encoded to/decoded from the
/// network.
pub trait WireReady: Send + Sync + Clone {
    /// How to decode from bytes
    fn from_bytes(data: &[u8]) -> Self;

    /// How to initialize self
    fn init(self) -> Self;

    // How to encode self to bytes
    fn to_bytes(&self) -> Vec<u8>;
}
```

Figure 5: Any object that needs to be sent across the network needs to implement the following trait.

The network layer can have different incoming messages and outgoing messages. The serialization implementation is purely up to the library users. In our examples, we use bincode [21], a binary coding scheme using serde, along with a length limited encoder/decoder to attach a length header to the messages that are sent (see Figure 6).

```
encode(m): [Length as usize; [Msg in bytes]]
```

Figure 6: The network layer takes messages that implement the WireReady trait, and encodes them as shown using the from_bytes and to_bytes functionality.

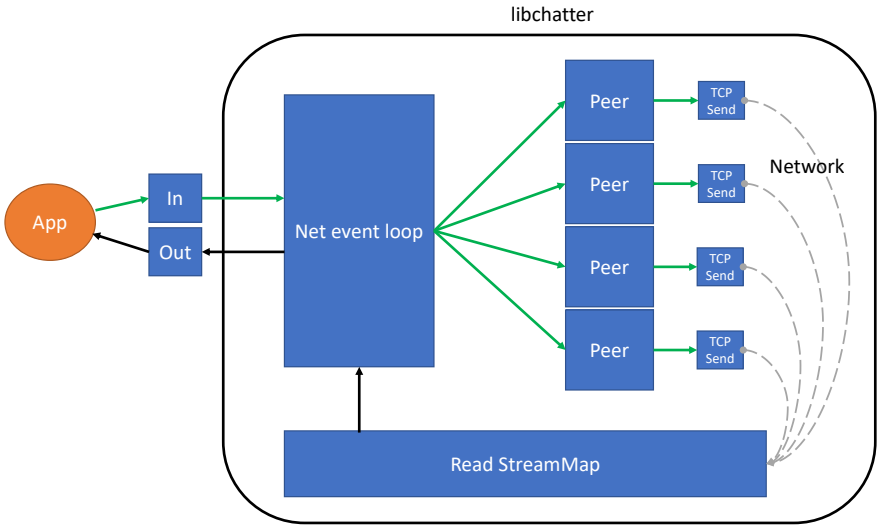


Figure 4: **Overview of the architecture of libchatter.** The application sees two opaque end points (send and receive end-points of a channel) that it can use to communicate messages with the network. The networking layer routes messages that need to be sent to the corresponding peer which are then concurrently sent over the network. The messages received over the network are automatically trigger the stream map to be processed by the net event loop which are then sent to the application for further processing.

**Safety.** The *Send* and *Sync* trait are necessary to send the messages to the tasks. The *Clone* requirement enables cloning of messages so that they can be broadcast to all the connected nodes.

**Semantic Checks.** Messages can contain semantic checks and self references. For instance, consider the following struct:

```
type PeerId = u64;
type Hash = [u8; 32]; // For SHA2

pub struct Block {
    pub proposer: PeerId,
    pub hash: Option<Hash>,
    ...
}

impl Block {
    fn compute_hash(&mut self) {
        ...
    }
}

pub enum ProtocolMsg {
    NewBlockProposal(PeerId, Block)
    Invalid,
}

impl WireReady for ProtocolMsg {
    ...

    fn init(self) -> Self {
        match &self {
            // Semantic check to ensure that the proposer in this message
            // is the same as the proposer in the block
            NewBlockProposal(proposer, block) => {
                if proposer != block.proposer {
                    return ProtocolMsg::Invalid;
                }
                self.hash = Some(self.compute_hash());
                return self;
            }
            _ => { return self; }
        }
    }
}
```

Figure 7: Struct illustrating the need for semantic check and initialization.

In this struct, the init method is used to ensure that the proposer of NewBlockProposal message is always the proposer in the block, and hence the network layer automatically rejects invalid messages. Library users can opt out of this by implementing a dummy init function, and performing semantic checks in the application layer. Also, note that the init abstraction guarantees that the valid NewBlockProposal messages contains blocks who will always have their hashes populated.

## 2.3 Concurrency

A common problem with developing networking libraries in Golang is the lack of fine-grained concurrency control. Rust allows multiple runtimes to execute asynchronous tasks, and therefore our network library design takes advantage of this feature. In the current version of the library, we use tokio [34] as the runtime to run tasks (also known as green threads).

The networking layer can be made it to spawn threads in its own runtime which will run with using a fixed number of cores, and the application can take control of all the remaining cores available in the system. We illustrate this with an example as shown below.

```
// Create a runtime for network 1 that uses 2 threads
let net1_rt = tokio::runtime::Builder::new_multi_thread()
    .enable_all()
    .worker_threads(2)
    .build()
    .unwrap();

// Setup network 1 to use net1_rt (All tasks will use only 2 kernel threads)
let net1 = net::futures_manager::Protocol::<Transaction, ClientMsg>::new(...);
let (net1_send, net1_recv) = net1_rt.block_on(
    net1.client_setup(
        util::codec::EnCodec::new(),
        util::codec::Decodec::new(),
    )
);

// Similarly setup network 2
let net2_rt = ...;
let net2 = ...;
let (net2_send, net2_recv) = ...;

// Setup a runtime for the application (Use all available threads)
let core_rt = tokio::runtime::Builder::new_multi_thread()
    .enable_all()
    .worker_threads(12)
    .build()
    .unwrap();

// Start the application using the core_rt
core_rt.block_on(
    application(
        net1_send,
        net1_recv,
        net2_send,
        net2_recv,
        ...
    )
);
```

Figure 8: Concurrency control in libchatter. Different networking components can be configured to have their own concurrency limits thereby allowing better resource management and prevent starvation of the core application.

In the example from Figure 8, the tokio runtimes `net1_rt` and `net2_rt` are both configured to use 2 worker threads (i.e., kernel threads). All the threads spawned by the networking layer in `net1` will only be running in 2 worker threads at a time. On a system with 16 cores, this spares 14 other worker threads for use by the core application `core_rt` (to maintain databases, logging, respond to messages, etc.)

This is contrast to Golang, where there is no known mechanism to control the degree of concurrency. In libp2p, the networking layer drives the core application, resulting in a loss of concurrency control.

# 3 Performance

## 3.1 Localhost stress tests

We perform a test on localhost so that we can ignore the interference from the network and focus solely on the computational overheads of the protocols.

**Experiment Setup.** For the libp2p implementation of this experiment [35, 36], we use the built-in floodsub protocol. All nodes subscribe to a topic $A$ while a designated leader will subscribe to an additional topic $B$. Once all nodes are connected, the leader publishes a message (500 bytes) to topic $A$. When nodes receive this message, they relay it back to leader with topic $B$, so nodes other than the leader ignore it. When the leader receives the relay from all nodes, it starts the process over by publishing another message. We repeat this process for $10,000$ iterations to find the average latency. We run the experiments on a machine with `Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz` with 24 physical cores (48 hyper threaded cores), 128 GB RAM running Arch Linux.

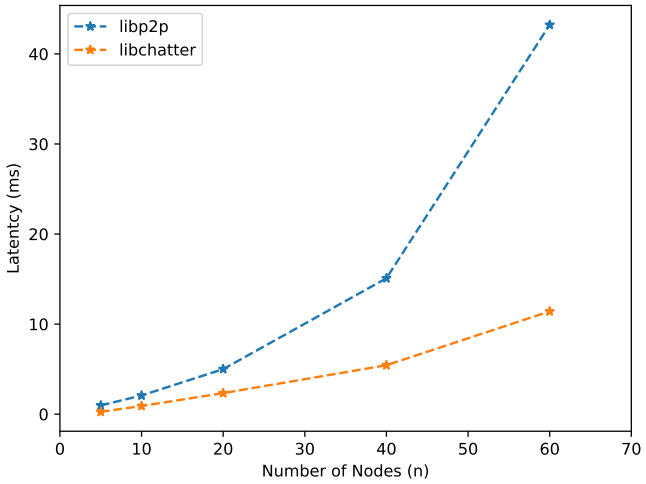


Figure 9: Performance of libp2p vs. libchatter when running a simple protocol locally (without network interference).

In Figure 9, we start with $n = 5$, and we observe that libchatter already performs $2\times$ better than libp2p. As we scale up to $n = 60$, we observe that libchatter scales much better ($4\times$) than libp2p. This is because the protocol is driven by the network layer, and the protocol and the network layer are tightly coupled, and therefore if a single node receives all the messages, the processing done by the single node will slow down the entire network.

Another source of overhead is that the various network events all being funneled into the swarm's event loop, thereby slowing the progress of the protocol. libchatter avoids this by allowing the work-stealing runtimes (from Tokio) to handle errors in tasks thereby reducing the load on the central event loop.

## 3.2 Drand vs. BRandPiper

As another case study, we compare the BRandPiper Protocol [1] which is cryptographically more intensive and communicates more messages with Drand [4] a simple random beacon protocol in Golang.

**Setup.** We run the experiment on $n$ `t2.small` machines all of which were located in the same area (us-east-2 Ohio region) with default instance settings. We run Drand with the TLS certificates disabled (to fairly compare it with libchatter). The `time_discrepancy_ms` field is output by Drand for every beacon as the time difference from the start of the round to the time in which it obtains the beacon. Drand 99.9% refers to the number of beacons produced if the 99.9% percentile of the `time_discrepency` were estimated as the actual computation and communication times, and Drand 100% refers to the longest computation and communication times. For RandPiper, we use two accumulators: bilinear and merkle tree based accumulator with the merkle tree based accumulator incurring $O(n^2 \log n)$ communication cost. (More details about the protocol and the experiments can be found here [1].)

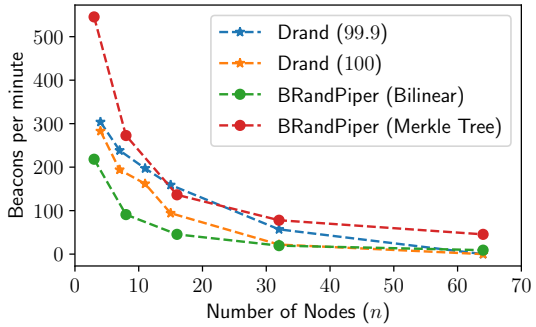


Figure 10: Performance of libp2p vs. libchatter by comparing RandPiper [1] and Drand [4].

**Analysis.** Figure 10 shows the impact of a good networking library and language choice on the performance of protocols. RandPiper uses verifiable secret sharing primitives with 11 rounds of $O(n^2)$ communication per beacon, but still manages to outperform Drand: a simple random beacon where the nodes sign the hash of a counter, and the hash of the aggregated signature serves as the random beacon. Despite being unoptimal in terms of communication, BRandPiper based on the merkle tree accumulator is the most efficient beacon.

*Caveats and conclusion.* There are several other factors that cause Drand to have a much lower performance such as the use of Databases, lack of cryptographic optimizations, lack of concurrency control, and Golang's garbage collector. By writing the protocol in Rust, we get rid of the garbage collector automatically, and libchatter provides efficient communication along with concurrency control that leads to an improved performance. There is a clear scope for improvement for Drand as shown by RandPiper.

## 3.3 Other Experiments

For $n = 4$ nodes, we also performed experiments in AWS (Arch linux using `c5-4xlarge` located in us-east-2) by implementing the Apollo protocol [12] in Golang using libp2p where we obtained throughputs of $\approx 10$ Kops/s (see experiment details here [12]) and latencies of $\approx 2$ seconds. By moving to a custom TCP implementation, we managed to improve the throughputs to $\approx 40$ KOps/s while reducing the latencies to $\approx 1.5$ seconds.

By moving the implementation of Apollo to Rust, we initially managed to improve the throughputs to $\approx 110$ KOps/s and latencies to $\approx 200$ ms. Since Apollo is a round-robin protocol, using the concurrency control features of libchatter, and by using futures channels in libchatter (futures channels allow us to peek into the channel to see if more messages are available), we managed to bring the final throughputs to $\approx 350$ KOps/s and latencies to $\approx 100$ ms.

# 4 Future Work

We aim for the following features in the near and long term future of this project:

1. *UDP and TLS support.* The current implementation only uses plain TCP communication, but in the near future we will be adding UDP and TLS network implementations.

2. *Custom polling.* Different protocols might have different priorities when polling messages. For instance, in round-robin protocols, messages from the current proposer may have a higher priority than messages from other nodes when messages from multiple nodes are available, in order to prevent timing out on the current proposer. Similarly, in stable-leader protocols, the messages from the leader might have a higher priority than the messages from other nodes. In a future version, we aim to support such custom polling mechanisms for messages from the network efficiently.

3. *Support common Network Layouts.* In a future version, the library should support common network configurations such as all-to-all, all-to-one, star, and finally custom graphs specified in a config file.

4. *Mock Network implementation.* A mock in-memory network implementation to test the correctness of Byzantine fault-tolerant protocols. This allows the protocols to check the behavior of protocols during time-outs, going offline, reconfiguration, etc.

5. *WAL.* An efficient write-ahead logger designed specifically for distributed protocols to allow tracing of failures. The logs should allow easy replication of the failure using the mock network.

# References


[1] A. Bhat, N. Shrestha, A. Kate, and K. Nayak, "Randpiper – reconfiguration-friendly random beacons with quadratic communication," Cryptology ePrint Archive, Report 2020/1590, 2020, https://eprint.iacr.org/2020/1590, To appear in ACM SIGSAC CCS 2021.

[2] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup," Cryptology ePrint Archive, Report 2021/100. https://eprint. iacr. org/2021/100, Tech. Rep., 2021.

[3] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "Hydrand: Practical continuous distributed randomness," in *2020 IEEE Symposium on Security and Privacy (SP). IEEE*, 2020.

[4] Drand, "Drand - a distributed randomness beacon daemon." [Online]. Available: https://github.com/drand/drand

[5] I. Cascudo and B. David, "Scrape: Scalable randomness attested by public entities," in *International Conference on Applied Cryptography and Network Security*. Springer, 2017, pp. 537–556.

[6] I. Cascudo, B. David, O. Shlomovits, and D. Varlakov, "Mt. random: Multi-tiered randomness beacons," Cryptology ePrint Archive, Report 2021/1096, 2021, https://ia.cr/2021/1096.

[7] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Aggregatable distributed key generation," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer International Publishing, 2021, pp. 147–176.

[8] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures." in *CCS '20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1751–1767. [Online]. Available: https://doi.org/10.1145/3372297.3423364

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

[10] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[11] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, 2016.

[12] A. Bhat, A. Bandarupalli, S. Bagchi, A. Kate, and M. Reiter, "Apollo – optimistically linear and responsive smr," Cryptology ePrint Archive, Report 2021/180, 2021, https://ia.cr/2021/180.

[13] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 654–667.

[14] N. Shrestha, I. Abraham, L. Ren, and K. Nayak, "On the Optimality of Optimistic Responsiveness," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 839–857.

[15] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[16] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 568–580.

[17] libp2p, "libp2p/rust-libp2p: The rust implementation of the libp2p networking stack." [Online]. Available: https://github.com/libp2p/rust-libp2p

[18] Diem, "diem/diem: Diem's mission is to build a trusted and innovative financial network that empowers people and businesses around the world." [Online]. Available: https://github.com/diem/diem

[19] A. Bhat, "Libchatter," Sep. 2021, original-date: 2020-12-08T06:00:09Z. [Online]. Available: https://github.com/adithyabhatkajake/libchatter-rs

[20] Serde-Rs, "serde-rs/serde." [Online]. Available: https://github.com/serde-rs/serde

[21] 2021. [Online]. Available: https://docs.rs/bincode/1.3.3/bincode/index.html

[22] "libp2p/go-libp2p," Sep. 2021, original-date: 2015-09-30T23:24:32Z. [Online]. Available: https://github.com/libp2p/go-libp2p

[23] "libp2p-rs," Sep. 2021, original-date: 2017-03-24T20:05:11Z. [Online]. Available: https://github.com/libp2p/rust-libp2p

[24] "CPP-Libp2p," Sep. 2021, original-date: 2019-09-17T12:07:29Z. [Online]. Available: https://github.com/libp2p/cpp-libp2p

[25] "py-libp2p," Sep. 2021, original-date: 2018-09-17T05:20:41Z. [Online]. Available: https://github.com/libp2p/py-libp2p

[26] "libp2p/js-libp2p," Sep. 2021, original-date: 2015-07-23T21:05:43Z. [Online]. Available: https://github.com/libp2p/js-libp2p

[27] "jvm-libp2p," Sep. 2021, original-date: 2019-05-29T20:28:05Z. [Online]. Available: https://github.com/libp2p/jvm-libp2p

[28] "ConsenSys/quorum," Sep. 2021, original-date: 2016-11-14T05:42:57Z. [Online]. Available: https://github.com/ConsenSys/quorum

[29] "Substrate," Sep. 2021, original-date: 2017-11-07T18:08:53Z. [Online]. Available: https://github.com/paritytech/substrate

[30] Tendermint, "tendermint/tendermint: Tendermint core (bft consensus) in go." [Online]. Available: https://github.com/tendermint/tendermint

[31] "Protocol Buffers - Google's data interchange format," Sep. 2021, original-date: 2014-08-26T15:52:15Z. [Online]. Available: https://github.com/protocolbuffers/protobuf

[32] "github - vmware/concord-bft: concord byzantine fault tolerant state machine replication library 2021," 2021. [Online]. Available: https://github.com/vmware/concord-bft

[33] chriskohlhoff, "chriskohlhoff/asio," Sep. 2021, original-date: 2011-02-15T05:18:45Z. [Online]. Available: https://github.com/chriskohlhoff/asio

[34] Tokio-Rs, "tokio-rs/tokio." [Online]. Available: https://github.com/tokio-rs/tokio

[35] C. Fike, "libchatter-rs with benchmark code," Aug. 2021, original-date: 2021-08-31T22:43:25Z. [Online]. Available: https://github.com/coltonfike/libchatter-rs

[36] ——, "coltonfike/p2p-benchmarks," Sep. 2021, original-date: 2021-08-30T19:28:41Z. [Online]. Available: https://github.com/coltonfike/p2p-benchmarks