Application Design Specifications
Peeriod

1 Overview

Based on our paper [1], we developed an application which implements the proposed protocol. Our intention has not been to merely adopt the concepts described in the paper but also to try finding a good balance between security and usability. We want to provide a solution which is working "out of the box" and is usable by the general public, not only by a technologically literate minority. Many applications providing solutions for secure communications or anonymizing connections are developed with only technical details in mind, at the same time missing good usability concepts which are able to transform the technical complexity into an easy-to-use interface.

Several considerations have been taken into account during the process of development and we will outline them on the following pages. Our implementation is far from perfect, of course, and it cannot be perceived as secure until its underlying concepts and itself will have been sufficiently reviewed, criticized and penetrated by others. Peeriod is part of our final thesis and our time frame for its development consisted of three months. Many implementation details must be improved, are yet missing or have been done differently in our testing environment and need be changed in the future (changing Kademlia RPCs from TCP to UDP being the most significant example, see below). Already planned enhancements or unsolved known problems will be mentioned in particular.

Thus, this specifications document shall help others to better understand what and why we did things the way we did. We want to cover all parts of the protocol: maintaining the network topology, constructing Onion Routing circuits, finding file locations, requesting and transmitting files. In short, describing the lifespan of a node in the network: how is it able to join, how can it stay, how does it find things, how does it download them?

At last, the Appendix provides a detailed description of all message types used. For simplicity we will refer to the software implementation as 'Peeriod'.

2 Topology

The peer-to-peer network is based on the distributed hash table (DHT) system of *Kademlia*, using a XOR metric as a notion of distance. In this part we assume general knowledge of Kademlia's remote procedure calls (RPCs); a very good and short overview of its protocol can be found at [2].

Every message regarding the topology of the network starts with a header which includes:

– The ID of the intended receiver node.
– Information about the originator of the message, i.e. the node ID and a list of addresses which the node can be reached with. Consequently every topology message carries useful contact information for the buckets.
– The message type.

General notes on the current state of implementation:

1. To locate files Peeriod uses a flooding-based search algorithm only, thus there is no implementation of the FIND_VALUE and STORE RPCs yet. However, Kademlia's key-value storage is planned and will be implemented.

2. *All* communication in Peeriod is TCP-based only, also Kademlia's RPCs. This has proved useful in a testing environment where reliable transport is advantageous, but is obviously inconvenient in the long term compared to connectionless communication. Thus, transferring the sending of Kademlia RPCs and flooding-based messages from TCP to UDP is a high-priority enhancement!
3. We assume a node can be reached via multiple addresses, i.e. $\langle$IP, Port$\rangle$ pairs. This becomes especially important for proxies (see below).

Although we assume familiarity with Kademlia, we will take a closer look at our implementation of the node lookup algorithm and how and when buckets are refreshed. Furthermore, we will describe how a node is able to find another node for initially joining the network. At last we will explain how a node maintains information about other nodes and how the network is extended with a proxy functionality. Proxies allow nodes without open ports to participate in the network.

2.1 Node Lookups and Bucket Refreshes

This Section describes our implementation of the algorithm used for locating k nodes near a given 160 bit identifier id, where k is a system-wide constant for the maximum number of nodes stored in a bucket. It uses the concept of loose parallelism and is comparable to the one proposed in [2]. We also changed the name of FIND_NODE to a more descriptive FIND_CLOSEST_NODES. The implementation is as follows:

1. Construct an empty $confirmedList$ which can hold up to k entries. This will be filled with the resulting list of close nodes.
2. Construct an empty *probeList* which holds the nodes yet to probe, i.e. send FIND_CLOSEST_NODES messages to.
3. Note: Both $confirmedList$ and *probeList* must always be sorted by distance to id (from close to far).
4. Pick the α nodes from the node's buckets whose IDs are closest to id. α is a system-wide constant determining the degree of parallelism.
5. Send parallel FIND_CLOSEST_NODES messages (with id) to these α nodes.

6. Set a *parallelismTimeout*, using a constant number of milliseconds. This timeout indicates when to send new FIND_CLOSEST_NODES messages.
7. If one of the probed nodes responds with a FOUND_CLOSEST_NODES message, it is added to the *confirmedList* at the right position. The returned close nodes contained in the FOUND_CLOSEST_NODES message are inserted into the *probeList*.
8. As soon as the *parallelismTimeout* elapses, the next α nodes are taken from the *probeList* and FIND_CLOSEST_NODES messages are sent to them in parallel. They are then removed from the *probeList* as well. Then set the *parallelismTimeout* again.
9. When the *probeList* is empty, a *cycleTimeout* is set, using another constant number of milliseconds. As soon as a FOUND_CLOSEST_NODES message (containing information about nodes close to *id*) is received, the *cycleTimeout* is revoked, and reset when the *probeList* becomes empty again.
10. A node lookup is considered finished as soon as:
    (a) Either the *confirmedList* is full, i.e. holds k entries.
    (b) Or the *confirmedList* isn't full, but the *probeList* is empty (all nodes have been probed) and the *cycleTimeout* has elapsed.

Peeriod uses 3 for α, 20 for k and 1.5 seconds for the *parallelismTimeout*.

To avoid iterating over all nodes in the routing table (the accumulation of all k-buckets) when a node A looks for k close nodes for a given *id*, the following short algorithm is used:

1. Find the bit with the highest order n in which A's ID and *id* differ.
2. Iterate over all buckets with index i, where $0 \leq i \leq n$ and try to get the k closest nodes to *id*.
3. If all these buckets together contain less than k nodes in total, iteratively look for nodes in the buckets with index $n+1, n+2, ...$ Stop as soon as k nodes have been found or all buckets have been checked.

2.2 Joining and maintaining the network

In Peeriod, a node's ID and its routing table are persisting and are thus not cleared at runtime. For now we assume a node is able to find an active initial node (how an initial node can be found is explained in the next Section). With the help of this *entry node* the node joins and maintains its network as follows:

1. If the node does not already have an ID it generates one.
2. It finds an entry node and inserts it into the appropriate bucket, PING-ing an old node if necessary. This happens if the buckets have been persisted from a previous runtime.
3. The node starts an iterative node lookup for its own ID.
4. If this lookup cycle doesn't return any nodes, a new entry node is being searched for.
5. As soon as the lookup cycle for the node's own ID is completed, the node checks if it needs any proxies and starts requesting proxies if necessary (proxies are described in the next but one section).

6. The node refreshes all buckets farther away than its closest neighbor it has gained knowledge of via the lookup. The closest neighbor will be in the accessed (definition below) bucket with the lowest index.
7. Once the buckets have been refreshed, the node's entrance into the network is considered successful.

This process is comparable to the one described in [2] as well.

Generally, when a node is added to a bucket (or moved from 'least recently seen' to 'most recently seen'), the bucket is considered 'accessed.' Unaccessed buckets are refreshed periodically. In Peeriod, this is hourly. Refreshing a bucket means performing a lookup operation for a random ID within the bucket's range.

2.3 Finding an entry node

In order for a node to join the network, it needs to find an active, already participating *entry node*. As the protocol only tries to hide file queries and transfers but not the participation itself, various possibilities exist.

One of our main goals of Peeriod is that it can work out of the box. A user should not be forced to meticulously configure the application on starting it for the first time, unless she decides otherwise. Thus, the search for an entry node is the only part of the protocol where we 'allow' centralized processes to slip in.

We want to provide a diversity of possibilities to find an entry node, and the application should gradually be extended with more and more node discovery options. Possibilities to publish a node's location information may range from using social media accounts to posting in an IRC channel.

The current implementation features:

a) Discovery through HTTP servers

A node POSTs its ID, IP and open ports to an HTTP server in an hourly interval. Nodes trying to find initial nodes GET a random entry node from the server. The source code for HTTP servers like this could be made public so generally everyone using e.g. a shared web hosting service can provide a node discovery service, if wanted.

Thus, a possible future feature could be flooding lists of these discovery services through the network.

b) Discovery through persisted routing table

This is the most basic form of node discovery. If the node has access to buckets which have been saved in a previous runtime, it randomly chooses nodes from it.

A node trying to find an entry node repeatedly tries to gain knowledge about possible nodes. Once it has received information about a node, it sends a PING message to it in order to check whether it is active or not. If the PINGed node responds with a PONG message, it can be used as an entry node. Of course a node can PING a lot of potential nodes in parallel and only stop PINGing as soon as one has answered.

2.4 Node Proxies

Node proxying describes the concept of relaying messages bound for an unreachable node through another – reachable – node which has a direct connection to the unreachable one. This concept shall give users who cannot (or do not want to) enable port forwarding the possibility to participate in the network nonetheless, of course under the condition that the network holds enough other nodes which are publicly reachable.

For example, assuming there is a node A which is unreachable from 'outside', and there are publicly reachable nodes B and C. We further assume A is sending a UDP datagram containing a PING message to C. As A's application is not listening on any open port, C would have no possibility to answer with a PONG message. This situation could be solved with A using B as a proxying node. B can be reached from outside, so A initiates a TCP connection with B and asks B to be its proxy. If B accepts, A and B keep the connection alive until B should no longer be a proxy, due to whatever reasons. A now uses B's contact information as its own: when A sends a PING to C it prepends B's address to the message. C can now easily respond to A with a PONG, for every topology related message contains the IDs of the message originator as well as the intended receiver. B receives the PONG message, notices that the message is not intended for itself, and relays it to A. C never needs to know if A has a proxy or not.

Obviously this scheme can only work if B is an honest node. In order to avoid relying on a single node only, A should always try to maintain a fixed number of proxies and substitute them regularly. Regularly changing proxies however is not implemented yet. For now, new proxies are only requested if old ones die. Moreover, proxies are also used merely for topology related messages. They do not play any role in 'anonymous' messages, where only TCP connections to specific ⟨IP, Port⟩ pairs matter!

In Peeriod, a proxy is requested using a PROXY_REQUEST message over a TCP connection. This session must be used by the requested node to either accept the request with a PROXY_ACCEPT message or reject it with PROXY_REJECT. For relaying messages from a proxy to the actually intended receiver, the PROXY_THROUGH message type is used.

A node is considered 'proxy capable' if it is publicly reachable and if the number of nodes it already acts as a proxy for does not exceed a certain self-imposed limit.

3 Anonymity and Onion Routing

On top of this layer – a node maintaing a routing table – the anonymity layer is constructed. A node's objective is to permanently maintain a multitude of Onion Routing circuits which it constructs with the help of the reachable addresses stored in its routing table.

Messages relatable to the anonymity layer are transferred over TCP only. Node IDs no longer play a role. TCP sessions and ⟨IP, Port⟩ pairs (or, in a stricter setup, only IP addresses) are used to identify communication with other nodes.

If within an Onion Routing circuit the TCP connection between two nodes is ended (due to whatever reasons), the whole circuit is rendered useless and must be torn down.

This Section starts with a description of the cryptographic parameters used, explains how in the current implementation nodes for an Onion Routing circuit are picked from the routing table and how an Onion Routing circuit is constructed/ maintained.

As a node aims towards always having *many* circuits to query and request files, an analogy to the many-headed Greek mythological creature 'Hydra' suggests itself. Thus, instead of Onion Routing circuits, the application's source code refers to 'hydra circuits'. In this document we will use the shorthand 'circuit'.

3.1 Cryptographical parameters

For authenticated encryption of a circuit session between two nodes, we use 128-bit AES in Galois Counter Mode, with a random 96-bit initialization vector for each message. Unfortunately, in the current implementation, no additional authenticated data (AAD) is used per message yet, but a high-priority enhancement is to introduce sequence numbers for encrypted communication in order to prevent basic replay attacks. See Section 8 for more information about current implementation issues.

One of the main goals of Peeriod is to avoid a public-key infrastructure. For the input material of the key derivation, we use a Diffie-Hellman key agreement with the 2048-bit MODP group (id 14) as defined in RFC 3526 [3].

For key derivation we use the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) as proposed in [4], with SHA-256 as a hash function, the exchanged Diffie-Hellman secret as input keying material, a varying 128-bit salt value and a zero-length string as info input for the expanding part of the derivation.

For all other hashing operations, we use SHA-1.

3.2 Picking nodes for a circuit

On the anonymity layer, node IDs are no longer respected. In this sense, a 'node' constitutes an unique $\langle$IP, Port$\rangle$ combination. In a stricter setup this can be reduced to the IP address only, but in our testing environment, we stick to the $\langle$IP, Port$\rangle$ combination.

Assuming a node wants to construct a circuit, it at first picks a batch of nodes it will use as relay/exit nodes for the circuit. 'Picking a node' means randomly choosing any $\langle$IP, Port$\rangle$ pair from the maintained routing table. Then for each consecutive node the circuit should be extended with, a batch of nodes used for the additive sharing scheme is picked in the same fashion. The current implementation of picking nodes obeys the following rules:

1. A picked relay node cannot be part of any batch used for additive message sharing.

2. Each additive sharing batch of nodes has a threshold j of known nodes, meaning that a maximum of j nodes which participated in previous sharing rounds can be picked again.
3. Each 'unsuccessful' try of picking a random node, i.e. choosing a random $\langle$IP, Port$\rangle$ from the routing table which does not fulfill the above mentioned requirements, increments a counter. Once that counter exceeds some self-imposed limit, the node waits for some time so that the routing table may get updated with new nodes in the meantime.

Peeriod currently uses 3-4 relay nodes for a circuit and 4 nodes for an additive batch, which means that a message distributed with an additive sharing batch is divided into 5 shares. The number of shares is a network-wide constant!

At the moment the additive sharing scheme is implemented with a single hop, i.e. the 5 shares meet immediately at the intended relay node. Future enhancements include the implementation of multiple-hop sharing as described in [1].

3.3 Circuit construction and maintenance

In this Section we describe the process and message flow of constructing a circuit in the implementation's current state. We take a look at the flow from three different perspectives: Firstly, from the viewpoint of a node initiating a circuit which it intends to use for anonymous communication. Secondly, from the perspective of a node participating in the additive sharing scheme. Finally, from the viewpoint of the node which the circuit is extended/initially constructed with.

As stated earlier, in a circuit, connections to a predecessor/successor node are not reestablished. Two nodes in a circuit use the same connection from start to finish. Once the TCP connection between them has ended, the whole circuit must be torn down. 'Tearing down' refers to merely killing the connection to the predecessor and successor. This can be done anywhere in the circuit and thus initiates a chain reaction of ending TCP connections until all nodes in the circuit have cleaned up their links.

Generally, two connected nodes within a circuit share a *circuitId* which identifies their connection, i.e. a *circuitId* is always shared between a node and its predecessor/successor, however it is always assigned by the predecessor. In Onion Routing, *circuitIds* are used for being capable of correctly relaying messages by relating the *circuitId* shared with a predecessor to the *circuitId* of a connection to the successor. Usually only the two nodes sharing a *circuitId* have knowledge of it. For example, in a circuit $A \longrightarrow B \longrightarrow C \longrightarrow D$, the initiator node does not know the *circuitIds* between $B \longrightarrow C$ or $C \longrightarrow D$.

However in our design, a node maintaining a circuit must be able to provide other nodes with information about how he can be reached by external nodes without disclosing its identity. This is achieved in the following way (using A, B, C, D from above):

Not only do A and a node the circuit has been extended with (e.g. D, but of course B and C as well), derive symmetric encryption keys from their shared secret, but also a 128-bit *feedingIdentifier* which A can use to anonymously 'publish' together with the ⟨IP, Port⟩ pair of D. Thus, an external node can send a message to D containing the *feedingIdentifier* shared between A and D. As D can relate a specific connection to the used *feedingIdentifier*, the message can be piped back through the circuit. Hence, A is externally reachable through theoretically every relay node in its circuit without needing to announce its identity.

Circuit construction from an initiator's point of view: We assume a node A has picked a batch of nodes B, C, D it wants to use as relay nodes from its routing table. The rest is done as follows:

1. A generates a random 128-bit *uuid* which acts as an identifier for all shares of the additive sharing scheme.
2. A furthermore generates a random 128-bit *circuitId* for the circuit connection it is about to establish with B.
3. A picks nodes for the additive sharing scheme (as-nodes) used to send the different shares of the cell-creation message to B.
4. Now A generates its half of the Diffie-Hellman (DH) key agreement (A's public key).
5. A computes additive shares which together will result in A's public key. The number of shares is the amount of as-nodes + 1.
6. A wraps each share in a CREATE_CELL_ADDITIVE message, marking it with *uuid*. A prepends *circuitId* to one of these messages. This is the 'initiator message'. All other CREATE_CELL_ADDITIVE messages are now wrapped up again in an ADDITIVE_SHARING message containing the address of B.
7. A sends each ADDITIVE_SHARING message to one as-node.
   The last CREATE_CELL_ADDITIVE message containing *circuitId* is sent directly to B. The established TCP connection to B is being kept open.
8. A sets a timeout which indicates the time it will wait for a response from B (via the said TCP connection).
9. The following things can happen:
   (a) **The timeout elapses and A has received no response**
   This is considered an error. As B is the first node in the circuit, A kills its connection to B and must construct a fresh circuit.

   (b) **A's connection to B dies**
   This is considered an error as well. Again, A must construct a fresh circuit.

   (c) A **receives a CELL_CREATED_REJECTED message from** B
   A checks whether the *circuitId* and *uuid* provided in the message match the ones it originally sent. If yes, and the cell request was not rejected, it computes the shared secret and its hash from the B's public key provided in the message payload, and checks whether the computed hash and the one in the message match. If yes, A derives a key for encrypting outgoing communication, a key for decrypting incoming communication, and a $feedingIdentifier$. B is now considered a valid relay node of the circuit. If the cell request was rejected, A substitutes B with another node from its routing table, picks new as-nodes and repeats the process.
   The failure of one of the checks renders the circuit unusable though: A kills its connection to B and must construct a new circuit.

The difference between the initial cell and the consecutive extension of the circuit lies in the 'initiator message'. Assuming A extends its circuit further with the node C:

1. A merely generates *uuid*.
2. A picks a fresh batch of as-nodes.
3. A generates its DH public key and computes the additive shares.
4. All but one shares are sent to the as-nodes. The last share is sent to B, encrypted and piped through the circuit, in an ADDITIVE_SHARING message indicating B to further extend the circuit with C.
   B can then choose a *circuitId* itself before sending the so created 'initiator message' (CREATE_CELL_ADDITIVE) to C.
5. A waits for a reaction which it obviously expects from B in the form of a CELL_CREATED_REJECTED message piped back through the circuit. The handling of the (non-)reaction follows analogously to the creation of the first cell.

Circuit construction from an additive sharing node's point of view:
An 'additive sharing node' is a node which receives an ADDITIVE_SHARING message from some other node. As an ADDITIVE_SHARING message actually just consists of the concatenation of the ⟨IP, Port⟩ pair of the intended receiver and an appendant CREATE_CELL_ADDITIVE message, all the additive sharing node has to do is to send the unwrapped message to the included address. The connection over which the node received the ADDITIVE_SHARING message can then be closed.

Circuit construction from a potential relay node's point of view:
A 'potential relay node' is a node receiving a CREATE_CELL_ADDITIVE message. Its objective is to wait until it has received a complete batch (identifiable by *uuid*) of CREATE_CELL_ADDITIVE messages.

Assuming a node receives such a message, it handles it as follows in the current implementation:

1. If the node doesn't already know the *uuid* of the message, it sets a timeout. If it elapses and this batch could not be completed, any potential remaining messages are discarded.
2. The node checks if the message holds a *circuitId*, i.e. is the 'initiator message'. If yes, the connection through which it received the message is being kept open. Otherwise the connection is closed after the message receipt.
3. If a batch is complete, the node checks whether it can be part of a circuit. That means that the number of active circuits it acts as a relay/exit node for does not exceed a self-imposed limit.
4. If it *cannot* be part of the circuit, it sends back a CELL_CREATED_REJECTED message (in the clear) without DH public key and hash via the TCP connection of the received 'initiator message'. This indicates a rejection.

5. If it *can* be part of the circuit, the node generates its half of the DH handshake, computes the shared secret and hash digest, derives the symmetric keys and *feedingIdentifier*, and finally sends back a CELL_CREATED_REJECTED message containing its DH public key and the shared secret's hash digest. It now considers the originator of the 'initiator message' as a valid predecessor.
6. The node is now able to decrypt messages coming through the circuit. If it receives an ADDITIVE_SHARING message through it, the node generates its own *circuitId* and sends the contained parameters of the decrypted message in an 'initiator message' to the intended node, waiting for a response (only for a limited time, of course, before killing the connection).
7. If it receives a rejection, that is the response does not include a DH public key, the connection to the node to extend with is killed. Otherwise it is considered as a valid successor. In both cases the (encrypted) message is piped back through the circuit.

In this way a node tries to maintain a constant number of circuits. If a circuit can no longer be used, a new one is created. It goes without saying that – no matter if it is from the perspective of a relay node or of an originator – a node within a circuit encounters any protocol non-compliance or encryption/decryption errors, the circuit is torn down.

4 Finding files

Peeriod shall support two kinds of locating file information and their whereabouts. One is a 'classic' key-value storage in a way that distributed hash tables are designed for: information about a file is stored on nodes with IDs close to the hash digest of the file. Because Peeriod's initial implementation shall first and foremost act as a testing environment, the conventional key-value storage is a feature which will be implemented in a future release.

The second possibility – which is the one currently featured in Peeriod – is a flooding-based keyword search propagating through the network.

'File information and its whereabouts' refers to a block of data containing a file's name, its exact size, its hash digest and of course information about how to contact the node providing the file. This contact information is nothing more than a list of valid exit nodes chosen from its circuits by providing the file node, i.e. a list of $\langle$IP, Port$\rangle$ pairs with their appropriate *feedingIdentifier*.

Thus, a node has the potential to 'publish' this file info, either via the key-value storage or by responding to a flooding-based search query (see below). Then another node gains the possibility to 'feed' a circuit of this node with the help of the *feedingIdentifier* (as an exit node can relate a known *feedingIdentifier* to a specific circuit it participates in). In short, it can send a message to a node without needing to know its identity.

Upon this principle the whole concept of finding a file is built, be it via querying the key-value storage or by flooding the network.

4.1 Flooding-based search

This Section explains how a node anonymously initiates a flooding-based search and how it is able to receive responses to the exact same query. *How* an efficient propagation of a query can be put into practice is explained in [1] and the paper referenced therein.

Assuming a node wants to flood the network with a specific search query, it pipes a QUERY_BROADCAST message through all its maintained circuits. A QUERY_BROADCAST message signifies an intended node in the circuit to start flooding the network with a search query. The message consists of the payload carrying the query, e.g. some keywords, and a randomly chosen list of ⟨IP, Port, *feedingIdentifier*⟩ triples referencing exit-nodes of its circuits. These exit nodes can be used by responding nodes to return their results. Furthermore, the QUERY_BROADCAST message includes a random *broadcastId* used to identify the query.

A node in one of the circuits receiving this QUERY_BROADCAST message can now initiate a flooding-based query. It also appends a current timestamp to the query, so other nodes receiving the broadcast query can decide whether the query is still valid – and thus must be propagated – or whether it can be discarded, as it is too old. Moreover, the included *broadcastId* is memorized by nodes receiving the query, in order not to propagate queries twice.

If a node receiving the query finds a match in the batch of its shareable files, it constructs a QUERY_RESPONSE. This response message consists of the *broadcastId* it is referring to, information about proposed files and a list of exit nodes (⟨IP, Port, *feedingIdentifier*⟩ triples) of the respondent's circuits – so the querying node gains knowledge about how to contact the respondent in case of a desired file transferal.

Because the respondent must not disclose its identity either, the QUERY_ RESPONSE message is sent through one of its maintained circuits with the instruction (EXTERNAL_FEED message) to 'feed' (GOT_FED message) the response to one of the exit nodes included in the query. Before actually sending the GOT_FED message however, the exit node which is about to feed another node at first sends a FEED_REQUEST message. This is to ensure that the other side is still part of a circuit to which it can relate the *feedingIdentifier*. This request is either accepted (FEED_REQUEST_ACCEPT) or rejected FEED_REQUEST_REJECT. In the latter case the exit node tries to feed another circuit (if present).

This 'feeding of circuits' allows both the querying and the responding node to stay anonymous.

5 Requesting and transfering files

This Section explains the situation of a node which has received a response to one of its queries deciding to download one of the proposed files.

Wrapping up the current situation, two nodes can now anonymously communicate with each other by always appending a list of ⟨IP, Port, *feedingIdentifier*⟩

triples to a message and letting an exit node of one of their circuits 'feed' a circuit of the opposite side.

In the following, 'A sends a message to B' refers to A knowing of B's exit nodes. A appends the list of its own exit nodes to the message, pipes it through one of its circuits in an EXTERNAL_FEED message, instructing the receiving exit node to feed a circuit of B. Supposing node A wants to download a file proposed in a response sent by node B, the process of requesting and downloading a file is as follows:

1. A sends to B a SHARE_REQUEST message containing the hash digest of the file A wants to download. A furthermore adds the public key of a Diffie-Hellman handshake to the message, as well as a 16-byte *randomIdentifier*. If B answers, B needs to mark his response with this *randomIdentifier*, so A is able to correctly relate the response to its file request.
2. If B receives the SHARE_REQUEST message and it is in fact able to provide the requested file, it generates its own half of the DH handshake, computes the shared secret and derives an incoming key for decryption and an outgoing key for encryption. Moreover, B derives another 16-byte identifier which A must mark its next message with. B can now respond with a SHARE_RATIFY message, including the DH public key, the hash digest of the shared secret, as well as (already encrypted) the filename and exact size of the file it is about to upload. B marks the SHARE_RATIFY message with the *randomIdentifier*.
3. If A receives the SHARE_RATIFY message, it can compute the shared secret itself, derive the keys and the expected identifier of the next message. A decrypts the filename and filesize and checks if it matches the information it already received through the response to the query. If everything matches, A starts requesting chunks of the file.
4. What then follows is a constant question-and-answer flow: A requests a chunk of data with a BLOCK_REQUEST message, marking the message with the derived identifier and appending another random identifier to the encrypted message (which again B must use for its next message). B responds to the request with a BLOCK message including the data chunk, also appending the next expected identifier. A processes the received data, requests the next chunk, B responds with the chunk etc. etc. One side always marks its message with the identifier it received in the most recent message and provides on its part the opposite side with an identifier for the next expected message.
   The identifier which a message is marked with is always sent in the clear, whereas the next expected identifier is included in the encrypted payload.
5. In this fashion the whole file is transferred until either something goes wrong (e.g. one side fails to reply within a given time frame, an encryption/decryption error happens, the file transfer is manually aborted...) or A acknowledges a complete download by sending a BLOCK_REQUEST asking for a chunk starting at the last byte of the file + 1.

6 Indexing files and matching queries

In order to make files available to others, users can add folders to their 'shared folders', whose contents will be analyzed and indexed. For this analysis and indexation process, Peeriod utilizes a unitized plugin architecture.

Plugins register on specific file formats; when a folder is being indexed, parts of a contained file are passed to the interfaces of the appropriate registered plugins, each of which classifies the file by creating its own index which is then stored in a database. Hence, the database can contain multiple perceptions of the very same file. The benefit of this modular design is that one is able to easily add and remove different data analysis methods, which all act independently from each other.

Therefore, given arbitrary keywords, a valid file query is created by an activated plugin by populating its parameters with these keywords. Moreover, plugins can provide their own detailed search fields to the user interface. Thus, a node receiving such a detailed query can significantly narrow down potential results:

Incoming queries are run against the database which returns matching indices. In the best case matching indices are associated to the plugin which was used to construct the query with in the first place. As each plugin *always* adds the name and the hash digest of the file to its created index, a basic match-keyword-to-filename fallback is provided regardless of currently used plugins.

Sensitive data (file system information etc.) is stripped off each index before issuing a response. As soon as the querying node receives such a response, it runs its own query against the included indices again, adding a 'result score' (the higher, the more likely the indice represents a desired file). This two-step verification is a basic mechanism for preventing that spam or results which have been tampered with are regarded as valid results and are presented to the user.

As opposed to conventional client-server search engines using a central index, in Peeriod obviously a querying node cannot rely on receiving results in a ranked order. Hence, there seems to be a certain contradition between low-latency and high-score results. On the one hand results should be returned to the user interface as fast as possible – speed is paramount. On the other hand, however, there is always the possibility that results with a higher score may still be received. This challenge is a usability problem, though, which can mainly be solved by the user interface.

7 User interface

Peeriod's user interface resides in the browser (in the current implementation in the form of a Chrome browser extension). This has several reasons. One is to avoid the difficulty which interfaces of cross-platform applications have to face when trying to embed them into the overall usability of an operating system. Often the resulting compromise tries to, but unfortunately fails to fit into its surroundings, leaving behind a slightly awkward user experience.

Web browsers, however, are inherently being considered as windows to the vast world of the Internet – a fact which greatly increases the willingness of an user to accept and come to terms with unconventional and new interfaces. Moreover, browser technologies increasingly influence operating systems, displacing or merging into known environments (for example Google's Chromebook).

Regarding Peeriod, the only part of the application which is still optically situated within the operating system's environment is a background application whose small icon is visible in the status/menu bar.

Hoisting the user interface into the browser necessarily means that it must also be started when the application is launched – and vice versa. Because many users constantly have their web browsers running, this might effectively increase the number of nodes in the network, stabilizing and maintaining its topology and anonymity in the background.

Putting the interface inside a browser also means being able to access it from almost every device: Having a single machine within a local network which is running the Peeriod application and searching/downloading from a mobile device or even a TV is thus perfectly possible.

8 Technologies used and additional feature list

The Peeriod project is an experiment and the application should be regarded front and foremost as a proof-of-concept implementation.

Concepts and technologies may change, may never change, or may be exchanged completely. Nevertheless and for the sake of completeness we want to provide a description of the technologies/platforms used in the current implementation.

Peeriod's source is written for Node.js [5], a platform utilizing Google Chrome's JavaScript runtime v8 [6] and libuv [7], a C library for asynchronous I/O. The desktop application is created for node-webkit [8], a runtime based on Chromium [9] and Node.js. Node-webkit combines the event-processing of both Node.js and Chromium, and provides a way for them to access each other's context. This gives developers the possibility to develop native, cross-platform applications with web technologies only.

Hence, Peeriod's core is written entirely in JavaScript, a fact which contributed greatly to permitting a rapid development.

For storing a node's contact information in the buckets, we use the Lightning Memory-Mapped Database (LMDB) [10], a compact and fast key-value database.

For indexing files and matching them against incoming queries, ElasticSearch [11] (a search server which is based on Apache Lucene [12]) is utilized in combination with Apache Tika [13] for analysis of file contents.

The user interface is set within a Chrome extension [15], its reactive data rendering is based on React [16]. Peeriod's core communicates with its user interface via WebSockets with added multiplexing capabilities.

The following is a list of current implementation issues and future features (not ordered by priority):

1. **GCM authentication tag**
   We decided not to rely on unstable releases of the above mentioned technologies. Unfortunately when encrypting/decrypting with AES in Galois Counter Mode, Node.js exposes a way to access the resulting authentication tag only in its unstable version 0.11.13 at the time of writing. Therefore Peeriod currently

uses an authentication tag of 128 zero bits for all its messages. The authentication part is skipped. As this is a severe security issue of the implementation, it obviously will be fixed as soon as possible.

2. **Message sequence numbers as AAD**
   Related to the *authentication tag* issue, using message sequence numbers as additional authenticated data (AAD) in encrypted communication should be used. This is not implemented yet.

3. **Use UDP for Kademlia RPCs**
   A main planned enhancement is switching from TCP to UDP transport when sending Kademlia RPCs and flooding-based queries. This will include changes in the message protocol though.

4. **Implementation of a basic key-value storage**
   This planned feature refers to storing information about a file on nodes with IDs close to the file's hash digest.

5. **Additional node discovery options**
   More concepts for how a node entering the network can find its initial contact node should be implemented.

6. **Port forwarding with UPnP**
   Peeriod requires users to manually open ports in their router if they do not want to rely on proxies or wish to be part of other nodes' circuits as well. This process of enabling port forwarding can be simplified using the Universal Plug and Play (UPnP) Protocol.

7. **Multiple hops in additive sharing**
   The additive sharing scheme can be refined by using multiple hops when relaying shares, so the initiator of a circuit is able to further hide the fact *that* it is extending a circuit.

8. **Regularly changing circuits**
   Peeriod currently only tears down constructed circuits when encountering behaviour which is not compliant to the protocol, encryption/decryption errors, or when any TCP connection within the circuit ends. This means that theoretically a node may never change it's circuits from application launch to exit. This can be changed to regularly tearing down circuits and exchanging them with new ones.

9. **DoS prevention**
   Analysis of and defense mechanisms against denial-of-service attacks should be improved.

10. **Bandwidth control and round-trip time**
    If needed, Peeriod takes up all your bandwidth. Users should however be able to impose limits on bandwidth usage. Moreover, circuit speed may be improved by measuring the round-trip time for ping-pong messages and choosing nodes accordingly.

11. **Parallel data block transmission when downloading/pausable downloads**
    Presently when downloading a file from another node, data blocks are requested and acknowledged one by one, until the whole file has been transferred. This scheme can be improved by requesting multiple blocks from the uploader node in parallel, using varying ciruits. Furthermore, a mechanism for pausing and resuming downloads can be provided.

12. **Seeding files from multiple providers**
    Related to parallel block transmission between the downloader and only one uploader, file transferal can be significantly sped up by downloading one file from multiple sources, if present. For example this mechanism is being successfully used by the BitTorrent protocol [14].

13. **Improved file system handling**
    Regarding indexing and analysing files which a node wants to provide, the mechanism of detecting and handling changes in the file system like renaming, moving or copying files should be improved. Moreover, there are still problems occuring when using e.g. external drives as file source.

14. **'Raw node' applications**
    A useful feature would be to provide application bundles which are decoupled from the user interface as well as the file provision mechanisms. Thus, one is able to run 'raw nodes' which only exist in order to stabilize the topology and improve the anonymity of other users.

15. **Sandbox**
    The process of matching search queries to files and the analysis of file contents is happening within a sandbox whose implementation must be significantly improved or even rebuilt completely.

16. **Third-party software components**
    A very nice feature would be providing possibilites of hooking third-party plugins to the application in order to empower interested contributors to develop own search algorithms and file content analysis. Without proper code review or satisfying sandboxing mechanisms though, a truly decentralized distribution of third-party plugins must remain for the time being a dream of the future.

17. **Spam/manipulated results avoidance**
    More research and effort can be put into using mechanisms to further avoid spamming and tampering with search results.

18. **Enrich search results with type-specific information**
    In the future, responses to queries shall contain attributes which are specific to the respective file type, for example showing thumbnail images or providing content summaries.

19. **GUI: Parallel queries**
    Although in Peeriod a user is able to handle file queries in parallel, there is currently no resemblance of simultaneous queries in the user interface.

20. **Adapting search fields**
    Each search plugin handles its query-to-file matching differently. So an improvement would be to implement 'advanced search fields' into the user interface for detailed filtering capabilities. Thus, when for example looking for text documents by a specific author, the name of the author can already be included in the query to signifcantly narrow down results.

21. **GUI: Personalized interface and visual feedback of status in network**
    The graphic user interface should necessarily mirror a node's current status in the network, i.e. has it successfully joined, how many circuits does it maintain, does it need a proxy, how many circuits is it a relay node of, etc. By displaying how many bytes a node has already uploaded since installation of Peeriod, a user may also build up some personal connection to the node (numbers going up!).

22. **External viewers**
    Peeriod's application design includes the possibility to potentially search and download from everywhere within a home network, even a TV. How nice would it be if you could search, download, *and even watch* a movie using only your TV?

23. **Securing the data stream to the user interface**
    As just mentioned, Peeriod's user interface can theroetically be accessed from everywhere within a local network. Therefore, the possibility of securing this data stream from Peeriod's core to the user interfaces must be taken into consideration.

A Message formats

This appendix contains a listing of the different message formats Peeriod uses in its current implementation. The message formatting is far from perfect and in certain cases message padding should be implemented to impede guessing remaining path lengths within circuits by message size analysis.

Each protocol message begins with four bytes indicating its size, i.e. the number of bytes which will follow. In the following we refer to this amount of bytes after the four size indicating bytes as the 'message'. 0x00000000 merely indicates a TCP heartbeat.

We differentiate between *topology messages* (non-anonymous messages including node IDs and contact/proxy information etc.) and *hydra messages* (anonymity-related messages without any topology-relevant information).

If a message is topology relevant, the first 20 bytes are reserved for the ID of the intended receiver. Otherwise if the first 20 bytes are all zero, it is a hydra message. This differentiation will be altered in a future release, because these 20 bytes can actually be reduced to one indicator byte in the beginning for hydra messages. Moreover, this means that a node with the id 0 may not exist. This is a current painful and useless misconception.

A.1 Topology messages

All topology messages start with a header containing:

- 20 bytes for the ID of the intended receiver.
- 20 bytes for the ID of the sender.
- A varying amount of bytes for the address block of the sender providing information about how the sender can be reached. For each ⟨IP, Port⟩ pair applies:
  - 1 byte for the kind of IP: 0x06 for IPv6 addresses, 0x04 for IPv4 addresses.
  - Then the IP address itself: 16 bytes for IPv6, 4 bytes for IPv4.
  - At last 2 bytes for the open port.

  The end of the address block is indicated by the octet 0x05.
- The header is concluded by two octets for the type of the payload. (Note: This can actually be reduced to one byte. We will never cross the limit of 256 message types...)

The header is directly followed by the payload of the according message type. A list of topology-related payload types in alphabetical order and their content follows:

ADDRESS_CHANGE (0x5005)
The payload is empty. Indicates a node that the sender's address has changed, but does not expect a reaction (as opposed to PING).

BROADCAST_QUERY (0x4252)
This type of message is propagated through the network when performing a flooding-based search. Its payload consists of:

- 16 bytes for the identifier of the broadcast.
- 8 bytes for a timestamp indicating when the query has been initiated.
- A varying number of bytes for the actual query payload.

FIND_CLOSEST_NODES (0x4649)
The payload merely consists of 20 bytes for ID which was sought-after.

FOUND_CLOSEST_NODES (0x464f)
The payload consists of:

- 20 bytes for the ID which was sought after.
- A varying amount of bytes for the list of found close nodes.
  For each node applies:
  - 20 bytes for the ID of the found node.
  - The found node's address block (formatted equally to the address block in the header)

PING (0x5049)
The payload is empty.

PONG (0x504f)
The payload is empty.

PROXY_ACCEPT (0x5002)
The payload is empty. PROXY_ACCEPT signals to the receiver that the sender accepts its request for a proxy.

PROXY_REJECT (0x5003)
The payload is empty. PROXY_REJECT signals to the receiver that the sender rejects its request for a proxy.

PROXY_REQUEST (0x5001)
The payload is empty. This message asks the receiver to act as a proxy for the sender.

PROXY_THROUGH (0x5004)
The payload is the whole message (header and payload, but without size-indicating bytes) which was sent to the proxy and which it now passes on to the actual intended receiver.

A.2 Hydra messages

As stated earlier, hydra messages begin with 20 zero bytes and do not carry any topology-relevant information. The 21st octet indicates the payload type. In the following list, 'message' refers to the payload starting from the 22nd byte.

ADDITIVE_SHARING (0x01)
The message starts with the ⟨IP, Port⟩ combination of the node to which the included payload must be relayed to. A node receiving such a message takes the payload, wraps it within a CREATE_CELL_ADDITIVE message and sends it to the specified node.

- The ⟨IP, Port⟩ pairs of the node to relay the included payload to:
  - One octet for the IP type: 0x06 for IPv6, 0x04 for IPv4
  - The IP: 16 octets for IPv6, 4 octets for IPv4
  - 2 octets for the number of the open port.
- The address is directly followed by the complete payload (without message type indicator byte!) of a CREATE_CELL_ADDITIVE message.

CELL_CREATED_REJECTED (0x05)
A CELL_CREATED_REJECTED message is a response to a request asking to be part of a circuit (i.e. the response to a full batch of CREATE_CELL_ADDITIVE messages). CELL_CREATED_REJECTED messages must be sent to the originator of the 'initiator message' of an additive sharing batch.

- 16 bytes for the *circuitId* which was extracted from the 'initiator message'.
- 16 bytes for the *uuid* of the additive sharing scheme this response refers to.
- If the request is not rejected: 20 bytes for the hash digest of the shared secret.
- If the request is not rejected: 256 bytes for the padded generated Diffie-Hellman public key.

CREATE_CELL_ADDITIVE (0x02)
Messages of this type are used for constructing circuits and carry one share of the additive sharing scheme.

- 1 byte indicating whether it is an 'initiator message' or not. 0x00 for yes, 0x01 for no.
- 16 bytes for the *circuitId* if it is an 'initiator message'.
- 16 bytes for the *uuid* of the additive sharing batch this message belongs to.
- 256 bytes for one share of the padded Diffie-Hellman public key.

ENCRYPTED_DIGEST (0x04) and ENCRYPTED_SPITOUT (0x03)
These two message types are used for sending encrypted messages through a circuit.

An ENCRYPTED_SPITOUT message is one which originates from a circuit's initiator and which carries a layered-encrypted payload. A node receiving such a message decrypts the payload and exchanges the *circuitId* from the message with the *circuitId* it shares with its successor, before sending it on.

All ENCRYPTED_SPITOUTs consist of:

- 16 bytes for the *circuitId*
- 12 bytes for the initialization vector
- The complete encrypted payload
- 16 bytes for the authentication tag
  (only present if message has reached receiver)

On decrypting the encrypted payload, the first decrypted byte indicates whether the message has received its destination (and must not be sent on) and can be interpreted (0x01) or not (0x00). Thus, a node receiving an ENCRYPTED_SPITOUT message decrypts it and checks whether it is the intended receiver itself. If not, it merely removes the first byte from the decrypted payload, exchanges the *circuitId* and send it on.

On the other hand, if the node is the intended receiver, the last 16 bytes of the message may not be decrypted, as they constitute the authentication tag. Messages within circuits are only authenticated edge-to-edge. The resulting fully decrypted payload is treated as a hydra message (without the initial 20 zero bytes), i.e. the first byte of the fully decrypted payload again indicates the message type.

As soon as the key negotiation within a circuit is completed, the relay nodes and the initiator of the circuit always expect ENCRYPTED_SPITOUT or ENCRYPTED_DIGEST messages and do not react to cleartext messages.

FILE_TRANSFER (0x06)
FILE_TRANSFER messages relate to some sort of file transferal, querying and requesting. FILE_TRANSFER messages always start with a 16 byte *transferIdentifier* which can have different meanings. The *transferIdentifier* is directly followed by one octet indicating the subtype of the message – thus FILE_TRANSFER messages can be split up again into different submessages. After the indicator byte the payload directly follows.

A.3 File transfer submessages

In logical order. File transfer messages often contain a *feeding-node-block*, i.e. the list of exit nodes (⟨IP, Port, *feedingIdentifier*⟩ triples) which can be fed messages to pipe them back through their related circuits.

A feeding-node-block consists of the following parts:

- 1 byte indicating the number of nodes.
- For each node applies:
  - 16 bytes for the node's *feedingIdentifier*.
  - 1 byte for the IP type: 0x04 for IPv4 or 0x06 for IPv6.
  - The IP address: 16 bytes for IPv6, 4 bytes for IPv4.
  - 2 bytes for the open port.

EXTERNAL_FEED (0x02)
transferIdentifier: 16 zero bytes. The payload is:

- The feeding-node-block with which the receiver determines who feed the payload to (as described above)
- The payload to feed: This is the whole payload of a FILE_TRANSFER message (i.e. *transferIdentifier* and submessage indicator byte).

FEED_REQUEST (0x08)
transferIdentifier: 16 bytes for the *feedingIdentifier* of the node which is about to get fed.

The payload is empty. A node receiving a FEED_REQUEST message must either reject (FEED_REQUEST_REJECT) if it is no longer part of a circuit which it can relate the *feedingIdentifier* to, or accept (FEED_REQUEST_ACCEPT).

FEED_REQUEST_ACCEPT (0x09)
transferIdentifier: Must be similar to the *transferIdentifier* used in the FEED_REQUEST this message refers to.

The payload is empty.

FEED_REQUEST_REJECT (0x10)
transferIdentifier: Must be similar to the *transferIdentifier* used in the FEED_REQUEST this message refers to.

The payload is empty.

GOT_FED (0x03)
This is the corresponding side of an EXTERNAL_FEED message, i.e. the message type a node uses when feeding an exit node of a circuit. The *transferIdentifier* in this case is the *feedingIdentifier* of the node which gets fed, so it knows which circuit to choose.

The payload is the same complete one extracted from the EXTERNAL_FEED message.

Although GOT_FED messages can of course contain encrypted information (see share messages below), the message itself is sent in cleartext from one circuit to the other.

QUERY_BROADCAST (0x01)
QUERY_BROADCAST messages are piped through a circuit to an exit node, signaling to the receiver to initalize a flooding-based search with the contained query body. The *transferIdentifier* is chosen randomly and must be used as *broadcastId* by the recipient when starting to flood the network. The payload of a QUERY_BROADCAST message consists of a feeding-node-block chosen by the message originator, and the actual query body. The node receiving such a message uses the contained information to correctly construct a BROADCAST message.

QUERY_RESPONSE (0x04)
QUERY_RESPONSE messages are fed to an exit node which pipes it back through its circuit. Messages of this type are responses containing file suggestions. The *transferIdentifier* of a QUERY_RESPONSE message matches the identifier of an earlier received query which the file suggestions refer to.

The following file transfer message types are used when actually requesting and transmitting a file. They are always sent anonymously from the originator to the intended recipient by feeding each other's circuits with EXTERNAL_FEED and

GOT_FED messages. They always contain a feeding-node-block (only unencrypted in SHARE_REQUEST messages) to signal to the opposite side how the originator of the message can be reached.

SHARE_REQUEST (0x05)
The *transferIdentifier* is chosen randomly. The recipient responding with a SHARE_RATIFY message must use this *transferIdentifier* for its response.

– The feeding-node-block chosen by the sender.
– 20 bytes for the hash digest of the requested file.
– 256 bytes of the sender's padded Diffie-Hellman public key.

SHARE_RATIFY (0x06)
If a requested node can in fact provide the desired file, it responds with a SHARE_RATIFY message. A message of this type concludes the key exchange and confirms the file request by providing its exact size and name. The *transferIdentifier* must match the identifier used in the earlier received SHARE_REQUEST message.

– 256 bytes for the sender's padded Diffie-Hellman public key.
– 20 bytes for the hash digest of the shared secret.
– Already encrypted with a derived symmetric key:
  - The feeding-node-block chosen by the sender.
  - 8 octets for the size of the provided file.
  - A varying number of bytes for the UTF-8 string representation of the provided file's name.

ENCRYPTED_SHARE (0x07)
The payload of ENCRYPTED_SHARE messages is – as the name already states it – completely encrypted. These messages are sent between two parties (by feeding each other's circuits) as soon as they have negotiated symmetric keys for file transferal. The *transferIdentifier* of ENCRYPTED_SHARE messages is constantly changing: A node A always includes in the message's encrypted payload the identifier which a node B must use for a successive message. An only exception is the first message following a SHARE_RATIFY message, as in this case the expected *transferIdentifier* can be derived from the shared secret.

The first octet of the decrypted payload of an ENCRYPTED_SHARE message indicates the type of another three possible submessages:

SHARE_ABORT (0x01)
A SHARE_ABORT message gracefully signals the abortion of a file transmission. Thus, SHARE_ABORT messages can be sent by either downloader or uploader. The payload consists of information about the shared file to impede forged abortion messages:

– 8 bytes for the file's size.
– 20 bytes for the file's hash digest.
– A varying number of bytes for the UTF-8 string representation of the file's name.

A SHARE_ABORT message does not include an identifier for a successive message, as it marks the end of a file transmission.

BLOCK_REQUEST (0x02)
Requests a block of data taken from the file, starting from the specified byte position. It consists of:

– The feeding-node-block chosen by the sender.
– 8 bytes for the position of the the first byte of the expected block within the whole file.
– 16 bytes for a random identifier which must be used by the recipient in a successive message.

BLOCK (0x03)
The response to a BLOCK_REQUEST message, carrying a block of data.

– The feeding-node-block chosen by the sender.
– 8 bytes for the position of the first byte of the expected block within the whole file.
– 16 bytes for a random identifier which must be used by the recipient in a successive message.
– A varying number of bytes for the block of data.

References

[1] J. Pirnay, J. Röder. *Peeriod: An Anonymous Approach for Decentralized Overlay Networks*. 2014.

[2] *Kademlia: A Design Specification*
http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html

[3] *RFC 3526: More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*
http://rfc-editor.org/rfc/rfc3526.txt

[4] *RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*
http://tools.ietf.org/rfc/rfc5869.txt

[5] http://nodejs.org

[6] https://code.google.com/p/v8/

[7] https://github.com/joyent/libuv

[8] https://github.com/rogerwang/node-webkit

[9] http://chromium.org

[10] http://symas.com/mdb/

[11] http://elasticsearch.org

[12] http://lucene.apache.org

[13] http://tika.apache.org/

[14] http://bittorrent.org/

[15] https://developer.chrome.com/extensions

[16] http://facebook.github.io/react/