SAND Protocol Specification

Version 1.0

1 General Aspects

1.1 Protocol Description

SAND (recursive acronym for *SAND Anonymous Distribution*) is an application layer network protocol on top of the TCP/IP stack. It makes use of a worldwide P2P network to allow anonymously distributing files on the internet.

The participants in the network are:

- Peers: Normal nodes which exchange files between them
- DNL Nodes: Nodes which serve the role of providing an initial set of neighbors for new peers. The addresses of these nodes should be publicly known. DNL nodes share a distributed list of all the peers in the network between them, which is periodically synchronized.

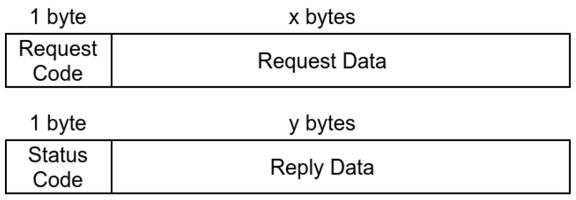
In this P2P network each node can act as proxy for another node to provide anonymity in communication. The two parties which exchange a file between them are completely unknown to each other. This is achieved by the means of a file searching algorithm that recursively propagates a request through the network while probing each node until the file is found.

SAND can additionally use UPnP to programmatically add a port mapping to the Internet Gateway Device. This will allow incoming messages on the port on which the client application is listening, if the device is sitting behind a NAT capable gateway. Other mechanisms for NAT traversal can be used as well, depending on the protocol implementation.

1.2 Messages

The protocol consists of a number of messages (also called requests), which may or may not receive a reply, depending on the type of the message. The request contains a 1-byte code which identifies the request type. The rest of the message is occupied by auxiliary data which are specific to each request type. The reply format is similar, but instead of a request code, it contains a 1-byte status code.

The status code can be interpreted in various ways, depending on the request type.



Request codes are divided in 8 categories, each having 32 values:

- Codes 0 - 31: Reserved / experimental
- Codes 32 - 63: Peer discovery
- Codes 64 - 95: File searching
- Codes 96 - 127: File transfer
- Codes 128 - 255 (4 categories): Unused

The following sections will present the main purpose and formats for the type of messages exchanged between nodes. If the reply format is missing, then the reply only contains the status code, unless specified otherwise. If the request format is missing too, then the request only contains the request type.

1.3 Status Codes

The only status codes defined by the specification are:

- 0 - Success
- 1 - Generic Error

Other status codes may be defined by implementations.

2 Peer Discovery Messages

2.1 PULL (32)

The PULL message requests IPv4 addresses of other nodes in order to expand its list of neighbors. This message can be sent to an existing neighbor or a DNL node. The number of addresses returned can be smaller than the number of addresses requested. The list of addresses from the reply can be missing if the status code indicates an error.

1 byte	1 byte
Request Code	Address Count

1 byte	1 byte	4 bytes	4 bytes	
Status Code	Address Count	Address 1	Address 2	...

2.2 PUSH (33)

PUSH can be sent by a node to make itself known to its new neighbors of which addresses were retrieved using the PULL message. The nodes which are receiving this message will add the address of the message source to its list of neighbors.

2.3 BYE (34)

BYE should be sent by a node to notify its neighbors and one of the DNL nodes that it will leave the network so they can remove the node from the neighbor list and perform proper cleanup. There is no reply for this message.

2.4 DEAD (35)

Used to notify a node's neighbors and DNL nodes that some other nodes were found to be dead (not responding). These dead nodes did not send the BYE message before leaving (this may be due to a software crash or power failure). The nodes which are receiving this message should check if the specified nodes are indeed dead by pinging before removing them from the neighbor list.

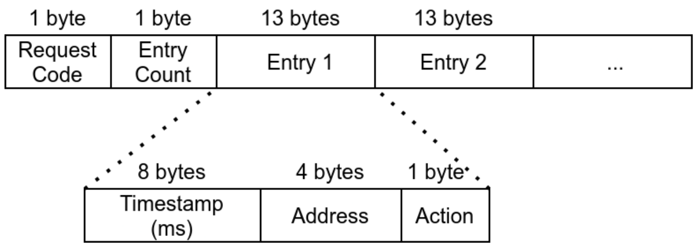
1 byte	1 byte	4 bytes	4 bytes	
Request Code	Address Count	Address 1	Address 2	...

2.5 PING (36)

PING is used in conjunction with the DEAD message. Pings a node to confirm that it indeed left the network without sending the BYE message.

2.6 DNLSYNC (37)

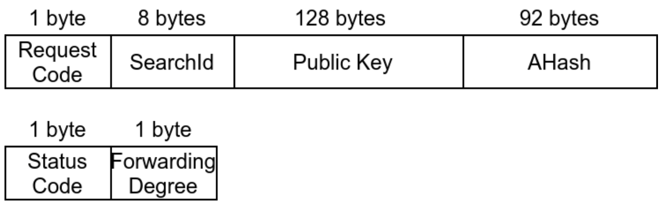
This is a special message sent between DNL nodes to synchronize the distributed data structure. Updates are accumulated and periodically sent in bulk using a single message. The message data itself contains a list of entries, each representing an update. An entry contains the type of the action (0 for *Add Address* and 1 for *Remove Address*) , the address to be added or removed and the time point at which the action was originally performed. The timestamp is crucial for resolving potential conflicting updates - for example a DNL node informs the others that a peer has joined the network while another informs that it left instead.



3 File Searching Messages

3.1 SEARCH (64)

This is the central part of the search request recursive propagation sequence. *SearchId* is an unique randomly generated number. *Public Key* is the 1024-bit RSA public key of the search initiatior. *AHash* is the identifier of the file to be searched for. It is the concatenation of the SHA-3-512 of the file content and the SHA-3-224 of the file name concatenated with the textual representation of the file size. A node receiving this message should remember (for a limited, implementation defined time) the SearchId along with address of the node from where it came. This information is vital for forwarding potential OFFER messages to the right peers. It is also useful for ignoring other SEARCH messages with the same SearchId - this prevents search cycles. The message is then forwarded to some, or all neighbors. The reply contains the number of peers to which the message will be forwarded on the next hop, i.e. the number of neighbors minus one. For now, this number is not used for anything, but it might be in the future (for blind alley notifications for example). The Public Key should be regenerated for each SEARCH message to avoid identification by an adversary.

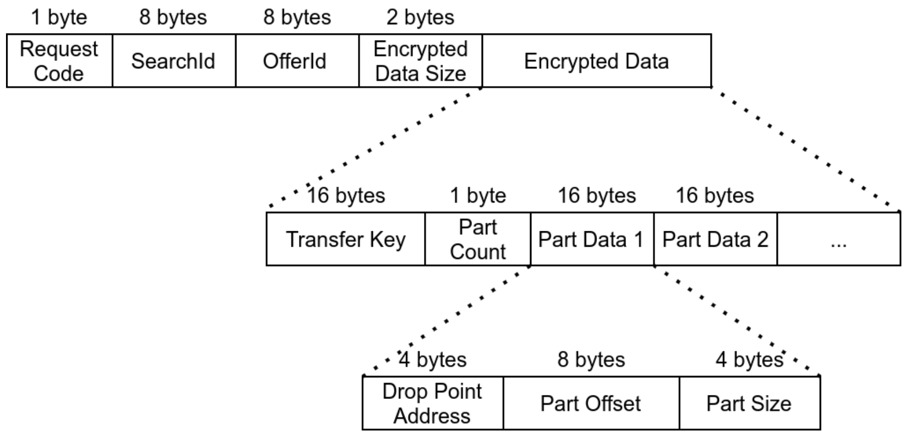


3.2 OFFER (65)

This message is initiated by a peer willing to provide a file requested by the means of a SEARCH message. It is also propagated recursively, but directly through the succesful search path, without branching out to all neighbors. This is achieved by using the stored SearchId and request source by each node in the respective search path. Propagation stops when this message is recieved by the search initiator.

The message itself contains the related SearchId and another randomly generated unique number, called *OfferId*. The rest of the message contains information encrypted using the Public Key contained in the SEARCH message. OfferId will be subsequently used for facilitating the established communication between the search initiator and the offer initiator, which are completely unknown to each other. In order for transfer confirmations to work correctly, each node that this message passes through has to remember the OfferId along with the address of the local source. Another use for OfferId is to uniquely identify a file transfer. The encrypted part of the message contains a transfer key and a list of part data structures used for the actual file transfer and are therefore discussed in the section detailing file transfers.

A useful feature of file searching is the path caching mechanism, in which multiple peers collectively remember the network path to the provider of a certain file. Multiple paths can exist for the same file. No single node knows the whole path, they just cache the addresses of the next hops for a given AHash. A future SEARCH request for this AHash will be forwarded just through the cached paths, instead of branching out to all local neighbors.



3.3 UNCACHE (66)

In the unfortunate and inevitable event that a peer which is part of a cached search path leaves the network, the part of the path up to the current node is left invalidated. There needs to be some kind of mechanism for notifying the previous node that the next hop is no longer alive. The UNCACHE command does just that. It is propagated backwards through the cached path which was invalidated, so the peers can remove the path information from their local cache. A cached path can branch out in multiple linear paths. The cache invalidation message is propagated only up to the branching point. If this message reaches the file requester, then the searching procedure is restarted without using any cache.

1 byte	92 bytes
Request Code	AHash

3.4 CONFIRMTRANSFER (67)

This message is initiated by the same peer which initiated the search request. It is propagated backwards through the same path which was followed by the related OFFER message. It leverages the stored information on each node to find its way to the offer initiator, along with the OfferId parameter in the message data. When the offer initiator receives this message, it starts the file transfer process.

1 byte	8 bytes
Request Code	OfferId

4 File Transfer Messages

4.1 REQUESTPROXY (96)

The REQUESTPROXY message is sent by the parties of the file transfer to verious nodes in the following mannger: The encrypted part of the OFFER message contains a list of "part data" structures. Each part data represents a part of the transfered file. Each such part will be sent by the offerer to a different node, randomly chosen from its list of neighbors before initiating an offer. If he does not have enough neighbors, then more addresses should be requested from a DNL node using PULL. Before making the offer, the offerer has to request the randomly chosen nodes to have the role of proxies in the transfer, using the REQUESTPROXY message. A node may or may not accept such a request. When accepting, we say that the node becomes a "drop point" in this transfer. When the transfer begins, the offerer will start uploading a part of the file to each drop point. The party which requested the respective file will know where to get the parts - the addresses are recorded in the part data structures.

This message type will also be utilized by the file requester, as he will not retrieve the parts directly from the drop points, but will ask other nodes to do this on his behalf. These nodes are called "lift proxies".

The message content only has one field, the size of the part to be handled by the proxy. The status code specified whether or not the node accepts to have the role of lift proxy or drop point. The acceptance criteria are dependent on implementation or configuration.

1 byte	8 bytes
Request Code	Part Size

4.2 CANCELPROXYREQUEST (97)

A proxy request may be canceled by one of the parties. This type of message is implementation defined and may not be implemented at all if not needed.

4.3 INITUPLOAD (98)

INITUPLOAD is used to initialize a part upload between nodes. The upload procedure initialized by this message is performed:

- From file offerer to drop point.
- From lift proxy to file requester.

The OfferId is used to identify the transfer and the file. The drop point will know how large is the part based on the REQUESTPROXY message received earlier, matched against the address of the message sender.

1 byte	8 bytes
Request Code	OfferId

4.4 UPLOAD (99)

The start of the part upload sequence by a node has to be preceded by an outgoing INITUPLOAD message or an incoming INITDOWNLOAD message. The part is transfered in chunks, using multiple UPLOAD messages. The transfer is identified by the part receiver by matching the sender address of the INITUPLOAD and UPLOAD messages. The file chunks may be sent out of order, and thus the message contains the chunk offset in the current part. The proxies involved in the transfer won't need to know the offset of the part in the file. The part is encrypted by the file offerer before uploading it to the drop point, using the transfer key recorded in the OFFER message. In this way, only the the two main parties involved in the transfer are able to view the file without fearing that a third party may intercept the transfer and read the file content. A node involved in the transfer does not have to wait for the whole part before sending the received chunks to the next hop.

1 byte	4 bytes	4 bytes	"Size" bytes
Request Code	Offset	Size	Data

4.5 FETCH (100)

FETCH is the message send by the file requester to a lift proxy to ask it to fetch a file part from a drop point. The ”Address” field specifies the address of the drop point.

1 byte	8 bytes	4 bytes
Request Code	OfferId	Address

4.6 INITDOWNLOAD (101)

INITDOWNLOAD is sent by a lift proxy to a drop point, to inform the drop point node to start uploading the stored file part associated with the given OfferId to the lift proxy.

1 byte	8 bytes
Request Code	OfferId