ScatterStream Design

Masala

Table of Contents

ScatterStream

Abstract

The stream package provides an implementation of an asynchronous, reliable, and encrypted communication protocol on top of the Katzenpost network using a storage service abstraction.

Overview

Stream provides provided for establishing reliable bidirectional communication channel between a pair of clients using a shared secret, and a key-value scratchpad service for exchanging messages. From the shared secret, sequences of message storage addresses (32 bytes) and symmetric message encryption keys are derived. Each client runs protocol state machines that fetch, transmit and acknowledge frames of data, and re-transmit unacknowledged frames in order to provide a reliable delivery of data via a lossy storage service. Storage addresses are mapped to nodes published in Katzenpost's Directory Authority system, which runs a service called "Map" that provides a simple lossy storage service where content is limited to a configurable buffer size and automatically expire.

Protocol description

Stream establishes an asynchronous bidirectional reliable communication channel between 2 parties: by convention, one is a Listener and the other the Dialer. The 2 parties must exchange a shared secret, which is used to derive secrets for encrypting and addressing Frames of a Stream. Each Frame is identified by a cryptographically secure random ID chosen from a deterministic CSPRNG so that each party knows the sequence of Frame ID's used to address messages to and from the other party. Similarly, a deterministic sequence of symmetric Frame encryption keys are used to encrypt each Frame. Frames are written to a storage service, so that each party may send or receive messages while the other party is offline, and Frames are acknowledged by transmitting the greatest sequential frame seen. Frames that are not acknowledged are periodically re-transmitted, so that reliable delivery is provided end-to-end by the clients and not the storage service.

Establishing secrets

A cryptographically strong (32 byte) shared secret is used as initial keying material and expanded by HKDF (HMAC-based Key Derivation Function) to produce 4 secrets which are the seeds for a deterministic CSPRNG (SHA256 of the seed + an 8 byte counter) that produce the sequences of Frame ID's and encryption keys for each peer. This implies that each party knows the other party's keys and can impersonate the other party, which is not considered to be a problem for the 2 party design of Stream, however future work may establish secrets from a cryptographic handshake so that read-only capabilities can be exchanged so that multiple readers may read messages from a peer without being able to modify ciphertext on the storage service.

Frame Encryption

The txFrame method is responsible for encrypting and transmitting frames. It uses secretbox from the NaCl library for encryption. Encryption keys are derived from hashing the frame encryption key with the frame ID, and a 24 byte random nonce is prepended to the ciphertext.

Key Derivation

Stream uses HKDF (HMAC-based Key Derivation Function) for deriving encryption keys and frame ID sequence seeds from the provided shared secret.

WriteKey: This is a pointer to a [keySize]byte array, representing the encryption key used for encrypting data frames before transmitting them. It is used in the txFrame function to derive the encryption key for a specific frame during transmission. This key ensures the confidentiality and integrity of the transmitted data.

ReadKey: Similar to WriteKey, ReadKey is a pointer to a [keySize]byte array, representing the decryption key used for decrypting incoming data frames. It is used in the readFrame function to derive the decryption key for a specific frame during reception. This key is crucial for decrypting and processing the received data.

WriteIDBase: This is a common.MessageID (a 32-byte array) representing the base for deriving message IDs during frame transmission. It is used in the txFrameID function to generate a unique identifier for each transmitted frame. The message ID is combined with a frame-specific value to create a unique identifier for each frame.

ReadIDBase: Similar to WriteIDBase, ReadIDBase is a common.MessageID representing the base for deriving message IDs during frame reception. It is used in the rxFrameID function to generate a unique identifier for each received frame. The message ID is combined with a frame-specific value to create a unique identifier for each frame.

Data Frames

The fundamental unit of communication is a "Frame." Frames are CBOR-encoded go structs, and a Frame contains metadata and payload data. Metadata of a frame is the type of the frame (StreamStart, StreamData, or StreamEnd), and an acknowledgment sequence number (Ack). The Payload field in the Frame holds the actual data being transported.

Types of Frames

There are 3 FrameTypes: StreamStart, StreamData, and StreamEnd. The first frame in a Stream must be type StreamStart, following data frames are StreamData, and the final frame must be a StreamEnd.

Stream State

The Stream type manages the state of communication. It includes parameters such as PayloadSize, WindowSize, MaxWriteBufSize, and others.

Addr: is the "address" of a Stream which is the shared secret used to establish the Stream. **Initiator:** is true if the Stream state was established by the Listener party. **WriteKey:** and ReadKey are used to derive frame encryption secrets **WriteIDBase:** The secret derived from the shared secret used to derive each frame ID written. **ReadIDBase:** The secret derived from the shared secret used to derive each frame ID read. **PayloadSize:** is the frame payload length, and must not change after a Stream has been established. **ReadIdx:** The counter corresponding to the current frame being requested, and used to derive the storage location address in combination with ReadIDBase. **WriteIdx:** The counter corresponding to the next frame to be written and used to derive the storage location address in combination with WriteIDBase. **AckIdx:** The counter that keeps track of the last acknowledgement sent to the peer. **PeerAckIdx:** The counter tracking the last acknowledgement sent from the peer. **WindowSize:** The number of frames ahead of the remote peer's PeerAckIDxthat will be transmitted. Peers which peers must agree on WindowSize as Stream does not presently support dynamically adjusting WindowSize. **MaxWriteBufSize is the buffered bytes that Stream will hold before blocking calls to Write.

Stream has separate states for reading (RState) and writing (WState) that correspond to the reader and writer routines. Both finite state machines have 3 valid states: StreamOpen, StreamClosing, and StreamClosed.

Stream worker routines

Each Stream has two goroutines, a reader and writer routine, which are responsible for handling read and write operations, respectively. The reader continuously polls for incoming frames, processes acknowledgments, and updates the read buffer. When the ReadIdx exceeds the AckIdx + WindowSize it prodes the writer routine to send an Acknowledgement, even if there is no Payload data to send. The writer routine transmits frames of data from the write buffer when available, re-transmits unacknowledged frames, and Acknowledges received frames. The writer routine waits until there is more data to send or is signalled by the reader routine in order to send an Acknowledgement. Retransmissions are accomplished by the TimerQueue worker routine that waits until a timeout has occurred and re-sends an unacknowledged frame.

Receiving Frames

The reader routine is responsible for reading frames from a stream. If a read operation returns an error, it increases the interval that it waits before trying to read another frame. On successful reads, the fetch interval is decreased. When a StreamEnd frame is encountered, the reader routine terminates.

Transmitting Frames

The writer routine is responsible for reading frames of data from the write buffer, chunking data into frames, sending frames to the storage service and adding frames to the re-transmission queue. When there is data available to send and WriteIdx-PeerAckIdx < WindowSize, the writer sends Frames of data. When the WindowSize is reached, the writer routine sleeps until woken by the reader routine to acknowledge data or woken by a call to Write. In the latter case, the writer may buffer addtional data but must not transmit additional frames.

Re-Transmission of UnAcknowledged Frames

When the writer routine sends a frame, a copy of the frame is placed into a re-transmisison queue. Unacknowledged frames are periodically re-transmitted. Contents of the re-transmission queue are preserved and restored by Save and LoadStream.

Acknowledging Received Frames

When a sequential frame is successfully read the reader routine updates the AckIdx to ReadIdx, and increments ReadIdx. If ReadIdx exceeds AckIdx by the WindowSize, the reader must signal to the writer routine to transmit an acknowledgement.

Finite State Machine

A Stream consists of two finite state machines, which correspond to a reader and writer thread, each of which can be in state StreamOpen, StreamClosing, or StreamClosed. When a Stream is created, it starts in state StreamOpen.

Transition to StreamClosing (writer):

The Close method is responsible for initiating the process of closing a stream. When Close is called, it first checks whether the stream is in the StreamOpen state. If so, it sets the WState to StreamClosing. After setting the state to StreamClosing, it triggers the doFlush method to wake up the writer goroutine, allowing it to transmit any pending frames, including a final frame indicating the end of the stream. The StreamClosing state indicates that the stream is in the process of closing, and no new data can be written to it.

Transition to StreamClosing (reader):

The reader StreamClosing state is entered when the Close method is called. While in the StreamClosing state, the reader routine continues fetching and processing incoming frames.

Transition to StreamClosed (reader):

If the reader routine encounters a final frame of type StreamEnd during the StreamClosing state, it transitions to StreamClosed.

Transition to StreamClosed (writer):

The writer goroutine (writer method) continuously checks the state of the stream. When it observes that the WState is set to StreamClosing, it proceeds to finalize the stream. During the finalization, it sets the WState to StreamClosed and transmits a final frame of type StreamEnd. After transmitting the final frame, it signals the onStreamClose channel to unblock any blocked calls to Close. The StreamClosed state indicates that the stream is closed, and no further data can be written or read.