One-pager 6

Martin Vassor

(Character count: 3000, `tr -d '[:blank:]' < lib.rs | wc -c`.)

```
/*
   There are 3 components:
    - A sessionHandler which manage new sessions and session re-creation
    - A Session, which is stateful
    - A SessionClient, by definition the initiator of the session.

   The goal is to ensure eventual recovery of a Session failure (loss of consistency or crash).

   The client is charged to keep all messages. Hence, there is no garbage in the Server if the
   session is interrupted, and the client can use these messages to reconstruct a session (method
   `recall_session`).

   Typical usecase:
    Upon detection of failure by F: delete S (shutdown)
    If the error is client-side, not our problem, either it completely stops or try to recover
    If the error is Server-side, the next message from the client is used to reconstruct the
    Session (lazy restart). Transparent from client-side.
 */


trait SessionHandler {
    /* Side-effect: sends an initialization vector to the client. */
    fn initiate_session<T, K>(self) -> (T, K) where T: Session, K: FailureDetector;

    /* Reconstruct a session based on the sequence preceding sm. The sequence of messages is
       verified. The session is reconstructed by locally replaying the messages (same as in Manetho
       protocol). */
    fn recall_session<T, K, S>(self, sm: S) -> (T, K)
```

```
        where T: Session, K: FailureDetector, S: SealedMessage;

    /* Route the message to its Session, if none, call `recall_session`. */
    fn receive_and_route_messages<S, T>(self, sm: S) -> T where S: SealedMessage, T: Session;

    /* Generator of initialization vectors. */
    fn next_key(self) -> i64;

    /* Get all generated initialization vector (depending on the implementation, they are not
       necessary stored, if they can be recomputed). */
    fn prev_keys(self) -> [i64];
}


trait Session {
    type State;

    /* Receives a message. In case of UpdateMessage, apply the update as side effect. If it is a
       request message, sends the current State. */
    fn receive_message<T>(self, T) -> () where T: Message;

    /* This is called by the failure detector. */
    fn delete_session() -> ();
}

trait Message {
    /* Seals the message, typically with a cryptographic hash of the content of
       the message and the seal of the previous message. */
    fn seal<S>(self, key: i64) -> S where S: SealedMessage;
}

trait UpdateMessage : Message {
    /* Returns the update, which is an application of a State in itself. */
    fn get_update<S>(self) -> (Fn(S::State) -> S::State) where S: Session;
}

trait SealedMessage {
    fn get_seal(self) -> i64;

    /* Contains a way to have the previous message (either stored directly, either via a request
       callback to the client). This allow to reconstruct the sequence of send messages.
       Reconstructing this sequence serves two purposes:
        - Ensuring that the seal of each message is valid, ensuring the validity of the sequence.
        - Since the session state is uniquelly determined by the UpdateMessages, the sequence
        allows to rebuild the state.
       */
    fn get_previous_message<S>(self) -> S where S: SealedMessage;

    fn get_message_content<M>(self) -> M where M: Message;
}

trait RequestMessage : Message {
}

trait SessionClient {
    /* Returns a Client initialized with the initialization vector sent by the SessionHandler.
     */
    fn initiate_session() -> Self;

    /* Sends a message. Returns a State if the message is a RequestMessage. */
    fn send<M, S>(self, M) -> Option<S::State> where M: Message, S: Session;
}

trait FailureDetector {
    /* Creates a failure detector for the session. */
    fn create_fd<S>(s: S) -> Self where S: Session;

    /* True if failure is detected. */
    fn has_failed() -> bool;
}
```