MODULE *HKFM*

EXTENDS *Integers*, *Sequences*
CONSTANTS *Client*, *Song*
VARIABLES *inbox*, *state*

Type definitions (kinda sorta) and other useful stuff...

There are various places where we want to refer to all variables at once, so it's useful to define a *vars* tuple.

$$vars \triangleq \langle inbox, state \rangle$$

The constant *Client* is the set of all clients, represented however we want. It's defined externally, in the model.

We define *Node* to be the set of all nodes in the system, including the server. We don't care how the server is represented, only that it doesn't clash with any of the clients. We use the TLA+ CHOOSE operator to express this.

$$\begin{aligned} Server &\triangleq \text{CHOOSE } x : x \notin Client \\ Node &\triangleq Client \cup \{Server\} \end{aligned}$$

These terms relate to the *playhead*. A *playhead* has two fields:

i: the current track in the *playlist*
t: the number of seconds into that track

When $i = -1$ it means we're not playing anything.

$$\begin{aligned} Idx &\triangleq Nat \cup \{-1\} \\ Playhead &\triangleq [i : Idx,\ t : Nat] \\ Stopped &\triangleq [i \mapsto -1,\ t \mapsto 0] \end{aligned}$$

A *playlist* is a sequence of songs from the constant set *Song*. More formally, *Playlist* is the set of all sequences of songs.

$$Playlist \triangleq Seq(Song)$$

Every node has a *State* containing their current *playlist* and *playhead*.

$$\begin{aligned} State &\triangleq [playlist : Playlist,\ playhead : Playhead] \\ InitState &\triangleq [playlist \mapsto \langle\rangle,\ playhead \mapsto Stopped] \end{aligned}$$

Clients send "add", "seek", and "skip" messages to the server and the server sends "sync" messages to all clients whenever its state changes. The term *Message* is the set of all possible messages that can occur.

$$\begin{aligned} Message \triangleq\ & [action : \{\text{"sync"}\},\ data : State] \cup \\ & [action : \{\text{"add"}\},\ data : Song,\ sender : Client] \cup \\ & [action : \{\text{"seek"}, \text{"skip"}\},\ data : Playhead,\ sender : Client] \end{aligned}$$

The *TypeOK* formula states that *inbox* must be a function from nodes to sequences of messages and *state* must be a function from nodes to states. We can ask *TLC* to check that *TypeOK* is an invariant, meaning it will find circumstances where *inbox* and state end up looking wonky. It's also useful to have as a high level type declaration for these variables.

$$\begin{aligned} \mathit{TypeOK} \triangleq\ & \land\ \mathit{inbox} \in [\mathit{Node} \to \mathit{Seq}(\mathit{Message})] \\ & \land\ \mathit{state}\ \in [\mathit{Node} \to \mathit{State}] \end{aligned}$$

Client Actions

A client sends the "add" message to the server with the name of the desired song in the *data* field.

$$\begin{aligned} &\mathit{SendAdd}(\mathit{self},\ \mathit{song}) \triangleq \\ &\quad \text{LET} \\ &\qquad \mathit{msg} \triangleq [\mathit{action} \mapsto \text{"add"},\ \mathit{data} \mapsto \mathit{song},\ \mathit{sender} \mapsto \mathit{self}] \\ &\quad \text{IN} \\ &\qquad \land\ \mathit{inbox}' = [\mathit{inbox}\ \text{EXCEPT}\ ![\mathit{Server}] = \mathit{Append}(\mathit{inbox}[\mathit{Server}],\ \mathit{msg})] \\ &\qquad \land\ \text{UNCHANGED}\ \mathit{state} \end{aligned}$$

A client sends the "seek" message to the server with the intended new state of the *playhead* in the *data* field.

$$\begin{aligned} &\mathit{SendSeek}(\mathit{self}) \triangleq \\ &\quad \text{LET} \\ &\qquad \mathit{playhead} \triangleq \mathit{state}[\mathit{self}].\mathit{playhead} \\ &\qquad \mathit{newPlayhead} \triangleq [\mathit{playhead}\ \text{EXCEPT}\ !.t = @ + 1] \\ &\qquad \mathit{msg} \triangleq [\mathit{action} \mapsto \text{"seek"},\ \mathit{data} \mapsto \mathit{newPlayhead},\ \mathit{sender} \mapsto \mathit{self}] \\ &\quad \text{IN} \\ &\qquad \land\ \mathit{playhead} \neq \mathit{Stopped} \\ &\qquad \land\ \mathit{inbox}' = [\mathit{inbox}\ \text{EXCEPT}\ ![\mathit{Server}] = \mathit{Append}(\mathit{inbox}[\mathit{Server}],\ \mathit{msg})] \\ &\qquad \land\ \text{UNCHANGED}\ \mathit{state} \end{aligned}$$

A client send the "skip" message to the server with their current *playhead* state in the *data* field.

$$\begin{aligned} &\mathit{SendSkip}(\mathit{self}) \triangleq \\ &\quad \text{LET} \\ &\qquad \mathit{playhead} \triangleq \mathit{state}[\mathit{self}].\mathit{playhead} \\ &\qquad \mathit{msg} \triangleq [\mathit{action} \mapsto \text{"skip"},\ \mathit{data} \mapsto \mathit{playhead},\ \mathit{sender} \mapsto \mathit{self}] \\ &\quad \text{IN} \\ &\qquad \land\ \mathit{playhead} \neq \mathit{Stopped} \\ &\qquad \land\ \mathit{inbox}' = [\mathit{inbox}\ \text{EXCEPT}\ ![\mathit{Server}] = \mathit{Append}(\mathit{inbox}[\mathit{Server}],\ \mathit{msg})] \\ &\qquad \land\ \text{UNCHANGED}\ \mathit{state} \end{aligned}$$

A client receives the "sync" message from the server and updates their own state to match the contents of the *data* field.

$$\mathit{RecvSync}(\mathit{self}) \triangleq$$

```
  ∧ inbox[self] ≠ ⟨⟩
  ∧ LET
      msg  ≜  Head(inbox[self])
      tail ≜  Tail(inbox[self])
    IN
      ∧ msg.action = "sync"
      ∧ inbox' = [inbox EXCEPT ![self] = tail]
      ∧ state' = [state EXCEPT ![self] = msg.data]
```

Server Actions

This is a helper operation. All the server's actions consume the first message in the server's *inbox* and then broadcast the new state to all clients. That is, it places a "sync" message in the *inbox* of all clients.

```
ConsumeMsgAndBroadcastSync ≜
  LET
    msg ≜ [action ↦ "sync", data ↦ state'[Server]]
  IN
    inbox' = [n ∈ Node ↦ IF n = Server
                          THEN Tail(inbox[n])
                          ELSE Append(inbox[n], msg)]
```

The server receives an "add" message, appends the requested track to the *playlist*, then performs *ConsumeMsgAndBroadcastSync*.

If the *playhead* is currently *Stopped*, the server starts playing the newly-added song.

```
RecvAdd ≜
  ∧ inbox[Server] ≠ ⟨⟩
  ∧ LET
      server ≜ state[Server]
      msg    ≜ Head(inbox[Server])
    IN
      ∧ msg.action = "add"
      ∧ LET
          newPlaylist  ≜ Append(server.playlist, msg.data)
          newPlayhead  ≜ IF server.playhead = Stopped
                         THEN [i ↦ Len(server.playlist), t ↦ 0]
                         ELSE server.playhead
          newState     ≜ [playlist ↦ newPlaylist, playhead ↦ newPlayhead]
        IN
          ∧ state' = [state EXCEPT ![Server] = newState]
          ∧ ConsumeMsgAndBroadcastSync
```

The server receives a "seek" message, updates *playhead.t* to the specified value, then performs *ConsumeMsgAndBroadcastSync*.

```
RecvSeek ≜
  ∧ inbox[Server] ≠ ⟨⟩
  ∧ LET
      server ≜ state[Server]
      msg ≜ Head(inbox[Server])
    IN
      ∧ msg.action = "seek"
      ∧ state' = [state EXCEPT ![Server].playhead.t = msg.data.t]
      ∧ ConsumeMsgAndBroadcastSync
```

The server receives a "skip" message, increments *playhead.i* , sets *playhead.t* to 0, then performs *ConsumeMsgAndBroadcastSync*.

If *playhead.i* has gone beyond the bounds of the *playlist*, the server sets the *playhead* to *Stopped*.

```
RecvSkip ≜
  ∧ inbox[Server] ≠ ⟨⟩
  ∧ LET
      server ≜ state[Server]
      msg ≜ Head(inbox[Server])
    IN
      ∧ msg.action = "skip"
      ∧ LET
          newIndex ≜ server.playhead.i + 1
          newPlayhead ≜ IF newIndex < Len(server.playlist)
                        THEN [i ↦ newIndex, t ↦ 0]
                        ELSE Stopped
        IN
          ∧ state' = [state EXCEPT ![Server].playhead = newPlayhead]
          ∧ ConsumeMsgAndBroadcastSync
```

Randomly lose a message from an *inbox*

```
Remove(i, seq) ≜
  [j ∈ 1 .. (Len(seq) − 1) ↦ IF j < i THEN seq[j] ELSE seq[j + 1]]

LoseMsg ≜
  ∃ n ∈ DOMAIN inbox :
    ∃ i ∈ DOMAIN inbox[n] :
      ∧ inbox' = [inbox EXCEPT ![n] = Remove(i, inbox[n])]
      ∧ UNCHANGED state
```

The *Spec*

Init must be true for the first state of all behaviours that satisfy *Spec*

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ inbox = [n \in Node \mapsto \langle\rangle] \\
\quad \land\ state\ = [n \in Node \mapsto InitState]
\end{array}
$$

Next must be true for all steps in all behaviours that satisfy *Spec* . It is the disjunction of the client and server actions defined above.

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor \exists\, self \in Client,\, song \in Song : SendAdd(self,\, song) \\
\quad \lor \exists\, self \in Client : SendSeek(self) \\
\quad \lor \exists\, self \in Client : SendSkip(self) \\
\quad \lor \exists\, self \in Client : RecvSync(self) \\
\quad \lor\ RecvAdd \\
\quad \lor\ RecvSeek \\
\quad \lor\ RecvSkip \\
\quad \lor\ LoseMsg
\end{array}
$$

Last but not least . . . *Spec* itself.

You can read this as:

For every behaviour that satisfies *Spec* , *Init* is true for the first state and for every pair of states (a "step") either *Next* is true or or nothing changes (a "stuttering" step).

$$
\begin{array}{l}
Spec \triangleq \\
\quad Init \land \Box[Next]_{vars}
\end{array}
$$

If *Spec* is true for a behaviour then *TypeOK* is true for every state in that behaviour. THEOREM is (basically) a hint to the reader that we can use *TLC* to check this invariant.

THEOREM $Spec \Rightarrow \Box\, TypeOK$