MODULE *naive*

EXTENDS *Naturals*, *Sequences*

Our "Test Data". Each of these is a set of ids, relevant only in that they are distinct from each other.

CONSTANTS

USERIDS, A set of *userIds* to test with (one per user)
SERVERS, A set of *serverIds* (each one will "create" a new server)
METADATAS, A set of *metadata* versions.
IMAGES A set of image versions.

Our variables update each step and represent the state of our modeled system.

VARIABLES

These variables are relevant to the implementation.

databaseState, *databaseState*[*key*] = What is stored for this key
blobStoreState, *blobStoreState*[*key*] = What is stored for this key
serverStates, *serverStates*[*serverId*] = What the server is doing

This variable is used to observe the state of the system to check if it's doing the right thing. Think of it like the test harness.

Represents all the write requests and read responses sent to/from the system.
operations

Represents every variable in this model.

$vars \triangleq \langle databaseState, blobStoreState, serverStates, operations \rangle$

You strongly type math with math. Here is where we say which types are allowed for all our variables. *TypeOk* is set as an Invariant, which means we expect it always to be true. It will fail if false, effectively giving us type checking. The declarations that follow are part of *TypeOk*, separated to make it clearer.

Allows all the values to also be *UNSET*, which is a distinct value not to be confused for the others.

$UserIdVal \triangleq USERIDS \cup \{\text{"UNSET"}\}$
$MetadataVal \triangleq METADATAS \cup \{\text{"UNSET"}\}$
$ImageVal \triangleq IMAGES \cup \{\text{"UNSET"}\}$

Describes all possible states a server can be in.

$ServerStateVal \triangleq$
[
state : {
current:
"waiting", next: *StartWrite* or *StartRead*
after: *StartWrite*

```
            "started_write",   next: WriteMetadata or FailWrite
             after: WriteMetadata
            "wrote_metadata",   next: WriteBlobAndReturn or FailWrite
             after: StartRead
            "started_read",    next: ReadMetadata
             after: ReadMetadata
            "read_metadata"  next: ReadBlobAndReturn
        },
        userId : UserIdVal,
        metadata :  MetadataVal,
        image : ImageVal
    ]
```

Represents an action that occured on the *API* boundary. Used for observability.

$$\begin{aligned} \mathit{OperationValue} \triangleq\ & [\mathit{type} : \{\text{"READ"}, \text{"WRITE"}\}, \\ & \mathit{userId} : \mathit{UserIdVal}, \\ & \mathit{metadata} : \mathit{MetadataVal}, \\ & \mathit{image} : \mathit{ImageVal}] \end{aligned}$$

The full type specification for all variables in the system

$\mathit{TypeOk} \triangleq$

The database state contains a mapping of *userIds* to metadatas. It can also be "UNSET", representing a case where there is no *metadata*. Note: we make this specific to our problem. If this were a more general problem, it might look like:
databaseState $\in [\mathit{KEYS} \to \mathit{RECORDS}]$.

$\land\ \mathit{databaseState} \in [\mathit{USERIDS} \to \mathit{MetadataVal}]$

The blob store state contains a mapping of *userIds* to images. Note: we make this specific to our problem. If this was a more general problem, it might look like:
blobStoreState $\in [\mathit{KEYS} \to \mathit{BLOBS}]$.

$\land\ \mathit{blobStoreState} \in [\mathit{USERIDS} \to \mathit{ImageVal}]$

The *serverStates* store the current states for each server, allowing us to build a state machine describing our system. Implemented as a mapping between servers and all their possible states.

$\land\ \mathit{serverStates} \in [\mathit{SERVERS} \to \mathit{ServerStateVal}]$
$\land\ \mathit{operations} \in \mathit{Seq}(\mathit{OperationValue})$

When the model starts, everything begins unset. Unlike standard testing every possible state will be explored, so we don't need to initialize for specific scenarios.

$\mathit{Init} \triangleq$

$$\begin{aligned} & \land\ \mathit{databaseState} = [u \in \mathit{USERIDS} \mapsto \text{"UNSET"}] \\ & \land\ \mathit{blobStoreState} = [u \in \mathit{USERIDS} \mapsto \text{"UNSET"}] \\ & \land\ \mathit{serverStates} = [s \in \mathit{SERVERS} \mapsto [\mathit{state} \mapsto \text{"waiting"}, \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathit{userId} \mapsto \text{"UNSET"}, \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathit{metadata} \mapsto \text{"UNSET"}, \end{aligned}$$

$image \mapsto$ "UNSET"
]]

$\land\ operations = \langle\rangle$

State Machine: All of the states are functions of s (server), because the only actively modeled actors in this system are our servers, but there can be multiple working simultainiously.

$StartWrite(s) \triangleq$

Writing only starts when a server is waiting

$\land\ serverStates[s].state =$ "waiting"

This will try every combination of *userId*, *metadata* and image (one at a time). We store this throughout the state lifecycle. Next states will refer to this

$\land \exists\, u \in USERIDS,\ m \in METADATAS,\ i \in IMAGES :$

serverStates' means the next state of *serverStates*

$\land\ serverStates' = [serverStates$ EXCEPT update only server s

$![s].state =$ "started_write", update state

set values for the upcoming write

$![s].userId = u,$

$![s].metadata = m,$

$![s].image = i]$

Record the write for observability

$\land\ operations' = Append(operations,$

This is created with "record" symantics, which is why $\mapsto$ not $=$ is used

[

$type \mapsto$ "WRITE",

$userId \mapsto u,$

$metadata \mapsto m,$

$image \mapsto i$

])

We need to list every unchanged variable. Not changing is a behavior too.

$\land$ UNCHANGED $\langle databaseState,\ blobStoreState\rangle$

$WriteMetadata(s) \triangleq$ Represents a successful database write

Established an alias to make code more compact

LET $currentState \triangleq serverStates[s]$

IN

Metadata writing happens directly after write is started

$\land\ currentState.state =$ "started_write"

Database is transactional/consistent. We can therefore model this happening in one step

$\land\ databaseState' = [databaseState$ EXCEPT

$![currentState.userId] = currentState.metadata]$

$\land\ serverStates' = [serverStates \text{ EXCEPT }$ This is how the state advances
$\qquad ![s].state = \text{“wrote\_metadata”}]$
$\land\ \text{UNCHANGED } \langle blobStoreState,\ operations\rangle$

$WriteBlobAndReturn(s) \triangleq$ Represents a successful blob store write
$\text{LET } currentState \triangleq serverStates[s]$
IN
Metadata writing happens directly after write is started
$\land\ currentState.state = \text{“wrote\_metadata”}$
Blob store has read after write consistency. We can therefore model it happening in one step
$\land\ blobStoreState' = [blobStoreState \text{ EXCEPT}$
$\qquad ![currentState.userId] = currentState.image]$
$\land\ serverStates' = [serverStates \text{ EXCEPT }$ update only server s
Process done once blob is written
$\qquad ![s].state = \text{“waiting”}]$
$\land\ \text{UNCHANGED } \langle databaseState,\ operations\rangle$

$FailWrite(s) \triangleq$
In our model, a server can only fail if it is writing. We don't need to do this, but it cuts down state space we don't care about. We are worried about writes failing in a bad spot causing errors in future reads and writes. We don't model spontaneous read failures.
Will only get to this state if writing
$\land\ serverStates[s].state \in \{\text{“started\_write”},\ \text{“wrote\_metadata”}\}$
$\land\ serverStates' = [serverStates \text{ EXCEPT}$
$\qquad ![s].state = \text{“waiting”},$
$\qquad ![s].userId = \text{“UNSET”},$
$\qquad ![s].metadata = \text{“UNSET”},$
$\qquad ![s].image = \text{“UNSET”}]$
Nothing happens with the database and blob store. Everything this server did stays done, anything left undone stays undone.
$\land\ \text{UNCHANGED } \langle databaseState,\ blobStoreState,\ operations\rangle$

We model reading in detail because reading and writing are occurring at the same time, and may interact with each other in unexpected ways.

$StartRead(s) \triangleq$
Reading only starts when a server is waiting
$\land\ serverStates[s].state = \text{“waiting”}$
$\land\ \exists\, u \in USERIDS :$ When we start reading we pick a user id
$\qquad serverStates' = [serverStates \text{ EXCEPT }$ update only server s
$\qquad\qquad ![s].state = \text{“started\_read”},$
$\qquad\qquad ![s].userId = u]$
Reading doesn't changed stored state
$\land\ \text{UNCHANGED } \langle databaseState,\ blobStoreState\rangle$
$\land\ \text{UNCHANGED } operations$

```
ReadMetadata(s) ≜
    LET currentState ≜ serverStates[s]
    IN
     Once the read has started, the first thing we do is Read Metadata
     ∧ currentState.state = "started_read"
     ∧ serverStates' =
            [serverStates EXCEPT  update only server s
                ![s].state = "read_metadata",
                Assembles the read request from whatever is in the database for that user.
                ![s].metadata = databaseState[currentState.userId]]

     Reading doesn't changed stored state
     ∧ UNCHANGED ⟨databaseState, blobStoreState⟩
     ∧ UNCHANGED operations

ReadBlobAndReturn(s) ≜
    LET currentState ≜ serverStates[s]
    IN
     Blob is read after metadata is read
     ∧ currentState.state = "read_metadata"
     ∧ serverStates' = [serverStates EXCEPT  update only server s
                            ![s].state = "waiting",
                            Assembles the read request from whatever is in the database for
                            that user.
                            ![s].image = blobStoreState[currentState.userId]]

     ∧ operations' = Append(operations,
                            Read returns the state it built up during the read process.
                            [
                                type ↦ "READ",
                                userId ↦ currentState.userId,
                                metadata ↦ currentState.metadata,
                                image ↦ blobStoreState[currentState.userId]
                            ])
     Reading doesn't changed stored state
     ∧ UNCHANGED ⟨databaseState, blobStoreState⟩


The Next section determines what states will be chosen on every step.
Next ≜
     For every step, pick a server and have it advance one state
     ∃ s ∈ SERVERS :
        ∨ StartWrite(s)
        ∨ WriteMetadata(s)
        ∨ WriteBlobAndReturn(s)
```

$\lor$ *FailWrite*(s)
$\lor$ *StartRead*(s)
$\lor$ *ReadMetadata*(s)
$\lor$ *ReadBlobAndReturn*(s)

The *Spec* describes what the describe system DOES. First it starts in the *Init* state. Then for every step use *Next* state: represented as $\Box$*Next*. In temporal logic $\Box$ means "for all states." However let's imagine this is part of a larger system; sometimes this system will do nothing. That is represented by $[Next]\_vars$, meaning: *Next*$\lor$UNCHANGED *vars*. See learning material for a better explaination of temporal logic operators. Note: The spec in this case describes what the system DOES, not what it should do. Basically this is our system under test, and we describe Invariants (below) and properties (discussed later) to alert us if the system does something wrong/unexpected

$Spec \triangleq Init \land \Box[Next]_{vars}$

Invariants: These are things that should always be true about the system. If they become false during any step, an error will occur with a trace that shows you the series of steps that let it to be violated. This is very powerful. The first invariant we saw was *TypeOk*: the types are expected to always conform to the expected type system, and if not we want to know why.

ConsistentReads $\triangleq$
If there are no operations, they are consistent
$\lor$ $operations = \langle\rangle$
$\lor \forall i \in 1 \,..\, Len(operations)$: For every read operation
LET $readOp \triangleq operations[i]$IN
$\lor$ $\land$ $readOp.type =$ "READ"
There must exist a write operation
$\land$ $\lor \exists j \in 1 \,..\, i$:
LET $writeOp \triangleq operations[j]$IN
$\land$ $writeOp.type =$ "WRITE"
With the same data
$\land$ $readOp.userId = writeOp.userId$
$\land$ $readOp.metadata = writeOp.metadata$
$\land$ $readOp.image = writeOp.image$
$\lor$ Ignore unset reads
$\land$ $readOp.metadata =$ "UNSET"
$\land$ $readOp.image =$ "UNSET"
$\lor$ $readOp.type =$ "WRITE" Ignore writes

One of the best things about invariants is that if they were ever going to be tripped, you'll hear about it. Unlike testing, where sometimes a confluence of events leads to a test passing when it shouldn't, the model checker will try every possible state, so if it ever messes up, you'll know.

This is used for model checker configuration so the simulation doesn't go on forever.

*StopAfter*3*Operations* $\triangleq$
$Len(operations) \leq 3$