MODULE *objectstore*

EXTENDS *FiniteSets*, *Sequences*, *Naturals*, *TLC*

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in *RFC* 2119.

This specification defines

* A model of a consistent object store: a consistent store of data and *metadata*, one without any notion of a "directory hierarchy". It is intended to model object stores such as *Amazon S*3, and includes its multipart *PUT API*.

* An *API* for communicating with object stores from *Hadoop* filesystems.

It is intended to be a foundation for defining algorithms with worth with *S*3, such as the *s*3*guard* commit algorithm.

CONSTANTS

Paths,	the non-finite set of all possible valid paths
PathsAndRoot,	Paths and the "root" path; the latter is read-only
Data,	the non-finite set of all possible sequences of bytes
MetadataKeys,	the set of all possible *metadata* keys
MetadataValues,	the non-finite set of all possible *metadata* values
Timestamp,	A timestamp
Byte,	
Etag,	
MultipartPutId,	

PartId,
NonEmptyString

ASSUME $NonEmptyString \in (\text{STRING} \setminus \text{“”})$

ASSUME $PathsAndRoot \in \text{STRING}$
ASSUME $Paths \in (PathsAndRoot \setminus \text{“”})$

There are some *metadata* keys which are system *metadata* entries. Those MAY be queried but SHALL NOT be explictly set. (more specifically, they'll be ignored if you try.
ASSUME $MetadataKeys \in NonEmptyString$

ASSUME $MetadataValues \in \text{STRING}$

Timestamps are positive integers since the epoch.
ASSUME $Timestamp \in Nat \land Timestamp > 0$

Byte type
ASSUME $Byte \in 0 \,..\, 255$

Data is a sequence of bytes
ASSUME $Data \in Seq(Byte)$

ASSUME $Etag \in NonEmptyString$

ASSUME $MultipartPutId \in NonEmptyString$

Only 11,000 parts are allowed
ASSUME $PartId \in 1 \,..\, 11000$

There is a predicate to validate a pathname. This is considered implementation-specific.

It could be describable as a regular expression specific to each implementation, though constraints such as “no two adjacent '/' characters” might make for a complex regexp. Perhaps each *FS* would have a set of regexps which all must be valid for a path to be considered valid.
CONSTANT $is\_valid\_pathname(\_)$
CONSTANT $is\_valid\_metadata\_key(\_)$

All paths can be evaluated to see if their pathname is valid

ASSUME $\forall\, p \in Paths : is\_valid\_pathname(p) \in \text{BOOLEAN}$

All *metadata* keys can be evaluated for validity

ASSUME $\forall\, e \in MetadataKeys : is\_valid\_metadata\_key(e) \in \text{BOOLEAN}$

Substring match predicate
CONSTANT $starts\_with(\_, \_)$
ASSUME $\forall\, p,\, p2 \in \text{STRING} : starts\_with(p,\, p2) \in \text{BOOLEAN}$

The patch matching algorithm used in the list operation

CONSTANT $path\_matches(\_, \_, \_)$

This should really be defined by looking inside the strings.
It is: all paths starting with the prefix up to those ending in the suffix

ASSUME $\forall\, p \in Paths,\ prefix,\ delimiter \in \text{STRING} : path\_matches(p,\ prefix,\ delimiter) \in \text{BOOLEAN}$

CONSTANT $path\_matches\_prefix(\_, \_)$
ASSUME $\forall\, p \in Paths,\ prefix \in \text{STRING} : path\_matches\_prefix(p,\ prefix) \in \text{BOOLEAN}$

A function to return an *etag* of some data
CONSTANT $etag\_of(\_)$

A function to return an *etag* of a multipart operation; implementation specific
CONSTANT $etag\_of\_multipart\_operation(\_)$

Etags are strings, hence in $MetadataValues$.
ASSUME $\forall\, d \in Data : etag\_of(d) \in Etag$

This is commented out as it is not a requirement that etags are the same for an equivalent sequence of bytes. All that matters is that one is generated. ASSUME $\forall\, d,\ e \in Data$: $d = e \Rightarrow etag\_of(d) = etag\_of(e) \in \text{STRING}$

VARIABLE *store* The object store
VARIABLE *pending* Pending requests

Exception logic

$BadRequest \triangleq$ "BadRequest"
$NotFound \triangleq$ "NotFound"
$Success \triangleq$ "Success"

$MetadataEntry \triangleq [$
  $key : MetadataKeys,$ The key of the entry
  $value : MetadataValues$ the value of this *metadata* entry
$]$

$SystemMetadata \triangleq [$
  $size : Nat,$
  $created : Timestamp$

]

A store : $path \rightarrow$ (data, user-md, system-md)

update: PUT, $DELETE$ query: GET, $HEAD$, $LIST(path)$

$$
\begin{array}{l}
StoreEntry \triangleq [\\
\quad data : Data, \quad \text{the data in the entry} \\
\quad created : Timestamp, \quad \text{timestamp} \\
\quad etag : MetadataValues \\
\quad]
\end{array}
$$

$$
\begin{array}{l}
ListingEntry \triangleq [\\
\quad path : Paths, \quad \text{The path to the entry} \\
\quad data : Data, \quad \text{the data in the entry} \\
\quad created : Timestamp, \quad \text{timestamp} \\
\quad etag : MetadataValues, \\
\quad metadata : MetadataEntry \quad \text{it's a set} \\
\quad]
\end{array}
$$

The check for a path having an entry is pulled out for declaring invariants

$$has\_entry(s,\, p) \triangleq p \in \text{DOMAIN } s$$

$$
\begin{array}{l}
PendingMultipartPartRequest \triangleq [\\
\quad putId : MultipartPutId, \\
\quad part : PartId, \\
\quad data : Data \\
]
\end{array}
$$

$$
\begin{array}{l}
PendingMultipartPartResponse \triangleq [\\
\quad etag : Etag \\
]
\end{array}
$$

$$
\begin{array}{l}
PendingMultipartPutPart \triangleq [\\
\quad data : Data, \\
\quad etag : Etag \\
]
\end{array}
$$

A pending *Multipart* Upload has an *ID* and start timne, which is used to define the final create time of the committed operation

$$
\begin{array}{l}
PendingMultipartOperation \triangleq [\\
\quad id\text{: STRING}, \\
\quad path : Paths, \\
\quad started : Timestamp, \\
\quad parts : [PartId \rightarrow PendingMultipartPutPart] \\
]
\end{array}
$$

The store state invariant not only declares the type of the store, it declares attributes of the *has_entry* operator which are superfluous given the definition of *has_entry*() as the path being in the domain of the store. It's explicit for those implementors planning to write tests.

$$
\begin{array}{l}
StoreStateInvariant \triangleq \\
\quad \land\ store \in [Paths \to StoreEntry] \\
\quad \land\ pending \in [MultipartPutId \to PendingMultipartOperation]
\end{array}
$$

The initial state of the store is that it is empty.

Notice how this ignores the root entry, "".
This is a special entry: object stores are not filesystems: there is no root node equivalent to "/"

$$
\begin{array}{l}
InitialStoreState \triangleq \\
\quad \land\ StoreStateInvariant \\
\quad \land\ \text{DOMAIN}\ store = \{\} \\
\quad \land\ \text{DOMAIN}\ pending = \{\}
\end{array}
$$

Actions. Note how some post conditions are explicitly called out. They are superfluous, in the model, but they do declare final state for testability

PUT: update the store with the newly uploaded data. This definition is consistent: the store changes are immediately visible, even if there was an existing entry.

$$
\begin{array}{l}
doPut(path,\ data,\ current\_time,\ result) \triangleq \\
\quad \text{LET}\ validArgs \triangleq path \in Paths \land data \in Data \land current\_time \in Timestamp \\
\quad \text{IN} \\
\qquad \lor\ \land \neg validArgs \\
\qquad\quad \land\ result' = BadRequest \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle store,\ pending \rangle \\
\qquad \lor\ \land\ validArgs \\
\qquad\quad \land\ result' = Success \\
\qquad\quad \land\ \text{UNCHANGED}\ pending \\
\qquad\quad \land\ store' = [store\ \text{EXCEPT}\ ![path] = [data \mapsto data,\ created \mapsto current\_time,\ etag \mapsto etag\_of(data)]]
\end{array}
$$

GET: *path* → *data* as well as summary *metadata*

$$
\begin{array}{l}
doGet(path,\ result,\ metadata,\ data) \triangleq \\
\quad \text{LET} \\
\qquad validArgs \triangleq path \in PathsAndRoot \\
\qquad exists \triangleq has\_entry(store,\ path) \\
\qquad entry \triangleq store[path] \\
\quad \text{IN} \\
\qquad \lor\ \land \neg validArgs \\
\qquad\quad \land\ result' = BadRequest \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle store,\ pending \rangle
\end{array}
$$

$\lor \land validArgs$
$\land path = \text{“”}$
$\land result' = Success$
$\land$ UNCHANGED $\langle store, pending \rangle$
$\land data' = \{\}$

$\lor \land validArgs$
$\land \neg exists$
$\land result' = NotFound$
$\land$ UNCHANGED $\langle store, pending \rangle$

$\lor \land validArgs$
$\land exists$
$\land result' = Success$
$\land data' = store[path].data$
$\land metadata' = [created \mapsto entry.created, length \mapsto Len(entry.data), etag \mapsto entry.etag]$
$\land$ UNCHANGED $\langle store, pending \rangle$

HEAD: the *metadata* without the data

$doHead(path, result, metadata) \triangleq$
LET
$validArgs \triangleq path \in PathsAndRoot$
$exists \triangleq has\_entry(store, path)$
$entry \triangleq store[path]$
IN
$\lor \land \neg validArgs$
$\land result' = BadRequest$
$\land$ UNCHANGED $\langle store, pending \rangle$

$\lor \land validArgs$
$\land path = \text{“”}$
$\land result' = Success$
$\land metadata' = [created \mapsto 0, length \mapsto 0]$
$\land$ UNCHANGED $\langle store, pending \rangle$

$\lor \land validArgs$
$\land \neg exists$
$\land result' = NotFound$
$\land$ UNCHANGED $\langle store, pending \rangle$

$\lor \land validArgs$
$\land exists$
$\land result' = Success$
$\land metadata' = [created \mapsto entry.created, length \mapsto Len(entry.data), etag \mapsto entry.etag]$
$\land$ UNCHANGED $\langle store, pending \rangle$

$doDelete(path,\ result) \triangleq$
LET
$validArgs \triangleq path \in Paths$
$exists \triangleq has\_entry(store,\ path)$
IN
$\lor \land \neg validArgs$
$\land result' = BadRequest$
$\land$ UNCHANGED $\langle store,\ pending \rangle$

$\lor \land validArgs$
$\land \neg exists$
$\land result' = NotFound$
$\land$ UNCHANGED $\langle store,\ pending \rangle$

$\lor \land validArgs$
$\land exists$
$\land result' = Success$
$\land store' = [p \in (\text{DOMAIN } store \setminus path) \mapsto store[p]]$
$\land$ UNCHANGED $pending$

$doCopy(source,\ dest,\ current\_time,\ result) \triangleq$
LET
$validArgs \triangleq source \in Paths \land dest \in Paths \land current\_time \in Timestamp$
$exists \triangleq has\_entry(store,\ source)$
IN
$\lor \land \neg validArgs$
$\land result' = BadRequest$
$\land$ UNCHANGED $\langle store,\ pending \rangle$

$\lor \land validArgs$
$\land \neg exists$
$\land result' = NotFound$
$\land$ UNCHANGED $\langle store,\ pending \rangle$

$\lor \land validArgs$
$\land exists$
$\land result' = Success$
$\land store' = [store \text{ EXCEPT } ![dest] = [data \mapsto store[source].data,\ created \mapsto current\_time]]$
$\land$ UNCHANGED $pending$

The list operation returns the *metadata* of all entries in the object store whose path matches the prefix/suffix pattern.
*S*3 also returns a string sequence of common subpath underneath, essential "what look like directories"
$pathsMatchingPrefix(prefix,\ suffix) \triangleq \forall\, path \in \text{DOMAIN } store : path\_matches(path,\ prefix,\ suffix)$

$doList(prefix,\ suffix,\ result,\ listing) \triangleq$

```
LET
  validArgs ≜ prefix ∈ STRING ∧ suffix ∈ STRING
IN
  ∨ ∧ ¬validArgs
    ∧ result′ = BadRequest
    ∧ UNCHANGED ⟨store, pending⟩

  ∨ ∧ validArgs
    ∧ result′ = Success
    ∧ listing′ = [path ∈ pathsMatchingPrefix(prefix, suffix) ↦
        [path ↦ path, created ↦ store[path].created, length ↦ Len(store[path].data), etag ↦ store[path].et
    ∧ UNCHANGED ⟨store, pending⟩
```

Initiate a multipart *PUT*. The destination is specified; the create time of the final artifact is set to the current server time. A unique *ID* is returned. There is no requirement for the destination to be unique: multiple requests may target the same destination, with the order of the commit operation defining the order in which the results become visible.

```
doInitiateMultipartPut(dest, current_time, result, operationId) ≜
  LET
    validArgs ≜ dest ∈ Paths ∧ current_time ∈ Timestamp
    newPartId ≜ CHOOSE id ∈ MultipartPutId : ¬id ∈ DOMAIN pending
  IN
    ∨ ∧ ¬validArgs
      ∧ result′ = BadRequest
      ∧ UNCHANGED ⟨store, pending⟩

    ∨ ∧ validArgs
      ∧ result′ = Success
      ∧ UNCHANGED store
      ∧ operationId′ = newPartId
      ∧ pending′ = [pending EXCEPT ![newPartId] = [path ↦ dest, created ↦ current_time]]
```

PUT a single part for an operation

```
doPutPart(operationId, partId, part_data, result, etagResult) ≜
  LET
    validArgs ≜ operationId ∈ DOMAIN pending ∧ part_data ∈ Data ∧ partId ∈ PartId
    etagVal ≜ etag_of(part_data)
  IN
    ∨ ∧ ¬validArgs
      ∧ result′ = BadRequest
      ∧ UNCHANGED ⟨store, pending⟩

    ∨ ∧ validArgs
      ∧ result′ = etagVal
      ∧ etagResult′ = etagVal
      ∧ UNCHANGED store
```

$\land\ pending' = [pending \text{ EXCEPT}$
$\quad ![operationId] = [$
$\quad\ path \mapsto pending[operationId].dest,$
$\quad\ parts \mapsto [pending[operationId].parts \text{ EXCEPT } ![partId] = [data \mapsto part\_data,\ etag \mapsto etagVal]]$
$\quad\ ]$
$\ ]$

The commit operation is the most complex. The part list supplied defines the order in which the supplied parts are saved to the store. *TODO*: work out how to declare that all data is the ordered appending of the data of the list of parts. Recurse?

$doCommitMultipartPut(operationId,\ parts,\ result) \triangleq$
LET
$\quad upload \triangleq pending[operationId]$
$\quad validArgs \triangleq (operationId \in \text{DOMAIN } pending) \land (parts \in Seq(PartId))$
$\qquad \land (\forall\, p \in parts : p \in \text{DOMAIN } upload.parts) \land (\forall\, p \in \text{DOMAIN } upload.parts : p \in parts)$
$\quad alldata \triangleq \forall\ [part \in\ (1 \ldots Len(parts)\text{ -1}])\ Append(upload[parts[part]],\ upload[parts[part+1]])$
$\quad alldata \triangleq parts$
$\quad etag \triangleq etag\_of\_multipart\_operation(upload)$
IN
$\quad \lor\ \land \neg validArgs$
$\qquad \land result' = BadRequest$
$\qquad \land \text{UNCHANGED } \langle store,\ pending\rangle$

$\quad \lor\ \land validArgs$
$\qquad \land result' = Success$
$\qquad \land pending' = [p \in (\text{DOMAIN } pending \setminus operationId) \mapsto pending[p]]$
$\qquad \land store' = [store \text{ EXCEPT } ![upload.path] = [data \mapsto alldata,\ created \mapsto upload.created,\ etag \mapsto etag]]$

Abort the multipart put operation. All pending data is deleted; the pending operation record removed.

$doAbortMultipartPut(operationId,\ result) \triangleq$
LET
$\quad validArgs \triangleq operationId \in \text{DOMAIN } pending$
IN
$\quad \lor\ \land \neg validArgs$
$\qquad \land result' = BadRequest$
$\qquad \land \text{UNCHANGED } \langle store,\ pending\rangle$

$\quad \lor\ \land validArgs$
$\qquad \land result' = Success$
$\qquad \land \text{UNCHANGED } store$
$\qquad \land pending' = [p \in (\text{DOMAIN } pending \setminus operationId) \mapsto pending[p]]$

$doListMultipartPuts(prefix,\ suffix,\ result,\ listing) \triangleq$
LET
$\quad validArgs \triangleq prefix \in \text{STRING}$

IN

$\lor \land \neg validArgs$
$\land result' = BadRequest$
$\land$ UNCHANGED $\langle store, pending \rangle$
$\lor \land validArgs$
$\land result' = Success$
$\land listing' =$
$[path \in path\_matches\_prefix(prefix, suffix) \mapsto$
$[path \mapsto path, created \mapsto store[path].created, length \mapsto Len(store[path].data), etag \mapsto store[path].eta$
$\land$ UNCHANGED $\langle store, pending \rangle$

$PutInvariant \triangleq \forall p$ in $Paths$: $doDelete(p, Success) \Rightarrow \neg has\_entry(store', p)$

$DeleteInvariant \triangleq \forall p$ in $Paths$: $doDelete(p, Success) \Rightarrow \neg has\_entry(store', p)$

The amount of data you get back is the amount of data you are told comes back.

$GetLengthInvariant \triangleq$
$\forall path \in$ DOMAIN $store, sysMd \in SystemMetadata, data \in Data :$
$doGet(path, Success, data, sysMd) \triangleq > Len(data) = sysMd.length$

The *metadata* that comes from a *doHead*() MUST match that from a *doGet*()
See: *HADOOP*-11202 *SequenceFile* crashes with encrypted files that are shorter than *FileSystem.getStatus*(*path*)

$GetAndHeadInvariant \triangleq$
$\forall path \in$ DOMAIN $store, sysMd \in SystemMetadata, data \in Data :$
$doGet(path, Success, data, sysMd) \triangleq > doHead(path, Success, sysMd)$

The details you get back in a listing match the details you get back from a *doGet* call on the specific path
of course, on an eventually consistent object store, there may be lag

$ListAndGetInvariant \triangleq TODO$

now define action messages which can be queued for processing; we consider them to processed in a serial order

---

Action Records

$putAction \triangleq [$
$verb$: "PUT",
$path$: STRING,
$data : [Nat \to Nat]$
$]$

$deleteAction \triangleq [$

$verb$: "DELETE",
$path$: STRING
]

$getAction \triangleq$ [
$verb$: "GET",
$path$: STRING,
$data$: STRING
]

$headAction \triangleq$ [
$verb$: "HEAD",
$path$: STRING
]

$copyAction \triangleq$ [
$verb$: "COPY",
$source$: STRING,
$dest$: STRING
]

$listAction \triangleq$ [
$verb$: "LIST",
$prefix$: STRING,
$delimiter$: STRING
]

Process a request, generate a result.

$process(request, result, metadata, body, current\_time) \triangleq$
LET $verb \triangleq request.verb$
IN
$\lor verb =$ "PUT" $\land doPut(request.path, request.data, current\_time, result)$
$\lor verb =$ "GET" $\land doGet(request.path, result, metadata, body)$
$\lor verb =$ "HEAD" $\land doHead(request.path, result, metadata)$
$\lor verb =$ "DELETE" $\land doDelete(request.path, result)$
$\lor verb =$ "COPY" $\land doCopy(request.source, request.dest, current\_time, result)$
$\lor verb =$ "LIST" $\land doList(request.prefix, request.suffix, result, body)$

THEOREM $InitialStoreState \Rightarrow \Box StoreStateInvariant$

\ * Modification History
\ * Last modified *Wed Apr* 26 16:33:00 *BST* 2017 by *stevel*
\ * Created Sun *Jun* 19 18:07:44 *BST* 2016 by *stevel*