MODULE *yarnregistry*

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

This defines the YARN registry in terms of operations on sets of records.

Every registry entry is represented as a record containing both the path and the data.

It assumes that

1. operations on this set are immediate.
2. selection operations (such as ∀ and ∃ are atomic)
3. changes are immediately visible to all other users of the registry.
4. This clearly implies that changes are visible in the sequence in which they happen.

A multi-server Zookeeper-based registry may not meet all those assumptions

1. changes may take time to propagate across the *ZK* quorum, hence changes cannot be considered immediate from the perspective of other registry clients. (assumptions (1) and (3)).

2. Selection operations may not be atomic. (assumption (2)).

Operations will still happen in the order received by the elected *ZK* master

A stricter definition would try to state that all operations are eventually true excluding other changes happening during a sequence of action. This is left as an excercise for the reader.

The specification also omits all coverage of the permissions policy.

CONSTANTS

PathChars,	the set of valid characters in a path
Paths,	the set of all possible valid paths
Data,	the set of all possible sequences of bytes
Address,	the set of all possible address n-tuples
Addresses,	the set of all possible address instances
Endpoints,	the set of all possible endpoints

PersistPolicies, the set of persistence policies
ServiceRecords, all service records
Registries, the set of all possile registries
BindActions, all possible put actions
DeleteActions, all possible delete actions
PurgeActions, all possible purge actions
MknodeActions all possible mkdir actions

the registry
VARIABLE *registry*

Sequence of actions to apply to the registry
VARIABLE *actions*

Tuple of all variables.

$vars \triangleq \langle registry, actions \rangle$

Persistence policy
$PersistPolicySet \triangleq \{$
"", Undefined; field not present. PERMANENT is implied.
"permanent", persists until explicitly removed
"application", persists until the application finishes
"application-attempt", persists until the application attempt finishes
"container" persists until the container finishes
$\}$

Type invariants.
$TypeInvariant \triangleq$
$\wedge \forall p \in PersistPolicies : p \in PersistPolicySet$

An Entry is defined as a path, and the actual data which it contains.

By including the path in an entry, we avoid having to define some function mapping Path $\rightarrow$ *entry*. Instead a registry can be defined as a set of *RegistryEntries* matching the validity critera.
$RegistryEntry \triangleq [$

The path to the entry
$path : Paths,$

the data in the entry
$data : Data$
]

An endpoint in a service record

$Endpoint \triangleq [$
API of the endpoint: some identifier
api : STRING,

A list of address n-tuples
$addresses : Addresses$
]

Attributes are the set of all string to string mappings

$Attributes \triangleq [$
STRING $\mapsto$ STRING
]

A service record

$ServiceRecord \triangleq [$
ID – used when applying the persistence policy
$yarn\_id$: STRING,

the persistence policy
$yarn\_persistence : PersistPolicySet,$

A description
$description$: STRING,

A set of endpoints
$external : Endpoints,$

Endpoints intended for use internally
$internal : Endpoints,$

Attributes are a function
$attributes : Attributes$
]

Action Records

$$putAction \triangleq [$$
$$\quad type : \text{“put”},$$
$$\quad record : ServiceRecord$$
$$]$$

$$deleteAction \triangleq [$$
$$\quad type : \text{“delete”},$$
$$\quad path : \text{STRING},$$
$$\quad recursive : \text{BOOLEAN}$$
$$]$$

$$purgeAction \triangleq [$$
$$\quad type : \text{“purge”},$$
$$\quad path : \text{STRING},$$
$$\quad persistence : PersistPolicySet$$
$$]$$

$$mkNodeAction \triangleq [$$
$$\quad type : \text{“mknode”},$$
$$\quad path : \text{STRING},$$
$$\quad parents : \text{BOOLEAN}$$
$$]$$

Path operations

Parent is defined for non empty sequences

$$parent(path) \triangleq SubSeq(path, 1, Len(path) - 1)$$

$$isParent(path, c) \triangleq path = parent(c)$$

Registry Access Operations

Lookup all entries in a registry with a matching path

$$resolve(Registry, path) \triangleq \forall\, entry \in Registry : entry.path = path$$

A path exists in the registry iff there is an entry with that path

$$exists(Registry, path) \triangleq resolve(Registry, path) \neq \{\}$$

A parent entry, or an empty set if there is none

$$parentEntry(Registry, path) \triangleq resolve(Registry, parent(path))$$

A root path is the empty sequence

$isRootPath(path) \triangleq path = \langle\rangle$

The root entry is the entry whose path is the root path

$isRootEntry(entry) \triangleq entry.path = \langle\rangle$

A path p is an ancestor of another path d if they are different, and the path d starts with path p

$$isAncestorOf(path,\ d) \triangleq \\ \quad \land path \neq d \\ \quad \land \exists\, k : SubSeq(d,\ 0,\ k) = path$$

$$ancestorPathOf(path) \triangleq \\ \quad \forall\, a \in Paths : isAncestorOf(a,\ path)$$

The set of all children of a path in the registry

$children(R,\ path) \triangleq \forall\, c \in R : isParent(path,\ c.path)$

A path has children if the *children*() function does not return the empty set

$hasChildren(R,\ path) \triangleq children(R,\ path) \neq \{\}$

Descendant: a child of a path or a descendant of a child of a path

$descendants(R,\ path) \triangleq \forall\, e \in R : isAncestorOf(path,\ e.path)$

Ancestors: all entries in the registry whose path is an entry of the path argument

$ancestors(R,\ path) \triangleq \forall\, e \in R : isAncestorOf(e.path,\ path)$

The set of entries that are a path and its descendants

$$pathAndDescendants(R,\ path) \triangleq \\ \quad \lor \forall\, e \in R : isAncestorOf(path,\ e.path) \\ \quad \lor resolve(R,\ path)$$

For validity, all entries must match the following criteria

$validRegistry(R) \triangleq$

there can be at most one entry for a path.

$\land \forall\, e \in R : Cardinality(resolve(R,\ e.path)) = 1$

There's at least one root entry

$\land \exists\, e \in R : isRootEntry(e)$

an entry must be the root entry or have a parent entry

$\land \forall\, e \in R : isRootEntry(e) \lor exists(R,\ parent(e.path))$

If the entry has data, it must be a service record

$\land \forall\, e \in R : (e.data = \langle\rangle \lor e.data \in ServiceRecords)$

Registry Manipulation

An entry can be put into the registry iff its parent is present or it is the root entry

$$canBind(R,\ e) \triangleq$$
$$\quad isRootEntry(e) \lor exists(R,\ parent(e.path))$$

'*bind*() adds/replaces an entry if permitted

$$bind(R,\ e) \triangleq$$
$$\quad \land\ canBind(R,\ e)$$
$$\quad \land\ R' = (R \setminus resolve(R,\ e.path)) \cup \{e\}$$

mknode() adds a new empty entry where there was none before, *iff*
– the parent *exists*
– it meets the requirement for being "bindable"

$$mknodeSimple(R,\ path) \triangleq$$
$$\quad \text{LET}\ record \triangleq [path \mapsto path,\ data \mapsto \langle\rangle\ ]$$
$$\quad \text{IN} \quad \lor exists(R,\ path)$$
$$\qquad \lor (exists(R,\ parent(path)) \ \land canBind(R,\ record) \land (R' = R \cup \{record\}))$$

For all parents, the *mknodeSimpl*() criteria must apply. This could be defined recursively, though as TLA+ does not support recursion, an alternative is required

Because this specification is declaring the final state of a operation, not the implemental, all that is needed is to describe those parents.

It declares that the *mkdirSimple* state applies to the path and all its parents in the set R'

$$mknodeWithParents(R,\ path) \triangleq$$
$$\quad \land \forall\, p2 \in ancestors(R,\ path) : mknodeSimple(R,\ p2)$$
$$\quad \land\ mknodeSimple(R,\ path)$$

$$mknode(R,\ path,\ recursive) \triangleq$$
$$\quad \text{IF}\ recursive\ \text{THEN}\ mknodeWithParents(R,\ path)\ \text{ELSE}\ \ mknodeSimple(R,\ path)$$

Deletion is set difference on any existing entries

$$simpleDelete(R,\ path) \triangleq$$
$$\quad \land \neg isRootPath(path)$$
$$\quad \land\ children(R,\ path) = \{\}$$
$$\quad \land\ R' = R \setminus resolve(R,\ path)$$

Recursive delete: neither the path or its descendants exists in the new registry

$$recursiveDelete(R,\ path) \triangleq$$

Root path: the new registry is the initial registry again

$$\quad \land\ isRootPath(path) \Rightarrow R' = \{[path \mapsto \langle\rangle,\ data \mapsto \langle\rangle]\}$$

Any other entry: the new registry is a set with any existing
entry for that path is removed, and the new entry added

$$\wedge\ \neg isRootPath(path) \Rightarrow R' = R \setminus (resolve(R,\ path) \cup descendants(R,\ path))$$

Delete operation which chooses the recursiveness policy based on an argument

$$delete(R,\ path,\ recursive) \triangleq$$
$$\text{IF } recursive \text{ THEN } recursiveDelete(R,\ path) \text{ ELSE } simpleDelete(R,\ path)$$

Purge ensures that all entries under a path with the matching *ID* and policy are not there afterwards

$$purge(R,\ path,\ id,\ persistence) \triangleq$$
$$\wedge\ (persistence \in PersistPolicySet)$$
$$\wedge\ \forall\, p2 \in pathAndDescendants(R,\ path) :$$
$$(p2.attributes[\text{“yarn:id”}] = id \wedge p2.attributes[\text{“yarn:persistence”}] = persistence)$$
$$\Rightarrow recursiveDelete(R,\ p2.path)$$

resolveRecord() resolves the record at a path or fails.

It relies on the fact that if the cardinality of a set is 1, then the CHOOSE operator is guaranteed to return the single entry of that set, iff the choice predicate holds.

Using a predicate of TRUE, it always succeeds, so this function selects the sole entry of the resolve operation.

$$resolveRecord(R,\ path) \triangleq$$
$$\text{LET } l \triangleq resolve(R,\ path) \text{ IN}$$
$$\wedge\ Cardinality(l) = 1$$
$$\wedge\ \text{CHOOSE } e \in l : \text{TRUE}$$

The specific action of putting an entry into a record includes validating the record

$$validRecordToBind(path,\ record) \triangleq$$

The root entry must have permanent persistence

$$isRootPath(path) \Rightarrow (record.attributes[\text{“yarn:persistence”}] = \text{“permanent”}$$
$$\vee\ record.attributes[\text{“yarn:persistence”}] = \text{“”})$$

Binding a service record involves validating it then putting it in the registry marshalled as the data in the entry

$$bindRecord(R,\ path,\ record) \triangleq$$
$$\wedge\ validRecordToBind(path,\ record)$$
$$\wedge\ bind(R,\ [path \mapsto path,\ data \mapsto record])$$

The action queue can only contain one of the sets of action types, and by giving each a unique name, those sets are guaranteed to be disjoint

$$
\begin{array}{l}
QueueInvariant \triangleq \\
\quad \land \forall\, a \in actions : \\
\qquad \lor (a \in BindActions \land a.type = \text{“bind”}) \\
\qquad \lor (a \in DeleteActions \land a.type = \text{“delete”}) \\
\qquad \lor (a \in PurgeActions \land a.type = \text{“purge”}) \\
\qquad \lor (a \in MknodeActions \land a.type = \text{“mknode”})
\end{array}
$$

Applying queued actions

$$
\begin{array}{l}
applyAction(R,\, a) \triangleq \\
\quad \lor (a \in BindActions \land bindRecord(R,\, a.path,\, a.record)) \\
\quad \lor (a \in MknodeActions \land mknode(R,\, a.path,\, a.recursive)) \\
\quad \lor (a \in DeleteActions \quad \land delete(R,\, a.path,\, a.recursive)) \\
\quad \lor (a \in PurgeActions \land purge(R,\, a.path,\, a.id,\, a.persistence))
\end{array}
$$

Apply the first action in a list and then update the actions

$$
\begin{array}{l}
applyFirstAction(R,\, a) \triangleq \\
\quad \land actions \neq \langle\rangle \\
\quad \land applyAction(R,\, Head(a)) \\
\quad \land actions' = Tail(a)
\end{array}
$$

$$
Next \triangleq applyFirstAction(registry,\, actions)
$$

All submitted actions must eventually be applied.

$$
Liveness \triangleq \Diamond(actions = \langle\rangle)
$$

The initial state of a registry has the root entry.

$$
\begin{array}{l}
InitialRegistry \triangleq registry = \{ \\
\quad [path \mapsto \langle\rangle,\, data \mapsto \langle\rangle] \\
\}
\end{array}
$$

The valid state of the “registry” variable is defined as Via the *validRegistry* predicate

$$
ValidRegistryState \triangleq validRegistry(registry)
$$

The initial state of the system

$$
\begin{array}{l}
InitialState \triangleq \\
\quad \land InitialRegistry \\
\quad \land ValidRegistryState \\
\quad \land actions = \langle\rangle
\end{array}
$$

The registry has an initial state, the series of state changes driven by the actions, and the requirement that it does act on those actions.

$$
\begin{aligned}
&\mathit{RegistrySpec} \triangleq \\
&\quad \land\ \mathit{InitialState} \\
&\quad \land\ \Box[\mathit{Next}]_{\mathit{vars}} \\
&\quad \land\ \mathit{Liveness}
\end{aligned}
$$

Theorem: For all operations from that initial state, the registry state is still valid

THEOREM $\mathit{InitialState} \Rightarrow \Box\, \mathit{ValidRegistryState}$

Theorem: for all operations from that initial state, the type invariants hold

THEOREM $\mathit{InitialState} \Rightarrow \Box\, \mathit{TypeInvariant}$

Theorem: the queue invariants hold

THEOREM $\mathit{InitialState} \Rightarrow \Box\, \mathit{QueueInvariant}$
