YARN-117 changes

Goals

1. Make it possible to stop() a service without caring what state it is in
2. Move state checking logic ahead of subclasses' actions
3. Make service implementation robust against failure.
4. Reduce code needed to make walking service through its lifecycle as a main()-invoked operation.

For (any) management tools

1. Make registration of ServiceStateChangeListenerslpart of the Service interface, not just the AbstractService implementation
2. Add queryable log of service history.
3. Make notion of blocking waiting for other services explicit. Currently some services have service dependencies (Hadoop 1: DN => HDFS, TT=>JT, JT=>HDFS), but these are never explicitly exposed to management tools except, sometimes, through getSafeMode() probes.
4. Provide a history of lifecycle transitions: event and time, in a serializable form.

State Model

1. stop() is valid from all states; when invoked from STOPPED it MUST be a no-op.
2. When an implementation raises an exception during state transition it MUST record the exception raised, then invoke stop() on itself from the (undefined intermediate) state.
3. The fault raised during the transition, and the state that was being entered when the failure occurred must be queried from the Service interface

Service Interface

- STATE enum includes a string value and (explicitly) a numeric one -used for indexing into arrays and case statements.
- Callers can query exception and fault states
- method `waitForServiceToStop(timeout)` allows caller thread to block waiting for the service to stop. This can be used in a sequence of

  ```
  service.stop();
  service.waitForServiceToStop(0);
  ```

  to shut down a service robustly
- Method getBlockers() returns a map of <String,String> containing names & details of services (or anything else) upon which a service considers itself "blocked" from performing useful work on. The service should still be considered live from the perspective of monitoring tools (HA, ZK, etc), but not from the strict notion of performing useful work in a bounded time.

```
/**
 * Service LifeCycle.
 */
public interface Service {

  /**
   * Service states
   */
  public enum STATE {
    /** Constructed but not initialized */
    NOTINITED(0, "NOTINITED"),

    /** Initialized but not started or stopped */
    INITED(1, "INITED"),

    /** started and not stopped */
    STARTED(2, "STARTED"),

    /** stopped. No further state transitions are permitted */
    STOPPED(3, "STOPPED");

    /**
     * An integer value for use in array lookup and JMX interfaces.
     * Although {@link Enum#ordinal()} could do this, explicitly
     * identify the numbers gives more stability guarantees over time.
     */
    private final int value;

    /**
     * A name of the state that can be used in messages
     */
    private final String statename;

    private STATE(int value, String name) {
      this.value = value;
      this.statename = name;
    }

    /**
     * Get the integer value of a state
     * @return the numeric value of the state
     */
    public int getValue() {
      return value;
    }

    /**
     * Get the name of a state
```

```
   * @return the state's name
   */
  @Override
  public String toString() {
    return statename;
  }
}

/**
 * Initialize the service.
 *
 * The transition must be from {@link STATE#NOTINITED} to {@link STATE#INITED}
 * unless the operation failed and an exception was raised.
 * @param config the configuration of the service
 */
void init(Configuration config);


/**
 * Start the service.
 *
 * The transition should be from {@link STATE#INITED} to {@link STATE#STARTED}
 * unless the operation failed and an exception was raised.
 */

void start();

/**
 * Stop the service.
 *
 * This operation must be designed to complete regardless of the initial state
 * of the service, including the state of all its internal fields.
 */
void stop();

/**
 * Register an instance of the service state change events.
 * @param listener a new listener
 */
void register(ServiceStateChangeListener listener);

/**
 * Unregister a previously instance of the service state change events.
 * @param listener the listener to unregister.
 */
void unregister(ServiceStateChangeListener listener);

/**
```

```
 * Get the name of this service.
 * @return the service name
 */
String getName();

/**
 * Get the configuration of this service.
 * This is normally not a clone and may be manipulated, though there are no
 * guarantees as to what the consequences of such actions may be
 * @return the current configuration, unless a specific implentation chooses
 * otherwise.
 */
Configuration getConfig();

/**
 * Get the current service state
 * @return the state of the service
 */
STATE getServiceState();

/**
 * Get the service start time
 * @return the start time of the service. This will be zero if the service
 * has not yet been started.
 */
long getStartTime();

/**
 * Query to see if the service is in a specific state.
 * In a multi-threaded system, the state may not hold for very long.
 * @param state the expected state
 * @return true if, at the time of invocation, the service was in that state.
 */
boolean inState(STATE state);

/**
 * Get the first exception raised during the service failure. If null,
 * no exception was logged
 * @return the failure logged during a transition to the stopped state
 */
Throwable getFailureCause();

/**
 * Get the state in which the failure in {@link #getFailureCause()} occurred.
 * @return the state or null if there was no failure
 */
STATE getFailureState();
```

```
  /**
   * Block waiting for the service to stop; uses the termination notification
   * object to do so.
   *
   * This method will only return after all the service stop actions
   * have been executed (to success or failure), or the timeout elapsed
   * This method can be called before the service is inited or started; this is
   * to eliminate any race condition with the service stopping before
   * this event occurs.
   * @param timeout timeout in milliseconds. A value of zero means "forever"
   * @return true iff the service stopped in the time period
   */
  boolean waitForServiceToStop(long timeout);

  /**
   * Get a snapshot of the lifecycle history; it is a static list
   * @return a possibly empty but never null list of lifecycle events.
   */
  public List<LifecycleEvent> getLifecycleHistory();

  /**
   * A serializable lifecycle event: the time a state
   * transition occurred, and what state was entered.
   */
  public class LifecycleEvent implements Serializable {
    /**
     * Local time in milliseconds when the event occurred
     */
    public long time;
    /**
     * new state
     */
    public Service.STATE state;
  }

  /**
   * Get the blockers on a service -remote dependencies
   * that are stopping the service from being <i>live</i>.
   * @return a (snapshotted) map of blocker name-&gt;description values
   */
  public Map<String, String> getBlockers();
}
```

Key changes in AbstractService

- State model moved into its own class, modelling the FSM as an array of booleans to

indicate whether or not that transition was valid.

- subclasses required to extend innerInit() innerStart() innerStop(), methods, which are all protected and throw Exception.
- AbstractService.start(), init() and stop() methods do state check before invoking the inner methods, to verify state change is permitted before any side-effecting operations fail.
- If innerInit() or innerStart() raise an exception
    a. the exception is recorded into AbstractService.failureCause (iff this is non null)
    b. the event is logged at INFO
    c. the Service.stop() method is invoked to place the service in the stopped state.
    d. the exception is throw up, wrapping in a RuntimeException iff necessary.
- If innerStop() raises an exception, the exception is noted and logged, *but not rethrown.* Services are expected to be best effort operations from any state, and not to relay problems up.

ServiceStateException extends YarnException

New exception added to indicate problems during service state changes
- Raised during attempts to change to a state not directly reachable from the current state according to the state model.
- Wrapped around any exception that is not a RuntimeException when relaying exceptions caught in the Service.init()/start()/stop() methods.

```
public static RuntimeException convert(Throwable fault) {
  if (fault instanceof RuntimeException) {
    return (RuntimeException) fault;
  } else {
    return new ServiceStateException(fault);
  }
}
```

This ensures that any YarnExceptions raised during startup do not get wrapped with more exceptions and are simply relayed instead.

AbstractService

Key Changes

1. Implement the new methods in the Service interface
2. Have the state model checked before the subclass state actions take place; automate entering the stopped state if a state transition fails

Init:

```
public void init(Configuration conf) {
 // ISSUE: should null configuration be allowed or rejected?
 if (conf == null) {
    throw new ServiceStateException("Cannot initialize service "
                                    + getName() + ": null configuration");
  }
```

```
    enterState(STATE.INITED);
    this.config = conf;
    try {
      innerInit(conf);
      notifyListeners();
    } catch (Exception e) {
      noteFailure(e);
      stopQuietly(this);
      throw ServiceStateException.convert(e);
    }
  }
```

This

Start is almost identical

```
  public void start() {
    //enter the started state
    stateModel.enterState(STATE.STARTED);
    try {
      startTime = System.currentTimeMillis();
      innerStart();
      LOG.info("Service " + getName() + " is started");
      notifyListeners();
    } catch (Exception e) {
      noteFailure(e);
      stopQuietly(this);
      throw ServiceStateException.convert(e);
    }
  }
```

Stop is more complex as downgrades to a no-op if already stopped, wakes up any object listening for the stop operation to complete.

```
  public void stop() {
    //this operation is only invoked if the service is not already stopped;
    // it is not an error
    //to go STOPPED->STOPPED -it is just a no-op
    if (enterState(STATE.STOPPED) != STATE.STOPPED) {
      try {
        innerStop();
      } catch (Exception e) {
        //stop-time exceptions are logged if they are the first one,
        noteFailure(e);
        throw ServiceStateException.convert(e);
      } finally {
        //report that the service has terminated
        synchronized (terminationNotification) {
          terminationNotification.set(true);
          terminationNotification.notifyAll();
        }
        //notify anything listening for events
```

```
        notifyListeners();
      }
    } else {
      //already stopped: note it
      if (LOG.isDebugEnabled()) {
        LOG.debug("Ignoring re-entrant call to stop()");
      }
    }
  }
```

Ideally, these methods should all be `final`; marking them as that stops subclasses inadvertently overriding them, and finds all such actions at compile-time. However, that stops Mockito from successfully mocking the object, which is why they aren't

AbstractService state action methods

The override points for service implementations to perform their work are now moved into inner-methods

```
  protected void innerInit(Configuration conf) throws Exception {

  }

  protected void innerStart() throws Exception {

  }

  protected void innerStop() throws Exception {

  }
```

these are

1. Protected -and not intended to be directly invoked by anything except the AbstractService init/start/stop methods.
2. Empty: there is no need for direct subclasses of AbstractService to invoke their superclass, though that was done during migration for consistency with the superclass invocation policy needed for subclasses of CompositeService and others.
3. Marked as throwing Exception: there is no need for any implementation class to catch and wrap any exception raised -that will be done automatically by AbstractService, which only wraps exceptions that are not themselves RuntimeException instances.

CompositeService

1. CompositeService now stops *all* services, in reverse order, from any state. Previously: if

stop() was entered during init(), only those services that were already in the STARTED state were stopped() (which was done implicitly based on the index of the array).

Provided the innerStop() methods work from all states, stopping all services guarantees that the child services are all stopped -including when a startup fails mid-way through init or start.

If we want the existing behaviour back, then the stop() operation would look at each service and only stop() those in the started state. This would mean that calling stop() from INITED wouldn't stop any of the children.

2. In innerStop(), the first exception thrown by any child's stop() method is caught and then rethrown after all other child services have been stopped. Previously: logged and ignored.

ServiceOperations

- helper methods (startService(), stopService()) that offered robustness by doing state checks before the operations made simpler by relaying directly to the service, retaining only checks for null parameters where appropriate.
- Generic ServiceShutdownHook added -calls Service.stop() on its (weakly-referenced) service. This can be used as the shutdown hook for any service, to eliminate the need for service specific hooks
- ServiceListeners class added to manage listeners to a service robustly -the list of listeners is copied in a synchronized block before the notifications start, so that a notification can change the list (including removing itself) without triggering ConcurrentModificationException

Tests that broke

- Tests that assume after a failure in start(), the final state is INITED and not STOPPED.
- Mockito mocking of services triggered NPEs in AbstractService until init(), start() and stop() were made non-final again
- TestNodeStatusUpdater failed on failing services in the wrong state, the wrong exception text (class of YarnException no longer in the string), and the fact that when exceptions in start() are relayed up, RuntimeException instances are no longer wrapped in YarnExceptions (see: ServiceStateException.convert())

TestServiceLifecycle needed changes

- Its current implementation modelled the existing (flawed) design, including assertions that if init(), start() or stop() or were called twice, their counters would be incremented, as the state checks only took place after the subclass's actions.
- It expected IllegalStateException to be raised on illegal state transitions, these are now ServiceStateException

The test is a lot simpler now, as it shows that the service model is stricter.

Issues

- should init(Configuration) reject null configurations?
  If this is enabled, TestResourceLocalizationService.testResourceRelease() fails.
  When disabled, the new tests TestCompositeService.testInitNullConf() and TestServiceLifecycle.testInitNullConf() should be set to warn rather than fail on an init(null)
- ContainerManager: should stop() stop through all services (in reverse order), or copy the original design and only stop() those already in the started state, starting at one below the current value? What will it do when stop() is called from init()
  Given stop() is now valid from all states, stopping all services in reverse order is now a legal state transition -and the only one guaranteed to roll back from init reliably. But it relies on stop operations being robust in all states, which may be premature.
- Making init/start/stop final is the way to stop subclasses breaking things; forcing them to move to the new inner{Init,Start,Stop} methods. However, do that and TestResourceLocalizationService breaks because Mockito can't override the final methods
- Should exceptions in stop() be thrown or not?
  Yes: consistent with current model
  No: more robust against shutdown problems
  Status: they are thrown