Scheduler Improvements (WiP)

Checklist

- ☐ User Stories Documented
- ✓ User Stories Reviewed
- ✓ Design Reviewed
- ✓ APIs reviewed
- ✓ Release priorities assigned
- ✓ Test cases reviewed
- ✓ Blog post

Introduction

The current scheduler runs in the master and is a (almost) purely cron-based Quartz scheduler. This imposes certain limitations that we wish to in a redesign of the CDAP scheduler.

Goals

- Scalability: Quartz can only schedule as many programs as it has threads. Each threads remains occupied (even though it is just waiting) while the workflow is executing.
- Reliability: When the master fails over, it may take a few seconds to a minute. During that time schedules can be missed.
- Efficacy: Pure time-based scheduling only works well for data that arrives reliably and periodically. For any data that arrives at unknown times, the schedule has to trigger speculatively, and the workflow will do nothing if there is no data. This can be wasteful in terms of cluster resources.
- Expressivity: Certain scheduling constraints can hardly be expressed in terms of only time. For example, a schedule that fires whenever at least 1GB of data is available, or at least 4 hours have passed since the last run.

User Stories

1. In a large cluster, hundreds to thousands of workflows are scheduled to process incoming data at fixed times. The scheduler needs to be scalable enough to fulfill these schedules reliably.
2. The CDAP master fails over. No schedules should be skipped.
3. New data for a feed arrives once daily at unknown times. The SLA for processing is 3 hours after arrival. With cron-style scheduling, this workflow must run every 3 hours, but 7 out of 8 times it will do nothing. Instead the workflow should only run when the data arrives. This would reduce the number of runs and the processing latency.
4. Workflow B always reads the output of workflow A. Ideally, B should run immediately after A has successfully completed, and B should not run if A failed.
5. A data lake has thousands of feeds. Many of these feeds arrives rarely, and their SLAs are generous: They need to processed once a day at no particular time.
6. A data lake has a high load of feeds and processing, and at times it may not have enough resources to process all incoming data. Feeds can be marked with a priority, such that high priority workflows are always scheduled on time, whereas low-priority workflows can defer processing and catch up at times of lower load.

7. A data maintenance workflow for a feed needs to run once a day. It should run within a time window such as 10pm-6am. In some cases, it only needs to run if there is new data.
8. A workflow consumes (joins) multiple datasets. Ideally, it should be scheduled to run after all of its inputs have received new data. However, if some of its inputs do not receive new data, the workflow should be scheduled after a time of waiting; it will then consume the existing data in these inputs.
9. A workflow processes a feed that arrives at unknown times. When that data arrives, it may arrive in chunks over a time window, or it may be known that it always arrives in N chunks. The workflow should be triggered by the new data, but only after all data has arrived.
10. A workflow processes a feed that arrives frequently (for example, every second). The workflow should be triggered when data is available. However, it should not run every second, but always wait for enough data to accumulate, where "enough" needs to be quantified in some way (mega bytes, seconds elapsed, etc.)
11. A workflow needs to run at least every 4 hours. But if the data is bursty, it should also run sooner, as soon as 100GB of data have arrived.

Design

Here we need to design 2 aspects:

a. Improving the scalability of the scheduler so that it can support thousands of concurrent schedules.

b. Improving the expressive power of schedules by adding event-based, conditional, and composite triggers.

Scalability

- The Quartz scheduler is limited in scale because it requires an active thread per schedule; the thread blocks as long as the program triggered by the schedule is running. That limits the number of programs that can be scheduled and executed at the same time. A re-implementation of the time-based scheduler (be it without Quartz, or be it using Quartz in a different way) would greatly increase the number of schedules supported by a single instance of the scheduler.
- Multiple instances of the scheduler can scale out linearly. This requires that instances do not interfere with each other, that is:
  - either there is a disjoint assignment of schedules to instances
  - or all instances can share all schedules by keeping all state persisted, and by mutually ensuring only one instance works on the same schedule at the same time
- Multiple instances of the scheduler can run
  - either in each CDAP master. That means a secondary (standby) master must still have an active scheduler
  - in Yarn containers. This would further increase the Yarn container footprint of CDAP's master services. But it would decouple the scheduler from the number of masters, providing more scalability.

Expressiveness

Schedules can be enhanced with these features:

- A trigger based on a notification from TMS. Other system services emit these notifications, for example:
  - A new partition in a file set
  - A program has succeeded (or failed)
  - A stream has N GB of new data
- A repeated trigger, for example:
  - 4 new partitions in a file set
  - A program has succeeded N times
  - etc.
- A time span has passed since the last time an event has happened
- A composite trigger:
  - (AND) Multiple triggers must all occur
  - (OR) One out of several triggers must occur
  - An OR could also be expressed as two separate schedules
- Run constraints:
  - Limit on the number of concurrent runs (existing for time schedules)
    - Limit across a group of workflows/schedules
  - Limit on the number of runs per time unit (hour, day, etc)
  - Delay a schedule after it triggered until a configured time window (for instance, run a program N minutes after the required data arrives)
  - Conditionally execute based on cluster load/resource availability
  - Constraints should be able to suppress or delay the execution of a program

Terminology

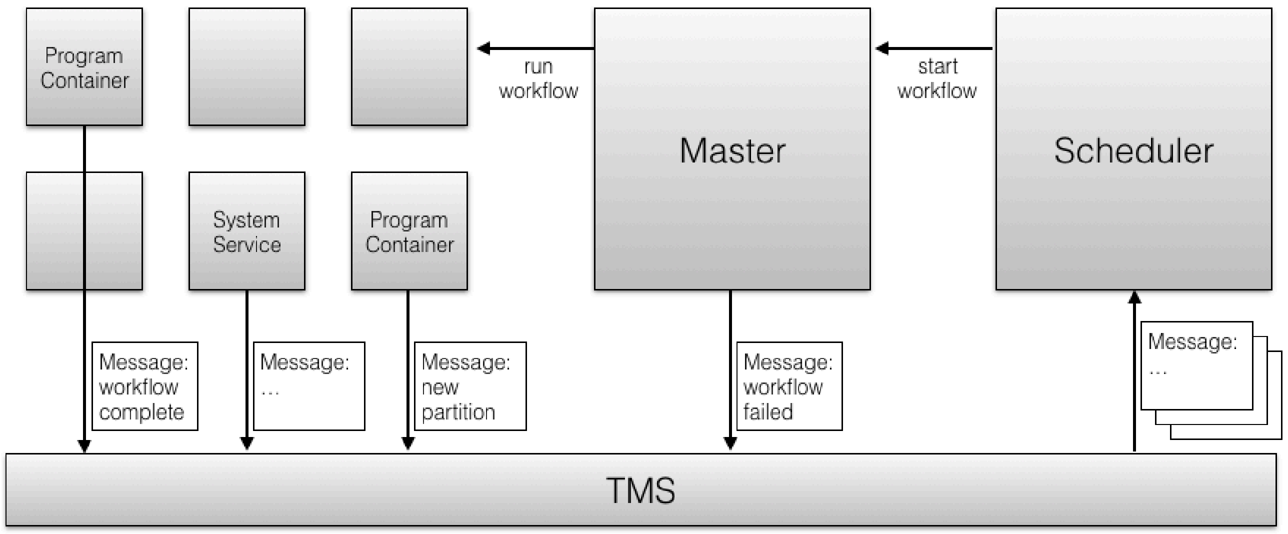
We will consistently use the following terms when talking about schedules:

- A schedule consists of the following:
  - The program to start, along with runtime arguments or options.
  - A trigger
  - A set of run constraints
- A trigger is a rule describing a condition that initiates the start of a program. A trigger can be
  - time-based
  - event-based
  - (possibly more in the future)
- A run constraint is a condition that needs to be fulfilled before the program is started.
  - A run constraint can either cancel or delay the start of the program
- A message is an object that contains an event
  - Messages are transported via TMS (transactional messaging service) topics
- A job represents a schedule one of whose triggers has been fired by an event.
  - A job is pending if one or more of its constraints or triggers are not fulfilled yet
  - What exactly is contained in a job is an implementation detail. Most likely it contains:
    - The schedule (or its id) that was triggeredThe trigger that caused this job to be created, and the event that fired it.
    - Additional triggers that have fired (and their events). This is in the future when we start supporting composite triggers.
    - The program to be started and its runtime arguments (or perhaps we only compute the runtime arguments when we start the program)
    - All of the schedule's run constraints, and whether and how they have been fulfilled.
- The (pending) job queue is the set of pending jobs maintained by the scheduler.
  - TODO: find a better name. Job queue? Pending set?

Approach

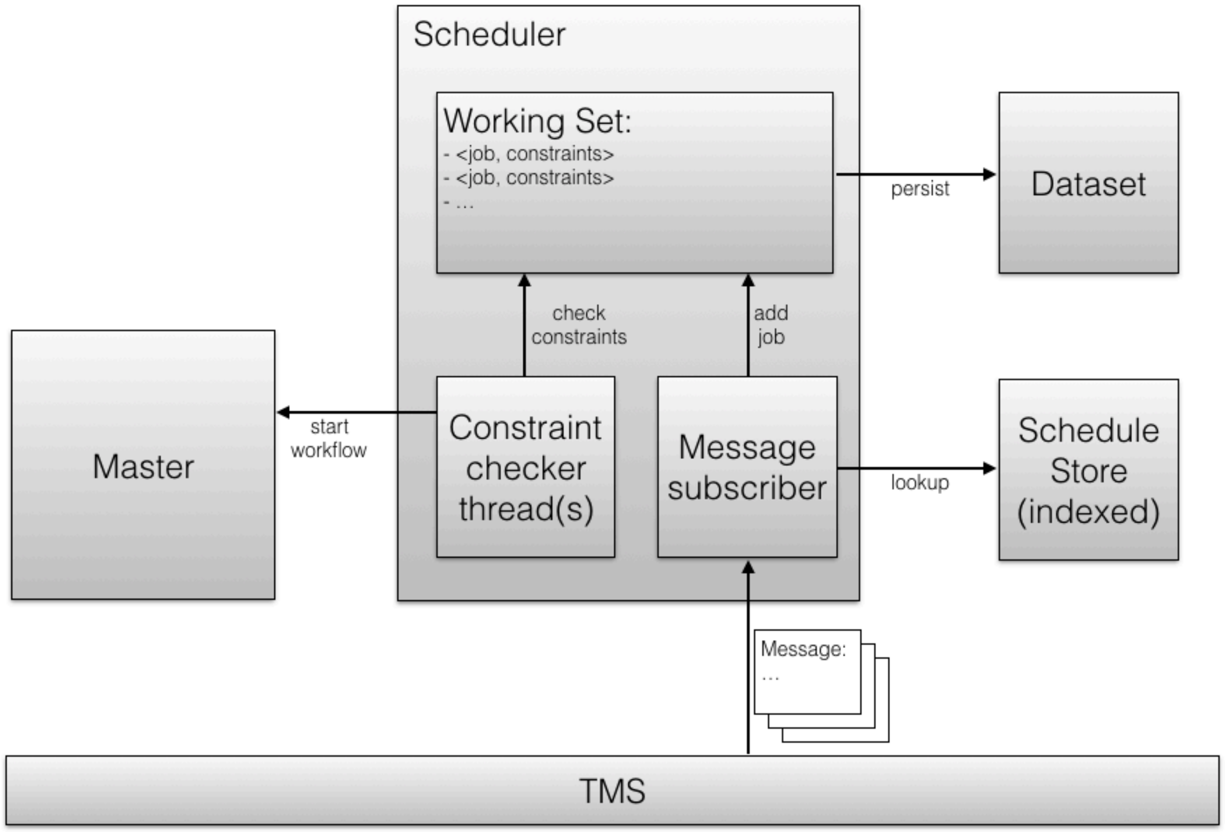
Event-based scheduling will rely on the Transactional Messaging System (TMS) in CDAP.

End-to-End Architecture



- Program container, system services, and the CDAP master will emit messages to TMS topics. These messages can be about program life cycle, data availability, or possibly other types of events.
- The scheduler subscribes to these messages transactionally, that is, it makes sure it never drops a messages.
- Based on the event described by a message, the scheduler starts workflows (and in the future, other program types).
- Some workflows have run constraints that prevent them from running or delay the start of the workflow.
  - The scheduler maintains internal state about what workflows have been triggered and what constraints they are depending on (the job queue of pending jobs)
  - When all constraints of the job are fulfilled, the scheduler starts the job's program.
  - More details below.
- To start a program, the scheduler relies on the CDAP master. That is, none of the program start and monitoring logic needs to move into the scheduler.
- The scheduler could run inside the CDAP master, or it could run in a system container for greater scalability.

Scheduler Internals



The scheduler has two main components:

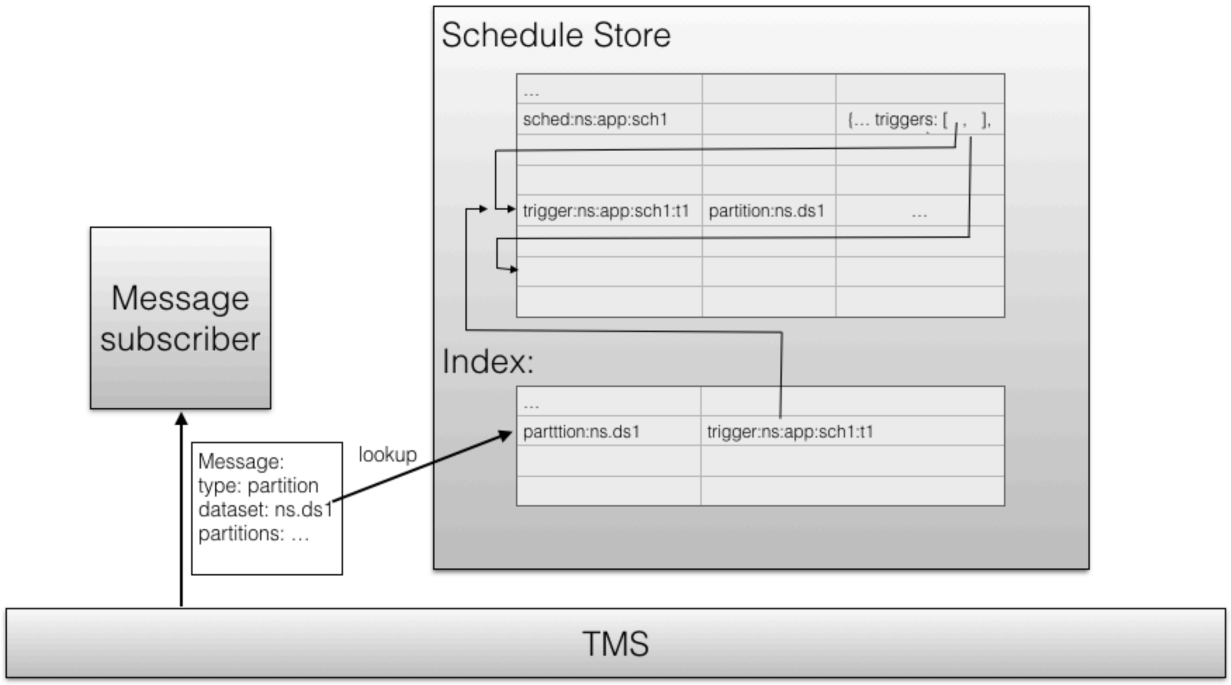
- A TMS subscriber that consumes messages.
  - For each event found in a message, it looks up the schedule store to find all schedules triggered by this event and creates or updates a job in the job queue for it. To make the schedule lookup efficient, the schedule store must be indexed or allow some other kind of reverse lookup.
  - After all jobs are created for a message, the message is marked as read, and it will never be consumed a again. To make sure that it never drops an event, the job must be persisted to the job queue in the same transaction.
- A constraint checker (TODO: find better name).
  - It validates the run constraints for the jobs in the job queue.
  - When all constraints for a job are fulfilled, it makes a call to the CDAP master to start the job, and removes the job from the job queue.

Schedule Store

The schedule store persists all schedules (whether they were created as part of app deployment or independently) in a way that allows looking up all scheduled possibly impacted by an event without requiring a full scan. That is, it needs some kind of reverse index by event key. For example, if a message arrives with an event about a new partition in a dataset, the event key would be (similar to) `partition:<namespace>:<dataset>`. Using this key, we should be able to find all schedules that have a partition trigger for that dataset. This is simply an indexed table if a schedule can only have one trigger. However, in the future, we will support multiple triggers per schedule (through a composition by AND or OR). That is, a single schedule will have multiple event keys. IndexedTable does not support multi-valued index columns, and we cannot use a fixed set of index columns, because the number of triggers in a schedule will be variable. That means we need an indirection: Every (non-composite) Trigger has one (or zero, if time-based) event key.

- Every trigger has a unique key. For simplicity, the user should not have to specify this, but we can generate that based on the schedule id.
- The schedule store is an indexed table with two types of entries:
  - Schedules. The entry lists in some form all the trigger keys
  - Triggers: These have an indexed column for the event key. The trigger key uniquely identifies the schedule
- When a message arrives:
  - construct an event key from the event in the message. For example, `partition:<ns>.<dsid>`
  - lookup the triggers that match this event key in the index
  - find the schedules that these triggers belong to
- When a schedule is added:
  - Generate a trigger key for each of its triggers
  - Insert the schedule and the triggers in the same transaction

- When a schedule is deleted:
    - Remove the schedule and all its trigger in the same transaction
- When a schedule is modified:This is not supported.
    - Because the semantics of pending jobs are hard to define.



JobQueue

The JobQueue is an intermediary queue, where a notification has arrived for a schedule, but the program has not yet been launched. This could either be because it is waiting for its trigger to be fulfilled or its constraints to be satisfied.

Once the Notification subscriber receives a notification from TMS, it will:

- Look up the schedules that are triggered by the notification.
- Look up any existing jobs in the job queue for these schedules
    - If a job exists in a state other than PENDING_LAUNCH, add the notification to the job
    - Otherwise, create a new job and add the notification to this new job

On the other side of this JobQueue, there will be a ConstraintChecker process that scans the job queue and checks the triggers/constraints of the Job, launching any jobs whose constraints are satisfied.

One potential issue is when the ConstraintChecker launches a job that has its constraints satisfied and removes the job from the JobQueue, it could have a transaction conflict if the Notification subscriber adds a notification to this job during the same time. To avoid such a transaction conflict, the ConstraintChecker will mark the job state as PENDING_LAUNCH, and the notification subscriber will know not to modify this job if it is in such a state.
Other operations may have transaction conflicts, but those should be fine since the attempting entity can retry.

In the first cut of this dataset, the ConstraintChecker will be performing continuous scan over the dataset (no indexing or caching in-memory). The downside to this is that it involves more hbase scans, but the advantage is its simplicity.

Suspend and Resume

through a separateUsers need to be able to suspend and resume schedules. For now, we will implement a simple logic:

- When a schedule is created (as part of app deployment, or through separate REST endpoints), it is initially suspended.
- For a suspended schedule, the TMS subscriber will ignore all events. That is, no jobs will be triggered and added to the job queue.
- Resuming the schedule means that the subscriber will start reacting to event for that schedule
- Suspending the schedule results in deleting all pending jobs for that schedule in the job queue.

This conforms with existing behavior.

In the future, we will want to support other modes of suspension:

- keep existing jobs in the queue and launch them as soon as their triggers and constraints are fulfilled, but do not add new jobs
- keep existing jobs and add new jobs, but instead of launching them when they are ready, add them to a "backlog" queue, such that they can run when the schedule resumes
- keep existing jobs and update them as new events arrive, but keep them in the job queue until the schedule resumes.
- question: if jobs remain in the job queue, should they still time out?

This needs to be defined more precisely, and the required modes should be driven by use cases.

TMS Topics

What topics will be used for TMS message transport?

- We will NOT segregate TMS topics by namespace. That is, we will not have a separate topic for each namespace, but all namespaces will share the same topic. This makes topic management siginificantly easier: We do not need to create/delete topics every time we create/delete a namespace. Also, the scheduler does not need to subscribe/unsubscribe every time a namespace is created or deleted. Scalability should not be an issue: In the end the scheduler needs consume all messages. We expect the volume of messages to be relatively low.
- We will have one topic per message type: Data availability and program lifecycle events. These topics can be statically configured in cdap-site.xml

TMS Message Design

The message sent on TMS will have the following fields:

- timestamp (long). Epoch timestamp of the event.
  eventType (String).
- payload (Json object), where the set of fields depends on the eventType.

For PFS Partition messages, it will contain the following fields:

- datasetId: the id (including namespace) of the dataset
- partitionKeys: GSON-serialized String of the list of partition keys that were added

Trigger Event

```
 public class TriggerEvent {
  private final long timestamp;
  // TODO: change to Enum
  private final String triggerType;
  // TODO: change to object?
  private final Map<String, String> properties;

  public TriggerEvent(long timestamp, String triggerType, Map<String,
String> properties) {
    this.timestamp = timestamp;
    this.triggerType = triggerType;
    this.properties = properties;
  }
}
```

API changes

A schedule will be represented as an object consisting of the following fields:

- Program:
    - Defines the program to launch when the schedule is triggered
    - Also includes runtime arguments
- Trigger:
    - This can either be a time trigger or a data trigger.
    - Stretch: also a program life cycle trigger.
    - In the future: also a composite trigger
- A set of Run constraints
    - Can delay or cancel a job after it has been triggered. Note that ConcurrencyConstraint and DurationSinceLastRunConstraint are vulnerable to a race condition, where two separate schedules can launch a program simultaneously after checking satisfaction of the constraint.
        - Concurrency: Defines an upper limit on the number of runs of the specified program
        - Delay: Duration after which the schedule is triggered that the program execution will be attempted.
        - Time Range: A time range that defines when the schedule is allowed to run. For instance (1-3AM).
        - Duration Since Last Run: A duration that must have passed since the last execution of the program.
        - more in the future
- A timeout for how long to wait for the run constraints to become fulfilled
    - Either cancel or force start of the program

Trigger Types:

- Time Trigger: Defined simply by a cron expression
- Data Trigger: Currently only one type of data trigger will be implemented:
    - PartitionedTrigger: Defined by a dataset Id and a number of partitions
    - Stretch: Stream Size Trigger, for compatibility with existing stream size schedules
- Stretch: Program Lifecycle Trigger
    - A program id and the type of lifecycle event (started, completed, failed)
- More in the future

Semantics of App Deployment

We allow creation of schedules as part of deploying an application; and we also support managing schedules separately through REST/CLI /UI. The interaction of these two paradigms is tricky: For example, when an app is updated(redeployed) with a different schedule than it previously had, should that schedule be updated? What if that schedule was changed though a REST API in the mean time, should redeployment of the app change the schedule again?

To understand this better, we need to look at different environments:

1. In a development environment, an engineer writes an app, deploys it, tests it, makes some changes, deploys again, etc. For convenience, she wants deployment to be self-contained, that is, deploying an app also creates all schedules, all datasets etc. If she changes the schedules in the app, she expects that redeploying updates the schedules just like it updates the programs of the app (and the datasets, but that is beyond the scope of this doc).
2. In a production environment, where the cluster is shared by different workloads, schedules need to adapted to cluster utilization and business requirements, and possibly coordinated across applications. The developer of a single app cannot possibly do that. Instead the operations team (ops) creates and manages the schedules after the application is deployed. When the application is redeployed (to fix a bug, add a feature, etc.), then that should not affect the existing schedules.

Scenario 1 represents exactly the semantics of pre-4.2 CDAP, and Scenario 2 represents pre-4.2 with the property (app.deploy.update.schedules) set to true in cdap-site.xml (or as an option for app deployment). We will keep this exact behavior:

- if app.deploy.update.schedules is true in cdap-site.xml, or given as an option when deploying the app, then the deployment will:
    - delete all existing schedules that are not given by the app
    - create schedules that are given by app and do not exist yet
    - update (that is, delete and recreate) all schedules that exist but are different from what the app specifies.
- if app.deploy.update.schedules is false in cdap-site.xml, or given as an option when deploying the app, then the deployment will:
    - neither delete nor create or update any schedules.
    - however, if a program is removed, then all its schedules are also deleted, and the same for deletion of an entire application
- the default forapp.deploy.update.schedules in cdap-default.xml is true - to accommodate the fact that there are more dev environments (including SDK) than production environments.

This preserves backward-compatibility with pre-4.2, with a difference in the implementation: The app spec, which used to be the source of truth for what schedules exist, will now represent only the schedules that were configured by the app. The source of truth is now the schedule store, and this is what the REST endpoints for managing schedules will work on, they will not touch the app spec.

New Programmatic APIs

New Java APIs introduced (both user facing and internal)

Schedule Builder

```
public class Schedules {

  public static Builder builder(String name) {
    return new Builder(name);
  }

  public static class Builder {
    private final String name;
    private String description;
    private List<Constraint> constraints;

    private Builder(String name) {
      this.name = name;
      this.description = "";
      this.constraints = new ArrayList<>();
    }

    public Builder setDescription(String description) {
      this.description = description;
      return this;
    }

    public Builder setMaxConcurrentRuns(int max) {
      if (max < 1) {
        throw new IllegalArgumentException("max concurrent runs must be
at least 1.");
      }
      constraints.add(new ConcurrencyConstraint(max));
      return this;
    }

    // fields that pertain more to data-triggered schedules
    public Builder setDelayMillis(long delayMillis) {
      // TODO: disallow from being called multiple times?
      constraints.add(new DelayConstraint(delayMillis));
      return this;
    }

    public Builder setTimeRange(int startHour, int endHour) {
      constraints.add(new TimeRangeConstraint(startHour, endHour));
      return this;
    }

    public Builder setDurationSinceLastRun(long delayMillis) {
      constraints.add(new DurationSinceLastRunConstraint(delayMillis));
      return this;
    }

    // TODO?: Add a method that allows adding a generic Constraint
object?
    // This will mean moving our Constraint interface to API, and
somehow packaging user class into Scheduler system

    public Schedule build(Trigger trigger) {
      return new Schedule(name, description, trigger, constraints);
    }
  }
}
```

Triggers

```
public class TimeTrigger {
  private final String cronExpr;

  public TimeTrigger(String cronExpr) {
    this.cronExpr = cronExpr;
  }
}

...

public class PFSTrigger {
  private final String pfsName;
  private final int numPartitions;

  public PFSTrigger(String pfsName, int numPartitions) {
    this.pfsName = pfsName;
    this.numPartitions = numPartitions;
  }
}
```

Constraints

```
public class ConcurrencyConstraint implements Constraint {

  private final int maxConcurrency;

  public ConcurrencyConstraint(int maxConcurrency) {
    this.maxConcurrency = maxConcurrency;
  }

  @Override
  public Result check(Schedule schedule, ConstraintCheckContext
context) {
    return null;
  }
}

...
```

```
public class DelayConstraint implements Constraint {

  private final long millisAfterTrigger;

  public DelayConstraint(long millisAfterTrigger) {
    this.millisAfterTrigger = millisAfterTrigger;
  }

  @Override
  public Result check(Schedule schedule, ConstraintCheckContext
context) {
    return null;
  }
}

...

public class TimeRangeConstraint implements Constraint {

  // only is satisfied within the range [startHour, endHour)
  // TODO: Allow minute granularity
  private final int startHour;
  private final int endHour;

  public TimeRangeConstraint(int startHour, int endHour) {
    this.startHour = startHour;
    this.endHour = endHour;
  }

  @Override
  public Result check(Schedule schedule, ConstraintCheckContext
context) {
    return null;
  }
}

...

public class DurationSinceLastRunConstraint implements Constraint {

  private final long millisSinceLastRun;

  public DurationSinceLastRunConstraint(long millisSinceLastRun) {
    this.millisSinceLastRun = millisSinceLastRun;
  }

  @Override
  public Result check(Schedule schedule, ConstraintCheckContext
context) {
    return null;
  }
}
```

Constraint

```
public interface Constraint {

  Result check(Schedule schedule, ConstraintCheckContext context);

  enum Result {
    SATISFIED,
    RETRY,
    NEVER;

    private final Long millisBeforeNextRetry;

    Result() {
      this(null);
    }

    Result(@Nullable Long millisBeforeNextRetry) {
      this.millisBeforeNextRetry = millisBeforeNextRetry;
    }
  }
}

...

public final class ConstraintCheckContext {
  private final int currentRunningInstances;
  private final List<RunRecord> history;
  private final TriggerEvent triggerEvent;

  public ConstraintCheckContext(int currentRunningInstances,
List<RunRecord> history, TriggerEvent triggerEvent) {
    this.currentRunningInstances = currentRunningInstances;
    this.history = history;
    this.triggerEvent = triggerEvent;
  }

  public int getCurrentRunningInstances() {
    return currentRunningInstances;
  }

  public List<RunRecord> getHistory() {
    return history;
  }

  public TriggerEvent getTriggerEvent() {
    return triggerEvent;
  }
}
```

Deprecated Programmatic APIs

New REST APIs

Path	Method	Description	Response Code	Response
/v3/apps/<app-id>	GET	Returns the application spec for a given application	200 - On success 404 - When application is not available 500 - Any internal errors	

Deprecated REST API

Path	Method	Description
/v3/apps/<app-id>	GET	Returns the application spec for a given application

CLI Impact or Changes

- Impact #1
- Impact #2
- Impact #3

UI Impact or Changes

- Impact #1
- Impact #2
- Impact #3

Security Impact

The TMS topics are not protected via authorization. This must be addressed in the future.

- Producers: For now, only PartitionedFileSet will emit notifications from user space
- Consumer: All messages are consumed by a system service as the CDAP user.

Miscellaneous Open Questions

- How will the notification metadata be exposed to workflow (metadata of the trigger)?
- How can PFS Consumer be leveraged?
- If a notification triggers many programs, how will atomicity be handled?
- How will scaling of the scheduler service be handled?
- Replication impact?
- Do we need CLI commands? Probably yes
- REST endpoints: JSON schema
- Semantic of Suspend and Resume

Work Items

- Notifications
  - Design topology of topics and schema of messages
  - Emit messages to TMS
    - when a program completes
    - when a partition is added to a PFS
    - others?

        - Security:
            - Need to emit from program runners/app fabric but also from dataset=application code
            - Need to subscribe from CDAP system service (scheduler)
- Specification
    - Design and implement API tp configure schedules, triggers and run constraints
        - as part of app deployment
        - separately from app deployment
- Scheduler
    - Implement as a system service running in Yarn
        - Impersonation
        - Workflows are still started by master?
    - Subscribe to TMS
    - Message-to-Schedule lookup (n-to-m mapping)
    - Atomicity of message consumption when a single event triggers multiple schedules
    - Run constraints
        - Max Concurrent runs
            - of same workflow
            - of other workflows in same "bucket" (how to define that)
        - Only run in specific time window (e.g. 1-pm-6am)
        - Delay workflow start by N minutes after message triggered
        - Run no sooner than N minutes after last run of the workflow
        - Unfulfilled constraint can lead to
            - skip this run of the schedule
            - wait until constraint is fulfilled
                - what if multiple runs triggered by the same schedule are waiting, do they all start when constraint become true? Or only one?
        - others?
    - job queue = all pending triggers (schedules that were triggered and are waiting for run constraints to become true)
        - Persistence to survive outage
        - Monitoring/polling of working and starting workflows when their constraints are fulfilled
    - Tools to debug / triage
        - debugger to inspect the state
        - emit metrics
        - record history in some way
    - Scalability
        - multiple instances of scheduler sharing the work
- Tech debt
    - Port existing schedulers to new framework
        - Quartz scheduler
        - Stream size scheduler
    - Migration of existing schedule store to new schema/format
- Test
    - Long running tests for all above scenarios
- Examples
- Documentation

Impact on Infrastructure Outages

System behavior (if applicable - document impact on downstream [YARN, HBase etc] component failures) and how does the design take care of these aspect

Test Scenarios

Test ID	Test Description	Expected Results
1	Configure a schedule to run whenever there is ANY new partition, but configure a minimum interval (5 minutes?) between runs. Add five partitions, distributed evenly across a five minute time window.	The first partition being added will trigger a workflow run, but the partitions added right after it shouldn't trigger a workflow run because sufficient time hasn't passed.
2	Configure a schedule with properties as program runtime args. After a scheduled program run completes, update the schedule with new properties via app deployment. Wait for another scheduled program run.	Scheduled program runs with schedule properties as runtime args, and after updating the schedule, program runs again with updated schedule properties as runtime args.
3	Configure an app with multiple schedules. Redeploy the app with	When updating schedule with app redeployment is not

	a different set of schedules but option of updating schedule with app redeployment is false. Trigger schedules in the updated app. Redeploy the app with some schedules removed, some schedules updated, and some new schedules added, and updating schedule with app redeployment is allowed.	allowed, listing schedules returns the original set of schedules. Schedules in the updated app won't launch any program unless they exists in the original app. When updating schedule with app redeployment is allowed, only schedules in the updated app can be listed and triggered.
4	Configure a schedule which requires 5 partitions. Add 1 partition and update the schedule via REST API with new properties, add 4 more partitions, and wait for 10 seconds. Send 1 more partition and wait for program to run.	Schedule should not launch the program after receiving 4 more partitions after update. After receiving totally 5 new partitions after update, schedule launches program with new properties.
5	Configure a schedule which requires 5 partitions and a 10 minutes delay constraint. Add 5 partitions and let the trigger to wait for delay constraint. Update the schedule via REST API to require 3 partitions with new properties and a 0.1 second delay constraint. Add 3 new partitions.	Schedule will launch the program with new properties after 3 new partitions are added after update.
6	Configure a schedule which requires 5 partitions. Add 4 partitions. Delete the schedule via REST API and immediately add 1 more partitions, and check for program status after 10 seconds.	Schedule should not launch the program
7	Configure a schedule which requires 5 partitions and a delay constraint. Add 5 partitions and let the trigger to wait for the 40 seconds delay constraint. Delete the schedule 500 milliseconds before the configured delay constraint arrives.	Schedule should not launch the program
8	Configure a schedule with a trigger and multiple constraints.	Schedule should only launch the program when the trigger and all constraints are satisfied

Releases

Release 4.2.0

Scope:

- Scheduler runs in CDAP master, only one instance is active
- Supported event types:
  - Availability of a new partition in a (T)PFS,
  - Completion of a program (stretch).
- Supported triggers:
  - Availability of N partitions
  - Successful completion of a program (stretch)
  - Failure of a program (stretch)
- Supported run constraints:
  - Concurrent runs of the same program
  - Concurrent runs of programs in the same "category" (stretch)
  - Delay execution by N time units
  - Minimal time elapsed since last run of the same program
  - Only run in a fixed time window
  - Timeout of a pending job: discard job
  - Timeout of a pending job: force run (stretch)
- Port existing features
  - Time trigger
  - Stream size (stretch)
  - Online upgrade step
- Diagnostic tools

Sprint 1: (demo 5/1)

- ✓ Finish Design
- ✓ Create TMS topic for new partition messages
- ✓ Publish messages for new partitions
- ✓ Define a schedule with single new partition trigger
- ✓ Scheduler to consume from TMS, find schedules, immediately start program.
  - ✓ no run constraints
  - ✓ no job queue

Sprint 2: (demo 5/8)

- ✓ Indexed schedule store
- ✓ Design semantics of deployment/update (Andreas)
- ✓ REST endpoints to delete/add/list schedules (Andreas)
- ✓ Implement job queue (Ali)
- ✓ Triggers (Ali)
    - ✓ based on N new partitions (Ali)
    - ✓ ~~by program completion (stretch)~~
        - ✓ ~~emit messages from program runner (workflow only)~~
        - ✓ ~~schedule triggered by successful run~~
    - ✓ time-based (Mao)
        - ✓ keep quartz (Mao)
        - ✓ when triggers, simply put into job queue (Mao)
- ✓ Run constraints (skip only): (Ali)
    - ✓ define common API
    - ✓ concurrent runs (Ali)
    - ✓ time elapsed since last run (Ali)
    - ✓ fixed time window (Ali)

Sprint 3: (demo 5/15)

- ✓ Online upgrade of existing schedules (Mao)
- ✓ Disable (formerly suspend) and enable (resume) (Andreas)
- ✓ ~~Triggers~~
    - ✓ ~~failed program run~~
- ✓ Run constraints with delay (Ali)
    - ✓ delay execution
    - ✓ timeout and discard
- ✓ Diagnostic tools:
    - ✓ inspect job queue (Ali)
    - ✓ inspect schedule store (Andreas)
    - ✓ emit metrics (Andreas)

Sprint 4: (demo 5/22)

- ✓ Integration test

Sprint 5: (demo 5/29)

- ✓ Long-running test completed

Release X.Y.Z

Related Work

- Work #1
- Work #2
- Work #3

Future work