Temporal Schedules: Reliable, Scalable, and More Flexible Than Cron

Temporal Schedules are a powerful feature of the Temporal platform that provide a durable and scalable way to execute tasks at specific times or on a recurring basis. They are a modern replacement for traditional cron jobs, offering enhanced reliability, observability, and control over your scheduled workflows.

With Temporal Schedules, you can define complex scheduling logic, manage overlapping executions, and ensure that your tasks run even in the face of failures or downtime. This document provides a comprehensive guide to understanding and using Temporal Schedules in your applications.

Why Use Temporal Schedules Over Cron?

Traditional cron jobs have been the standard for task scheduling for decades, but they come with several limitations, especially in distributed and high-availability systems. Temporal Schedules address these challenges by providing:

- **Enhanced Workflow Control and Observability**: Gain complete control over your automation processes. With Schedules, you can create, backfill, delete, describe, list, pause, trigger, and update Workflow Executions.

- **Flexible and Extensible Scheduling**: Schedules are more adaptable and facilitate the definition of Tasks with varied schedules and intervals. This adaptability allows for more sophisticated scheduling needs, such as handling overlapping runs.

- **Elimination of External Dependencies**: For Temporal users, Schedules remove the need to integrate external scheduling systems. This simplifies the Workflow process and reduces complexity.

- **Reliability and Scalability**: Designed for reliability and scalability, Temporal Schedules handle the complexities of distributed systems while ensuring your Workflows run as intended, even during failures.

Core Concepts

Schedule Spec

A Schedule Spec defines when a scheduled action should be executed. There are two types of specs:

- **Interval**: A simple, recurring time interval, like "every 30 minutes". You can also specify a phase offset to adjust the initial start time.

- **Calendar**: A cron-like expression for more complex scheduling, such as "at 2:00 PM on the first Monday of every month".

Overlap Policy

The Overlap Policy controls what happens when it's time to start a new workflow execution, but a previous one is still running. The available policies are:

- **Skip (Default)**: The new workflow execution is not started.
- **BufferOne**: The new workflow execution starts as soon as the current one completes. The buffer is limited to one.
- **BufferAll**: Allows an unlimited number of workflows to buffer. They are started sequentially.
- **CancelOther**: Cancels the running workflow execution and then starts the new one after the old one completes cancellation.
- **TerminateOther**: Terminates the running workflow execution and starts the new one immediately.
- **AllowAll**: Starts any number of concurrent workflow executions.

Catchup Window

The Catchup Window defines how long the Temporal service will try to execute missed actions after a service outage. The default is one year. If your actions are time-sensitive, you can set a smaller window (minimum 10 seconds).

Managing Schedules

You can manage schedules using the Temporal SDKs, the tctl command-line tool, or the Temporal Web UI.

Creating a Schedule

To create a schedule, you need to provide a unique Schedule ID, a spec, and an action.

TypeScript

```
import { Connection, Client } from '@temporalio/client';
import { temporalCommunityWorkflow } from './workflows';

async function run() {
  const client = new Client({
    connection: await Connection.connect(),
  });

  await client.schedule.create({
    scheduleId: 'my-first-schedule',
    spec: {
      cronExpressions: ['0 12 * * MON'], // Every Monday at 12:00 PM
    },
    action: {
      type: 'startWorkflow',
      workflowType: temporalCommunityWorkflow,
      args: ['my-workflow-argument'],
      taskQueue: 'schedules-task-queue',
      workflowId: 'my-scheduled-workflow',
    },
  });
}
```

```
run().catch((err) => {
  console.error(err);
  process.exit(1);
});
```

Python

```python
import asyncio
from datetime import timedelta
from temporalio.client import (
    Client,
    Schedule,
    ScheduleActionStartWorkflow,
    ScheduleIntervalSpec,
    ScheduleSpec,
)
from your_workflow import TemporalCommunityWorkflow

async def main():
    client = await Client.connect("localhost:7233")
    await client.create_schedule(
        "my-first-schedule",
        Schedule(
            action=ScheduleActionStartWorkflow(
                TemporalCommunityWorkflow.run,
                id="my-scheduled-workflow",
                task_queue="schedules-task-queue",
            ),
            spec=ScheduleSpec(
                intervals=
[ScheduleIntervalSpec(every=timedelta(minutes=5))]
            ),
        ),
    )

if __name__ == "__main__":
    asyncio.run(main())
```

Go

```go
package main

import (
    "context"
    "log"
    "time"
    "go.temporal.io/sdk/client"
    "example.com/myworkflows"
)

func main() {
    ctx := context.Background()
    temporalClient, err := client.Dial(client.Options{})
    if err != nil {
        log.Fatalln("Unable to create client", err)
    }
    defer temporalClient.Close()

    scheduleID := "my-first-schedule"
    workflowID := "my-scheduled-workflow"

    _, err = temporalClient.ScheduleClient().Create(ctx,
client.ScheduleOptions{
        ID: scheduleID,
        Spec: client.ScheduleSpec{
            CronExpressions: []string{"0 12 * * MON"},
        },
        Action: &client.ScheduleWorkflowAction{
            ID:        workflowID,
            Workflow:  myworkflows.MyWorkflow,
            TaskQueue: "schedules-task-queue",
        },
    })
    if err != nil {
        log.Fatalln("Unable to create schedule", err)
    }
}
```

Backfilling a Schedule

You can backfill a schedule to execute missed actions or to test a workflow ahead of its scheduled time.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
await handle.backfill({
  startTime: new Date('2023-01-01T00:00:00Z'),
  endTime: new Date('2023-01-02T00:00:00Z'),
  overlapPolicy: 'AllowAll',
});
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
now = datetime.utcnow()
await handle.backfill(
    ScheduleBackfill(
        start_at=now - timedelta(days=1),
        end_at=now,
        overlap=ScheduleOverlapPolicy.ALLOW_ALL,
    ),
)
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-first-schedule")
err = scheduleHandle.Backfill(ctx, client.ScheduleBackfillOptions{
    Backfill: []client.ScheduleBackfill{
        {
            Start:   time.Now().Add(-24 * time.Hour),
            End:     time.Now(),
            Overlap: enums.SCHEDULE_OVERLAP_POLICY_ALLOW_ALL,
        },
    },
})
```

Deleting a Schedule

You can delete a schedule by its ID. This will not affect any workflows that were already started by the schedule.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
await handle.delete();
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
await handle.delete()
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-
first-schedule")
err = scheduleHandle.Delete(ctx)
```

Describing a Schedule

You can get detailed information about a schedule, including its configuration and recent and upcoming executions.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
const description = await handle.describe();
console.log(description);
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
description = await handle.describe()
print(description)
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-
first-schedule")
description, err := scheduleHandle.Describe(ctx)
log.Println("Schedule Description:", description)
```

Listing Schedules

You can list all the schedules in a namespace.

TypeScript

```
const schedules = client.schedule.list();
for await (const schedule of schedules) {
  console.log(schedule.scheduleId);
}
```

Python

```
async for schedule in client.list_schedules():
    print(schedule.id)
```

Go

```
listView, err := temporalClient.ScheduleClient().List(ctx,
client.ScheduleListOptions{})
if err != nil {
    log.Fatalln("Unable to list schedules", err)
}
for listView.HasNext() {
    log.Println(listView.Next())
}
```

Pausing and Unpausing a Schedule

You can pause a schedule to temporarily stop it from executing workflows.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
await handle.pause('Taking the service down for maintenance');
// ...
await handle.unpause('Maintenance is complete');
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
await handle.pause(note="Taking the service down for maintenance")
# ...
await handle.unpause(note="Maintenance is complete")
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-
first-schedule")
err = scheduleHandle.Pause(ctx, client.SchedulePauseOptions{Note: "Taking
the service down for maintenance"})
// ...
err = scheduleHandle.Unpause(ctx, client.ScheduleUnpauseOptions{Note:
"Maintenance is complete"})
```

Triggering a Schedule

You can trigger a one-off execution of a schedule, regardless of its spec.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
await handle.trigger();
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
await handle.trigger()
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-first-schedule")
err = scheduleHandle.Trigger(ctx, client.ScheduleTriggerOptions{})
```

Updating a Schedule

You can update an existing schedule to change its spec, action, or policies.

TypeScript

```
const handle = client.schedule.getHandle('my-first-schedule');
await handle.update((prev) => ({
  ...prev,
  spec: {
    cronExpressions: ['0 18 * * FRI'], // Every Friday at 6:00 PM
  },
}));
```

Python

```
handle = client.get_schedule_handle("my-first-schedule")
async def update_func(schedule):
    schedule.spec = ScheduleSpec(
        intervals=[ScheduleIntervalSpec(every=timedelta(hours=1))]
    )
    return schedule
await handle.update(update_func)
```

Go

```
scheduleHandle, _ := temporalClient.ScheduleClient().GetHandle(ctx, "my-
first-schedule")
err = scheduleHandle.Update(ctx, client.ScheduleUpdateOptions{
    Do: func(input client.ScheduleUpdateInput) (*client.ScheduleUpdate,
error) {
        schedule := input.Description.Schedule
        schedule.Spec.CronExpressions = []string{"0 18 * * FRI"}
        return &client.ScheduleUpdate{Schedule: &schedule}, nil
    },
})
```

From Cron to Temporal Schedules

Converting your existing cron jobs to Temporal Schedules is a straightforward process. You can wrap your existing scripts in a Temporal Activity and then create a simple workflow that executes that activity. Once you have your workflow, you can schedule it using the methods described above.

This approach allows you to leverage the power of Temporal Schedules without having to rewrite your existing business logic. You can gradually migrate your cron jobs to Temporal and take advantage of the enhanced reliability, observability, and control that Temporal provides.