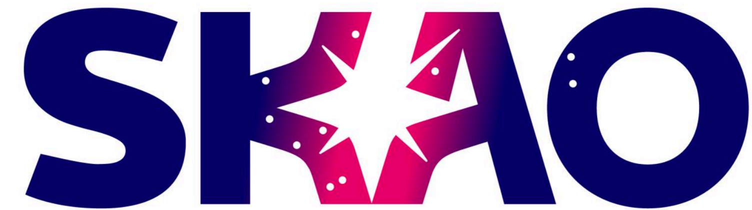
Scheduling Tango callbacks

Lessons learnt from debugging event-driven Tango at scale



Thomas Ives and Thomas Juerges

2026-06-09

Outline

Event-driven Tango at SKAO

Pitfall 1: EventConsumer thread is shared

Pitfall 2: Managing ad-hoc queues is bug prone

Pitfall 3: The EventConsumer scheduler is unfair

Pitfall 4: Subscriptions are serialised and block

Pitfall 5: DeviceProxy destructor unsubscribes

Pitfall 6: DeviceProxy construction is not “stateless”

How could Tango do better?

Event-driven Tango at SKAO

- SKAO is a large organisation making heavy use of the Tango event system
- Many tests come down to “did we receive this event in time”
  - **Sentiment**: “Tango’s event system doesn’t work”
  - **Honest answer**: “You’re holding it wrong”
- Explaining what “holding it right” looks like is complicated
- Our new `CallbackScheduler` is an attempt to make something easier to hold
  - Based on lessons learnt from debugging Tango event system problems

Pitfall 1: EventConsumer thread is shared

Pitfall 1: EventConsumer thread is shared

- There is a single event consumer thread for the whole device server process
- As a Tango device developer, you don't know who else is using the event consumer
- **Solution**: Tango callbacks should just put the event in a queue and a separate thread should process the event

Pitfall 2: Managing ad-hoc queues is bug prone

Pitfall 2: Managing ad-hoc queues is bug prone

Spot the bug(s):

```
class MyDevice(tango.server.Device):
    def init_device(self) -> None:
        ...
        self._event_queue = Queue(maxsize=256)

    def my_tango_callback(self, event: tango.EventData) -> None:
        try:
            self._event_queue.put(event)
        except Full:
            _ = self._event_queue.get()
            self._event_queue.put(event)
```

- **Solution**: Provide `CallbackScheduler` class that manages these queues

Pitfall 3: The EventConsumer scheduler is unfair

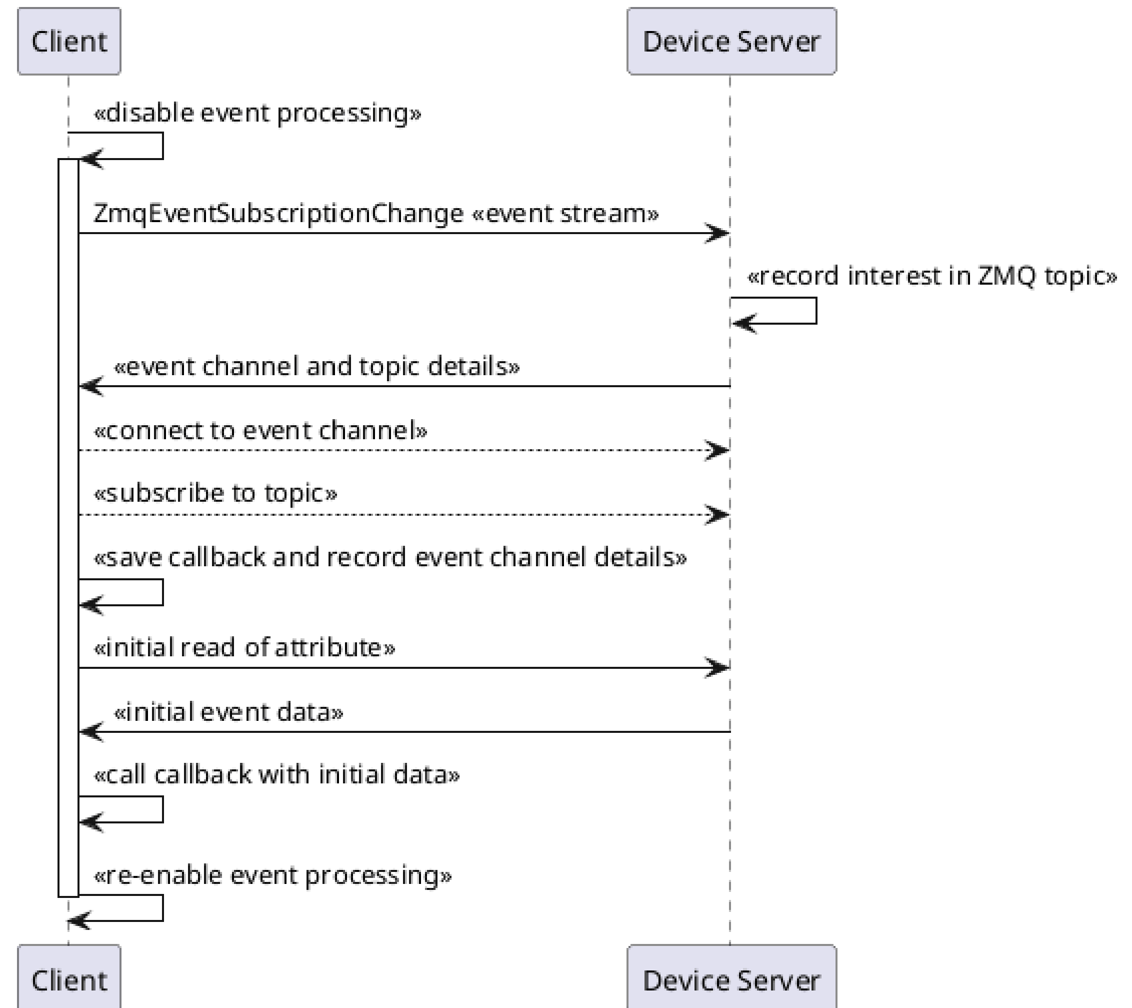
Pitfall 3: The EventConsumer scheduler is unfair

- Tango event consumer uses a single queue managed by ZMQ
- Events are processed in a first come, first serve order
- Leads to starvation if a callback cannot keep up with the rate of events
- **Solution**: `CallbackScheduler` uses multiple queues (1 per event stream by default) and schedules them with a priority system

Pitfall 4: Subscriptions are serialised and block

Pitfall 4: Subscriptions are serialised and block

- Only a single thread can subscribe at a time
- While it is subscribing no events can be processed
- The subscription requires 2 network calls!



Pitfall 4: Subscriptions are serialised and block

- **Solution**:
  - `CallbackScheduler` allows pre-subscription and will discard events while no callbacks are registered.
  - Event data and callbacks are stored together in the queue allowing updating the callbacks quickly without synchronisation with processing threads.

Pitfall 5: DeviceProxy destructor unsubscribes

Pitfall 5: DeviceProxy destructor unsubscribes

- cppTango event subscriptions are tided to the life cycle of the `DeviceProxy` that was used to subscribe
- When using PyTango, if this `DeviceProxy` ends up in a reference cycle we can end up unsubscribing at unpredictable times
- **Solution**: `CallbackScheduler` uses its own internal `DeviceProxy` objects whose lifetime is managed by the scheduler. It also provides a `shutdown()` method to cleanup all resources.

Pitfall 6: DeviceProxy construction is not “stateless”

Pitfall 6: DeviceProxy construction is not “stateless”

- At SKAO we do not keep our Tango databases around
- In CI we rebuild the database with each device server pod responsible for adding itself to the database
- Creating a `DeviceProxy` requires the device to be defined in the database, even though we don’t actually connect to the device itself
- During startup we often want to do stateless subscriptions to other devices, however, we cannot do this to a device not yet in the database!
- **Solution**: `CallbackScheduler` supports subscribing to a device TRL and will retry creating the `DeviceProxy` if it is not found in the database yet.

How could Tango do better?

How could Tango do better?

- Multiple batched subscriptions at once
  - Multiple subscriptions to the same server are batched into a single `ZmqEventSubscriptionChange` call
  - Clients can subscribe to multiple servers at the same time
  - `AysncRead` subscription is nice but it is not enough
- `"DB_DeviceNotDefined"` to happen at the same time as `"DB_DeviceNotExported"` (e.g. on `ping()` not `DeviceProxy` creation)
- Expose the `DeviceProxy::unsubscribe_all_events` method from PyTango so we don't have to rely on object cleanup

Any comments/questions?

- https://developer.skao.int/projects/ska-tango-base/en/latest/how-to/callback-scheduler.html