Tango event system - Agenda

- History
- Basic principles
- On the wire
- ZMQ usage
- Establishing connections
- Threading
- Miscellaneous event related info.

History

- First implementation in Tango 4 (03/2004)
  - Using notifd
    - OMG notification service implementation
- Decision to re-write event system in 2010
  - Get rid of additional process,….
- Second implementation in Tango 8 (07/2012)
  - Using ZeroMQ
  - Multicast event propagation added in v 8.1
    - Used somewhere ?

User point of view

- 8 event types
  - Change, Periodic, Archive
    - Fired by Tango lib or user code
  - Attribute configuration change, Device interface change
    - Fired by Tango lib
  - Data ready, Pipe, User
    - Fired by user code
- Data transferred within the event depends on event type

User point of view

- Firing event from a DS code
  - DeviceImpl::push_xxx_event()
- Client API to receive events:
  - DeviceProxy::subscribe_event()
  - DeviceProxy::unsubscribe_event()
  - Client writes a class inheriting from Tango::CallBack and re-defines the CallBack::push_event() method

User point of view

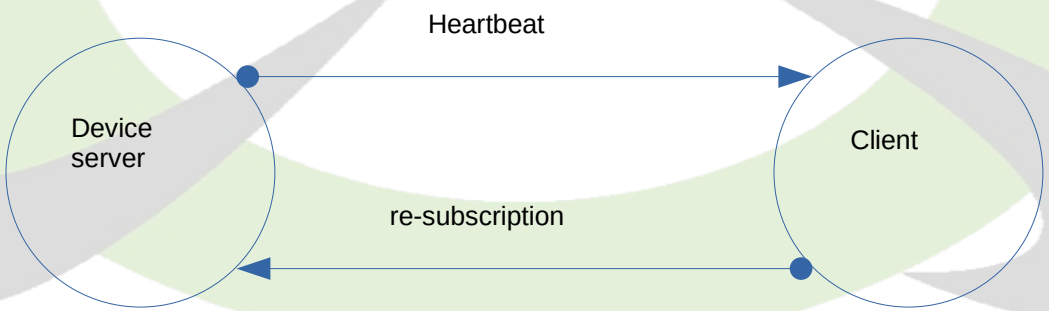
- Push / Pull model
  - Default is push model
    - When the event arrives, Tango pushes it to the user
  - Pull model implemented using event buffer in client process.
    - When the event arrives, it is pushed into the buffer
    - The buffer is managed as a round robin buffer
    - The client reads the buffer when he wants
  - DeviceProxy::subscribe_event() allows user to select pull model
  - DeviceProxy::get_events() to retrieve events from the buffer

Basic principle: Automatic change event firing

- For most of the attribute data types, user has to define what is a change
  - Ex for a PS device:
    - Event sent when generated current change
      - Do you want to be informed when current changes from 9.0 to 9.01 A (change is 0.01 A) or from 9 to 10 A (change is 1 A) ?
- It is required to regularly read the attribute to detect the change
  - It’s the Tango polling thread which fires the event
    - When fired by library, minimum event period is the polling period

Basic principle: Behind the scene - Heartbeats

- A client needs to know if the DS which should send the event is still alive
  - Heartbeat from DS to client
    - Every 9 sec (dedicated polling thread)
- A DS needs to know if there are still some clients interested in events
  - Regular re-subscription from client to DS
    - Every 200 sec (KeepAliveThread class)



Basic principles

- Behind the scene:
  - Heartbeat from DS to client(s)
    - Yet another event type
  - Re-subscription
    - Through a DS admin device command
- Late joiner problem
  - User callback called during the DeviceProxy::subscribe_event() call
    - Data from a synchronous attribute read done by library

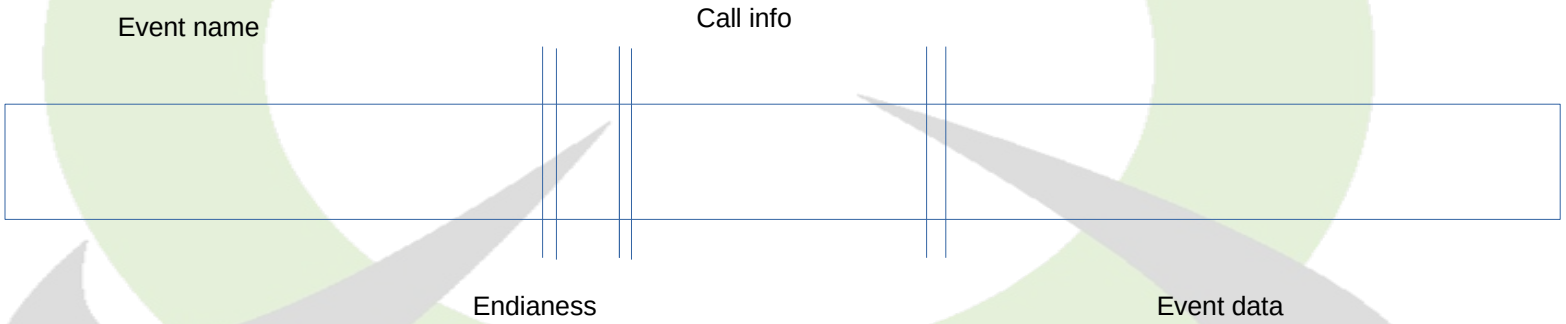
ZeroMQ

- A layer to build distributed system
  - Between threads within a process
  - Between processes within a host
  - Between hosts
- Supports several communication patterns
  - Request/Reply, Publish/Subscribe,...
- Only takes care of transporting data
  - No encoding provided

https://zeromq.org

On the wire

- Events are transmitted using ZMQ multipart message with
  - 3 parts for heartbeat event
  - 4 parts for other event types



On the wire

- Event name
  - Fully qualified event name
    - tango://acs.esrf.fr:10000/srmag/ps-qf8/c01-b/state.change
    - tango://acs.esrf.fr:10000/dserver/hsmaccess/c01.heartbeat
- Endianess
  - A single byte (0 Big endian – 1 Little endian)
- Call info and event data
  - Structure with data of different type
    - Encoded using the CORBA Common Data Representation (CDR)
    - Re-use the omniORB generated marshalling / un-marshalling methods
    - Re-use data type defined in the Tango IDL file (structures / sequences / ...)

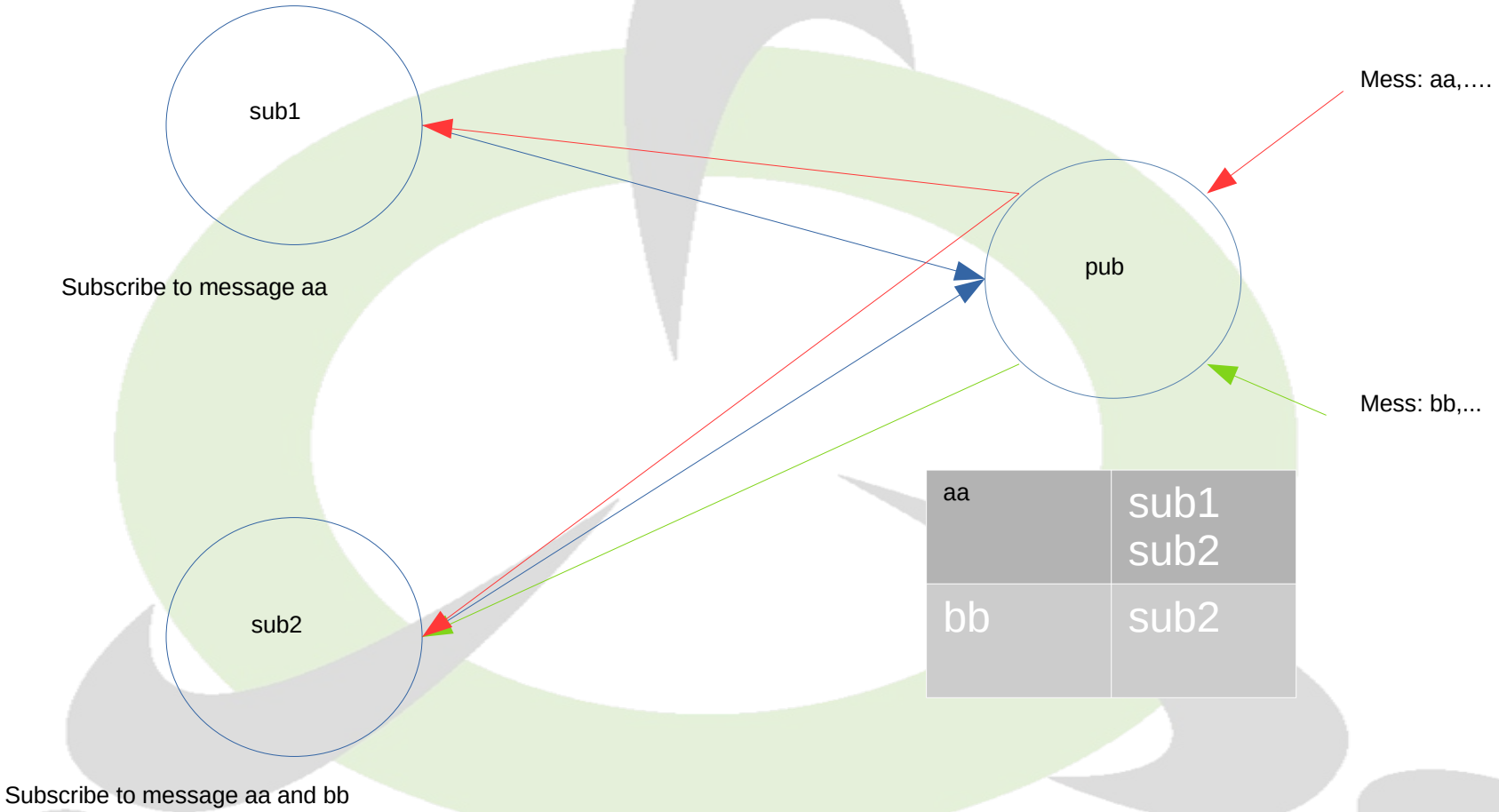
On the wire

- Call info
  - Structure with
    - Version, ctr, method_name, oid, call_is_except
      - method_name and oid are today unused
- Event data
  - Depend on the event type
  - Use data type defined in Tango idl file
    - AttributeValue_X for change, periodic, archive event
    - AttributeConfig_X for attribute configuration change event
    - AttDataReady for data ready event
    - DevIntrChange for device interface change event
    - DevPipeData for pipe event

ZMQ usage

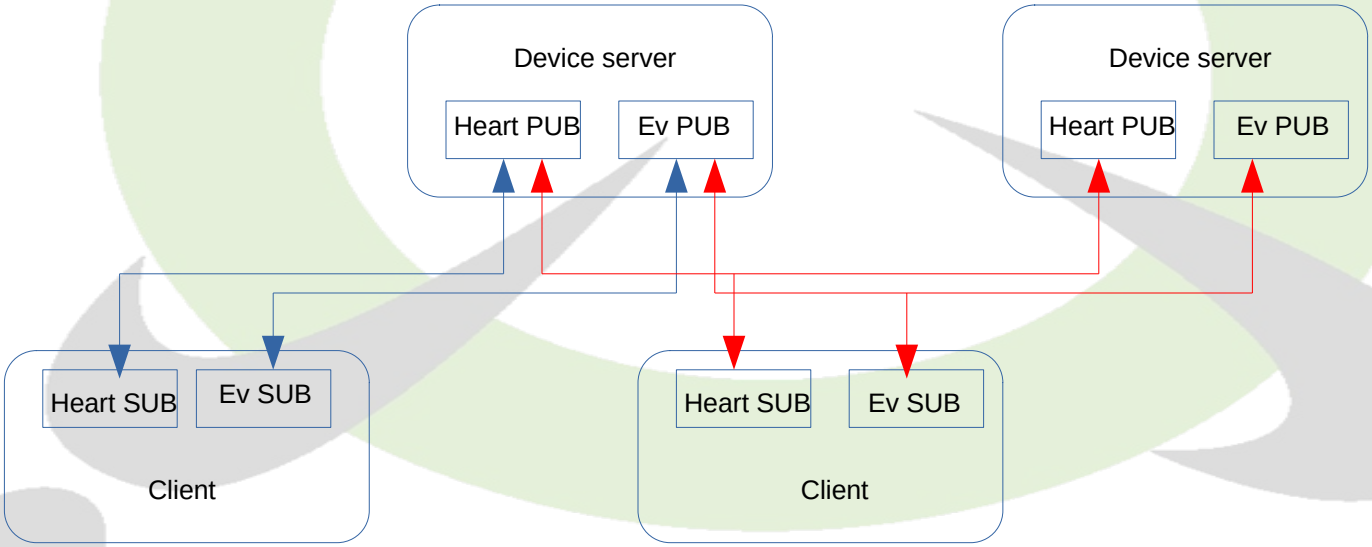
- Use publish / subscribe communication pattern
  - The publisher is the device server
  - The subscriber is the application
- We use ZMQ subscription forwarding
  - Subscription string is the first part of the multipart message: The event name
    - tango://acs.esrf.fr:10000/my/funny/dev/state.change
  - String compare in pub: warning many traps related to how tango host is defined in clients and DS
    - Lower / upper case
    - Use of network alias name for TANGO_HOST in DS but not in appli(s)
    - Use of localhost in TANGO_HOST in DS but not in appli(s)
    - CS with several TANGO_HOST (ESRF machine CS with acs:10000 and acs:11000)

ZMQ subscription



ZMQ usage

- Two kind of data to be transmitted
  - Heartbeat using couple PUB / SUB socket
  - Events using another couple PUB / SUB socket
- A DS has two PUB sockets
- A client has two SUB sockets



Establishing event connection

- The DS “bind” ZMQ PUB sockets to ephemeral port number
- Client has to retrieve the DS host IP and the two port numbers
  - A new command on the DS admin device: *ZmqEventSubscriptionChange* which returns those information (plus other things)
    - Try it with “info” as argin

Event connection and DS startup

- Many objects related to event stored in class ZmqEventSupplier
- The DS startup
  - Create instance of the ZmqEventSupplier class
  - Start the polling thread dedicated to heartbeat
    - Does nothing, just wait
- ZmqEventSupplier ctor:
  - Create heartbeat PUB socket
  - Bind it to ephemeral port
  - Detect host endianess
  - Init miscellaneous event related data

Event connection (DS point of view)

- On reception of the ZmqEventSubscriptionChange command
  - If not already done
    - Ask heartbeat polling thread to fire heartbeat event
    - Create event PUB socket
    - Bind it to ephemeral port

Event connection (client point of view)

- Many objects related to event stored in class ZmqEventConsumer
- ZMQEventConsumer ctor
  - Create the two SUB ZMQ sockets (for heartbeat and events)
- Several maps to store event related data
  - device_channel_map
    - Link device name – adm_device name
  - channel_map
    - Entry for connected DS (heartbeat)
  - event_callback_map
    - Entry for subscribed event (with callback ptr)

Event connection (client point of view)

- What has to be done during DeviceProxy::subscribe_event()
  - If not already done
    - Create instance of ZmqEventConsumer class
  - Retrieve DS admin device name
  - Build a DeviceProxy to that admin device
  - Execute command ZmqEventSubscriptionChange on admin device
  - If not already done
    - Connect the heartbeat SUB socket to the DS heartbeat PUB socket
  - ZMQ subscription with heartbeat event name
  - If not already done
    - Connect the event SUB socket to the DS event PUB socket
  - ZMQ subscription with event name
  - Read the attribute
  - Execute user callback

Event system threads

- User callback execution requires a thread
  - Thread code in ZmqEventConsumer class
    - The ZMQ thread
  - A ZMQ socket is not thread safe
    - Protect it with a mutex and lock it before accessing the socket (DS code)
    - Use a single thread to access it (Client code)
  - DeviceProxy::subscribe_event() executed by user thread != ZMQ thread
    - A need for a mechanism between user thread and ZMQ thread
      - Use ZMQ REQ / REP socket
    - The ZMQ thread also has a REP socket
    - During event subscription / un-subscription, user thread(s) create ZMQ REQ socket to ask ZMQ thread to execute control commands

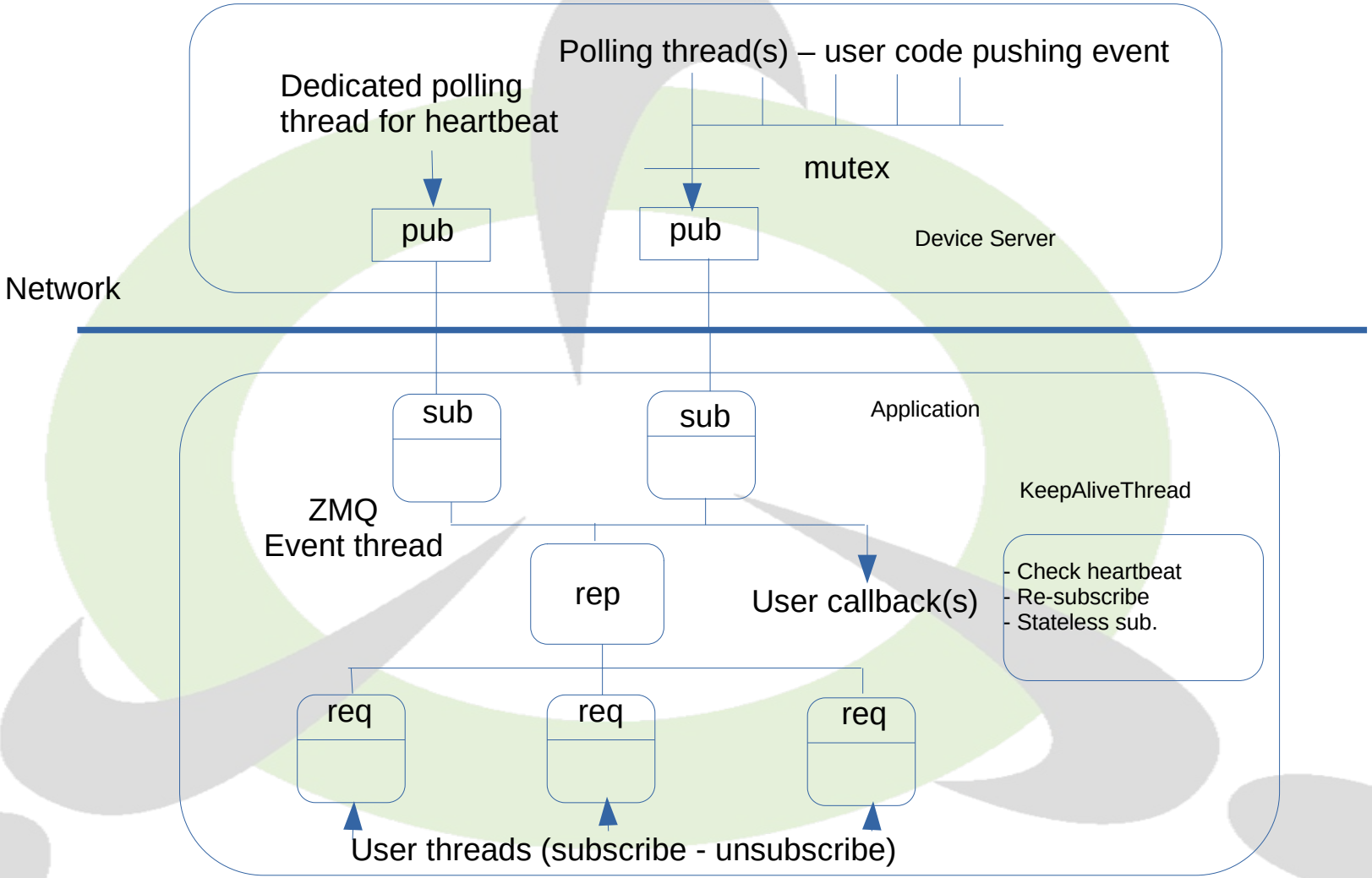
Event system threads: ZMQ thread

- What it does
  - Create the two SUB ZMQ sockets
  - Create the REP ZMQ socket
  - While true
    - Wait for data on those 3 sockets (zmq::poll()) ← blocking call
    - If data on heartbeat SUB
      - Retrieve in maps data for that DS
      - Update last heartbeat date
    - Else if data on event SUB
      - Retrieve in maps data for that event
      - Call user callback(s)
    - Else if data on control REQ socket
      - Execute control command

Event system threads

- The ZMQ thread control commands
  - Connect heartbeat
  - Disconnect heartbeat
  - Connect event
  - Disconnect event
  - Connect mcast event
  - Delay event / Release event

Event system (Tango threads)



Compatibility

- Tango IDL v4 (Tango 8) and v5 (Tango 9)
  - IDL AttributeValue struct changed
- Use case: DS using Tango 9, client 1 using Tango 9 and client 2 using Tango 8
  - DS: Tango 9 IDL 5 → AttributeValue_5
  - Client 1: Tango 9 → Knows AttributeValue_5
  - Client 2: Tango 8 → Does not understand AttributeValue_5 !!!
  - DS has to send the event twice with
    - AttributeValue_4 for old clients still using Tango 8
    - AttributeValue_5 for new clients using Tango 9

Compatibility

- IDL release added in the event name transferred on network
  - tango://acs.esrf.fr:10000/my/beautiful/dev/state.idl5_change
    - For Change / Archive / Periodic events due to IDL AttributeValue_5
    - For attribute configuration change events due to IDL AttributeConfig_5
- A new input parameter to the ZmqEventSubscriptionChange command
  - IDL version used by client

Re-connection

- In case the DS is stopped
  - No more heartbeat event
  - Every 10 sec, the client KeepAliveThread detects missing heartbeat and
    - Try to re-connect to DS admin device
    - If successful, re-subscribe the client
      - User callback called with fresh event data
    - Else
      - User callback called with error in DevErrorList argument

Re-connection

- In case of different event name between client and DS
  - Every 10 sec, the KeepAliveThread
    - Call user callback(s) with error
    - Try to re-subscribe including the synchronous read
      - Call user callback(s) with data read from synchronous call
  - The event system works strangely
    - Your callbacks are not executed when they should
      - Called twice every 10 sec
        - With error
        - With sync. read data

ZMQ HighWaterMark (HWM)

- ZMQ has its own buffer
  - If buffer gets fulled, events are silently discarded !
  - Buffer size defined by the ZMQ HWM
    - Tunable at different levels
      - Default value = 1000
      - Control system properties
      - Tango API: Tango::Util or Tango::ApiUtil classes call
      - Using Env Variables
    - Some doc about this tuning in
      - Developer’s guide / Advanced / Reference part / Control system specific

ZMQ HighWaterMark (HWM)

- An event counter in the ZMQ Call info part of the message transferred on the network
  - Client checks that between two consecutive events
    - new_ctr = previous_ctr + 1
  - In case new_ctr != previous_ctr + 1
    - CallBack(s) fired 2 times:
      - 1: With DevErrorList argument with one DevError struct
        - Desc field = “Missed some events! Zmq queue has reached HWM?”
        - Reason field = “API_MissedEvents”
      - 2: With event data just received

On the wire: ZMQ call info

- method_name
  - To specify which method has to be executed on remote object
    - We have only one feature: event system
- object identifier (oid)
  - To specify which object has to be called
    - We have only one entity (the event system)

DS admin device

- 2 events related commands
  - ZmqEventSubscriptionChange
    - For event connection
  - EventConfirmSubscription
    - For KeepAliveThread for heartbeat from client to DS
    - 3 input arguments per event
      - Device name
      - Attribute name
      - Event name

Some file names

- Client part
  - client/zmqeventconsumer.cpp
    - The Zmq thread (ZmqEventConsumer class code)
  - client/eventkeepalive.cpp
    - The KeepAliveThread code
  - client/event.cpp
    - Event subscription / un-subscription
  - client/event.h
    - Event related structures used in user API
  - client/eventconsumer.h
    - Event related structure / class definition
      - The maps used to store event related data

Some file names

- Server part
  - server/eventcmds.cpp
    - The admin device event related commands code
  - server/zmqeventsupplier.cpp and server/zmqeventsupplier.h
    - The ZmqEventSupplier class code
    - Code to push event(s)
  - server/eventsupplier.cpp
    - Code to decide when events must be pushed