TANGO
Connecting things together

Tango Communication Patterns

Tango Communication Patterns:

- Synchronous
- Asynchronous
- Grouped
- Publish-Subscribe (Events)

Generalities

- Synchronous, asynchronous and grouped communications are available for:
  - Command execution
  - Attribute(s) reading
  - Attribute(s) writing
- Publish-Subscribe (Events) communication pattern is only available for:
  - Attribute reading
    - Events are configured and send for individual attributes

Generalities

- Error management
  - All the exceptions thrown by the API are PyTango.DevFailed (Tango::DevFailed) exceptions
  - One catch (except) block is enough

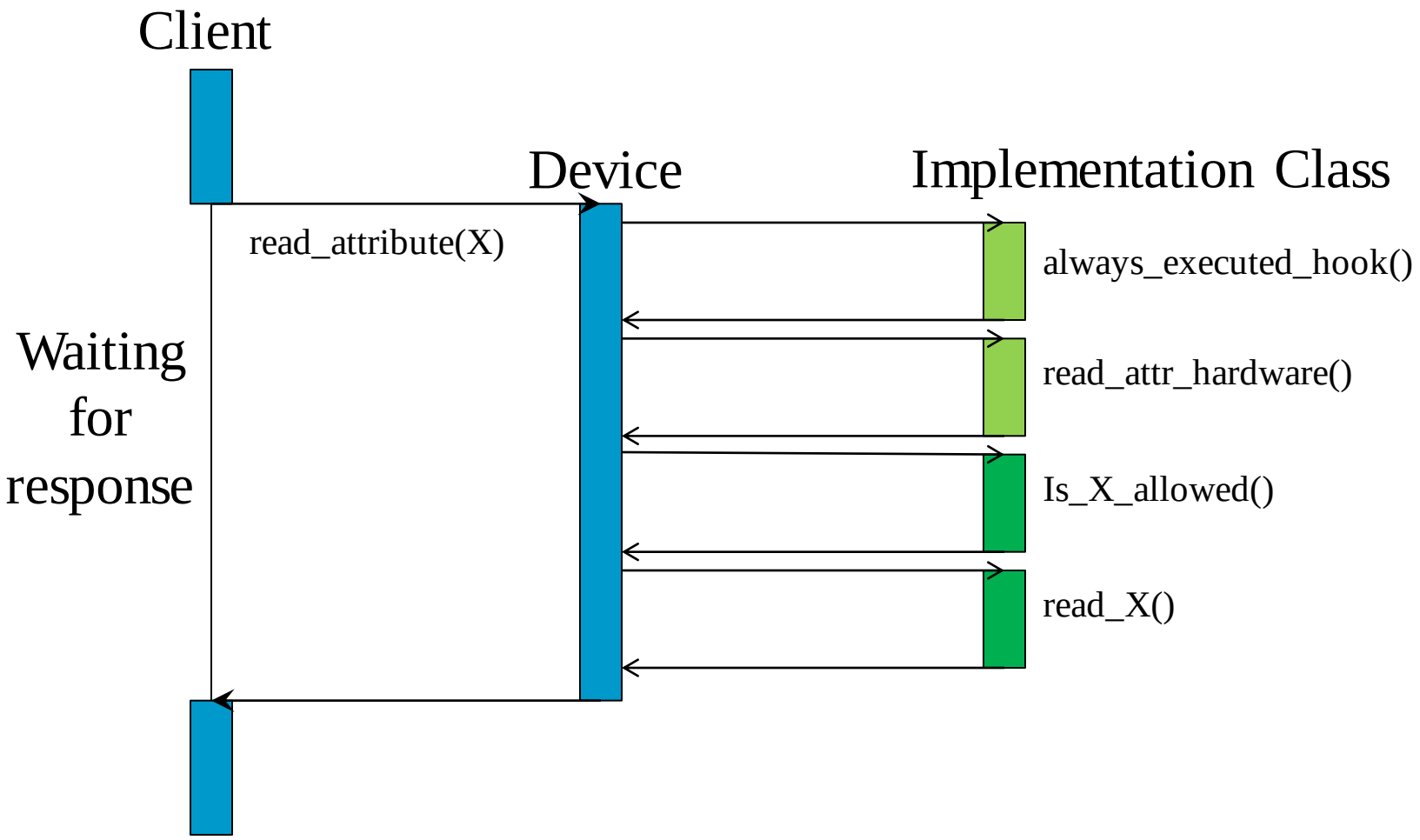
Synchronous Calls

- The DeviceProxy *read_attribute()* method reads an attribute
- The received data from attribute reading is stored in an DeviceAttribute object

```
Tango::DeviceProxy device("tango/test/2");
Tango::DeviceAttribute da;
string att_name ("Pressure");

double press;
da = device->read_attribute(att_name);
da >> press;
cout << "pressure value " <<  press;
```

Synchronous – 1 Attribute



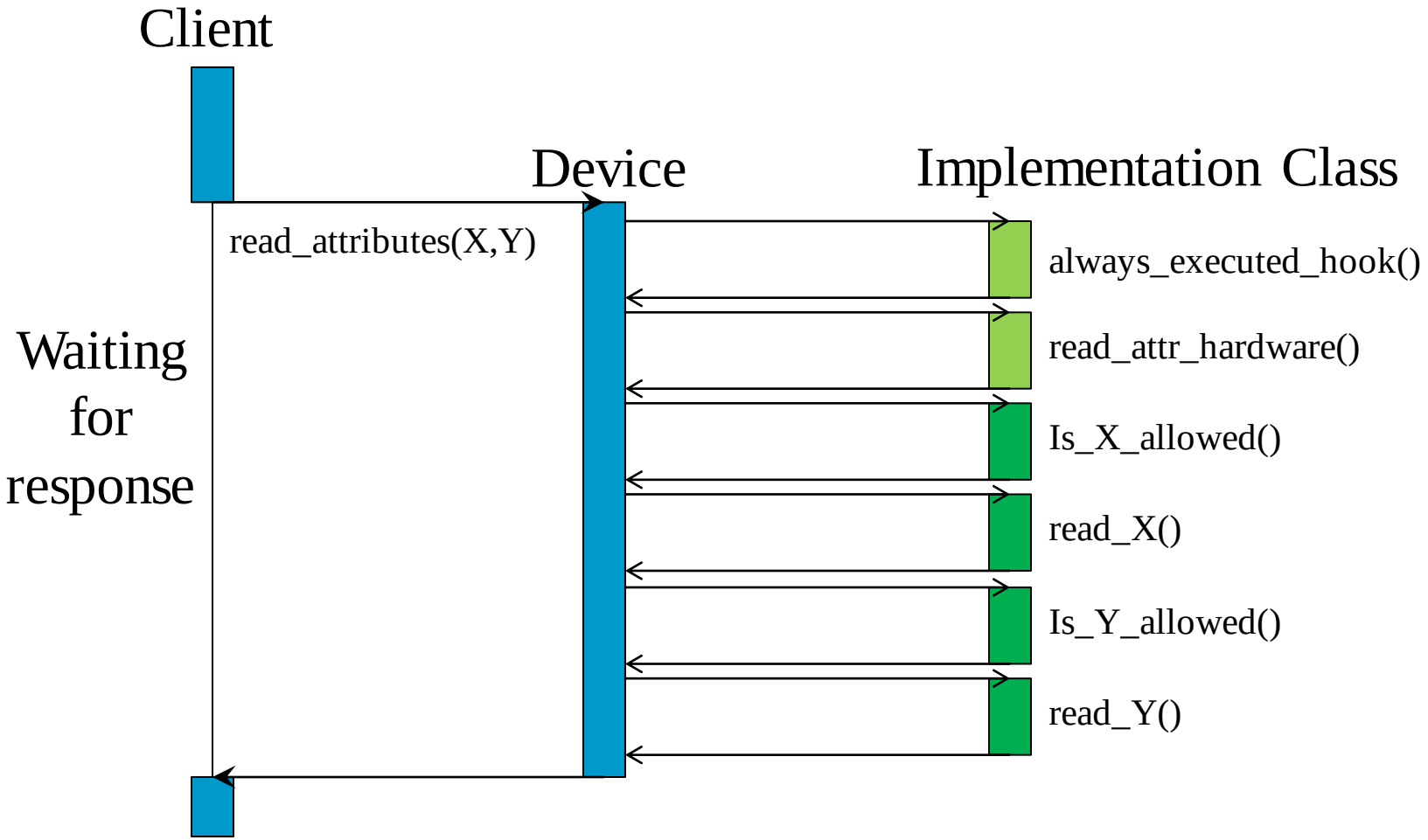
Synchronous Calls

- The DeviceProxy *read_attributes()* method reads a set of attributes
- The received data from attributes reading is stored in a vector of DeviceAttribute objects

```
Tango::DeviceProxy device("tango/test/1");
vector<DeviceAttribute> *devattr;
vector<string> attr_names;

attr_names.push_back("attribute_1");
attr_names.push_back("attribute_2");
devattr = device->read_attributes(attr_names);
short short_attr_1;
long long_attr_2;
(*devattr)[0] >> short_attr_1;
(*devattr)[1] >> long_attr_2;
cout << "long attribute value " << long_attr_2;
delete devattr
```

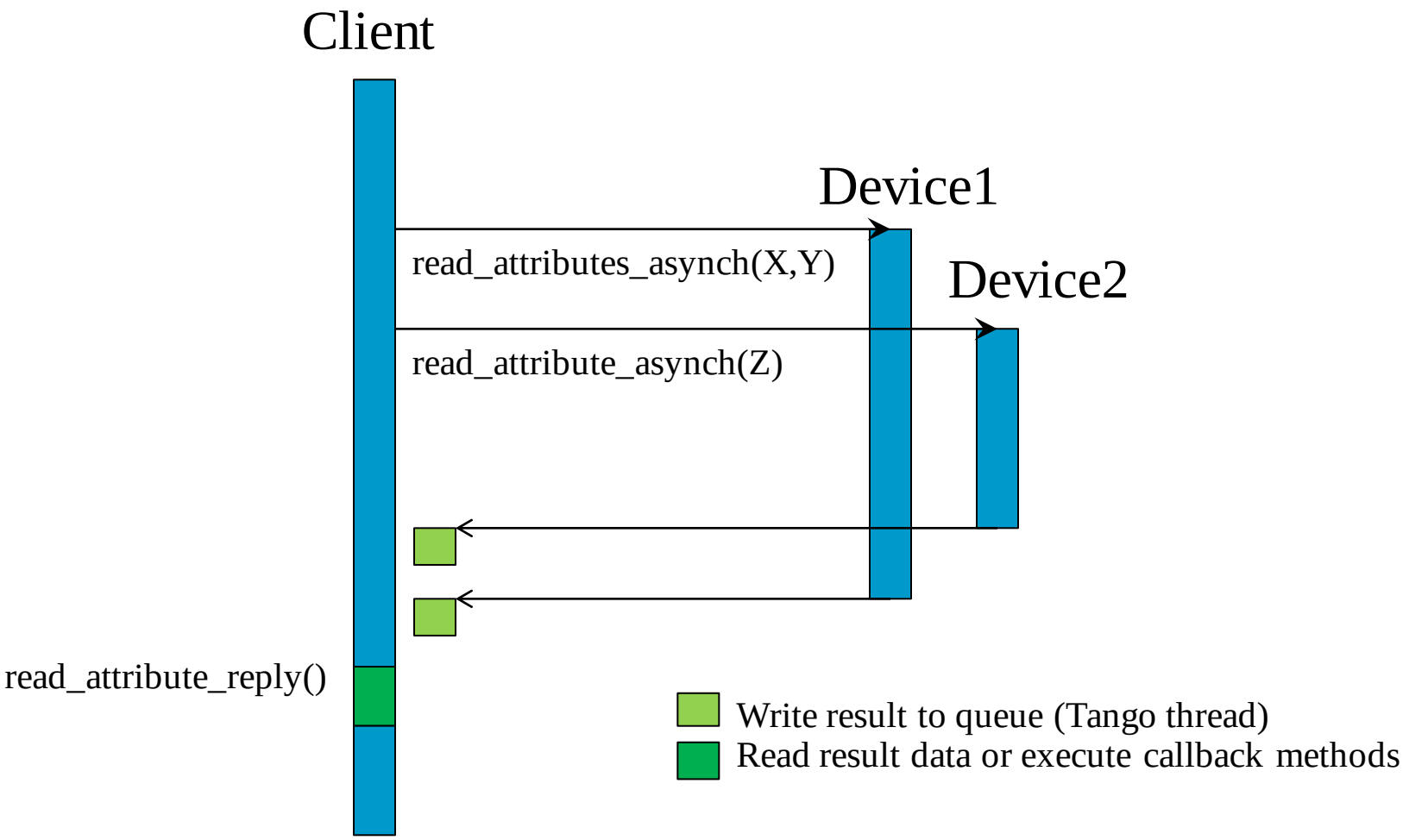
Synchronous – 2 Attributes



Asynchronous call

- The client sends a request to a device and does not block waiting for the answer.
- Tango supports two ways for clients to get requested answers
  - **Polling** (pull model)
    - The client decides when it checks for requested answers
      - With a non blocking call
      - With a blocking call
  - **Callback** (pull or push model)
    - The requested reply triggers a callback method
      - When the client requested it with a synchronization method (Pull model)
      - As soon as the reply arrives in a dedicated thread (Push model)

Asynchronous – Pull Model



Asynchronous – Pull Model

Send Requests

```
Tango::DeviceProxy device1(“tango/test/1”);
Tango::DeviceProxy device2(“tango/test/2”);

long call_id1
long call_id2

vector<string> attr_names;
attr_names.push_back("attribute_1");
attr_names.push_back("attribute_2");
call_id1 = device1->read_attributes_asynch(attr_names);

call_id2 = device2->read_attribute_asynch(“Pressure”);
…
…
```

Read Replies

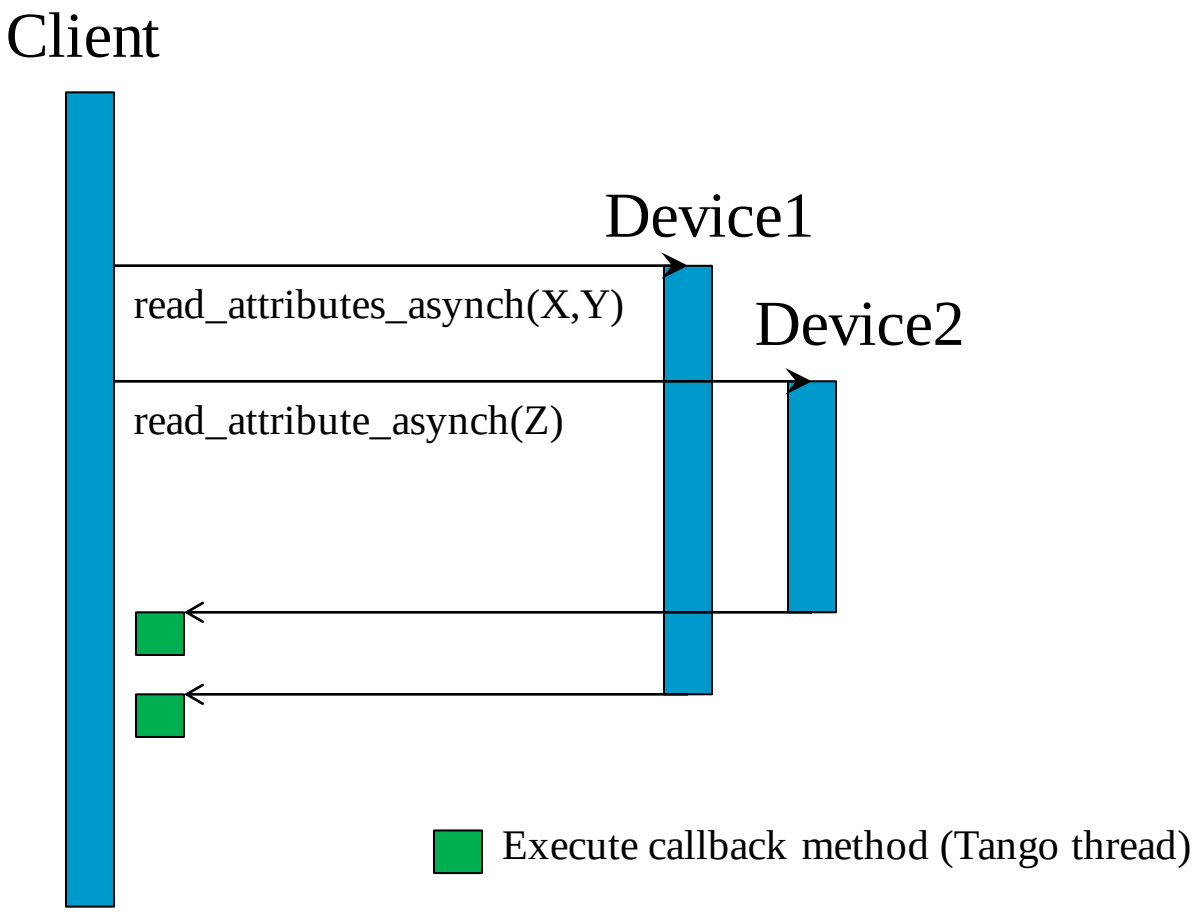
```
…
vector<DeviceAttribute> *devattr;
devattr = device1->read_attributes_reply(call_id1);

short short_attr_1;
long long_attr_2;
(*devattr)[0] >> short_attr_1;
(*devattr)[1] >> long_attr_2;
cout << “long attribute value " << long_attr_2;
delete devattr

DeviceAttribute da;
da = device2->read_attribute_reply(call_id2);

double press;
da >> press;
cout << “pressure value " << press;
```

Asynchronous – Push Model



Asynchronous – Push Model

```
class MyCallBack: Tango::CallBack{
  void attr_read(Tango::AttrReadEvent *);
  …
};

void MyCallBack::attr_read(Tango::AttrReadEvent  *att){
   cout << “CB from " << att->device->dev_name() << endl;
   if (!att->err){
     for (unsigned int i=0; i < att->attr_names.size(); i++) {
       cout << "Attribute read = " << att->attr_names[i] << endl;

       switch ( (* att->argout)[i]->get_type() ){
         case Tango::DEV_SHORT:
              short short_attr;
              (* att->argout)[i] >> short_attr;
              cout << “short value " << short_attr << endl;
              break;
         case Tango::DEV_LONG:
              long long_attr;
              (* att->argout)[i] >> long_attr;
              cout << “long value " << long_attr << endl;
              break;
```

```
         case Tango::DEV_DOUBLE:
              double press_attr;
              (* att->argout)[i] >>press_attr;
              cout << “pressure value " << press_attr << endl;
              break;
         }
       }
     }
   else{
       cout << "Error send to callback" << endl;
       Tango::Except::print_error_stack(att->errors);
   }
 }
…
Tango::DeviceProxy device1(“tango/test/1”);
Tango::DeviceProxy device2(“tango/test/2”);

MyCallBack  cb;

vector<string> attr_names;
attr_names.push_back("attribute_1");
attr_names.push_back("attribute_2");
device1->read_attributes_asynch(attr_names, cb);
device2->read_attribute_asynch(“Pressure”, cb);
```

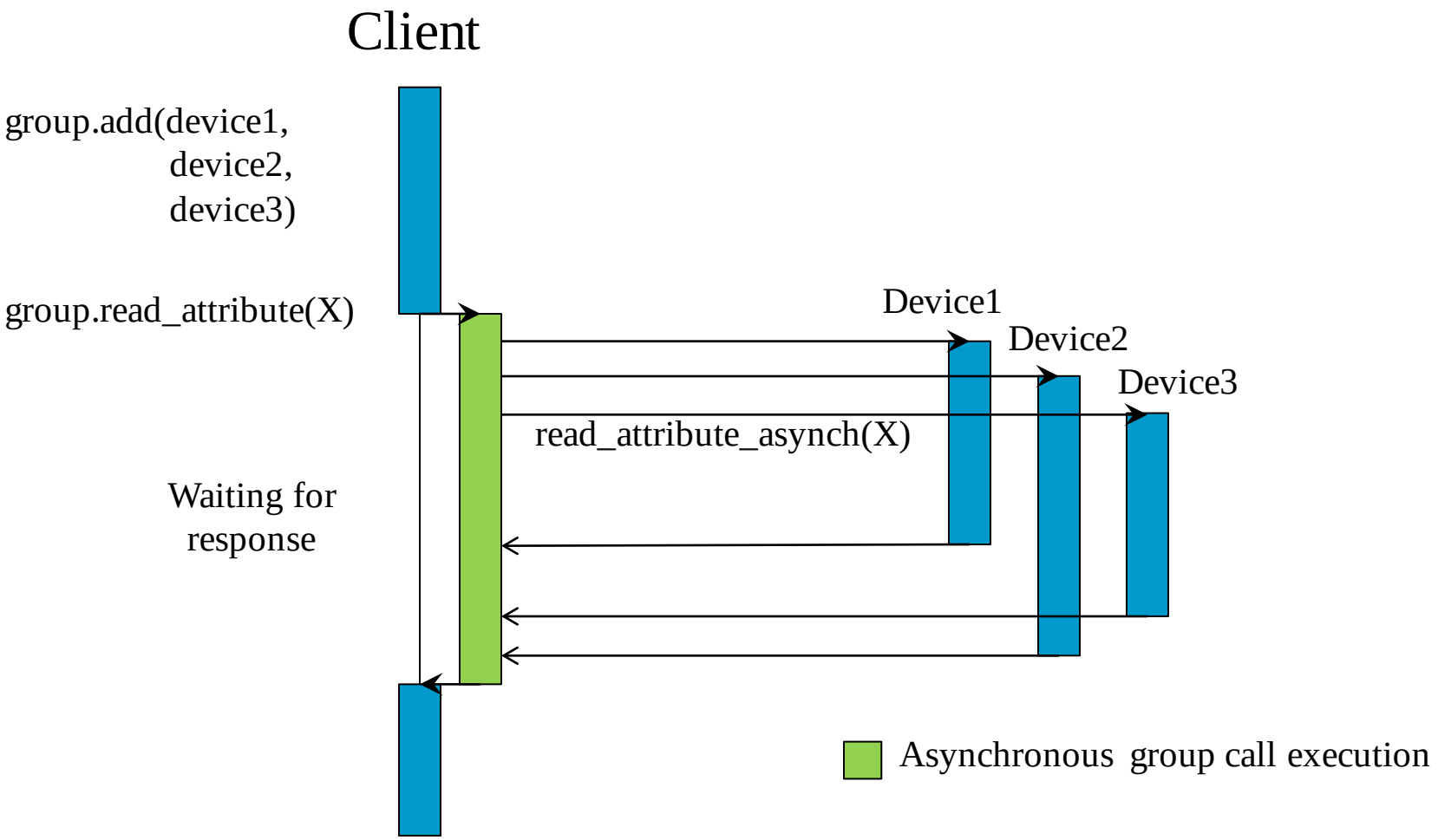
Group Call

- Provides a single point of control for a Group of devices
- **Group calls under the hood are executed asynchronously**!
- It's a hierarchical object (You can have a group in a group) with a forward or not forward feature

Group Call

- Using groups, you can
  - Execute one command
    - Without argument
    - With the same input argument to all group members
    - With different input arguments for group members
  - Read one or several attributes
  - Write one attribute
    - With same input value for all group members
    - With different input values for group members

Grouped Synchronous



Grouped Synchronous

Send Requests

```
vector<string> steererNames;
…
vector<string> attributes;
attributes.push_back("State");
attributes.push_back("Current");

vector<Tango::DevState> states (steererNames.size());
vector<double>currents(steererNames.size());
….
steerers = new Tango::Group ("Steerers");
steerers->add (steererNames);
…
Tango::GroupReply::enable_exception(true);
Tango::GroupAttrReplyList  steerer_data;
steerer_data = steerers->read_attributes (attributes);
```

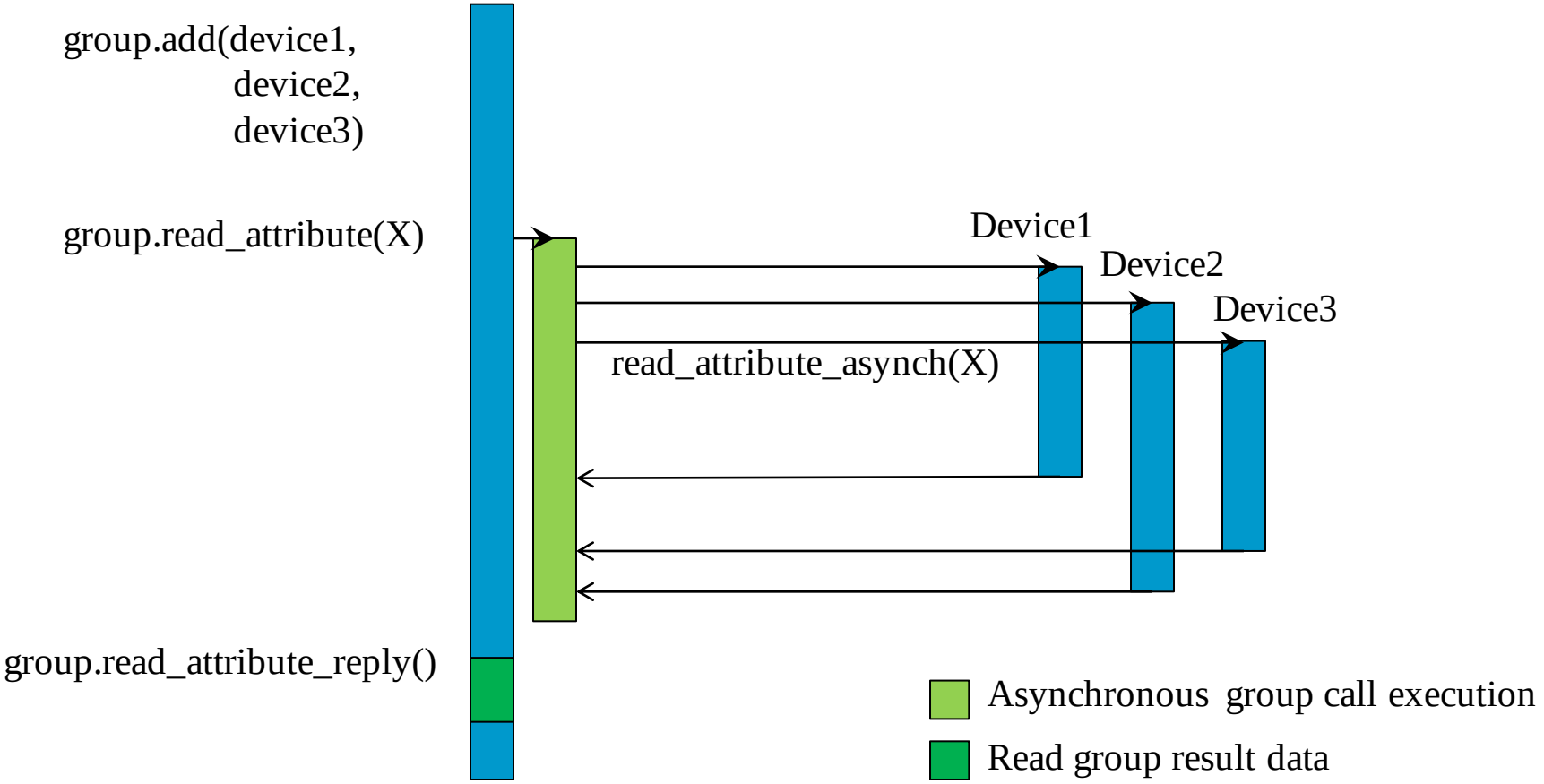
Read Replies

```
try{
  long index = 0;
  for (unsigned long i=0; i < steerer_data.size();
        i = i + attributes.size(), index++) {
     steerer_data[i]     >> states[index];
     steerer_data[i+1] >> currents[index];
  }
}
catch (const Tango::DevFailed &e){
  cout << "Group call failed" << endl;
  Except::print_exception(e);
}
steerer_data.reset();

cout <<  "States of all steerers:  " << states << endl;
cout<< "Currents of all steerers:  " <<currents<< endl;
```

Grouped Asynchronous

Client



Grouped Asynchronous

Send Requests

```
vector<string> steererNames;
…
vector<string> attributes;
attributes.push_back("State");
attributes.push_back("Current");

vector<Tango::DevState> states (steererNames.size());
vector<double>currents(steererNames.size());
….
steerers = new Tango::Group ("Steerers");
steerers->add (steererNames);
…
Tango::GroupReply::enable_exception(true);
long call_id;
call_id= steerers->read_attributes _asynch(attributes);
…
…
```

Read Replies

```
Tango::GroupAttrReplyList steerer_data;
steerer_data = steerers->read_attributes _reply(call_id, 2000);

try{
  long index = 0;
  for (unsigned long i=0; i < steerer_data.size();
        i = i + attributes.size(), index++) {
     steerer_data[i]    >> states[index];
     steerer_data[i+1] >> currents[index];
  }
}
catch (const Tango::DevFailed &e){
  cout << "Group call failed" << endl;
  Except::print_exception(e);
}
steerer_data.reset();

cout << "States of all steerers: " << states << endl;
cout<< "Currents of all steerers: " <<currents<< endl;
```

Event Generalities

- Basic events do not require any changes in the device server code
  - Configuration only
- But, event sending can also implemented in the device server code
- 6 types of events
  - Periodic, Change, Archive
  - Attribute configuration change, Data ready
  - User defined

Event Generalities

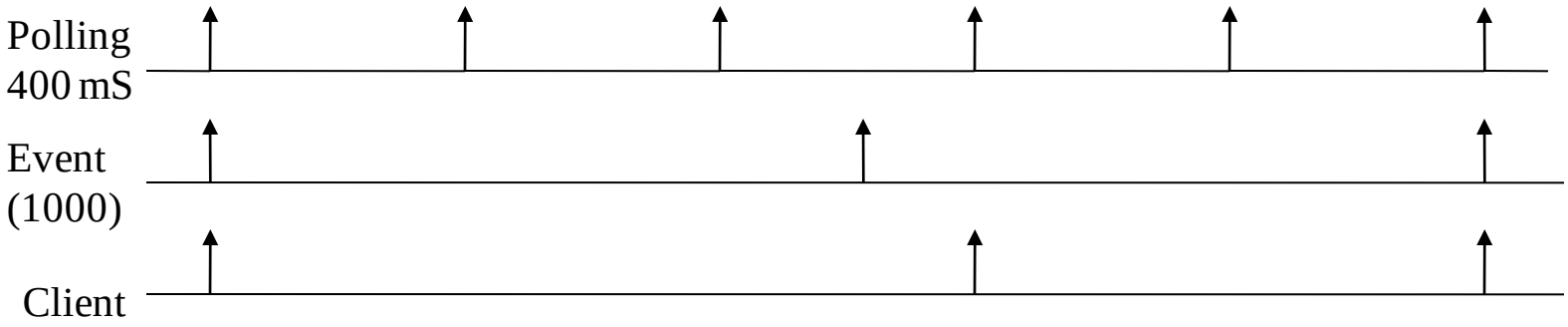
- Periodic event
  - Event pushed:
    - At event subscription
    - On a periodic basis
- Change event
  - Event pushed when
    - a change is detected in attribute data
    - a change is detected in attribute size (spectrum/image)
    - At event subscription
    - An exception was received by the polling thread
    - the attribute quality factor changes
    - When the exception disappears

Event Generalities

- Archive event
  - A mix of periodic and change events
- Attribute configuration change
  - Event pushed when:
    - At event subscription
    - The attribute configuration is modified with set_attribute_config()
- User defined event / Data ready event
  - Event pushed when the user decides it
  - Need to be implemented in the device server code

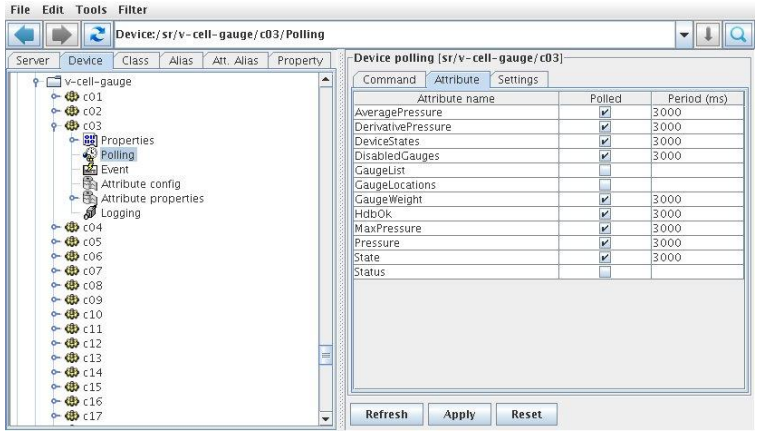
Event Configuration

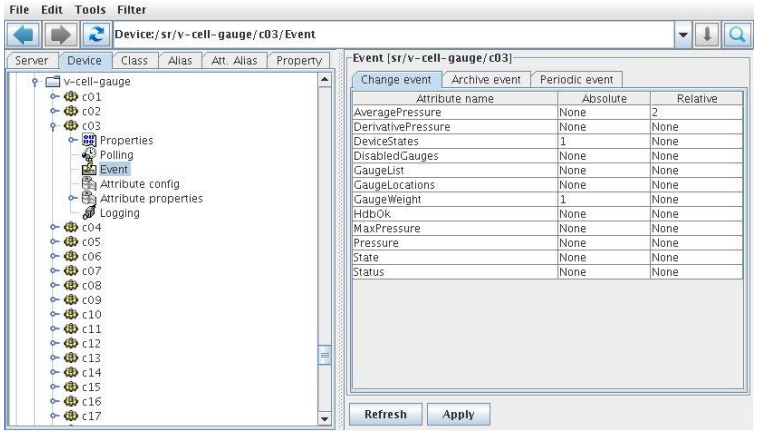
- Periodic event configuration
  - event_period (in mS).
    - Default is 1000 mS
    - Cannot be faster than the polling period
  - Polling period != event period
  - The event system does not change the attribute polling period if already defined



Event Configuration

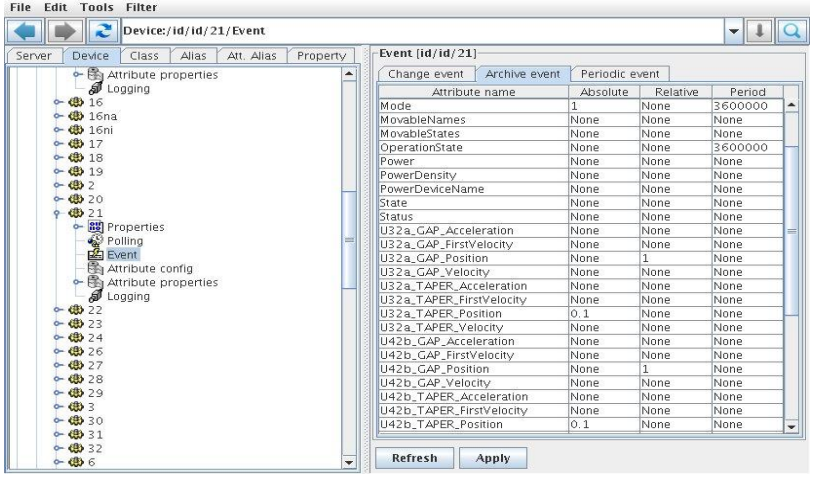
- Change event configuration
  - Checked at the polling period
  - rel_change and abs_change
    - Up to 2 values (positive and negative delta)
    - If both are set, relative change is checked first
    - If none is set -> **no change event!**





Event Configuration

- Archive event configuration
  - Checked at the polling period
  - event_period (in mS).
    - Default is 0 mS -> **no periodic archive event**!
  - rel_change and abs_change
    - Up to 2 values (positive, negative delta)
    - If both are set, relative change is checked first
    - If none is set -> **no archive event on change!**

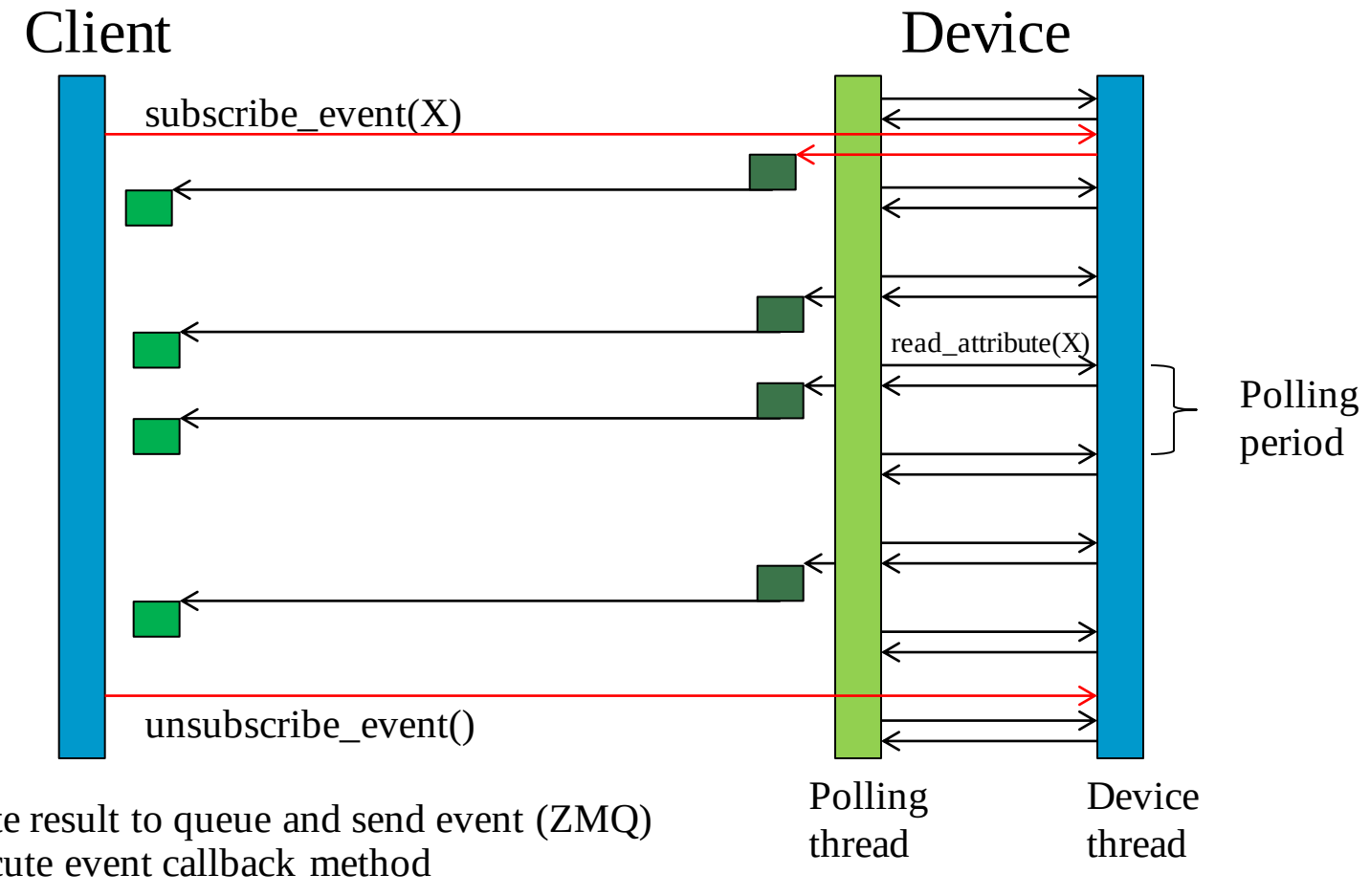


Event Client

- Client needs to subscribe for events
- The subscription can be stateless (in case the device server process does not run)
- Two ways for event reception
  - Push model
    - Callbacks as for asynchronous calls
    - Triggered on event reception
  - Pull model
    - Arriving events are queued
    - Queue size defined at subscription time
    - The client decides when it checks for received events

Event – Push Model

(Attributes only!)



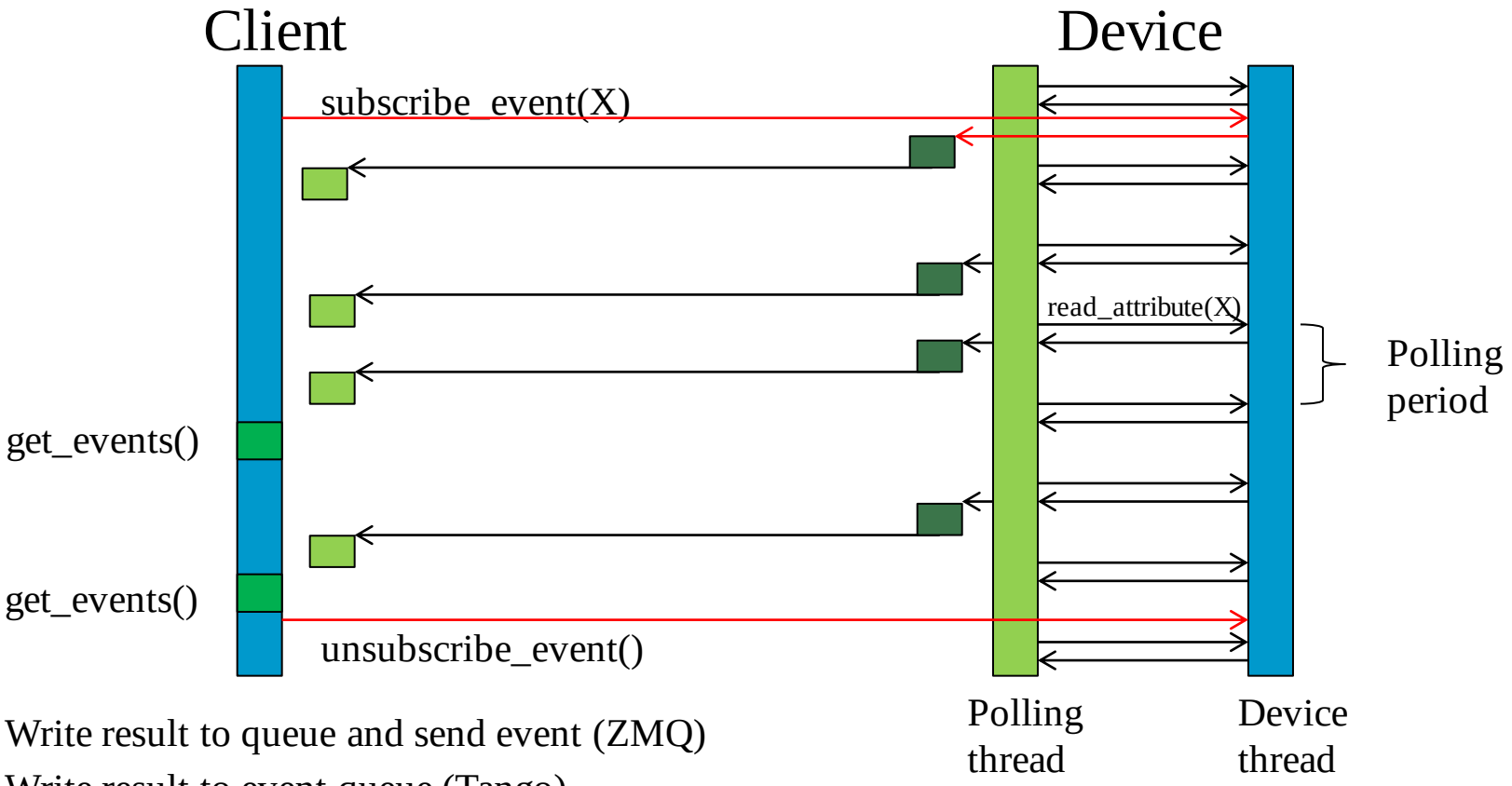
Event – Push Model

```
class EventCallBack: Tango::CallBack{
  void push_event(Tango::EventData *);
  …
};
void EventCallBack::push_event(Tango::EventData* event_data){
   cout << “CB from " << event_data->attr_name << endl;
   if (!event_data->err){
     if ( event_data->attr_value->get_quality() ==
          Tango::ATTR_INVALID ){
       cout << " Attribute data is invalid!" << endl;
     }
     else{
       switch ( event_data->attr_value->get_type() ){
         case Tango::DEV_SHORT:
            short short_attr;
            *(event_data->attr_value) >> short_attr;
             cout << “short value " << short_attr << endl;
            break;
        case Tango::DEV_LONG:
             long long_attr;
             *(event_data->attr_value) >> long_attr;
             cout << “long value " << long_attr << endl;
            break;
         case Tango::DEV_DOUBLE:
           double press_attr;
           *(event_data->attr_value)  >> press_attr;
            cout << “pressure value " << press_attr << endl;
            break;
         }
       }
     }
   else{
     cout << "Error send to callback" << endl;
     Tango::Except::print_error_stack(event_data->errors);
   }
 }
…
Tango::DeviceProxy device1(“tango/test/1”);
Tango::DeviceProxy device2(“tango/test/2”);
MyCallBack  cb;

Vector<int> eve_id(3);
eve_id[0] = device1->subscribe_event("attribute_1  ",
                Tango::CHANGE_EVENT, &cb);
eve_id[1] = device1->subscribe_event("attribute_2  ",
                Tango::CHANGE_EVENT, &cb);
eve_id[2] = device2->subscribe_event(“Pressure  ",
                Tango::CHANGE_EVENT, &cb);
…
device1->unsubscribe_event(eve_id[0]);
device1->unsubscribe_event(eve_id[1]);
device2->unsubscribe_event(eve_id[2]);
```

Event – Pull Model

(Attributes only!)



Event – Pull Model

```
Tango::DeviceProxy device1("tango/test/1");
Tango::DeviceProxy device2("tango/test/2");

Vector<int> eve_id(3);
eve_id[0] = device1->subscribe_event("attribute_1  ",
                Tango::CHANGE_EVENT, 1);
eve_id[1] = device1->subscribe_event("attribute_2  ",
                Tango::CHANGE_EVENT, 100);
eve_id[2] = device2->subscribe_event("Pressure  ",
                Tango::CHANGE_EVENT,  Tango::ALL_EVENTS);
…
…
While (xxx){
  if ( ! device1->is_event_queue_empty(eve_id[0]) ){
    Tango::EventDataList event_list;
    device1>get_events(eve_id[0], event_list);

   // event queue length = 1!
   Tango::EventData *event_data   = event_list[0];

    if (!event_data->err){
      if ( event_data->attr_value->get_quality() ==
          Tango::ATTR_INVALID ){
        cout << " Attribute data is invalid!" << endl;
      }
    else{
       short short_attr;
       *(event_data->attr_value) >> short_attr;
       cout << "short value " << short_attr << endl;
    }
  }

  if ( ! device1->is_event_queue_empty(eve_id[1]) ){
    Tango::EventDataList event_list;
    device1>get_events(eve_id[1], event_list);
    treat_long_events(event_list);
  }

  if ( ! device2->is_event_queue_empty(eve_id[2]) ){
    Tango::EventDataList event_list;
    device2>get_events(eve_id[2], event_list);
    treat_pressure_events(event_list);
  }
}
…
…
device1->unsubscribe_event(eve_id[0]);
device1->unsubscribe_event(eve_id[1]);
device2->unsubscribe_event(eve_id[2]);
```

Events Pushed from the Code

- Possible for change, archive, data ready and user events
- To push events manually from the code a set of data type dependent methods can be used:

  DeviceImpl.push_xxx_event (attr_name, ...)

  xxx = {change, archive, data_ready, 'user'}
- It is possible to push events from the code and from the polling thread at the same time

Events Pushed from the Code

- To allow a client to subscribe to events of non polled attributes the server has to declare that events are pushed from the code

  DeviceImpl.set_change_event(attr_name, implemented, detect = true)

  DeviceImpl.set_archive_event(attr_name, implemented, detect = true)

  - *implemented*=true inidcates that events are pushed manually from the code
  - *detect*=true triggers the verification of the same event properties as for events send by the polling thread.
  - *detect*=false, no value checking is done on the pushed value!
- Attribute configuration possible with Pogo

Event – Push Model

(Attributes only!)