Adoption process for existing ROS code

Zurich, March 21, 2018

Overview

One of our main goals at Klepsydra is to make the adoption process as light and smooth as possible. The API was built with this in mind and tools are provided to help with the adoption.

The procedure consists of two parts: application and wiring.

- At the application level, the goal is to have no longer dependencies with ROS. That means that a) ROS messages are replaced with Klepsydra objects (which are nothing but POCOs[1]) and b) that ROS API for publishing and subscription is replaced with Klepsydra API.
- We recommend to follow the composition root pattern[2] in order to do the wiring of the application. That means that only a few (or ideally one) classes are in charge of configuring the Klepsydra - ROS integration.

By using this procedure correctly, migration to Klepsydra of an existing ROS application should be very straightforward and with very little change in the code base.

[1] Plain Old C++ Objects. No dependencies to any libraries in them.

[2] A Composition Root is a (preferably) unique location in an application where modules are composed together.

Adoption Procedure

Application Level

As explained before, the application should be completely ROS agnostic. Furthermore, the application should also remain transparent to the underlying Klepsydra integration implementation, for instance, the application shouldn't know anything about Eventloop, Disruptor or blocking queues. In order to achieve that, classes needing to publish or subscribe to messages should have pointers to the following interfaces in the Klepsydra API:

```
class Example {
...
private:
   kpsr::Publisher<std::string> * _publisher;
   kpsr::Subscriber<std::string> * _subscriber;
}
```

Code generation tools

Klepsydra comes with a code generation tool that takes care of the conversion between Klepsydra Objects and ROS messages. This is done by proving a YAML file with a specific syntax (KIDL[3]) for each ROS message.

This tool generate always the Klepsydra Object class and the transformation class (or mapper) between this and the ROS message. Optionally, the MSG file can also be generated. We recommend to generate the MSG file for custom application messages and not generate it for builtin ROS messages.

The aim of this tool is that code currently using ROS messages can start using Klepsydra Objects with as little as possible intrusion.

Example 1. Custom Application ROS message KIDL file.

The following KIDL file is a typical application message structure. A very important property is the middlware/ exists. This property determines whether the ROS MSG file should be created or not as explained before.

```
className: Temperature
middlewares:
    - type: ROS
      namespace: tutorial
      exists: false
      className: Temperature
enums:
    - name: Units
      exists: false
      values:
         - CELSIUS = 0
         - FAHRENHEIT
         - KELVIN
fields:
    - name   : value
      type   : float32
    - name   : units
      type   : Units
```

[3] Klepsydra-IDL

In this case, the MSG file will be generated and it looks like that:

```
# Copyright (C) Klepsydra Robotics - All Rights Reserved
# Unauthorized copying of this file, via any medium is strictly prohibited
# Proprietary and confidential
# This code has been automatically generated, manual modification might be
inadvertently overridden.

float32 value
int32 units
```

The Klepsydra Object class looks like this:

```
/* Copyright (C) Klepsydra Robotics - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * Proprietary and confidential
 */

// This code has been automatically generated, manual modification might be
inadvertently overridden.

#ifndef TEMPERATURE_H_
#define TEMPERATURE_H_

// Include section.

// Klepsydra generated event class.
class Temperature {
public:
   // Autogenerated Enum.
   enum Units {
      CELSIUS = 0,
      FAHRENHEIT,
      KELVIN
   };

   // Default constructor.
   Temperature() {}

   // Main constructor.
   Temperature(
      float value,
      Units units)
      : value(value)
      , units(units)
   {}

   // Clone constructor. Needed by klepsydra core APIs.
   Temperature(const Temperature & that)
      : value(that.value)
      , units(that.units)
   {}

   // Clone method. Needed by klepsydra core APIs.
   void clone(const Temperature * that) {
      this->value = that->value;
      this->units = that->units;
   }

   // List of fields.
   float value;
   Units units;
};
#endif
```

And the mapper or transformation class looks like this:

```
/* Copyright (C) Klepsydra Robotics - All Rights Reserved
 * Unauthorized copying of this file, via any medium is strictly prohibited
 * Proprietary and confidential
 */

// This code has been automatically generated, manual modification might be
inadvertently overridden.

#ifndef TEMPERATURE_ROS_MAPPER_H
#define TEMPERATURE_ROS_MAPPER_H

#include "tutorial/Temperature.h"

#include "mapper.h"
#include "temperature.h"

namespace kpsr {
template<>

class Mapper<Temperature, tutorial::Temperature>
{
public:

   void fromMiddleware(const tutorial::Temperature & message, Temperature & event) {
      event.value = message.value;
      event.units = (Temperature::Units) message.units;
   }

   void toMiddleware(const Temperature & event, tutorial::Temperature & message) {
      message.value = event.value;
      message.units = event.units;
   }
};
}
#endif
```

Example 2. Builtin ROS message KIDL.

This is an example on how to apply the same approach to existing messages in ROS[4]. In this case the MSG is not generated.

More features

Many features are included in the KIDL code generation, including:

- Namespace
- Related class references
- ENUMS
- Vector of primitive types and custom classes
- Available for ROS, DDS and ZMQ
- Extension API for custom code generation.

---

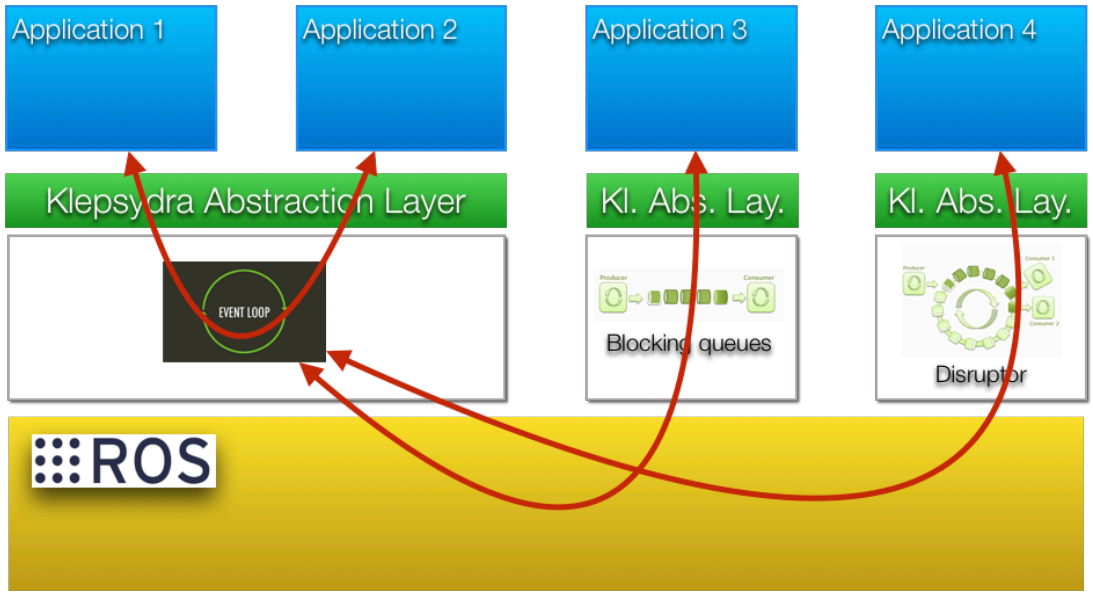
[4] This particular example is already included in Klepsydra as well as many other ROS builtin messages like geometry, MAVROS, sensors, etc.

```
className: LaseScanEvent
namespace: kpsr::sensors
middlewares:
    - type: ROS
      namespace: kpsr_ros_sensors
      exists: true
      className: LaserScan
      classNamespace: sensor_msgs
fields:
    - name   : angle_min
      type   : float32
    - name   : angle_max
      type   : float32
    - name   : angle_increment
      type   : float32
    - name   : time_increment
      type   : float32
    - name   : scan_time
      type   : float32
    - name   : range_min
      type   : float32
    - name   : range_max
      type   : float32
    - name   : ranges
      type   : float32[]
    - name   : intensities
      type   : float32[]
```

Application Wiring API

The wiring of the application consists of three parts:

- Klepsydra Communication setup (Blocking queue, Eventloop or Disruptor)
- Publishing to ROS
- Subscription from ROS



Klepsydra Communication Setup

All messages arriving from ROS will go first through one of the three communication implementations in Klepsydra (Blocking queue, event loop or disruptor). Also, communication between two classes can be done using the same mechanism without ROS or any other middleware.

The API to use these communication mechanism is very simple and straightforward, while at the same time is highly performing and with several advance tuning possibilities.

Event Loop Setup

The following code snippet shows how to setup an event loop in the system and how to use it.

```
1. kpsr::disruptor::EventLoopMiddlewareProvider<4> provider(nullptr);
2. provider.start();

3. kpsr::Subscriber<ExampleEvent> * subscriber =
provider.getSubscriber<ExampleEvent>("ExampleEvent");

4. kpsr::Publisher<ExampleEvent> * publisher =
provider.getPublisher<ExampleEvent>("ExampleEvent", 0, nullptr, nullptr);
```

This example shows the power of the Klepsydra API: in literally 4 lines, the event loop is setup including the publisher and subscripted pair. This is how to each line is interpreted:

- Line 1: Eventoop provider setup. The event loop needs two parameters: the size (passed as template param) and the container (used for monitoring)
- Line 2: The event loop has to be started. This starts the thread where all logic in the customer application will happen. This is the main feature of an event loop: one threads handles all messages.
- Line 3. Create a subscriber. Pair publisher/subscriber can be created limitless for the same event loop provider. The only requirement is that the name is different for each pair. In the case of the subscription, two parameters are required: the event class template and the name of the pair (or event name)
- Line 4: The publication requires a bit more configuration that the rest. Firstly, we need the same as the subscriber: class and event name. But then we have three more params. These are for performance use:
    - Pool size (bigger than 0 mean that an object pool will be created and they will be used when copying the event before publishing)
    - An initialiser function (std::function(void(Event &)>) that will be used by the object pool to initialise the constructed events. This for example can be used to reserve memory for vectors or images.
    - A cloner function. Used to copy the content of the publishing event instead of the default copy function.

Disruptor Setup

The disruptor pattern is quite a complex pattern. In the basic case of Klepsydra, we have limited its behaviour to have one producer and several subscriber, where each subscriber runs in its own thread:

```
1. kpsr::disruptor::DisruptorMiddlewareProvider<DisruptorTestEvent, 4>
   provider(nullptr, “test");
2. provider.getSubscriber()->start();
3. provider.getSubscriber()->registerListener("cacheListener",
eventListener.cacheListenerFunction);
5. provider.getPublisher()->publish(event);
```

The basic configuration of the disruptor consist mainly in providing the size, the event class, the container and a name. There are several other configurations, oriented to performance, but they are not included in this document.

Blocking queue setup

The setup of the blocking queues is very similar to the one of the event loop, with the main difference that one queue can only handle one publisher/subscriber pair:

```
1. kpsr::memstg::SafeQueueMiddlewareProvider<SQTestEvent> provider(nullptr, "event", 4,
0, nullptr, nullptr, false);
2. provider.start();
3. provider.getSubscriber()->registerListener("cacheListener",
eventListener.cacheListenerFunction);
4. provider.getPublisher()->publish(event1);
```

In this case the configuration is concentrated in the provider including container, object pool, etc. There is one extra parameter at the end which is a boolean that indicates whether the queue should block if it is full (false) or discard messages when full (true).

ROS Publisher

The following snippet shows the configuration of the ROS Publisher.

```
0. #include “primitive_type_ros_mapper.h"
1. ros::Publisher rosPublisher = nodeHandle.advertise<std_msgs::String>("test", 1);
2. kpsr::rosstg::ToRosMiddlewareProvider toRosProvider(nullptr);
3. kpsr::Publisher<std::string> * kpsrPublisher =
toRosProvider.getToMiddlewareChannel<std::string, std_msgs::String>(“kpsr_test", 1,
nullptr, rosPublisher);
```

The Klepsydra Publisher to ros is literally a two liner:

- Line 0: include the mapper. This line is necessary for the internals of Klepsydra to perform the conversion to ROS. Here the generated mappers or the Klepsydra builtin mappers can be provided.
- Line 2. This line setups the general ROS provider. It is necessary only one as it can be used for all publishers to ROS. It expects one parameter with is the container for performance monitoring.
- Line 3. It setups the actual to ROS publisher. This one expects a similar set of parameters as the event loop publisher. Namely the event class, event name, pool size and initialiser function. Plus the ROS native publisher itself.

ROS Subscriber

The following snippet shows the configuration of the ROS Subscriber.

```
0. #include “primitive_type_ros_mapper.h"
1. kpsr::disruptor::EventLoopMiddlewareProvider<8> provider(nullptr);
2. provider.start();
3. kpsr::Publisher<std::string> * publisher =
provider.getPublisher<std::string>("test", 0, nullptr, nullptr);
4. kpsr::rosstg::FromRosMiddlewareProvider fromRosProvider(nodeHandle);
5. fromRosProvider.registerToTopic<std::string,
std_msgs::String>("kpsr_ros_core_test_topic", 1, publisher);
```

Now, the setup of the subscription is bit more complex than the one for the publication. This is because the nature of Klepsydra abstraction layer: all message received from ROS goes through one of the three comm implementation of Klepsydra.

The general flow of this process is:

I. Klepsydra creates a special callback to ROS

II. When a message arrives, this callback transform the message to a Klepsydra Object and publish it using a Klepsydra publisher

III. Now the object is available in Klepsydra and any subscriber associated to the publisher can listen to it.

Now the code itself explained:

- Line 0: Mapper include as before.
- Lines 1-3: Event loop publisher creator. The same can be done using the other two implementations
- Line 4. Create a from ROS provider. Common to all events in the application.
- Line 5: Create the from ROS / Klepsydra subscriber for the specific event. The params are: ROS topic name, ROS queue size and Klepsydra Publisher.

Conclusion

This document shows the adoption procedure to use Klepsydra in an existing ROS application. At Klepsydra we tried to make the process as simple and straightforward as possible, keeping the performance as a top priority.

The main intrusion point that is required, is the use of the composition root pattern to configure the application wiring. The examples in the corresponding section showed why is this important and yet how simple it is to do it with the provided API.

We left a lot of extra elements in the API related to both performance and testing. This document is intended as introductory only and not as full comprehensive API description.