cROS - An Introductory Tutorial

Alberto Pretto
IT+Robotics S.r.l.
Sapienza University of Rome
cROS in a nutshell

→ cROS is a **lightweight ANSI-C implementation of the roscpp** library that provides topic subscribing and publishing, and service providing;
→ **Depends only on libC** and standard BSD sockets
→ cROS supports for each node multiple topics, subscriptions, etc... but is **single-thread**;
→ **Online message managements**: cROS dynamically manages standard and custom built messages in run-time and it does not require any offline message generator tool.
Motivations

Enabling a large number of embedded, low cost, custom built or legacy device to be ROS-compliant:

- No operating system required: just an ANSI C compiler and a standard TCP/IP stack implementation;
- The single thread implementation allows to easily meet the **real time requirements** often required in an industrial scenario.

Easy, quick and non-invasive ROS interface for M$ Windows applications.
First use case (2014)

Industrial project where manipulators were involved in warehouse tasks.

Robox uRC2:
- Custom operating system
- Hard real time constraints
ROS plumbing strategy

XML-RPC over TCP/IP  
TCPROS over TCP/IP
ROS plumbing strategy

ROS master node

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

ROS master node

Node “X”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

Node “X”

ROS master node

XML-RPC over TCP/IP

TCPROS over TCP/IP

Anyone is publishing /foo topic?
ROS plumbing strategy

ROS master node

Node “X”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

ROS master node

Node “X”

XML-RPC over TCP/IP

TCPROS over TCP/IP

Node “Y”
ROS plumbing strategy

ROS master node

Node “X”

Node “Y”

XML-RPC over TCP/IP

TCPROS over TCP/IP

Try, I’m publishing /foo topic!
ROS plumbing strategy

ROS master node

Node “X”

Node “Y”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

Node “X”

ROS master node

Ehy, “Y” is publishing /foo topic!

Node “Y”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

Node “X”

ROS master node

Node “Y”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

ROS master node

Node “X”

Subscribe to /foo topic

Node “Y”

XML-RPC over TCP/IP

TCPROS over TCP/IP
ROS plumbing strategy

ROS master node

Node “X”

XML-RPC over TCP/IP

Node “Y”

TCPROS over TCP/IP
ROS plumbing strategy

Node “X” ->... /foo topic data .... -> Node “Y”

ROS master node

XML-RPC over TCP/IP

TCPROS over TCP/IP
Plumbing in ROS: roscpp library

Multithreading;
Messages defined in C++ headers that should be “generated” by external tools;
Depends on several system library:
- **pthread**: Multithreading
- **C++ STL**: containers
- **BSD sockets**: networking
- **XmlRpc++**: XML-RPC protocol
- **Boost**: Shared pointers, mutex, filesystem, signal-slots, ...
- **Log4cxx**: logging
- **POSIX UNIX** standard libraries
Plumbing in cROS

select() syscall
Plumbing in cROS

select() syscall

XML-RPC listener
Plumbing in cROS

XML-RPC server → select() syscall → XML-RPC listener

XML-RPC server

XML-RPC server
Plumbing in cROS

XML-RPC client
XML-RPC client
XML-RPC client
XML-RPC server
XML-RPC server
XML-RPC server
XML-RPC listener

select() syscall
Plumbing in cROS

XML-RPC client
XML-RPC server
XML-RPC server
XML-RPC server
XML-RPC client
XML-RPC client
XML-RPC client
TCPROS listener

select() syscall
Plumbing in cROS
Plumbing in cROS

select() syscall

XML-RPC client

TCPROS client

TCPROS server

XML-RPC server

XML-RPC listener

TCPROS listener
Plumbing in cROS

Process just one event per cycle:
- open new connections;
- start new write and read actions;
- close dropped connections;
- ...

select() syscall
Plumbing in cROS

Process just one event per cycle:
- open new connections;
- start new write and read actions;
- close dropped connections;
- ...
Plumbing in cROS

Each connection (XML-RPC and TCPROS clients, XML-RPC and TCPROS servers, ...) is managed in a similar way of a process in a multitasking operating system, with:

- Associated file descriptor (i.e., the socket)
- Its own memory space
- A state (i.e., inside a finite state machine)
- Timing information
- ...

Plumbing in cROS

One cycle of the loop:

- **Function cRosNodeDoEventsLoop()**: useful for hardware without operating system, it should be used inside a cycle, e.g.:

```c
while(1)
{
    //Do here all the other non ROS operations
    cRosNodeDoEventsLoop( node );
}
```

Infinite loop:

- **Function cRosNodeStart()**
Prepare your *.msg files

Add in your CmakeLists.txt the following macros:

- `add_message_files( FILES my_message1.msg my_message2.msg ...)`
- `generate_messages( DEPENDENCIES std_msgs ...)`

These macros will generate C++ headers, to be used in the node implementation:

- `#include <std_msgs/Time.h>`
- `#include <std_msgs/Int16.h>`
- `#include <my_msgs/my_message1.h>`
Messages in cROS

cROS **dynamically generate and parse** messages and services if the message/service definition files (*.msg and *.srv) have been provided in a known path.

cROS supports standard and custom built message and service types and it provides an API to read and write the single messages fields: the **developer is in charge of providing the right combination filed name - type**.
Getting started with cROS

1. Create in the main() function a cROS node entity and tell it to connect to roscore:

   - char *node_host = "127.0.0.1";
   char *roscore_host = "127.0.0.1";
   short uint roscore_port = 11311;
   char *msg_path= ".";
   CrosNode *node =
   cRoscNodeCreate("/node_name", node_host, roscore_host, roscore_port, msg_path, NULL);

2a. Create a subscriber and supply a callback for received messages:

   - cRosApiRegisterSubscriber(node,"/topic_name", "std_msgs/String", callback_sub, NULL, NULL)
Getting started with cROS

2b. Create a publisher and supply a callback that sent messages with a given frequency:

- int loop_period = 1000;
  cRosApiRegisterPublisher(node,"/topic_name",
  "std_msgs/String", loop_period,
  callback_pub, NULL, NULL);

3. Run the endless loop:

- unsigned char exit = 0;
  cRosNodeStart( node, &exit );
  cRosNodeDestroy( node );
  return EXIT_SUCCESS;
Getting started with cROS

4a. Implement the callbacks for the published topics:

```c
static CallbackResponse
    callback_pub(cRosMessage *message, void* data_context)
{
    // Get the required message field
    cRosMessageField *data_field = 
        cRosMessageGetField(message, "data");
    if(data_field)
    {
        char *str = "Hello";
        cRosMessageSetFieldValueString( 
            data_field, str);
    }
    return 0;
}
```
4b. Implement the callbacks for the subscribed topics:

```c
- static CallbackResponse
  callback_sub(cRosMessage *message, void* data_context)

  
  cRosMessageField *data_field =
  cRosMessageGetField(message, "data");
  if(data_field)
  {
    // Do something with incoming data
    printf("%s", data_field->data.as_string);
  }
  return 0;
```

Getting started with cROS
cROS important headers

- cros_node.h
  - Node creation

- cros_api.h
  - Create publishers and subscribers

- cros_message.h
  - Read and write messages
Exercise 1: testing cROS

Download cROS using git:

- git clone
  https://github.com/ros-industrial/cros.git

Compile cROS using cmake and make:

- cd cros
- mkdir build
- cd build
- cmake ..
- Make
Exercise 1: testing cROS

Start the ROS master node
- roscore

From a different terminal, run a simple cROS sample publisher node:
- cd cros/build/bin
- ./talker

Test the new node using the rostopic diagnostic tool:
- rostopic list
- rostopic info chatter
- rostopic echo chatter
Compile cROS using only gcc

If the commands `cmake` and `make` are not available, basically:

- cd cros
- gcc -Iinclude -lc -o talker
  samples/talker.c src/*
Exercise 2: publish & subscribe

Follow tutorial 1, then from a new terminal run a simple cROS sample subscriber node:

- cd cros/build/bin
- ./listner

In the meanwhile, it is still possible to listen to the publisher node using rostopic or to run more subscribers:

- rostopic echo chatter
- ./listner
- ...

Modify the sample code talker.c adding a second published topic:

- Publish the standard topic type `std_msgs/Float64` with a new topic name (e.g. `double_chatter`) and a different loop_period (e.g., 1000 ms).
- Provide a new callback that generates a new data of type `std_msgs/Float64` (see include/cros_message.h):
  ```
  cRosMessageField *data_field = cRosMessageGetField(message, "data");
  if(data_field)
      data_field->data.as_float64 = count;
  ```
- Note: remember to add the message definition, e.g.:
  ```
  float64 data
  in samples/rosdb/std_msgs/Float64.msg and to run again cmake ..
  ```
Exercise 2: publish & subscribe

Test the talker node using rostopic:

- rostopic echo chatter
- rostopic echo double_chatter

Modify the sample code listener.c in order to subscribe to both the topics chatter and double_chatter printing in output the received messages

Test the modified listener node

Mix published and subscribed topics between talker.c and listener.c, i.e.:

- talker.c publishes chatter and subscribes to double_chatter
- listener.c publishes double_chatter and subscribes to chatter
Starting from `talker.c`, implement a service provider based on the ROS tutorial service example `add_two_ints_client` using the cROS function:

```c
    crosApiRegisterServiceProvider() defined in cros_api.h.
```

Use `/add_two_ints` as service name and `AddTwoInts` as service type.

Note: remember to add the service definition `AddTwoInts.srv`, e.g.:

```c
    int64 a
    int64 b
    ***
    int64 sum
```

in `samples/rosdb/`

Test the service using the ROS tutorial script:

```bash
    $ rosrun rospython_tutorials add_two_ints_client 5 4
```
Next steps toward cROS 1.0

Simplified API (small changes required)

Complete service subscription capabilities

Easier parameters management

Complete API documentation

Asynchronous publishing?

...