Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 2:

ROS and Roboscoop
Robots of today

- Many sensors and actuators
- Able to operate in familiar or expected environments
- Able to perform specialized tasks
Robots of the future

C-3PO
➢ Provides etiquette, customs, and translation assistance
➢ Has own thoughts and feelings

R2-D2
➢ Rescues people and robots
➢ Repairs other robots and complex hardware and software

Advanced robots must be able to operate and perform tasks in complex, unknown environments.

As robotics advances, we must be aware that robots can be both helpful and harmful.
Concurrency in robotics

Advanced robotic systems have many hardware components that can operate concurrently.

- Sensors and actuators can run in parallel.
- Locomotion and manipulators can run concurrently.
Concurrency in robotics
Multiprocessing, parallelism

- **Multiprocessing**: the use of more than one processing unit in a system
- **Parallel execution**: processes running at the same time

![Diagram showing multiprocessor system with two processes and two CPUs]

- Process 1: Move
- Process 2: Scan
Multitasking, concurrency

- **Interleaving**: several tasks active, running one at a time
- **Multitasking**: the OS runs interleaved executions
- **Concurrency**: multiprocessing and/or multitasking

P1: Go to goal  P2: Avoid obstacle
Concurrency

Benefits of introducing concurrency into programs:

- **Efficiency**: time (load sharing), cost (resource sharing)
- **Availability**: multiple access
- **Convenience**: perform several tasks at once
- **Modeling power**: describe systems that are inherently parallel
Roboscoop

Concurrency framework for robotics
Roboscoop software architecture

**Roboscoop**
- Library (set of primitives and tools for their coordination)
- Integration with other robotics frameworks
- External calls

**SCOOP**
- O-O Structure
- Coordination
- Concurrency

**ROS**
- Communication
- Navigation, image processing, coordinate transforms, visualization, ...
ROS: Open-source, meta-operating system for robots
ROS provides the services of an operating system, including
- hardware abstraction,
- low-level device control,
- implementation of commonly-used functionality,
- message-passing between processes, and
- package management

http://www.ros.org
Goals of ROS

- Support code *reuse* in robotics research and development.
- Enable executables to be individually designed and loosely coupled at runtime through its distributed framework of processes.
- Group processes for easy sharing and distribution.
- Enable the distribution of collaboration through its repositories.

Properties of ROS

- Thin
- Peer-to-Peer
- Multi-lingual: C++, Python, Lisp
ROS communication
ROS node

Node
- A process that performs computation
- Interchangeable with a software module
- Can generate data for and receive data from other nodes

A system is typically comprised of many nodes: robot control node, localization node, path planning node, perception node, etc.

Benefits of using nodes
- Fault-tolerance: crashes are isolated to individual nodes
- Reduction of code complexity
ROS topic

**Topic**
- Named bus over which nodes exchange messages
- Has anonymous publish/subscribe semantics.

A node can publish and/or subscribe to multiple topics. A topic supports multiple publishers and subscribers.
**ROS message**

**Message**: Strictly typed data structure used for communication between nodes

**Message description specification**
- Build-in types
- Names of Messages defined on their own
- Fixed- or variable-length arrays:
- Header type: `std_msgs/Header`:
  `uint32 seq`, `time stamp`, `string frame_id`
- Constants

Messages can be arbitrarily nested structures and arrays.

```plaintext
int16 x
uint32 y
sensor_msgs/LaserScan s
uint8[] data
float32[10] a
Header header
int32 z=123
string s=foo
```
common_msgs

- Messages that are widely used by other ROS packages
- Provide a shared dependency to multiple stacks, eliminating a circular dependency

Types of common_msgs

- **geometry_msgs**: Point, Pose, Transform, Vector, Quaternion, etc.
- **nav_msgs**: MapMetaData, Odometry, Path, etc.
- **sensor_msgs**: LaserScan, PointCloud, Range, etc.
ROS service

Service: A pair of strictly typed messages for synchronous transactions

Service description specification
- Request messages
- Response messages

Two messages are concatenated together with a ‘---’.
A service cannot be embedded inside another service.
Only one node can advertise a service of any particular name.

Node

Service Invocation

Response

Node

int16 x
uint32 y
---
string s
ROS master

Master

- Provides naming and registration services to nodes
- Tracks publishers and subscribers to topics and services
- Enables individual nodes to locate one another

Diagram:

- Camera
- Images
- Image Viewer

Arrows indicating:
- Advertise (Images) to Master
- Subscribe (Images) to Master
- Publication from Images to Camera
- Subscription from Image Viewer to Images
**TCPROS**
- Provides a simple, reliable communication stream
- TCP packets always arrive in order
- Lost packets are resent until they arrive.

**UDPROS**
- Packets can be lost, contain errors, or be duplicated.
- Is useful when multiple subscribers are grouped on a single subnet
- Is useful when latency is more important than reliability, e.g., teleoperation, audio streaming
- Suited for a lossy WiFi or cell modem connection.
ROS topic connection example

registerPublisher("camera", "image", "sensor_msgs/Image", "pub:123")

registerSubscriber("image_viewer", "image", "sensor_msgs/Image", "sub:456")

requestTopic("image_viewer", "image", [[TCPROS, "sub:567"]])

[1, "initialize communication", [TCPROS, "pub:234"]]

Image data message

- XMLRPC - TCPROS
ROS package

Package

- A software unit with useful functionality
- Aims to provide enough functionality to be useful but still lightweight and reusable in other software.
- Can contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, etc.

Useful packages for the class

TF: coordinate transformation  
RViz: 3D visualization
TF: Coordinate Transformation

```cpp
static tf::TransformBroadcaster br;
tf::Transform transform;
transform.setOrigin( tf::Vector3(x, y, 0.0) );
transform.setRotation( tf::Quaternion(theta, 0, 0) );
br.sendTransform(tf::StampedTransform(transform,
ros::Time::now(), "world", "robot1"));
```
Demo

- ROS publish/subscribe
- TF
- RViz
ROS coordinate frame conventions

Axis orientation
- \( x \): forward, \( y \): left, \( z \): up

Rotation representation
- Quaternion: \( x, y, z, w \)
  - Compact representation
  - No singularities
- Rotation matrix
  - No singularities
- \textbf{roll}: \( x \), \textbf{pitch}: \( y \), \textbf{yaw}: \( z \)
  - No ambiguity in order
  - Used for angular velocities
# Standard SI units

<table>
<thead>
<tr>
<th>Base Units</th>
<th>Derived Units</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quantity</strong></td>
<td><strong>Unit</strong></td>
</tr>
<tr>
<td>Length</td>
<td>Meter</td>
</tr>
<tr>
<td>Mass</td>
<td>Kilogram</td>
</tr>
<tr>
<td>Time</td>
<td>Second</td>
</tr>
<tr>
<td>Current</td>
<td>Ampere</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Build system: CMake

Build system

- A software tool for automating program compilation, testing, etc.
- Maps a set of source code (files) to a target (executable program, library, generated script, exported interface)
- Must fully understand the build dependencies

CMake

- Cross-platform build system
- Controls the build process using a CMakeLists.txt file
- Creates native makefile in the target environment

```bash
cmake_minimum_required(VERSION 2.8.3)
project(ProjectName)
add_executable(ExecutableName file.cpp)
```
ROS build system: catkin

**catkin**

- Official build system of ROS
- CMake with some custom CMake macros and Python scripts
- Supports for automatic 'find package' infrastructure and building multiple, dependent projects at the same time
- Simplifies the build process of ROS’s large, complex, and highly heterogeneous code ecosystem

**Advantages of using catkin**

- Portability through Python and pure CMake
- Independent of ROS and usable on non-ROS projects
- Out-of-source builds: can build targets to any folder

http://wiki.ros.org/catkin/Tutorials
Dependency management: package.xml

```xml
<package>
  <name>foo</name>
  <version>1.2.3</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="me@ethz.ch">Me</maintainer>
  <license>BSD</license>
  <url>http://www.ethz.ch/foo</url>
  <author>Me</author>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>
  <test_depend>python-mock</test_depend>
</package>
```

- **Required tags**
- **Package’s build system tools**
- **Packages needed at build time**
- **Packages needed at run time**
- **Additional packages for unit testing**

http://wiki.ros.org/catkin/package.xml
Dependency management: CMakeLists.txt

```cmake
# Minimum Cmake version
cmake_minimum_required(VERSION 2.8.3)

# Project name
project(foo)

# Dependent packages
find_package(catkin REQUIRED COMPONENTS roscpp)

# Include paths for the package
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS roscpp
  DEPENDS opencv
)

# Location of header files
include_directories(include ${catkin_INCLUDE_DIRS})

# An executable target to be built
add_executable(foo src/foo.cpp)

# Libraries to be built
add_library(moo src/moo.cpp)

target_link_libraries(foo moo)
```

http://wiki.ros.org/catkin/CMakeLists.txt
Roboscoop software architecture

**Roboscoop**
- Library (set of primitives and tools for their coordination)
- Integration with other robotics frameworks
- External calls

**SCOOP**
- O-O Structure
- Coordination
- Concurrency

**ROS**
- Communication
- Navigation, image processing, coordinate transforms, visualization, ...
SCOOP: a brief introduction

Simple Concurrent Object Oriented Programming

- Easy parallelization
- One more keyword in Eiffel (separate)
- Natural addition to O-O framework
- Retains natural modes of reasoning about programs
- Coordination is easy to express: close correspondence with behavioral specification[1]

To go straight, to avoid obstacles ...

Get the state of the robot
- Location and orientation
- Linear and angular velocity
- Sensory information

Control the velocity
Stop if there is a request for stopping (e.g., emergency stop)

P1: Go straight  P2: Avoid obstacle

**separate**: objects are potentially in a different region
r: **separate** ROBOT_STATE_SIGNALER
d: **separate** DIFFERENTIAL_DRIVE
s: **separate** STOP_SIGNALER

Obstacle
separate calls

feature

robot: separate ROBOT_STATE_SIGNALER -- Current robot's state
drive: separate DIFFERENTIAL_DRIVE -- Control robot's velocity
stop: separate STOP_SIGNALER -- Whether stop requested

start -- Start the control

local

    a, b: separate PRIMITIVE_BEHAVIOR

do

    create a.make (stop)
    create b.make (stop)
    start_robot_behaviors (a, b)

end

start_robot_behaviors (a, b: separate PRIMITIVE_BEHAVIOR)
do

    a.repeat_until_stop_requested (  
        agent a.avoid_obstacle (robot, drive, stop))
    b.repeat_until_stop_requested (  
        agent b.go_straight (robot, drive, stop))

end
Synchronization through preconditions

```
go_straight (a_robot: separate ROBOT_STATE_SIGNALER;
    a_drive: separate DIFFERENTIAL_DRIVE;
    a_stop: separate STOP_SIGNALER)
    -- Move robot unless stopped or an obstacle detected.
    require
        (not a_robot.is_moving and not a_robot.has_obstacle)
        or a_stop.is_stop_requested
    do
        if a_stop.is_stop_requested then
            a_drive.stop
        else
            a_drive.send_velocity (0.03, 0.0) -- 3cm/sec, no spinning
        end
    end
```
How do we cancel all processors?

GO_STRAIGHT (BEHAVIOR 1)

AVOID_OBSTACLE (BEHAVIOR 2)

STOP_SIGNALER

APPLICATION

is_stop_requested: BOOLEAN

set_stop (val: BOOLEAN)

stop.is_stop_requested

stop.set_stop(FALSE)

stop.is_stop_requested
Roboscoop

Coordination layer above SCOOP
Three-layer architecture
Synchronization: wait conditions
Interoperability through ROS (external calls)
Roboscoop repository structure

- **thymio_app**
  - app.e
  - controller
  - ...

- **roboscoop_lib**
  - controller
  - sequencer
  - sensor
  - common
  - util
  - signaler
  - ros
  - msg
  - actuator

- **roboscoop_ros**
  - msg
  - src
Communication with ROS nodes: publication

**Topic name:**
/aseba/events/sound_cmd

**Message type:**
asebaros/AsebaEvent

```cpp
pub: ROS_PUBLISHER[ASEBA_MSG]  
msg: ASEBA_MSG
create msg.make_with_two_values (0, sound_id)
create pub.make_with_topic ("/aseba/events/sound_cmd")
...
pub.publish (msg)
```
Communication with ROS nodes: subscription

- thymio_app
- roboscoop_lib
- roboscoop_lib/cpp
- ROS

**Topic name:**
/thymio_driver/odometry

**Message type:**
nav_msgs/Odometry

**Header**
- header
  - string child_frame_id
  - PoseWithCovariance pose
  - TwistWithCovariance twist

**subscriber**:
- separate ROS_SUBSCRIBER[ODOMETRY_MSG]

**signaler**:
- separate ODOMETRY_SIGNALER

...  

separate subscriber as sub, signaler as sig do

sub.subscribe ("/thymio_driver/odometry", agent sig.update_odometry)

end
class YOUR_APPLICATION feature

thymio: separate THYMIO_ROBOT -- The robot.
ros_spinner: separate ROS_SPINNER -- ROS spinner object for communication.

some_feature
    local
        robo_node: separate ROBOSCOOP_NODE
    do
        -- Initialize this application as a ROS node.
        robo_node := (create {ROS_NODE_STARTER}).roboscoop_node
        -- Create a robot object.
        create thymio.make
        -- Listen to ROS.
        create ros_spinner.make
        start_spinning (ros_spinner)
        -- Launch Thymio.
        launch_robot (thymio)
    end