ETH Zürich
Chair of Software Engineering

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.

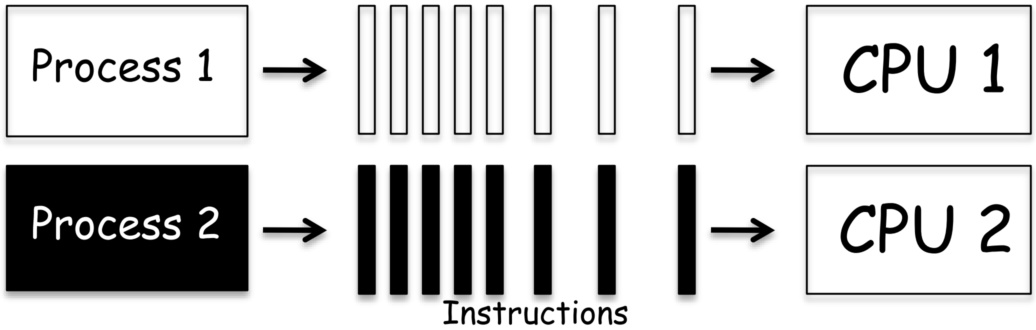
Concurrency in robotics

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

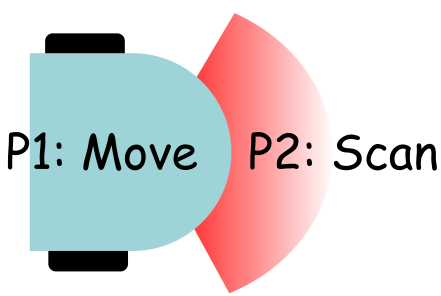
- Sensors and actuators can run in parallel.
- Locomotion and manipulators can run concurrently.

Concurrency in robotics

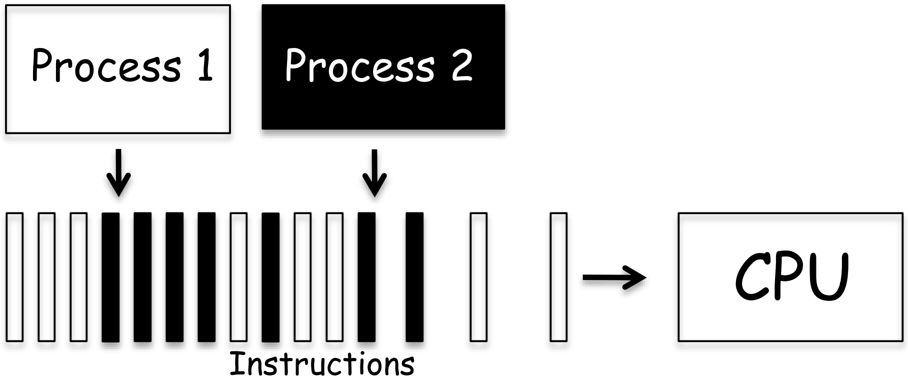
Multiprocessing, parallelism



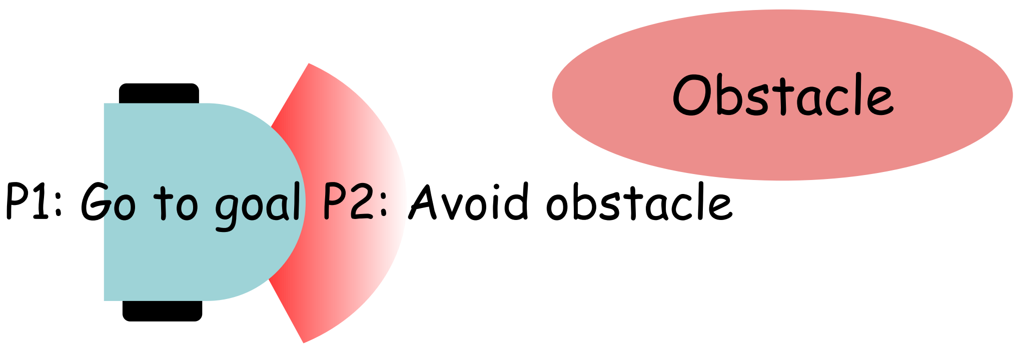
- Multiprocessing: the use of more than one processing unit in a system
- Parallel execution: processes running at the same time



Multitasking, concurrency



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



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: Robot Operating System

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

Quigely, M., et al. "ROS: an open-source Robot Operating System," IEEE International Conference on Robotics and Automation. 2009.

http://www.ros.org

ROS

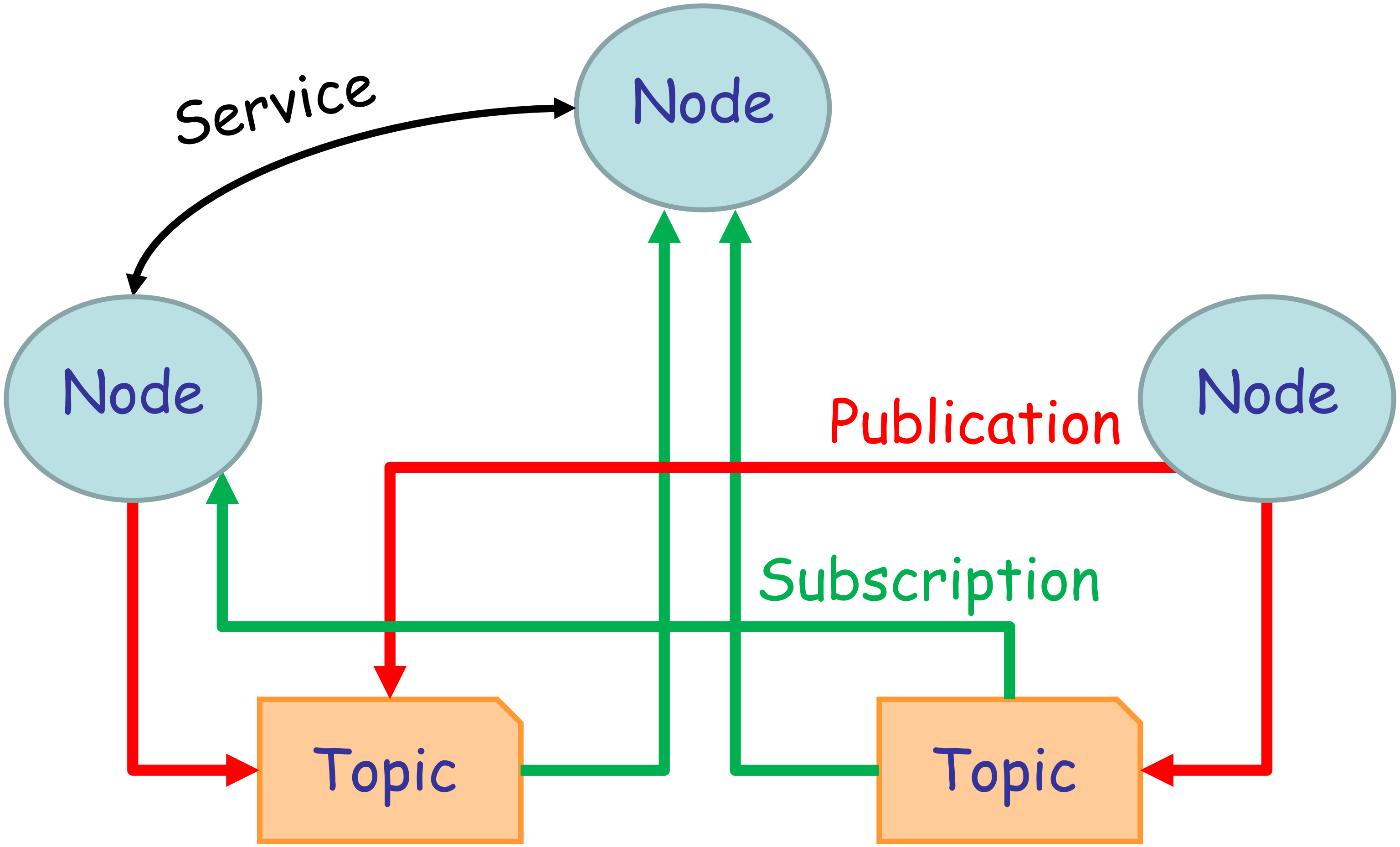
Goals of ROS

- 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

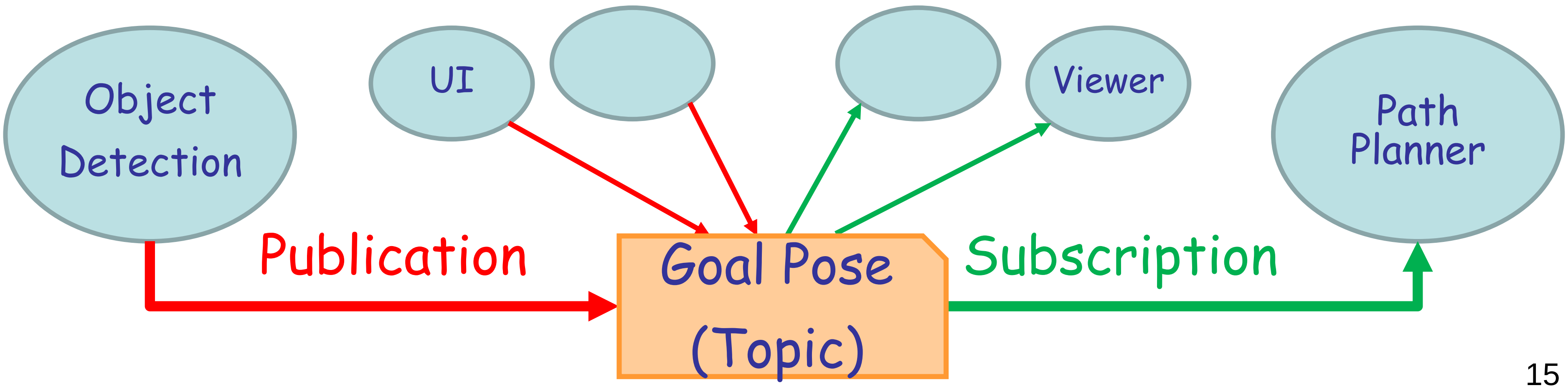
ROS topic

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

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

Messages can be arbitrarily nested structures and arrays.

common_msgs

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

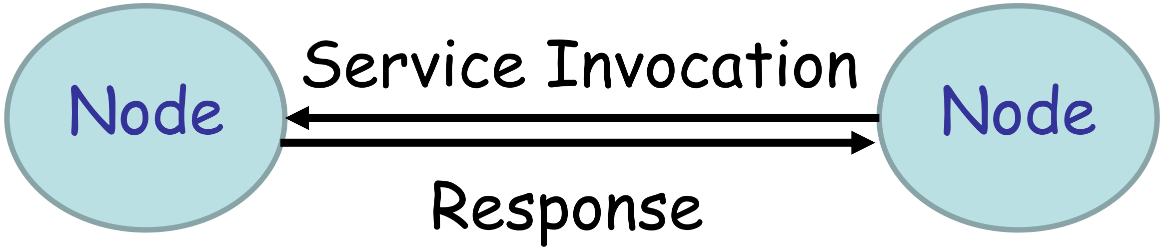
- Request messages
- Response messages

```
int16 x
uint32 y
---
string s
```

Two messages are concatenated together with a '---'.

A service **cannot** be embedded inside another service.

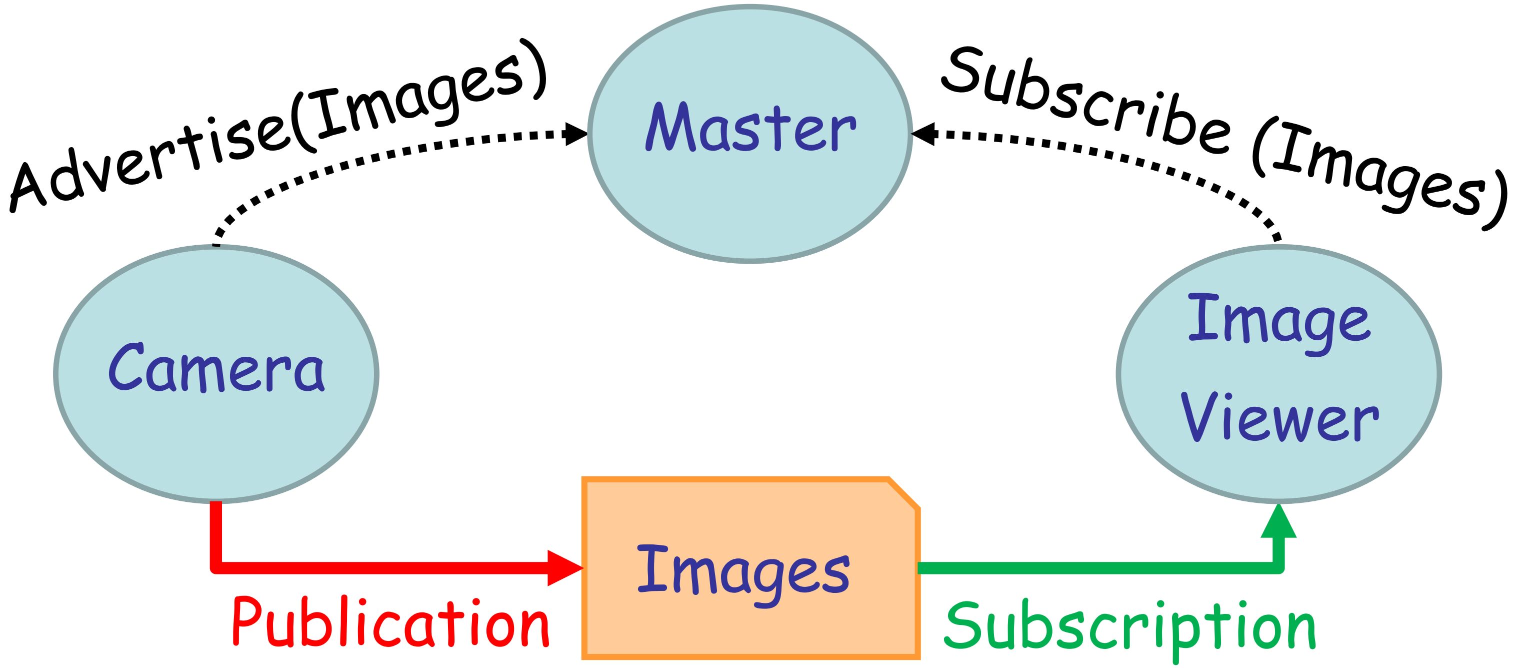
Only one node can advertise a service of any particular name.



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



Master and Slave APIs

- Manage information about the availability of topics and services
- Negotiate the connection transport
- **XML-RPC** (remote procedure call protocol using XML)
    - HTTP-based protocol
    - Stateless
    - Lightweight
    - Available in many programming languages

Return value format

- Status code: -1 (ERROR), 0 (FAILURE), 1 (SUCCESS)
- Status message: human readable string
- Value: defined by individual API calls

Master API

Register/unregister methods

- `registerService(caller_id, service, service_URI, caller_URI)`
- `unregisterService(caller_id, service, service_URI)`
- `registerSubscriber(caller_id, topic, topic_type, caller_URI)`
- `unregisterSubscriber(caller_id, topic, caller_URI)`
- `registerPublisher(caller_id, topic, topic_type, caller_URI)`
- `unregisterPublisher(caller_id, topic, caller_URI)`

Name service and system state

- `lookupNode(caller_id, node_name)`
- `lookupService(caller_id, service)`
- `getPublishedTopics(caller_id, subgraph)`
- `getTopicTypes(caller_id)`
- `getSystemState(caller_id)`
- `getUri(caller_id)`

Slave API

Receive callbacks from the Master

- `publisherUpdate(caller_id, topic, publishers)`
- `paramUpdate(caller_id, parameter_key, parameter_value)`

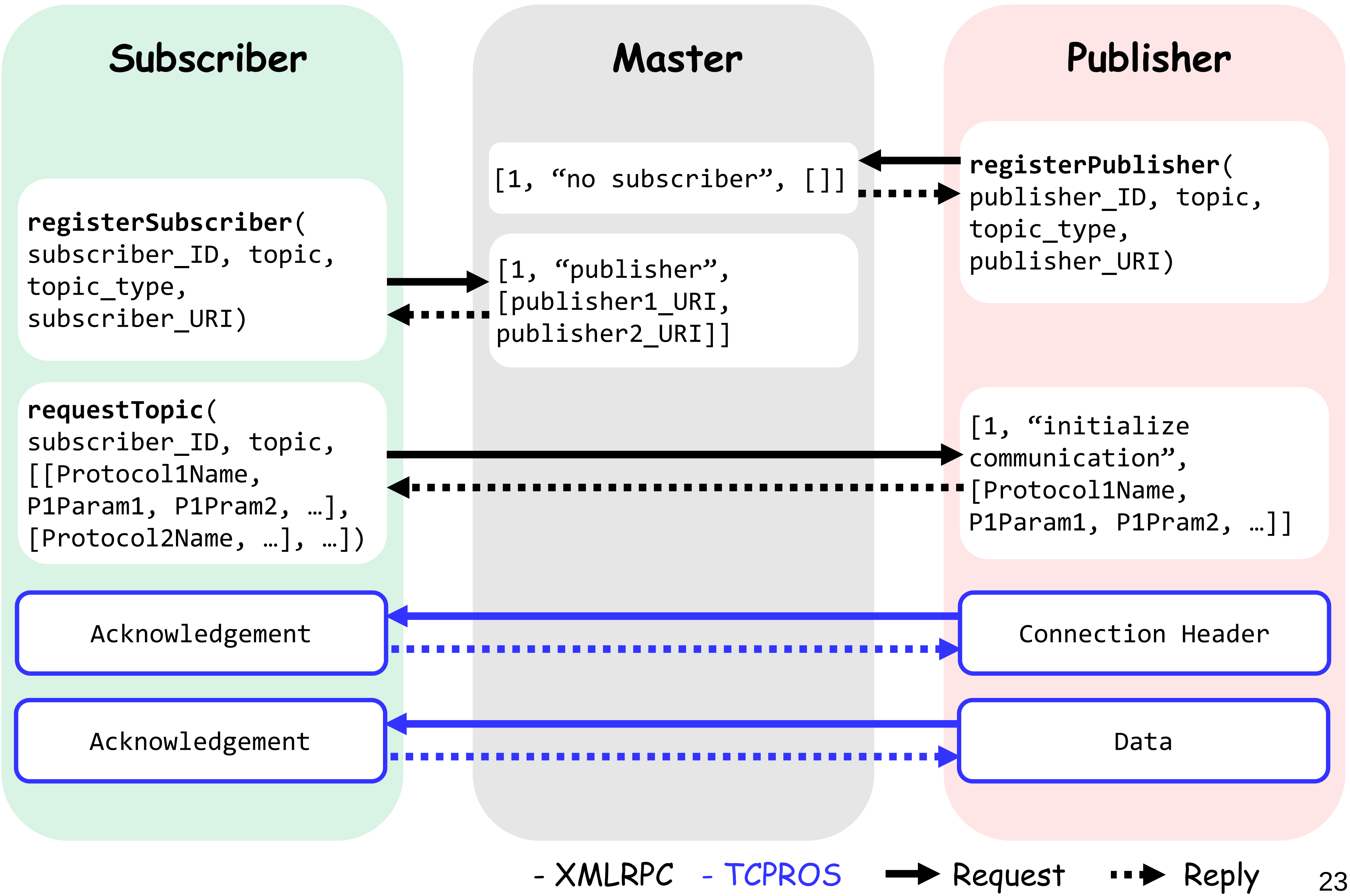
Negotiate connections with other nodes

- `requestTopic(caller_id, topic, protocols)`
- `shutdown(caller_id, msg='')`

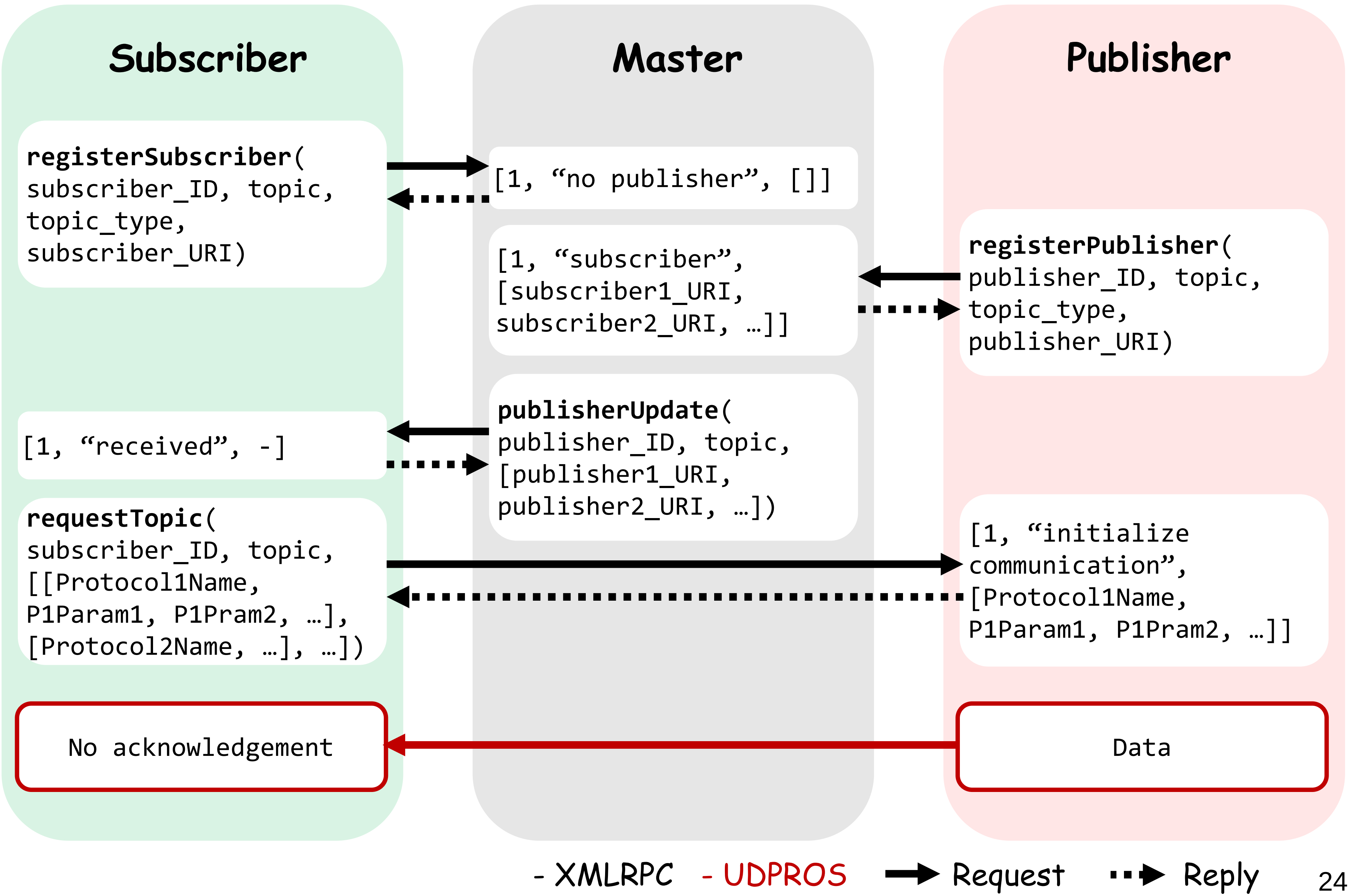
System state

- `getBusStats(caller_id)`
- `getBusInfo(caller_id)`
- `getMasterUri(caller_id)`
- `getPid(caller_id)`
- `getSubscriptions(caller_id)`
- `getPublications(caller_id)`

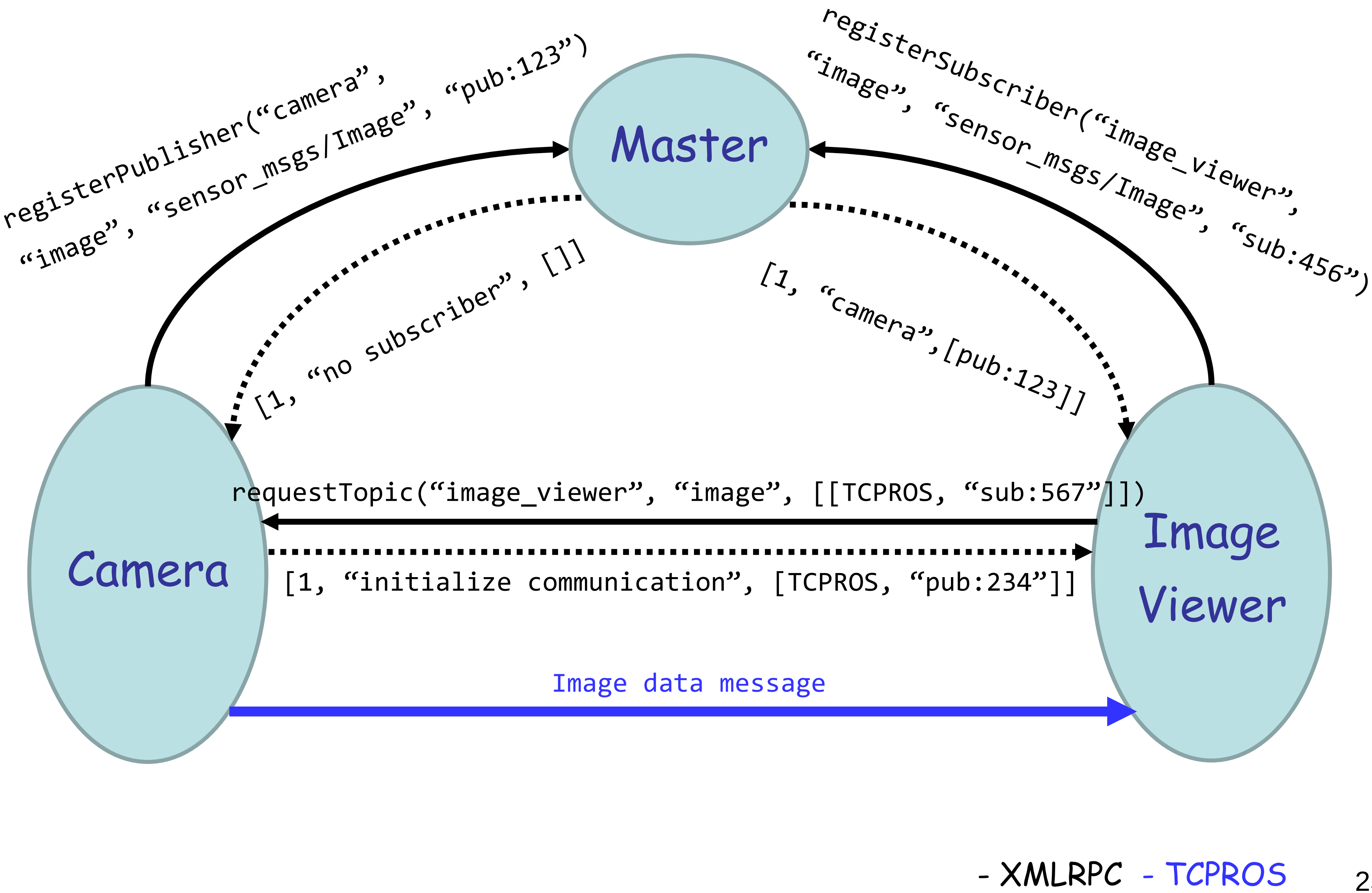
ROS topic connection



ROS topic connection



ROS topic connection example



ROS topic transport protocol

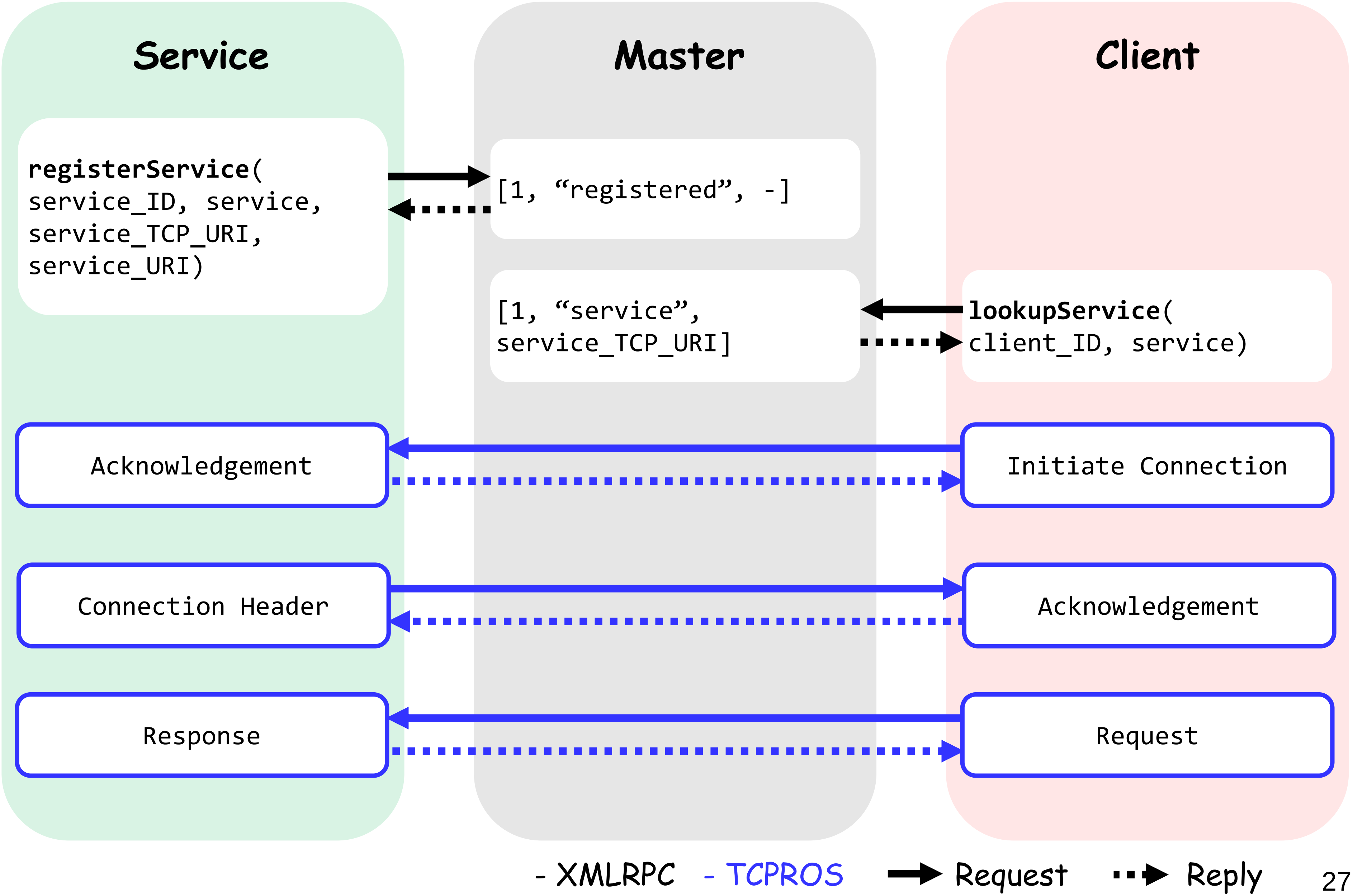
TCPROS

- 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 service connection

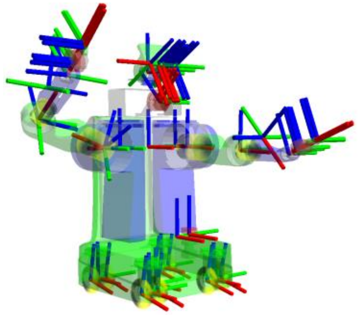


ROS package

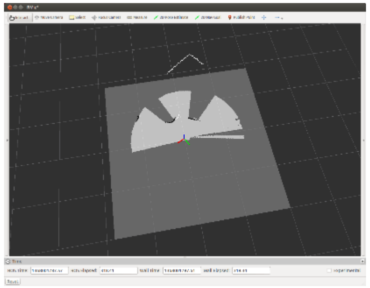
Package

- 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

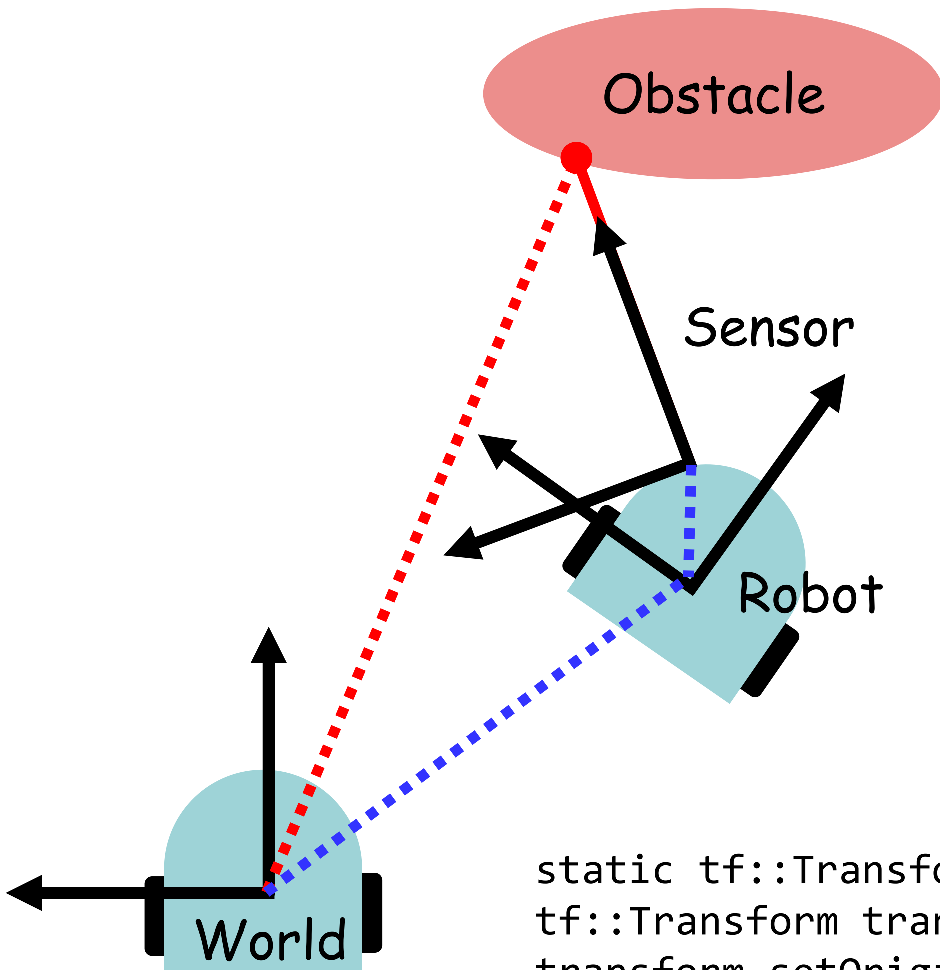


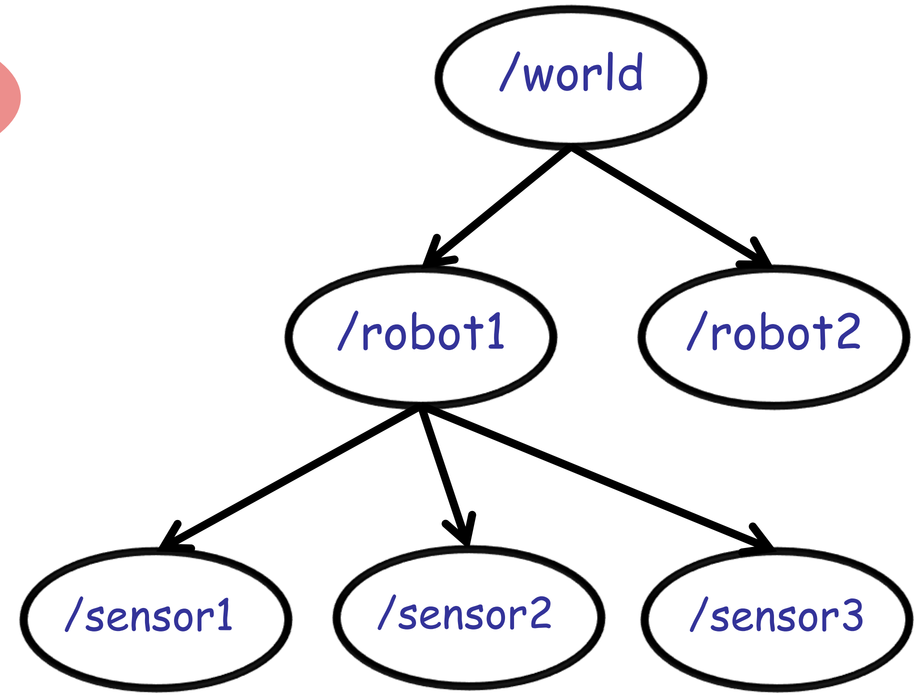
TF: coordinate transformation



RViz: 3D visualization

TF: Coordinate Transformation






```
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”);
```

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
- roll: x, pitch: y, yaw: z
  - No ambiguity in order
  - Used for angular velocities

ROS units

Standard SI units

Base Units		Derived Units	
Quantity	**Unit**	**Quantity**	**Unit**
Length	Meter	Angle	Radian
Mass	Kilogram	Frequency	Hertz
Time	Second	Force	Newton
Current	Ampere	Temperature	Celsius
		Power	Watt
		Voltage	Volt

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

```
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

```
<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 (name, version, description, maintainer, license)

Package's build system tools (buildtool_depend)

Packages needed at build time (build_depend)

Packages needed at run time (run_depend)

Additional packages for unit testing (test_depend)

http://wiki.ros.org/catkin/package.xml

Dependency management: CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)          Mimimum Cmake version

project(foo)                                   Project name

find_package(catkin REQUIRED COMPONENTS roscpp) Dependent packages

catkin_package(        Installs package.xml and generates code for find_package
   INCLUDE_DIRS include           Include paths for the package
   LIBRARIES ${PROJECT_NAME}      Exported libraries from the project
   CATKIN_DEPENDS roscpp          Other catkin projects this project depends on
   DEPENDS opencv                 Non-catkin CMake projects this project depends on
)

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

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

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

target_link_libraries(foo moo)       Libraries the executable target links against
```

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, …

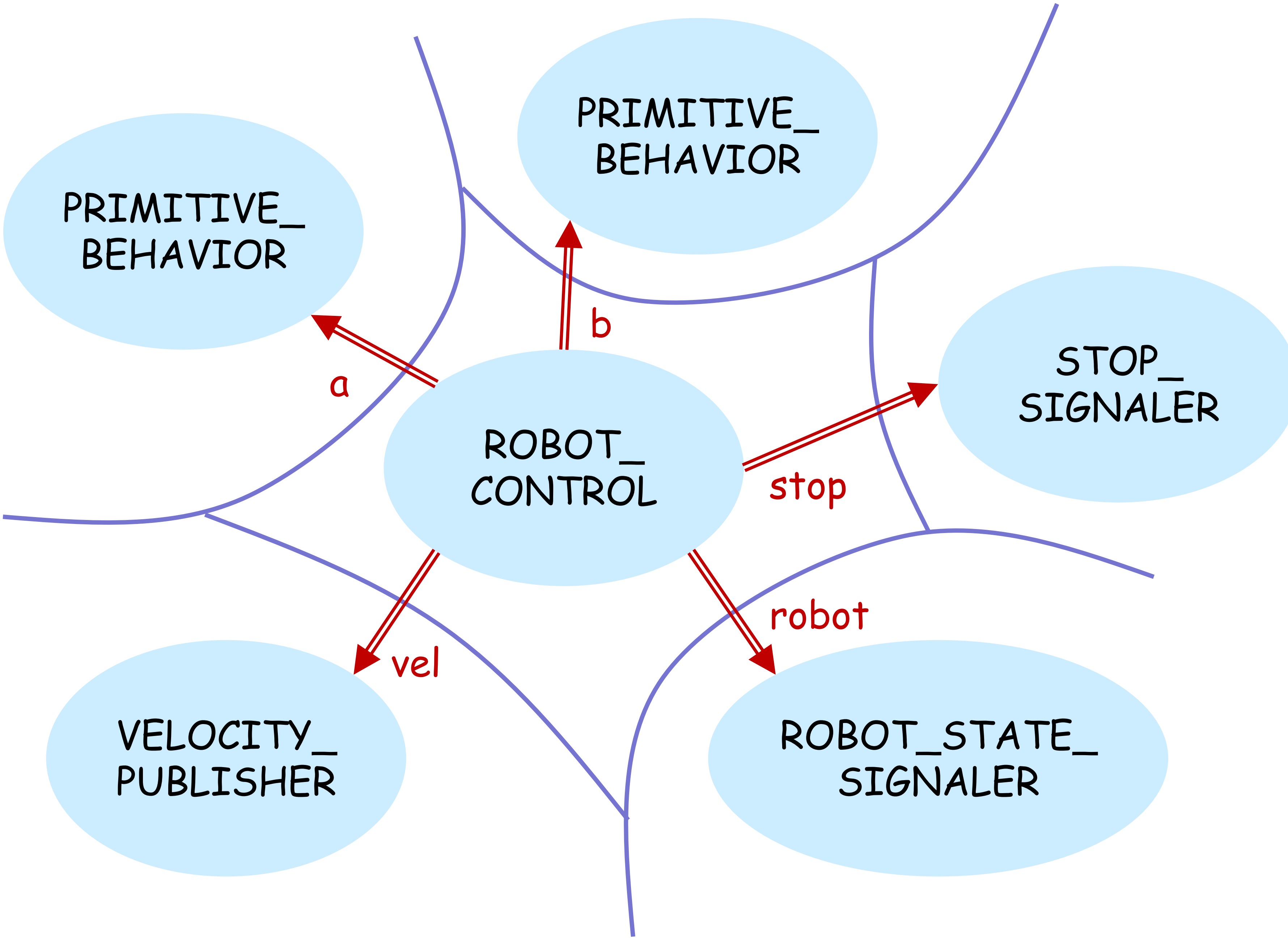
SCOOP: a brief introduction

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]

[1] Ramanathan, G. et al.: Deriving concurrent control software from behavioral specifications. IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1994-1999

Object and processor architecture



To go straight, to avoid obstacles ...

Get the state of the robot

- Location and orientation
- Linear and angular velocity
- Sensory information

Control the velocity

separate: objects are potentially on a different processor

r: **separate** ROBOT_STATE_SIGNALER

v: **separate** VELOCITY_PUBLISHER

s: **separate** STOP_SIGNALER

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

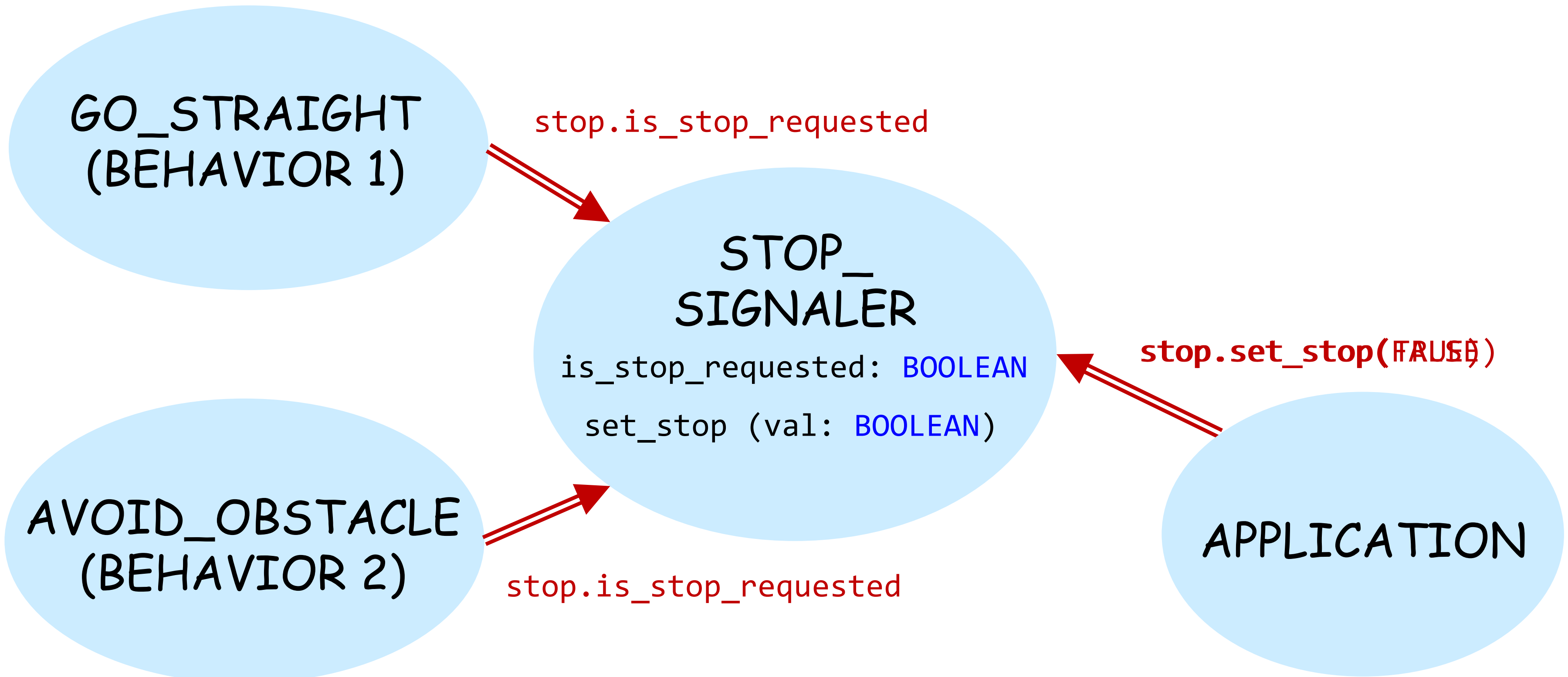
separate calls

```
feature
    robot: separate ROBOT_STATE_SIGNALER     -- Current robot's state
    vel: separate VELOCITY_PUBLISHER         -- 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_with_attributes (robot, vel, stop)
            create b.make_with_attributes (robot, vel, stop)
            start_robot_behaviors (a, b)
        end

    start_robot_behaviors (a, b: separate PRIMITIVE_BEHAVIOR)
        do
            a.avoid_obstacle_repeatedly
            b.go_straight_repeatedly
        end
```

Synchronization through preconditions

```
go_straight (a_robot: separate ROBOT_STATE_SIGNALER;
             a_vel: separate VELOCITY_PUBLISHER;
             a_stop: separate STOP_SIGNALER)
              -- Move robot unless stopped or an obstacle observed.
    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_vel.send_stop
        else
            a_vel.send_velocity (0.03, 0.0)  -- 3cm/sec, no spinning
        end
    end
```

How do we cancel all processors?



Roboscoop

Coordination layer above SCOOP

Three-layer architecture

Synchronization: wait conditions

Interoperability through ROS (external calls)

Roboscoop repository structure

roboscoop_app	application.e	_cpp	
roboscoop_lib	controller	sequencer	sensor
	common	utils	signaler
	ros	msgs	communication
roboscoop_ros	msg	src	

Communication with ROS nodes: publication

roboscoop_app → roboscoop_lib → roboscoop_lib/_cpp → ROS

```
THYMIO_SOUND_PUBLISHER

inherit

ASEBA_PUBLISHER

SOUND_PUBLISHER
```

```
ASEBA_PUBLISHER

SOUND_PUBLISHER
```

aseba_event_publisher.h

Topic name:
/aseba/events/sound_cmd

Message type:
asebaros/AsebaEvent

```
time stamp
uint16 source
int16[] data
```

Communication with ROS nodes: subscription

roboscoop_app | roboscoop_lib | …_lib/_cpp …_app/_cpp | ROS

THYMIO_SUBSCRIBER

inherit

ODOMETRY_SUBSCRIBER

ROS_SUBSCRIBER

ODOMETRY_SUBSCRIBER

subscriber.h

gen_subscriber.h

Topic name:
/thymio_driver/odometry

Message type:
nav_msgs/Odometry

```
Header header
string child_frame_id
PoseWithCovariance pose
TwistWithCovariance twist
```

Edit **topics.xml** (in roboscoop_app/_cpp/callbacks_gen)

Defines topics that the application subscribes to

Generate C++ subscriber (run script)

Write your custom subscriber class in Eiffel

Create an object of the generated class and pass it to your subscriber

Communication with ROS nodes: application

```
subscriber: THYMIO_SUBSCRIBER

some_feature
    local
        sub_name: C_STRING
        gen_subscriber_ptr: POINTER
    do
        create sub_name.make ("pregenerated_subscriber")
        gen_subscriber_ptr := c_new_gen_subscriber (1, sub_name.item)

        create subscriber.make_with_ptr (gen_subscriber)
    end

c_new_gen_subscriber (a_id: INTEGER; a_c_name: POINTER): POINTER
    external
        "C++ inline use %"gen_subscriber.h%""
    alias
        "return new GenSubscriber($a_id, $a_c_name);"
    end
```