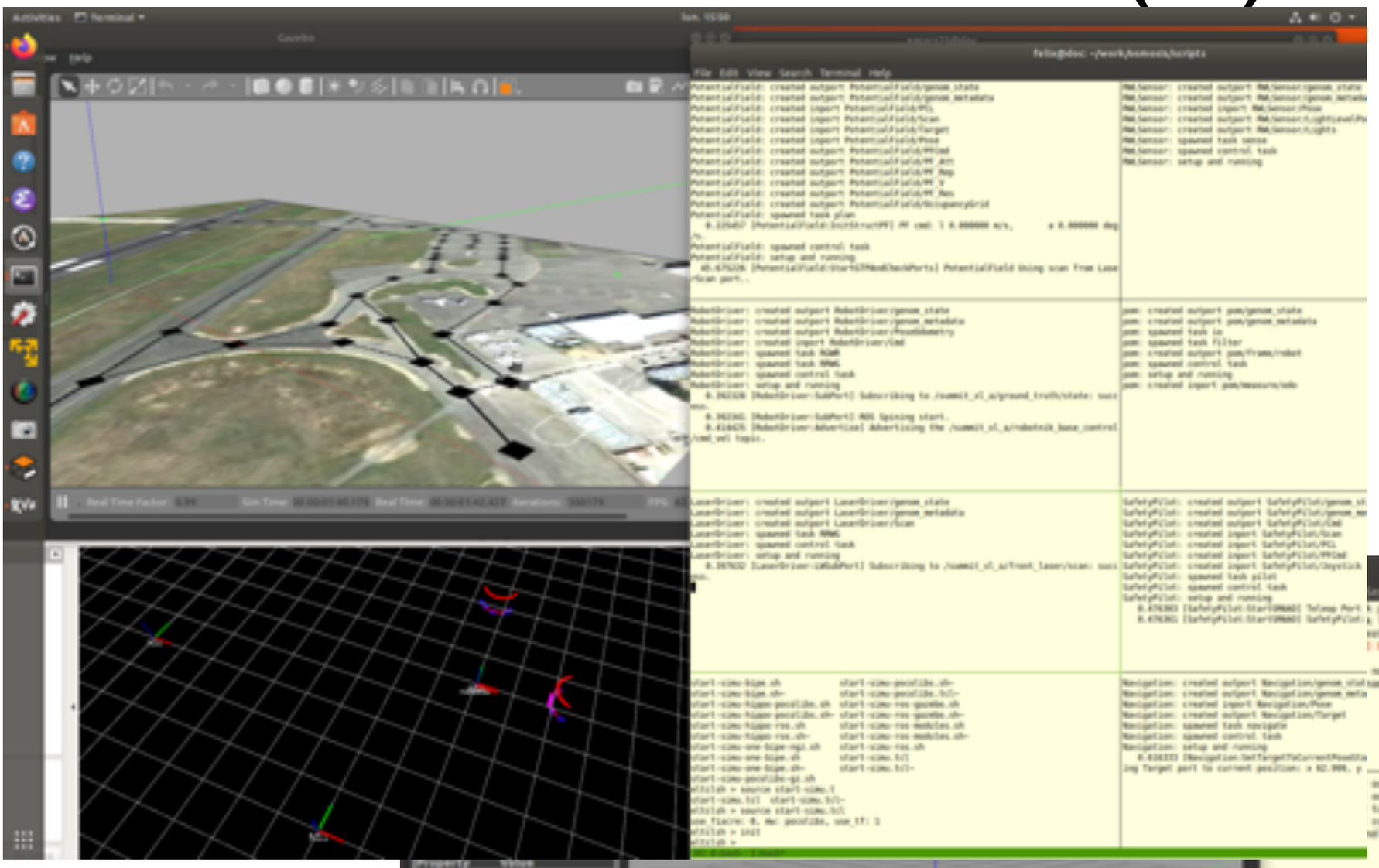
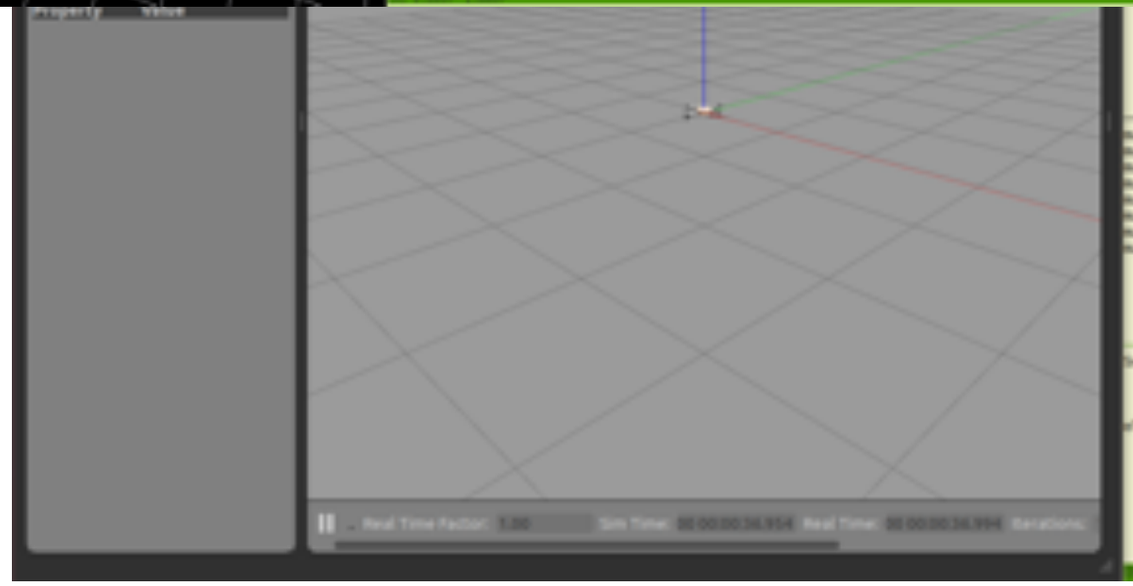
Robot Software Architecture and Robot Programming

Architecture, Middleware, Tools, Specification, Programming,Validation & Verification

Félix Ingrand
LAAS/CNRS
felix@laas.fr

Démo(s)





Demo

- What do you see? (simulation but correspond to real robot application)
- What are the sensors and effectors?
- How many programs?
- How do you program this?
  - langage, OS, hardware?
- What are the problems to address?

Demo

- From an experiment specification POV, what can go wrong?
  - Loss of sensors
  - Interpretation
  - Localization
  - etc

Demo

- From a “software” point of view, what can go wrong?
  - Memory leak
  - Time and CPU resources
    - schedulability (Janette Cardoso’s course)
  - Program crash
  - Program delayed

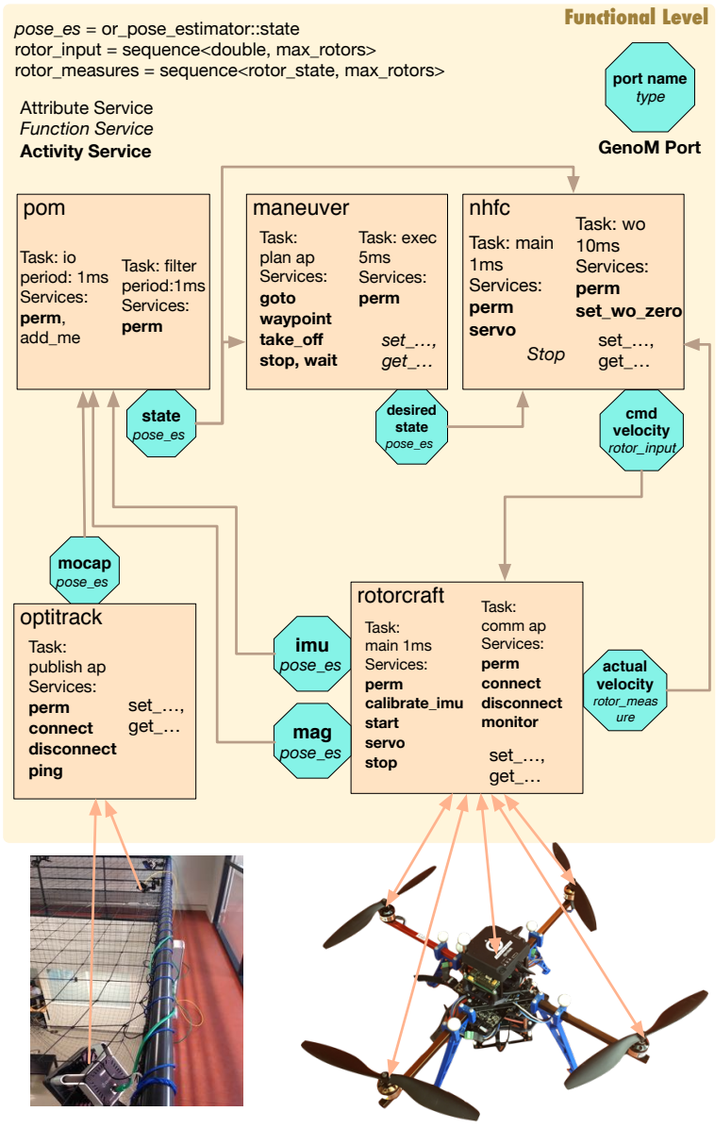
How do you really program these?

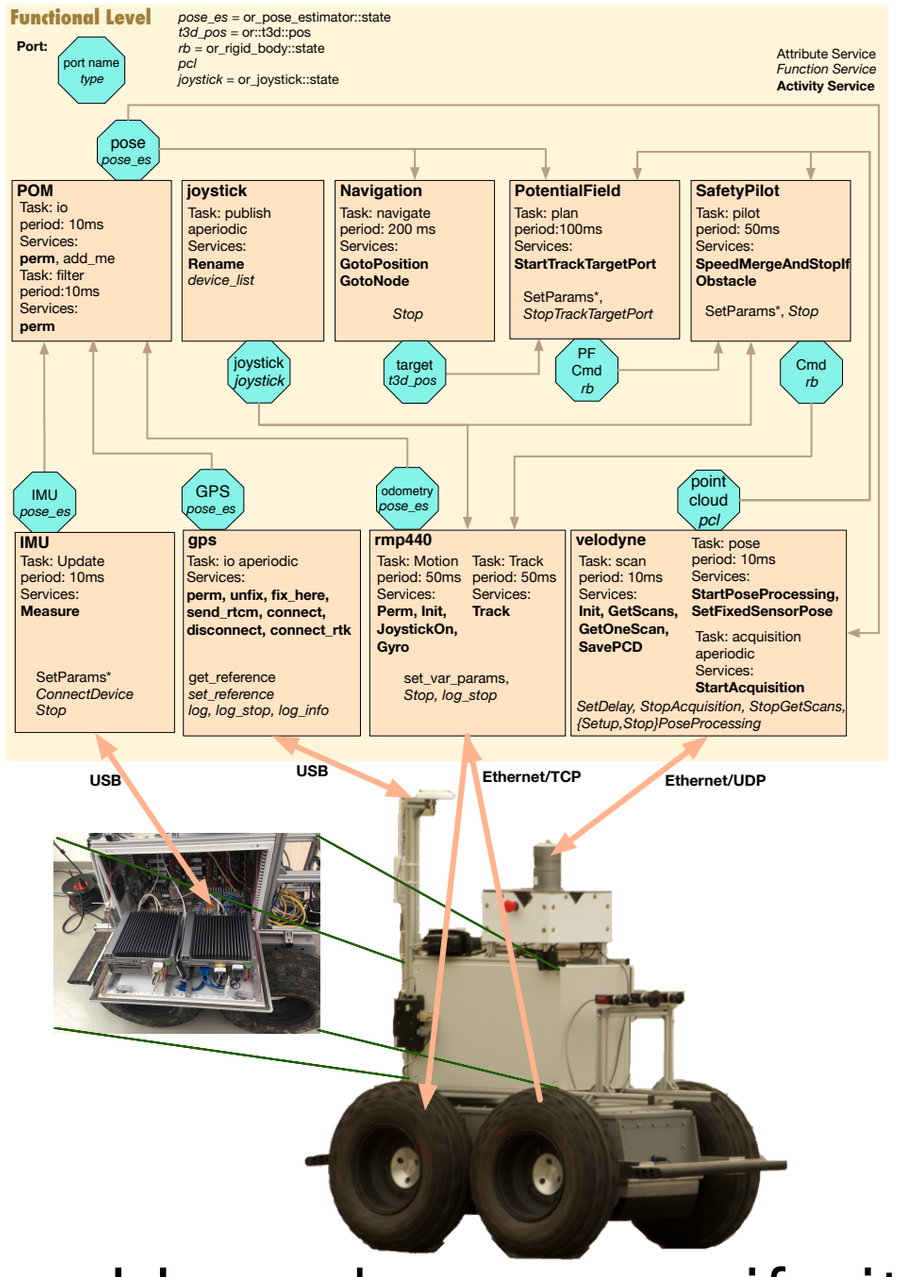
What can you guarantee?

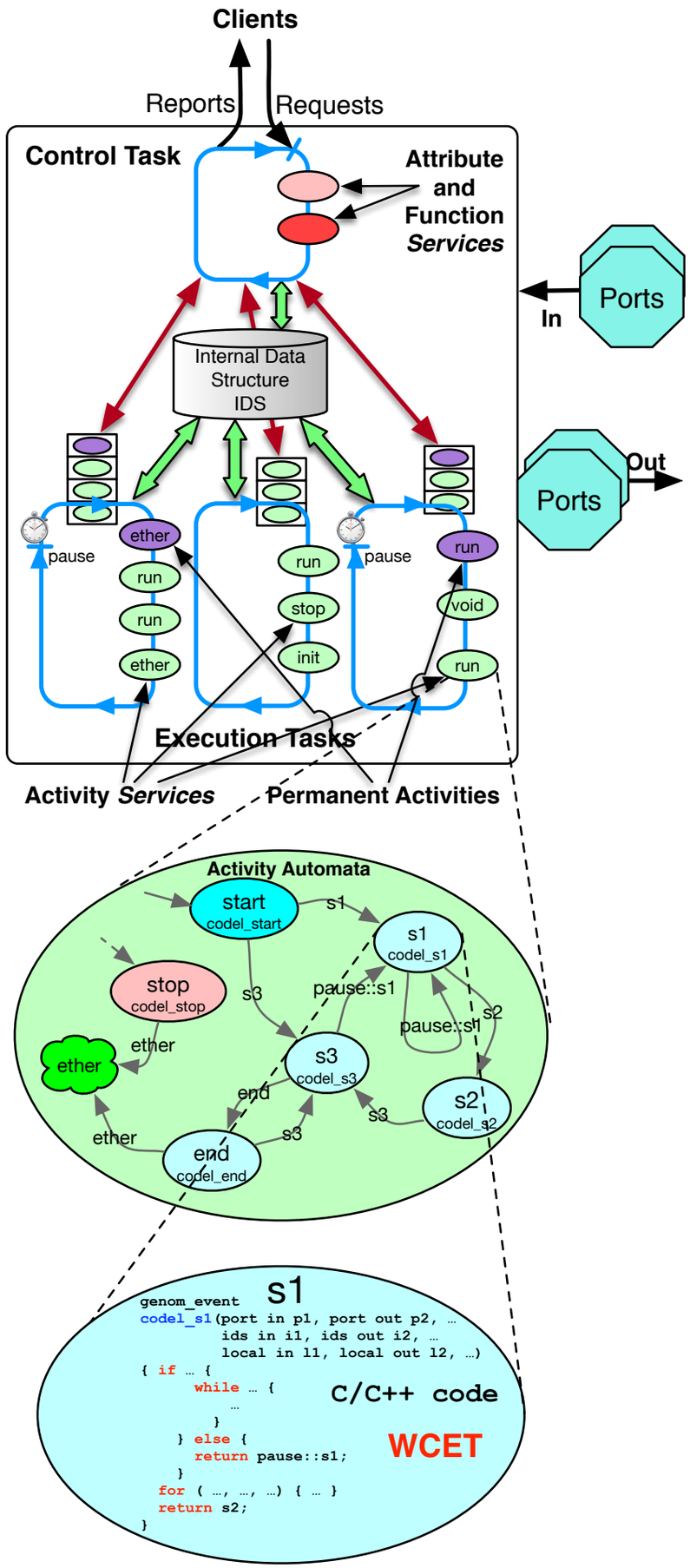
Plan UPSITECH

- 8 hours course, 4 hours BE
- Running examples (mobile robots, drone) -> to the final code, simulation **and** on the real robots (code available to all, VMWare and Docker).
- Robot software integration and architecture
- MiddleWare (ROS, ROS2, PocoLibs)
- Emphasis on ROS/ROS2 MW and Tools (**BE**)
- GenoM (**BE**)
- Validation and Verification of robotic systems
- V&V of functional components/layer (**BE**)
- Deployment of verification on the two running examples (**BE**)

The big picture… architecture and tools

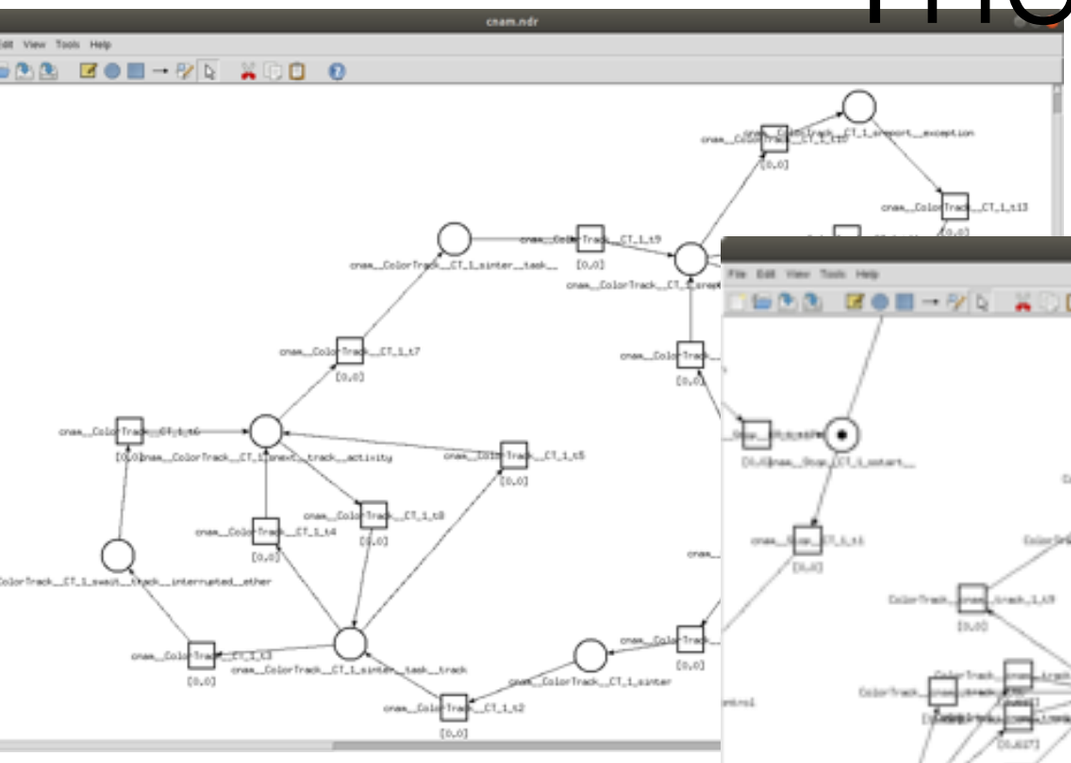


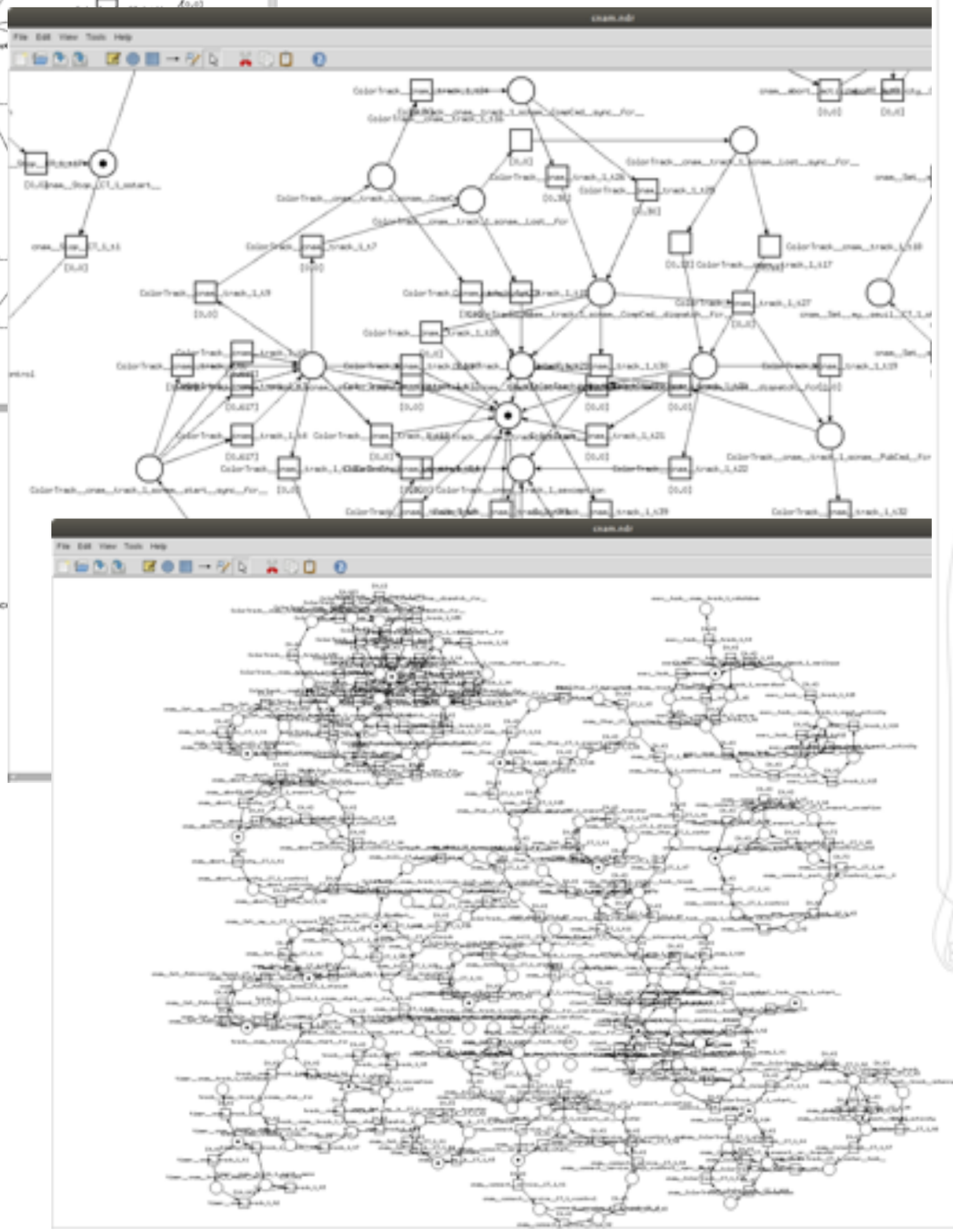


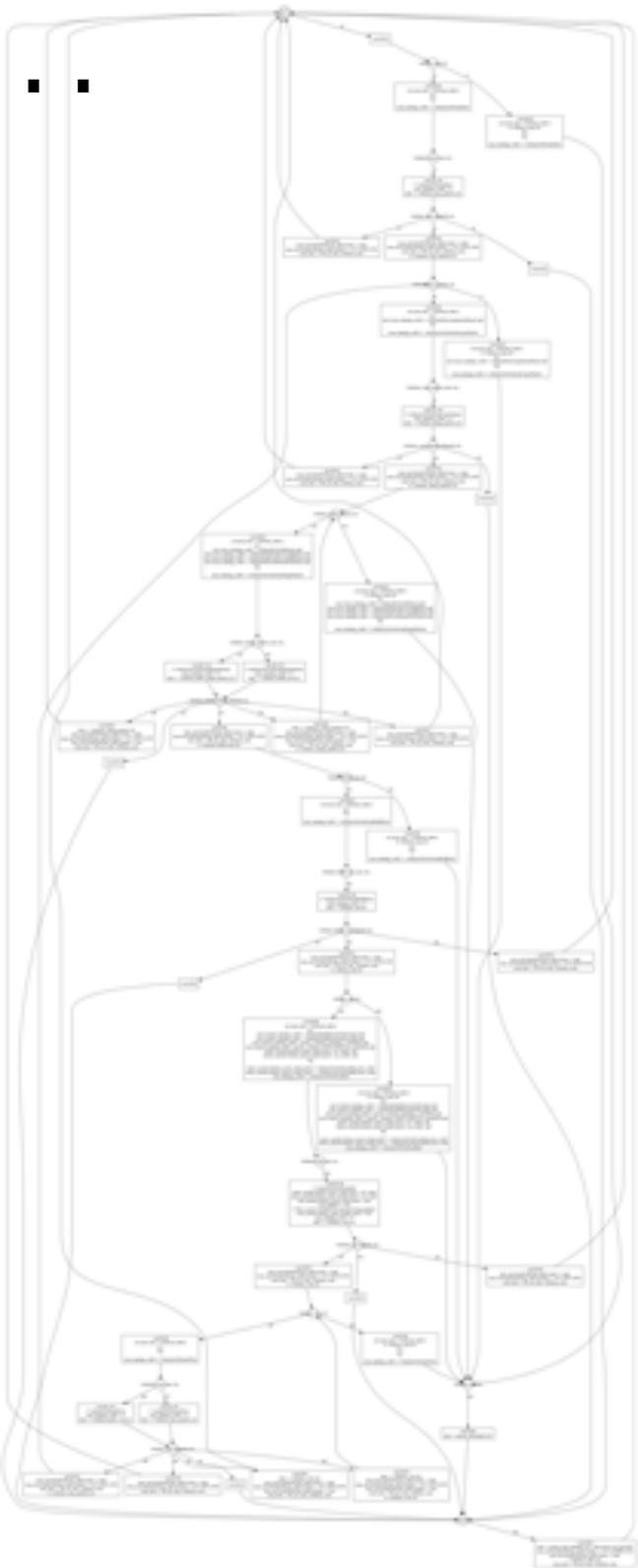


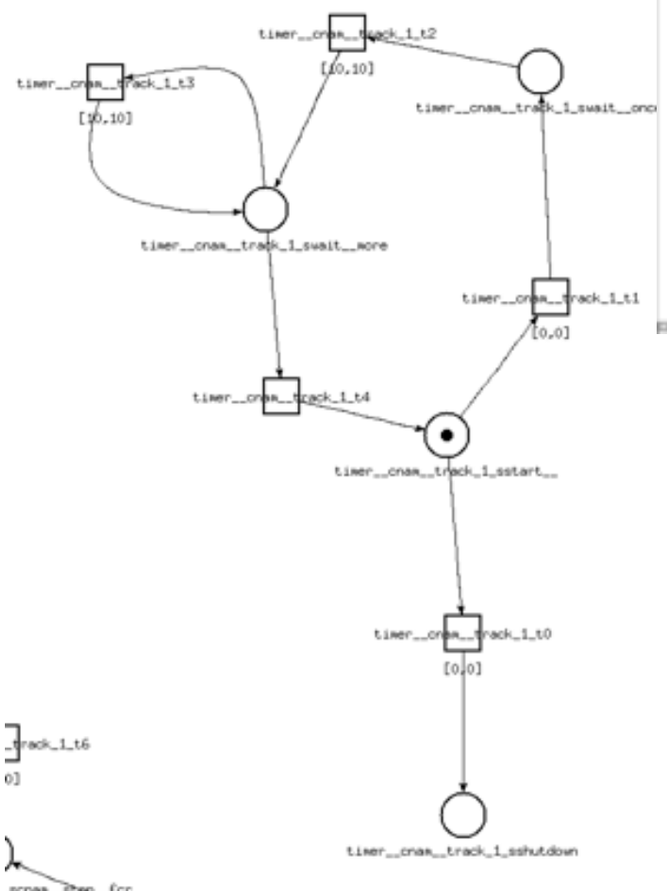
How do you verify it?

The big picture…
V&V Tools





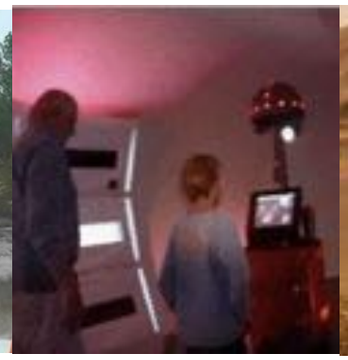




“Desired” Prerequisites

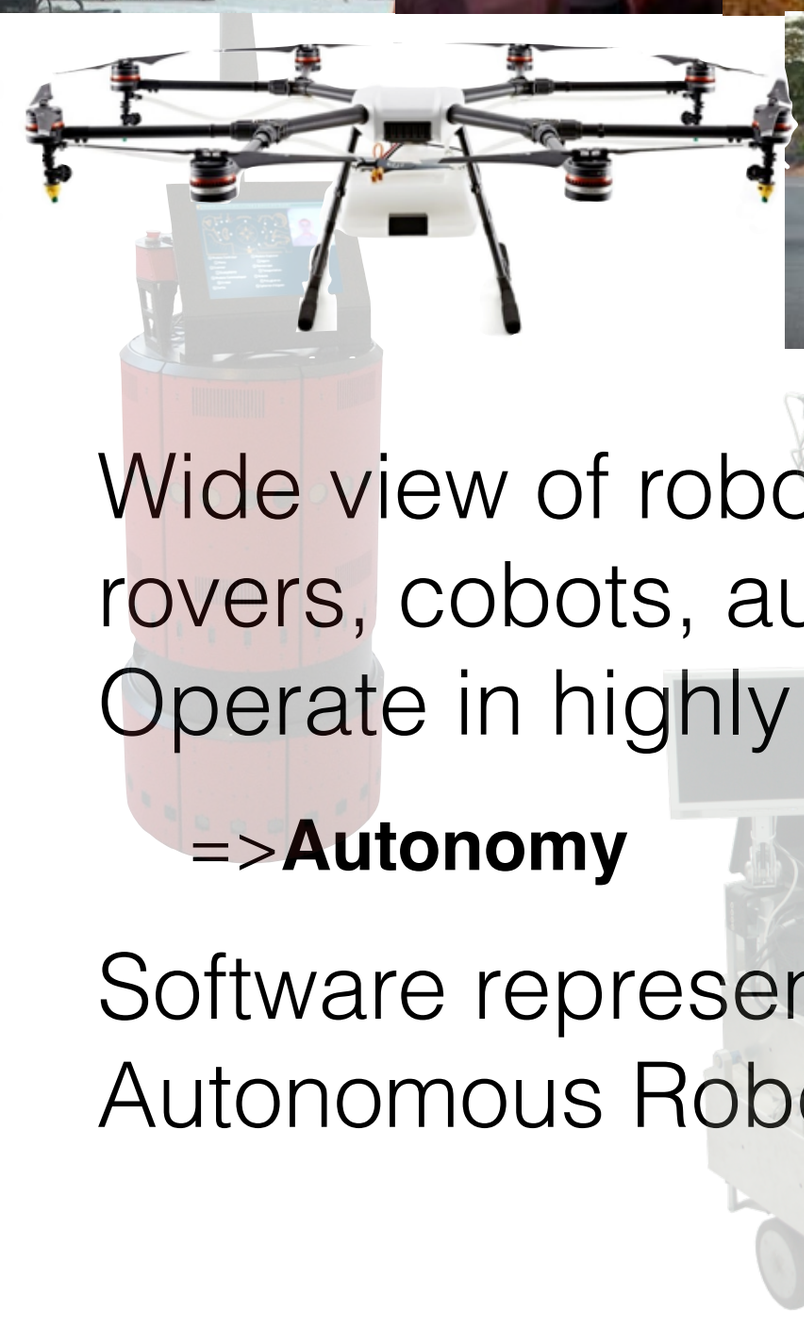
- Programming C/C++
- Real Time programming (e.g., Posix, thread, mutex)
- Formal models and methods
  - Petri net
  - Linear Temporal Logic

Autonomous “Robots” and Software









Wide view of robotics: service robots, exploration rovers, cobots, autonomous cars, drones, etc
Operate in highly variable and uncertain environments:

=>**Autonomy**

Software represents a large part of the development of Autonomous Robots

Integrating software

- Robotic software is a lot about integrating software
- Platforms are often Unix/Linux based
- RealTime OS?
- More and more robots have more than one computer and/or CPU
- More than one programs, and most of them are multi-threaded (multi-core processor)

Integrating software

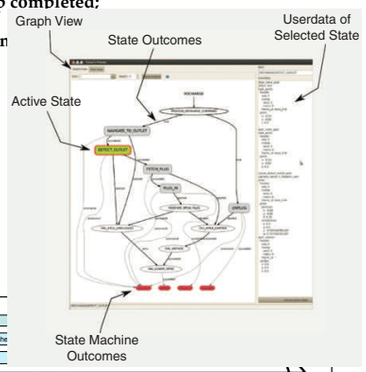
- Limitation of the computing platform
  - Enough CPU power/cycles?
  - Enough resources (memory, disk, bus, network bandwidth, etc)?
- Hardware integration (low level controller, drivers, sensors, etc)
  - bus (e.g. CAN)
  - serial lines, USB, etc
  - ethernet (TCP UDP)

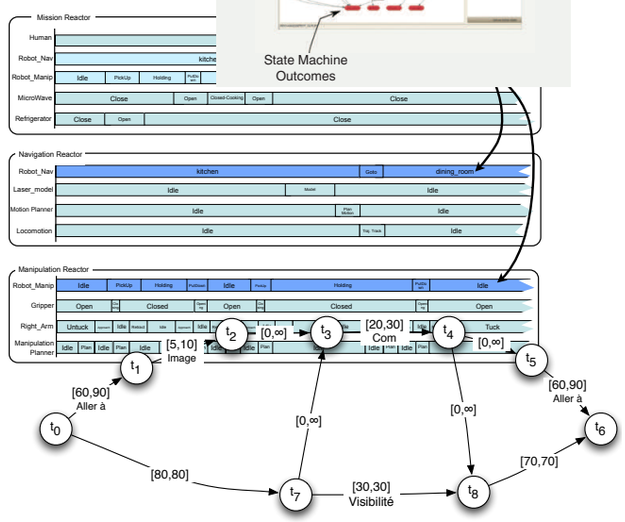
Complexity of integration

- Autonomous Robot “Software” are intrinsically heterogeneous:
  - built from components with different characteristics:
    - computational model (servo loop, tree/ graph search, automata, etc)
    - real-time requirements (hard real-time, anytime, etc)
    - interaction models (synchronous/ asynchronous, rendez-vous, broadcast, data stream, client-server, publish-subscribe, etc)

```
Goal deliverMail (int room)
{
  double x, y;
  getRoomCoordinates(room, &x, &y);
  spawn navigateToLocn(x, y);
  spawn centerOnDoor(x, y)
     with sequential execution previous,
       terminate in 0:0:30.0;
  spawn speak(“Xavier here with your mail”)
     with sequential execution centerOnDoor,
       terminate at monitorPickup completed;
  spawn monitorPickup()
     with sequential execution cen
}
```

```
(defop PickUpTray
 :invocation_part (! (PickUpTray $tray))
 :context (RobotLocation $robotLoc)                    ; The refinement depends of where
                                                       ; is the robot (kitchen or living_room)
 :body ((! (ArmsServerInit))                           ; Start the ArmServer
        (// ((! (Untuck LeftArm)))                     ; Untuck both arms
            ((! (Untuck RightArm))))                   ; in parallel
        (IF (? (EQUAL $robotLoc kitchen))              ; Robot is in the kitchen
          (! (TorsoMove Up))                           ; we grab the tray from the kitchen counter
         ELSEIF (? (EQUAL $robotLoc living_room))      ; Robot is in the living_room
          (! (TorsoMove Down))                         ; we grab the tray from coffee table
         ELSE
          ...
        (IF (? (Localize $tray $posla $posra))         ; Localize the tray
          (// ((! (MoveArm LeftArm $posla)))           ; Move both arms
              ((! (MoveArm RightArm $posra))))         ; in parallel
          (// ((! (CloseGripper LeftArm)))             ; Close the grippers
              ((! (CloseGripper RightArm))))           ; in parallel
          (! (JointMoveArmUp 20))
          ...
        <recovery when fail to localize the tray>
 ))
```

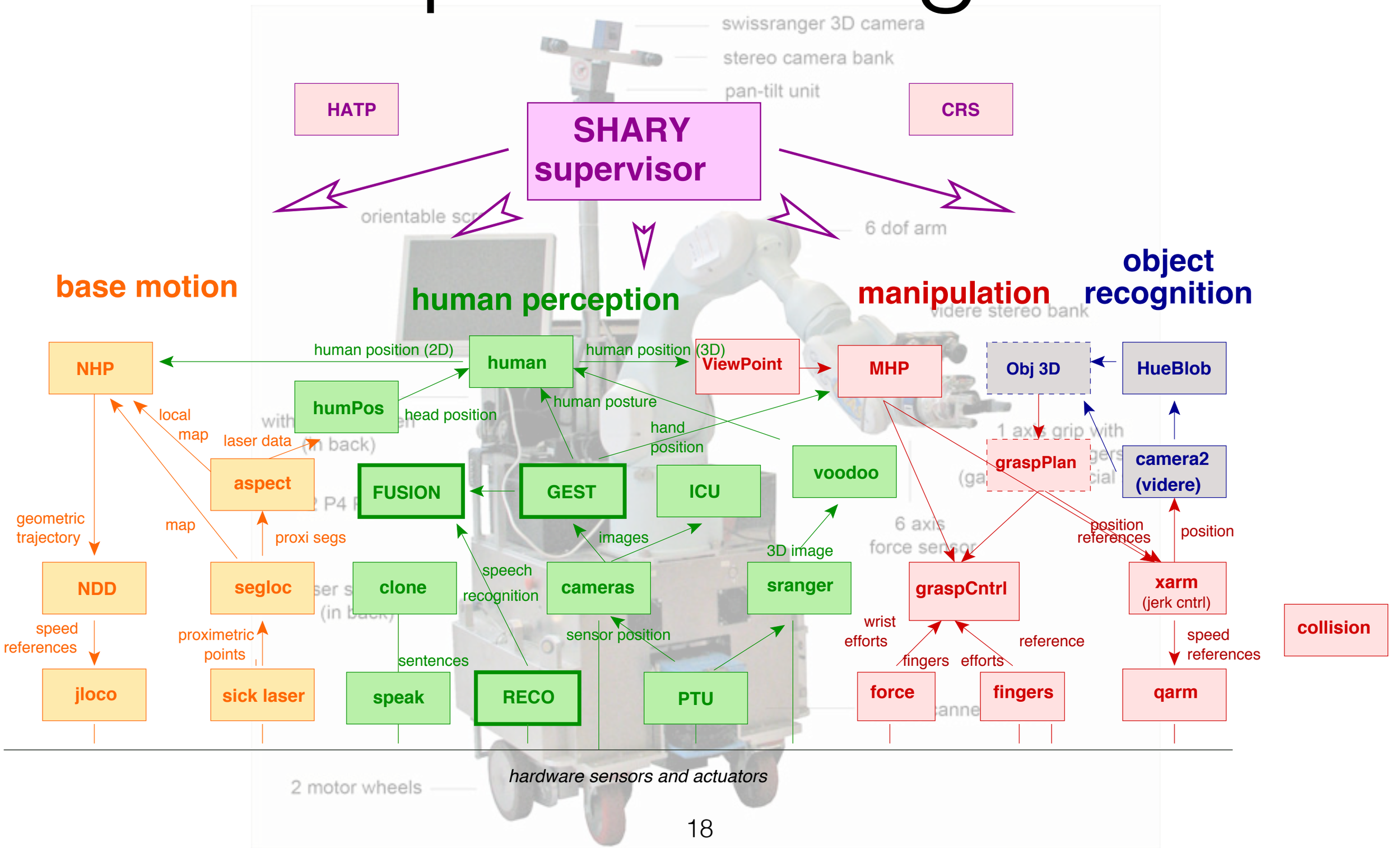




How do you program and integrate all these components, software, etc?

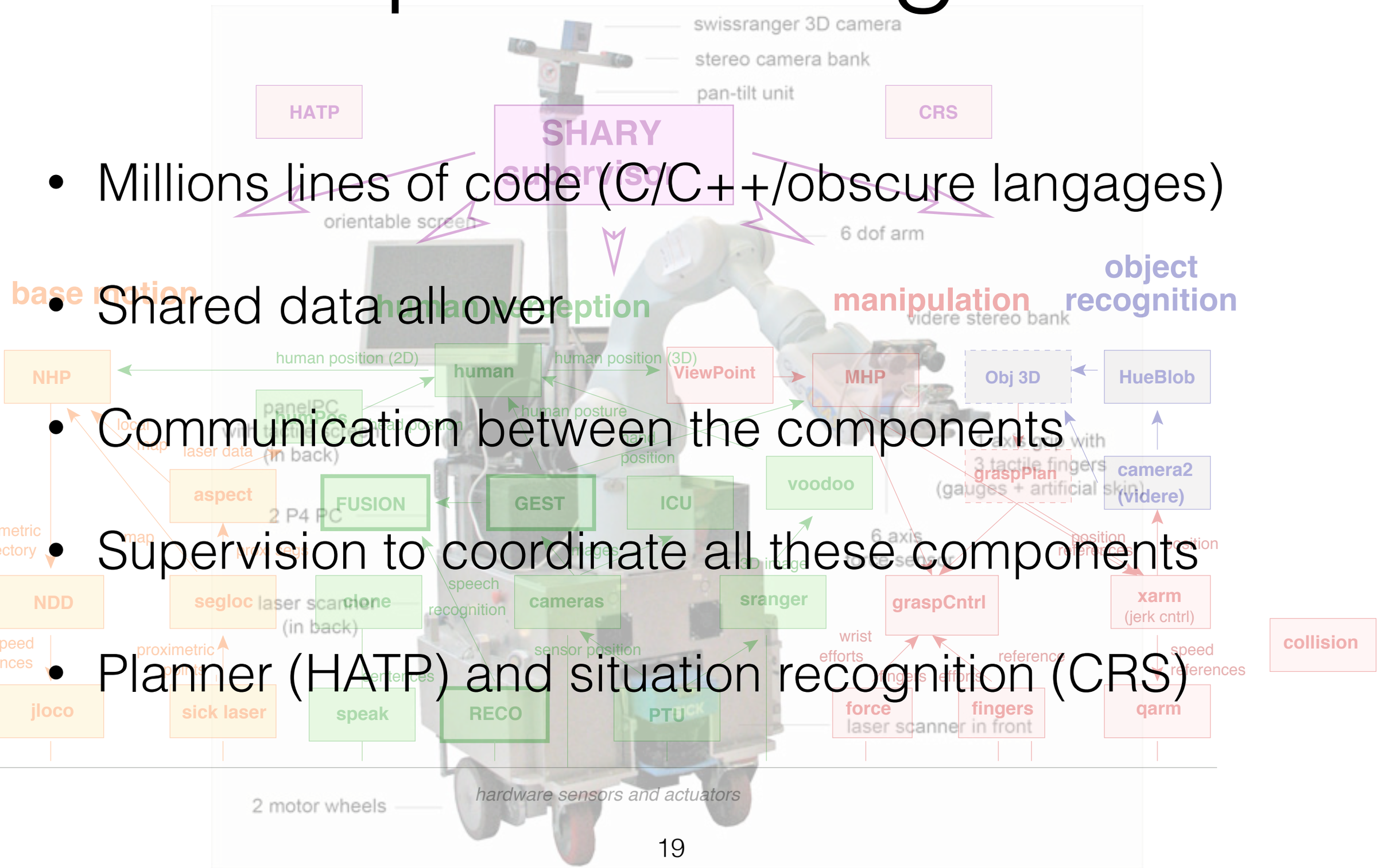
Example of integration

Example of integration



Example of integration

- Millions lines of code (C/C++/obscure langages)
- Shared data all over
- Communication between the components
- Supervision to coordinate all these components
- Planner (HATP) and situation recognition (CRS)



UBER car

Self Driving Uber sensor suite

7 Cameras
1 Laser
Inertial Measurement Units

Custom compute and data storage
360° radar coverage

Advanced Technologies Group

UBER

Need for an architecture

- How to organize the components with respect to:
  - timing constraints
  - resources needed (CPU, driver access, etc)
  - modularity, components based
- Provide a unified communication between the components
- Sharing data between the components
- Provide overall control of the platform

Need for an architecture

Preserve:

- reusability,
- portability,
- evolution,
- modularity,
- traceability,
- maintainability

"Designing a robot architecture is much more of an art than a science."

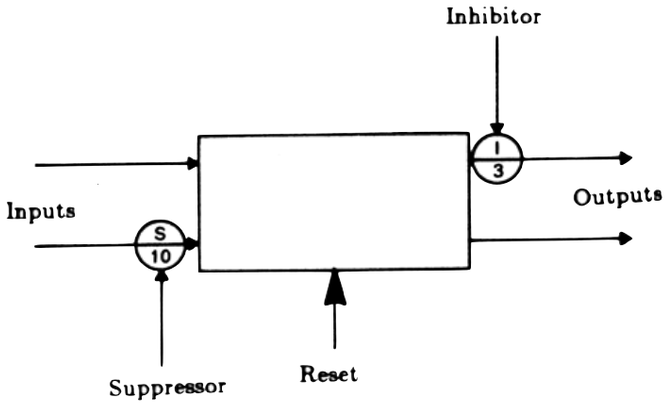
Handbook of Robotics, Robotic Systems Architectures and Programming.
David Kortenkamp, Reid Simmons

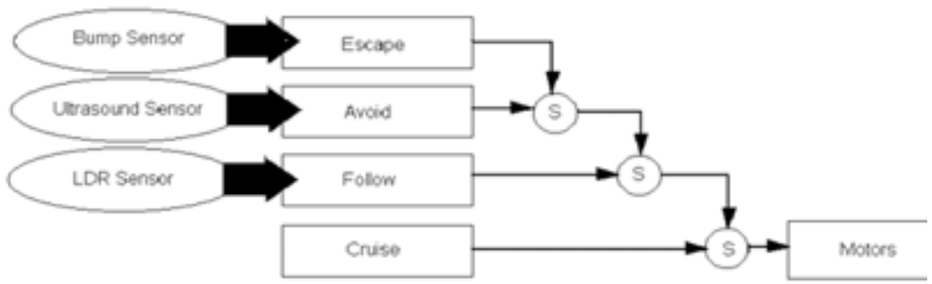
Robot Architecture

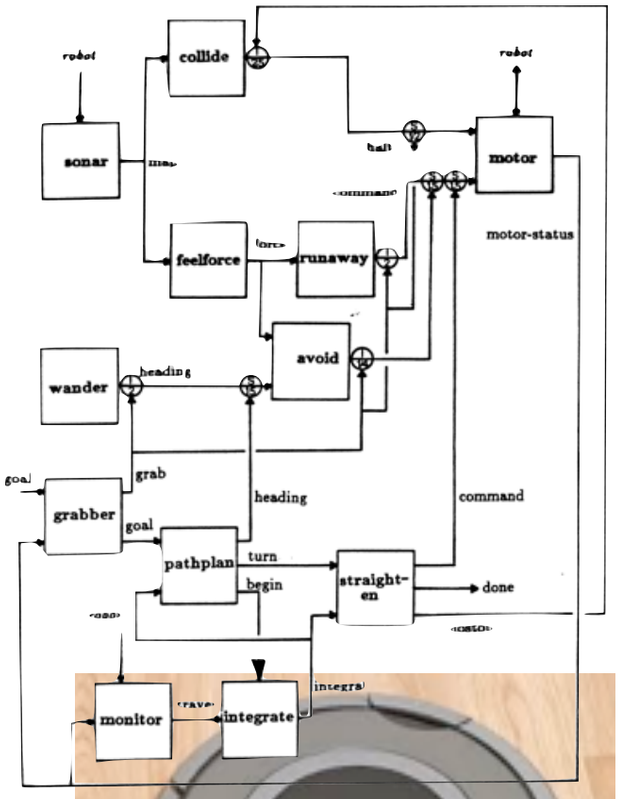
- How to organize the software along some guidelines supported by tools
- Hard to make a complete map of all the architectures available
- Some big families (subsumption, layered, teleo-reactive)
- Many specific architectures…

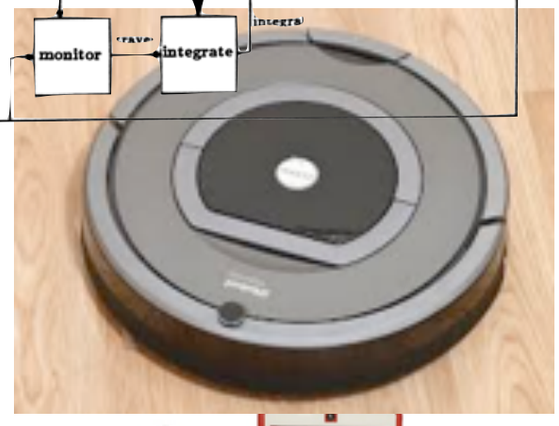
- But… tools and frameworks are really as important as architecture… if not more…

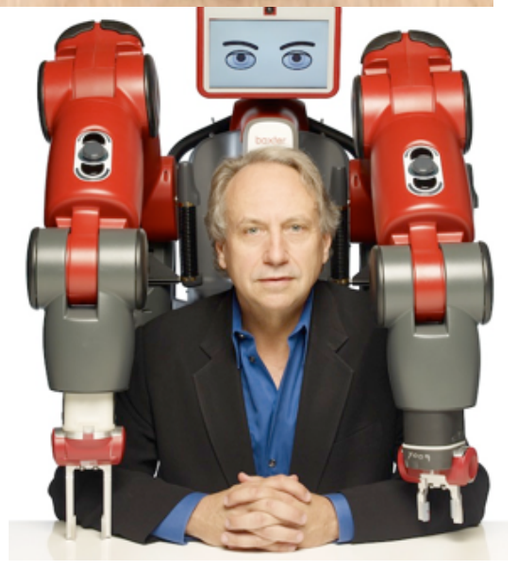
Subsumption Architectures





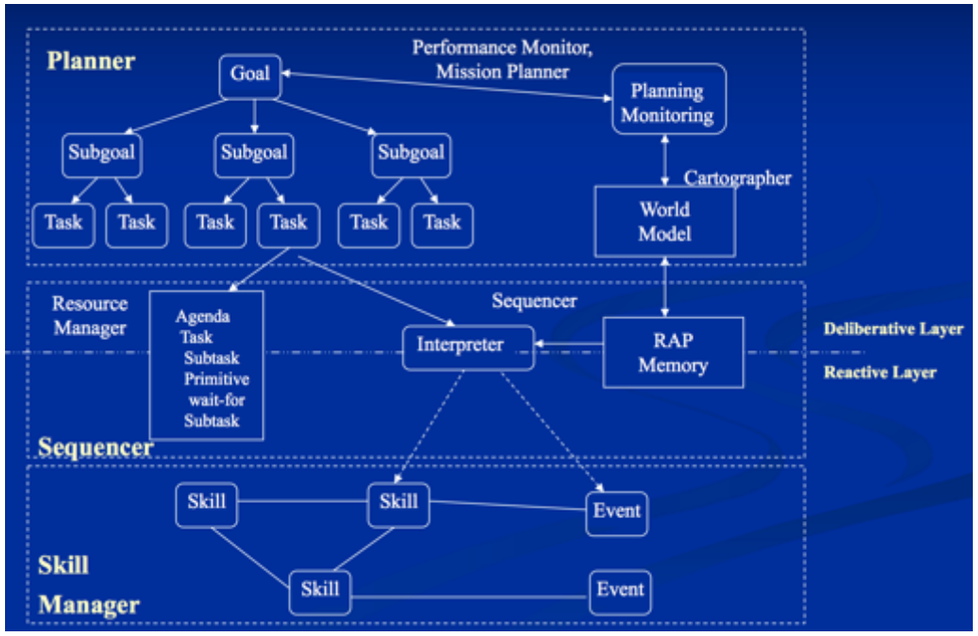


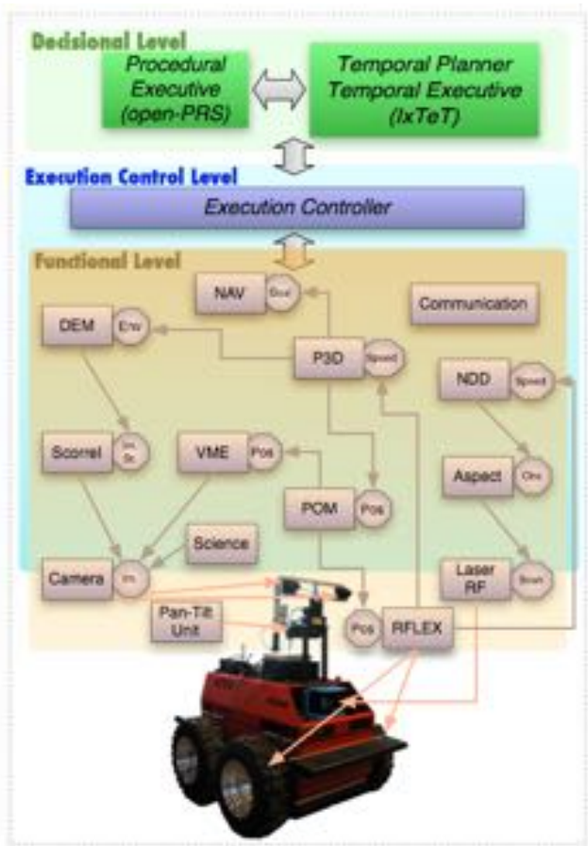


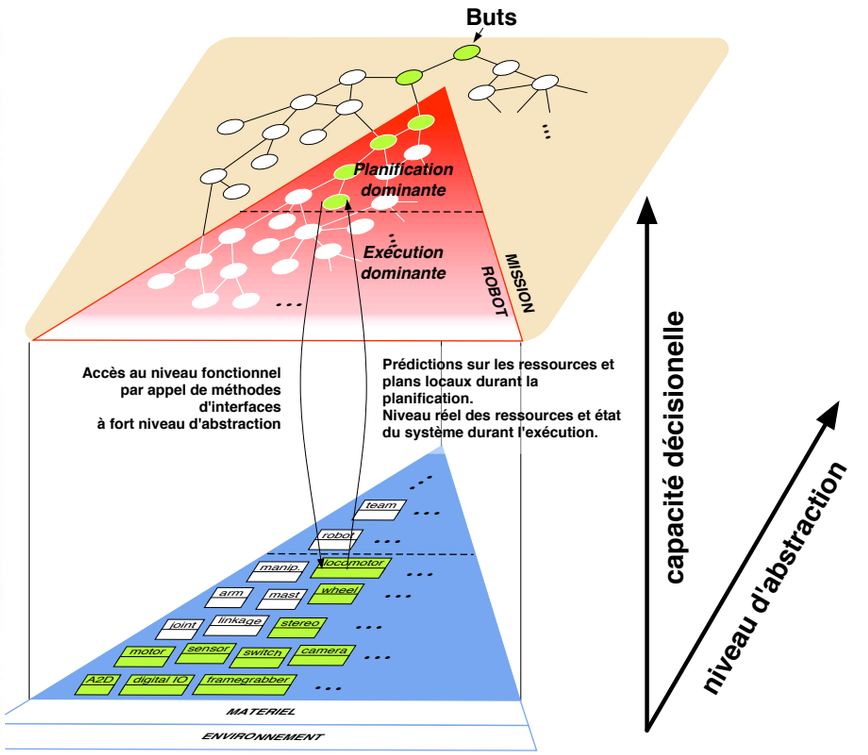


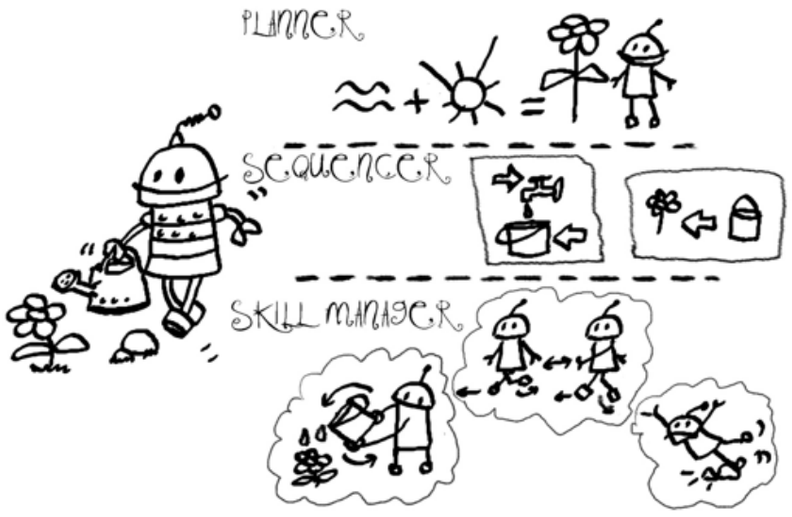
- Multiple components
- Arbitration required
- Popular and easy to setup for simple experiment
- Prove difficult to scale for complex experiments (e.g. an office mail delivery robot)

Layered Architectures





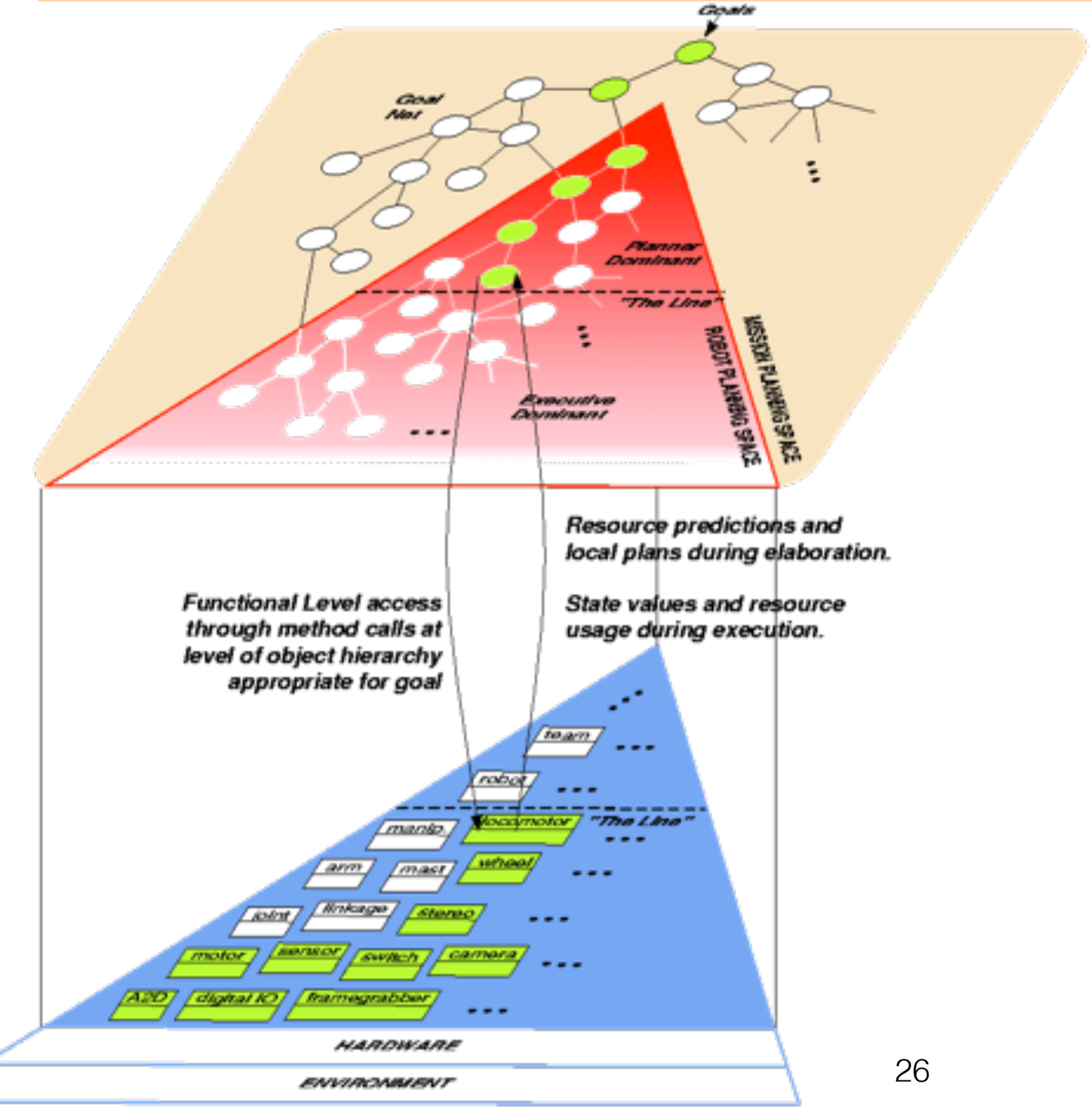




- Layered Architectures (3T, LAAS, CLARAty, Remote Agent, CIRCA, etc)
  - Layer and modular organization, communication, etc
  - 2 or 3 layers

CLARAty: A Two-Layered Architecture

CLARAty = Coupled Layer Architecture for Robotic Autonomy



THE DECISION LAYER:

Declarative model-based
Mission and system constraints
Global planning

INTERFACE:

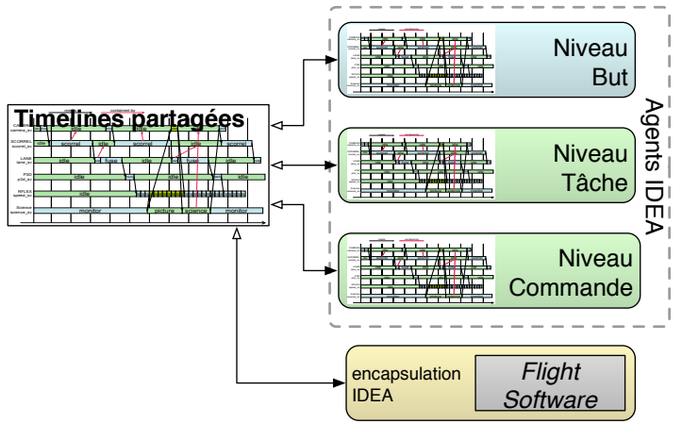
Access to various levels
Commanding and updates

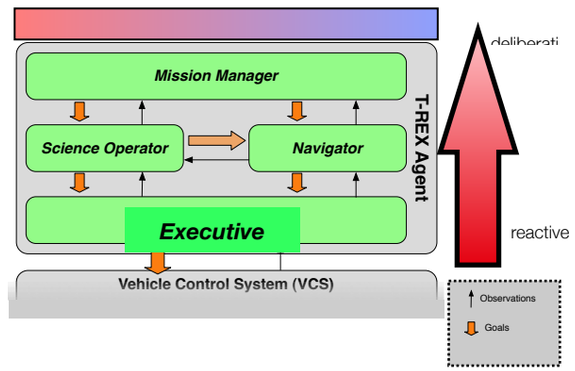
THE FUNCTIONAL LAYER:

Object-oriented abstractions
Autonomous behavior
Basic system functionality

Adaptation to a system

Teleo Reactive/Agent based Architectures

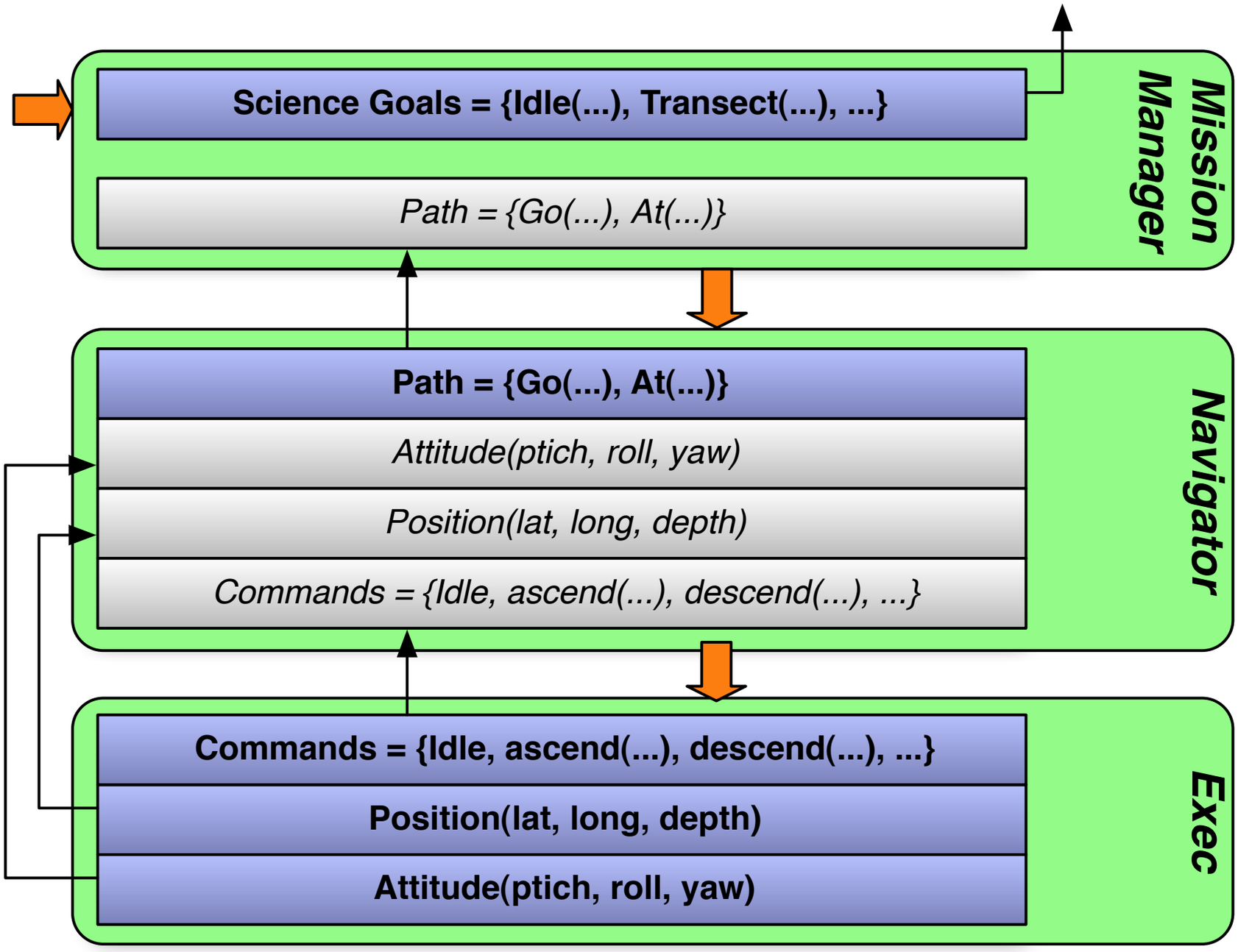




- IDEA and T-ReX
- agents (IDEA) or reactors (T-ReX)
- each is defined along a planning/execution paradigm
- each with a different planning horizon/time quantum

- State variables on timelines
- Temporal compatibilities between tokens on timelines
- Similar to constraint programming
- Allen temporal logic

T-ReX: Timelines and reactors

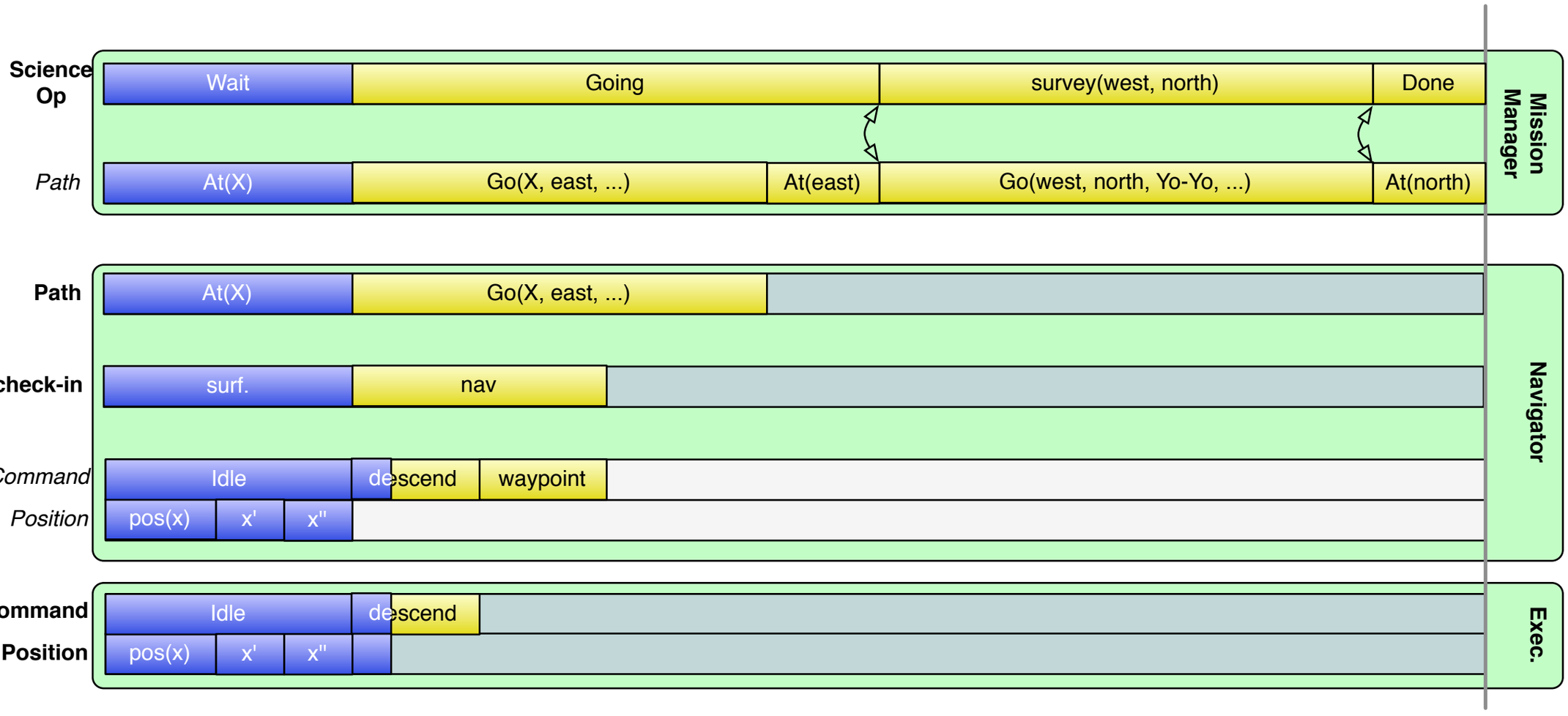


- π = 22 hrs
- λ = 20 s

- π = 60 s
- λ = 5 s

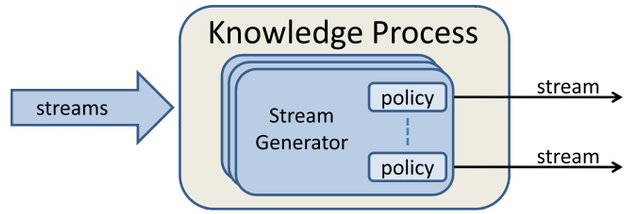
- π = 0 s
- λ = 0 s

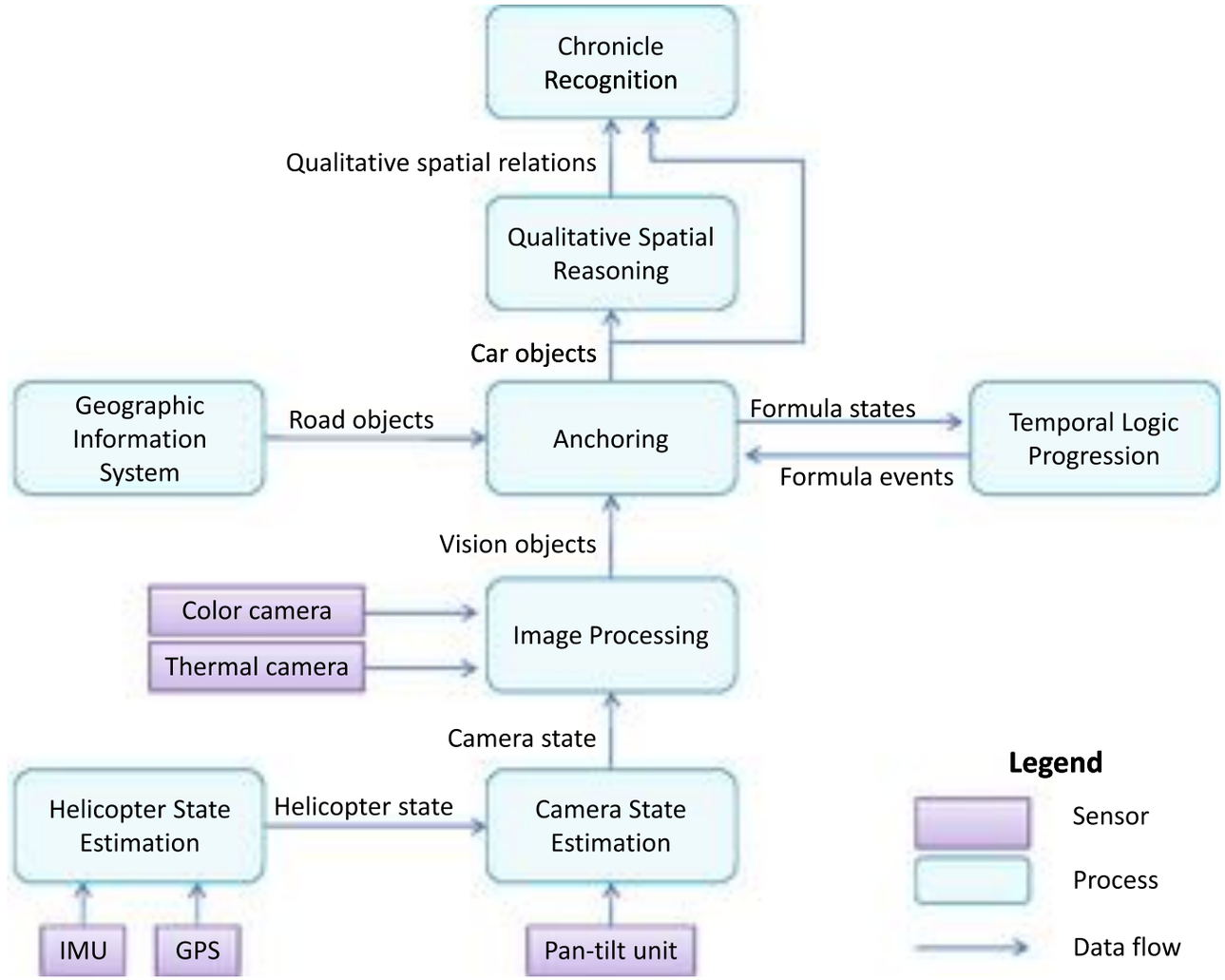
T-ReX: Example



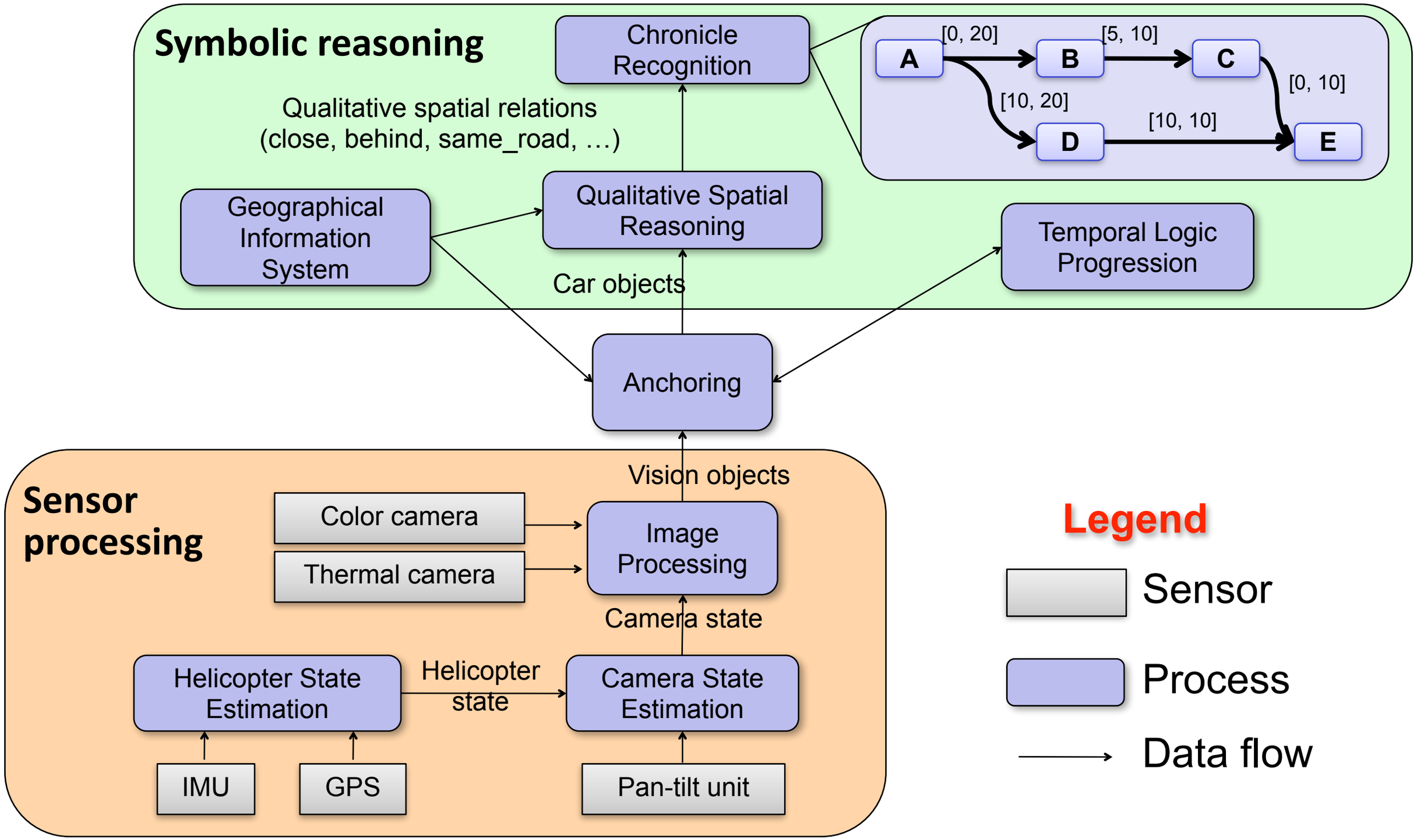
DyKnow: Stream Based architecture

- They build on a stream based formalism on knowledge process:
  - primitive, refinement, configuration, mediation KP
  - policies over processes (temporal constraints)
    - sample every
    - max delay
    - monotone/strict order
    - etc



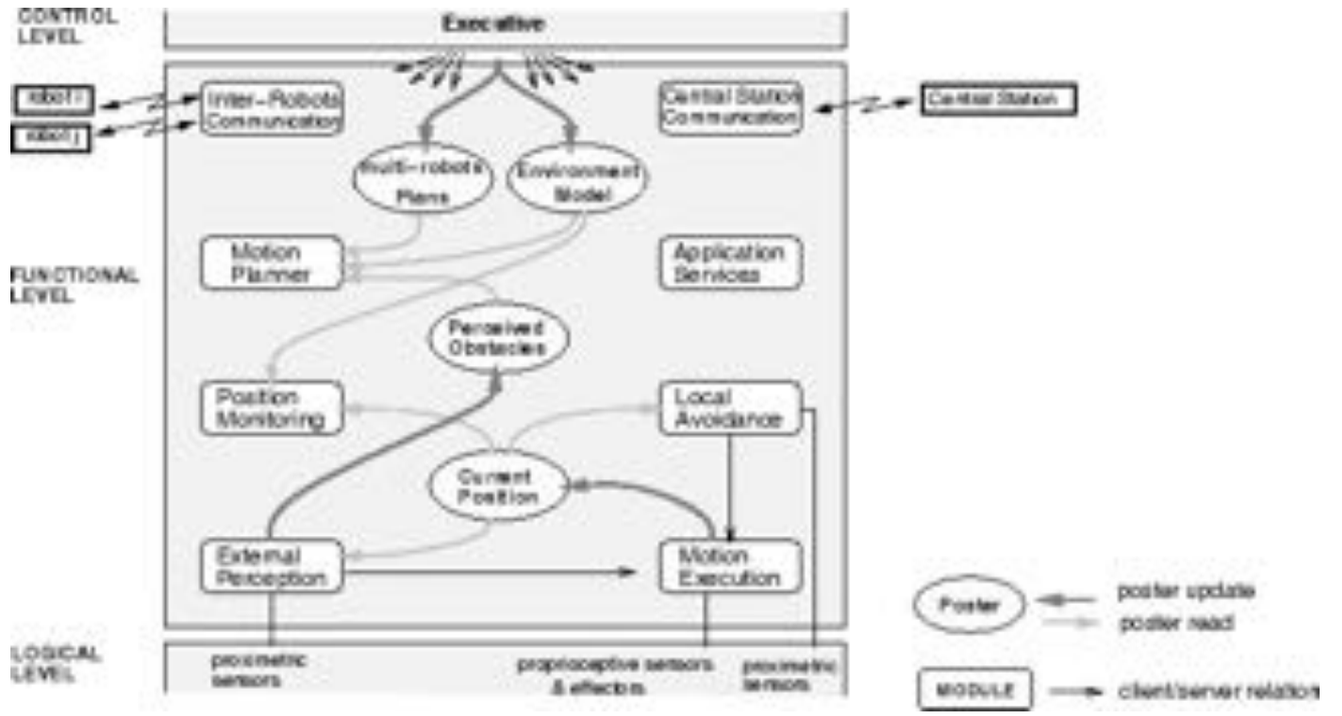


DyKnow Application: Traffic Monitoring



and many others…

- Multi robot architecture (e.g., Martha)

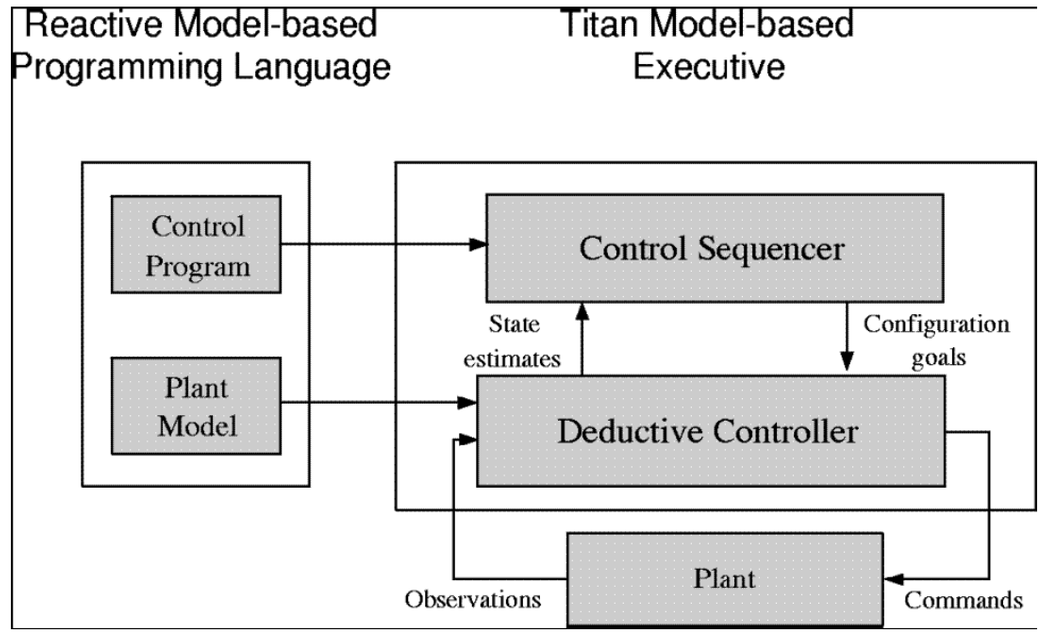


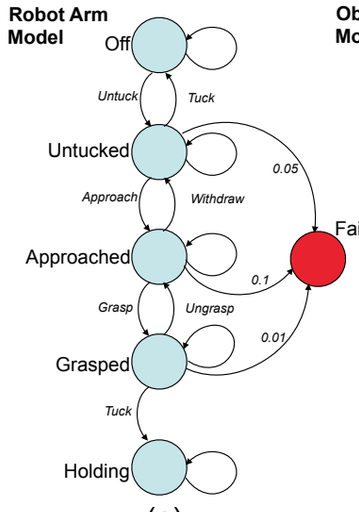
"Designing a robot architecture is much more of an art than a science."

Handbook of Robotics, Robotic Systems Architectures and Programming.
David Kortenkamp, Reid Simmons

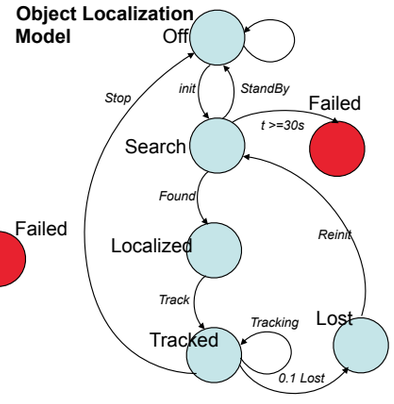
and many others…

- Model Based (e.g., RMPL)





(a)



(b)

```
PickUpObject()::
((do-watching (RA = holding)
  (do-watching (RA = Failed)
    (parallel
       (RA = Untucked)
       (OL = Search))
    (do-watching (OL = Failed)
       (OL = Localized))
    (when-donext (OL = Localized)
       (parallel
          (RA = Approached)
          (OL = Tracked))
    (when-donext  ((OL = Tracked) and
                   (RA = Approached))
                  (RA = Grasped)))
    )
    ;;; <Arm recovery>
  )
  ;;; Success
)
```

(c)

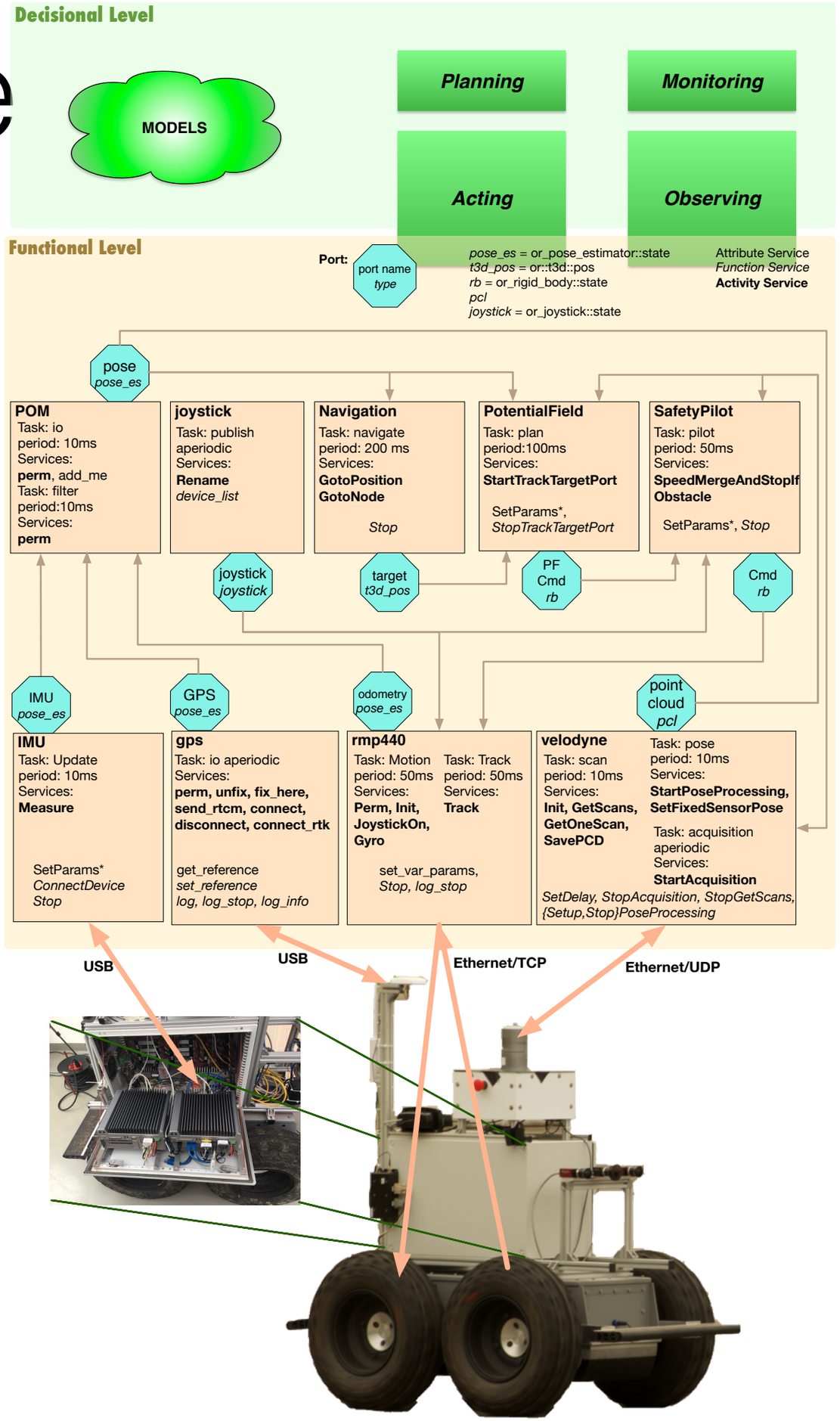
and many others…

“Designing a robot architecture is much more of an art than a science.”

Handbook of Robotics, Robotic Systems Architectures and Programming.
David Kortenkamp, Reid Simmons

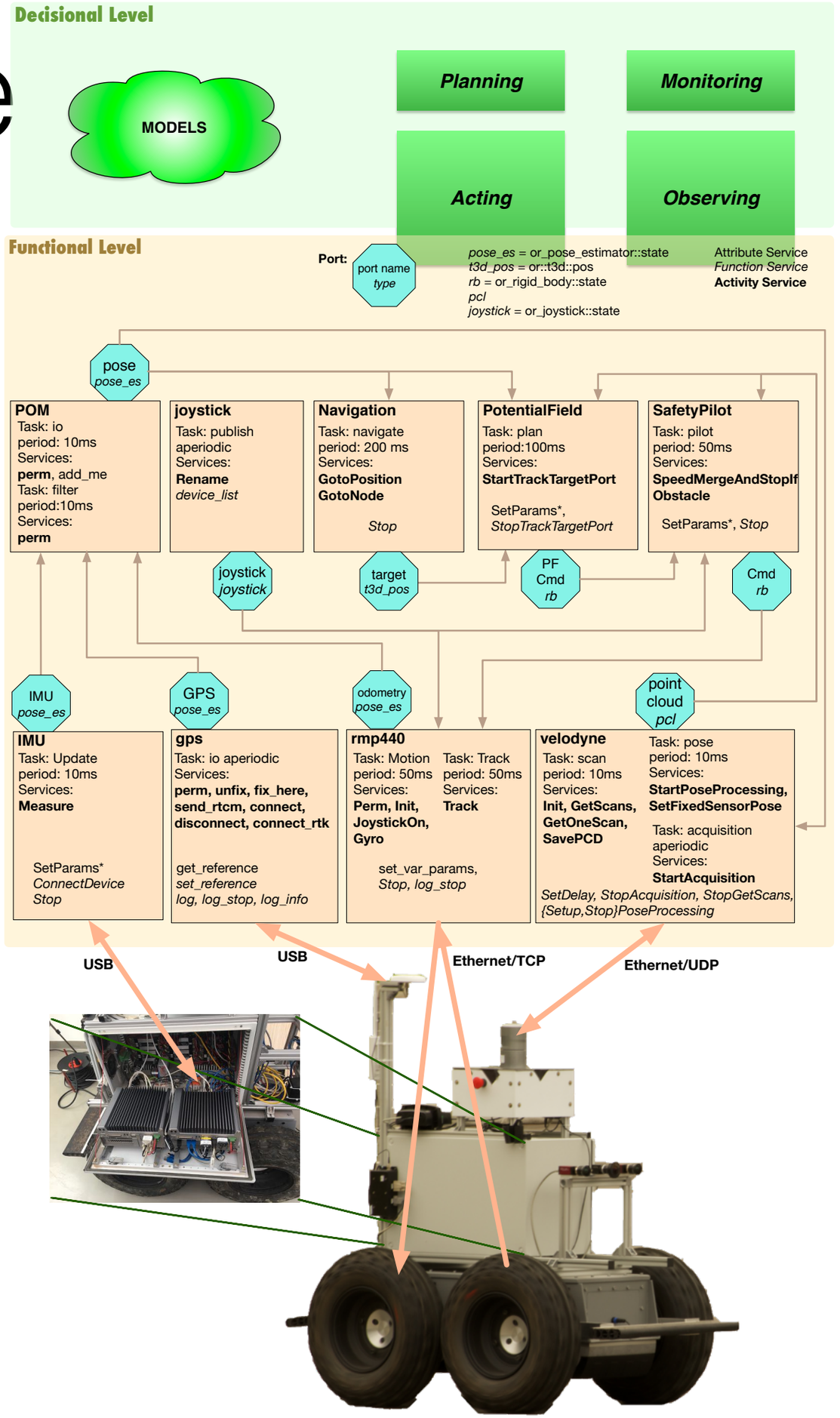
LAAS Architecture

- Has evolved over time and so have the associated tools
- Decisional level, now better defined
- Functional level
- Supported by tools, libraries and programs



LAAS Architecture

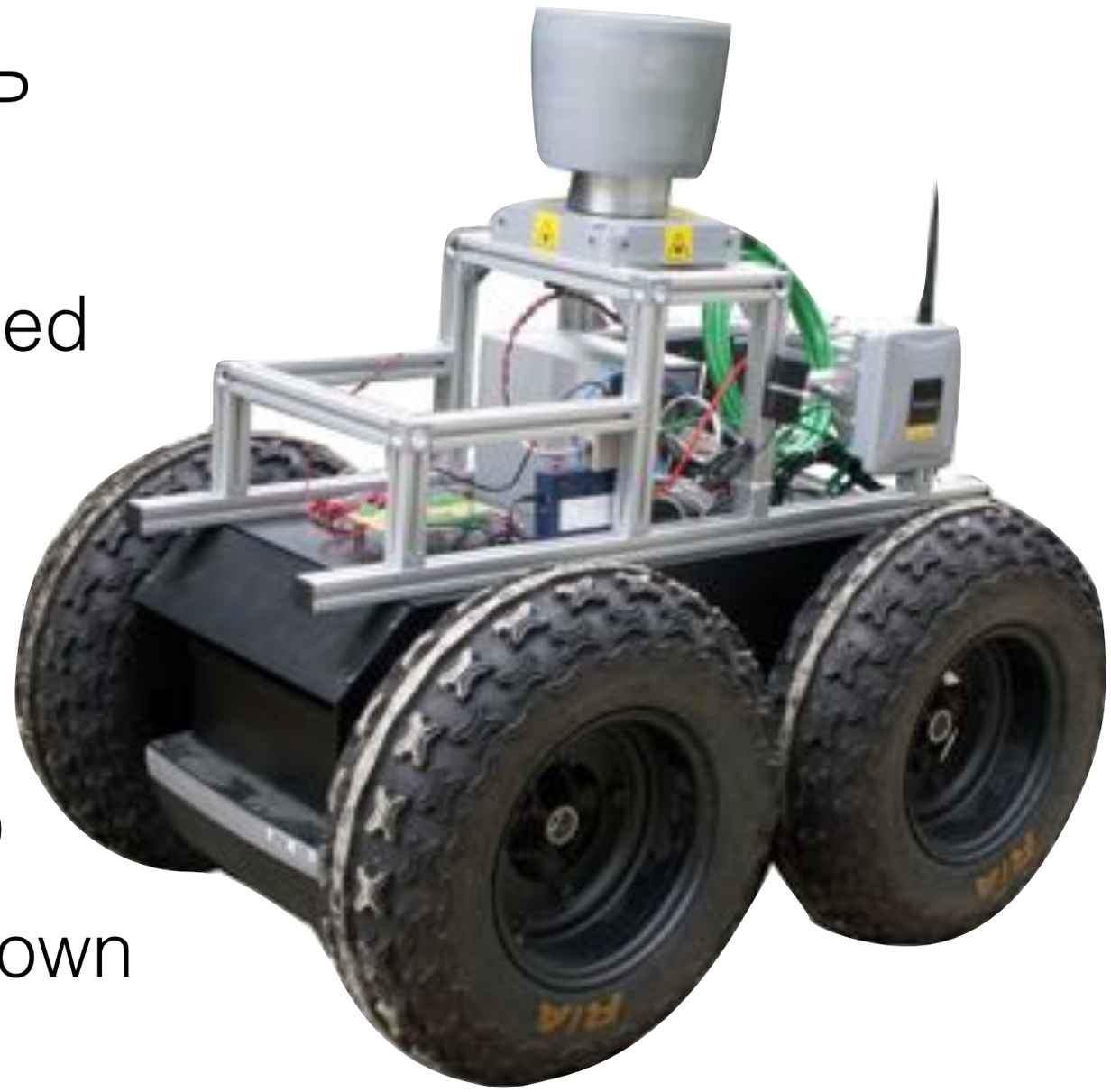
- Decisional Level
- Planning and Acting are now integrated
- Learning models



A Simple Example

Inspired from a real navigation LAAS Mana robot (SegWay RMP 400)

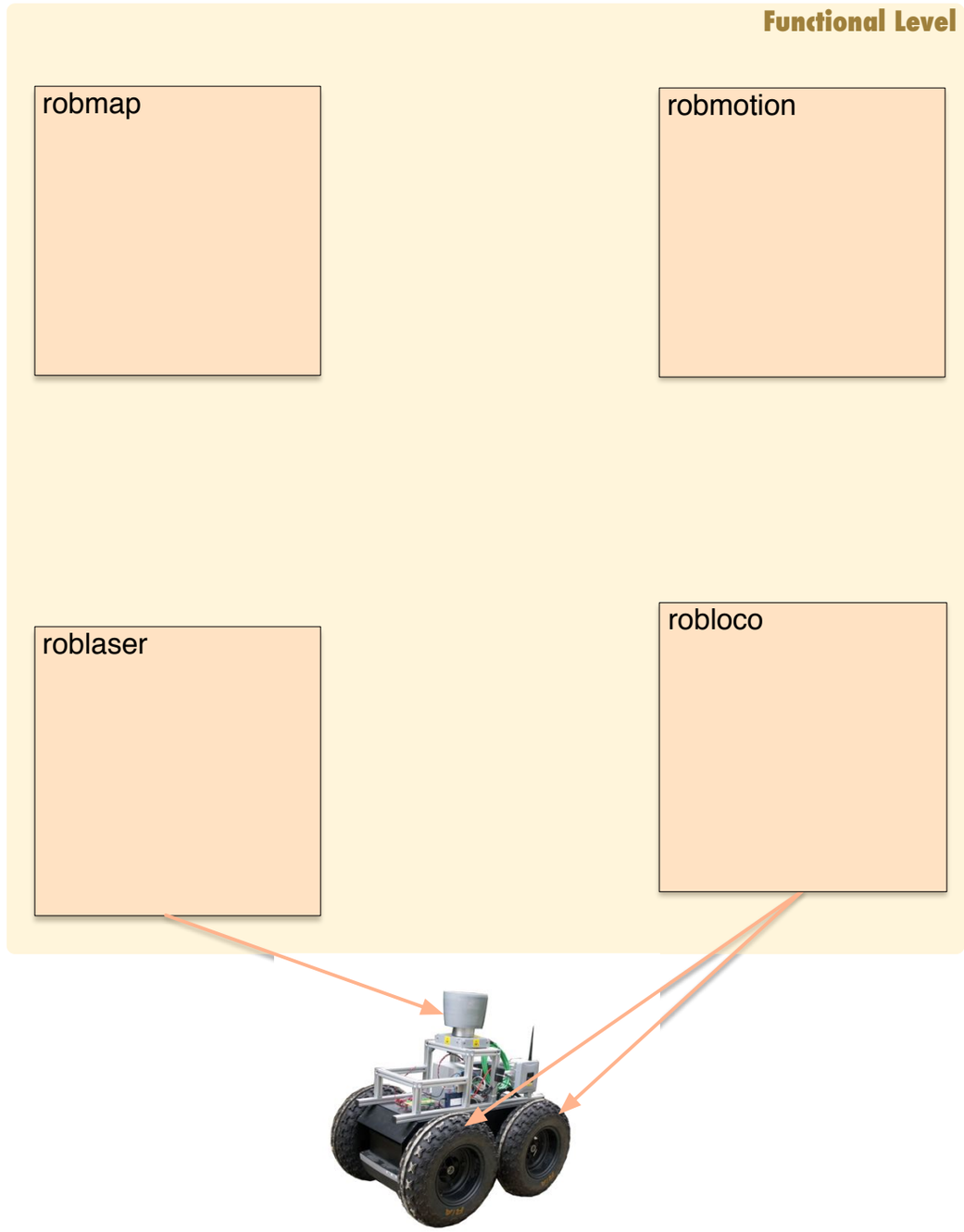
- A simple mobile robot equipped with:
  - wheels (**speed controller** and **odometry**)
  - laser range finder (scan for obstacles) (Velodyne 3D scan)
- Navigating in an a priori unknown environment with obstacles



Modules/Components

How many modules/components do we need?

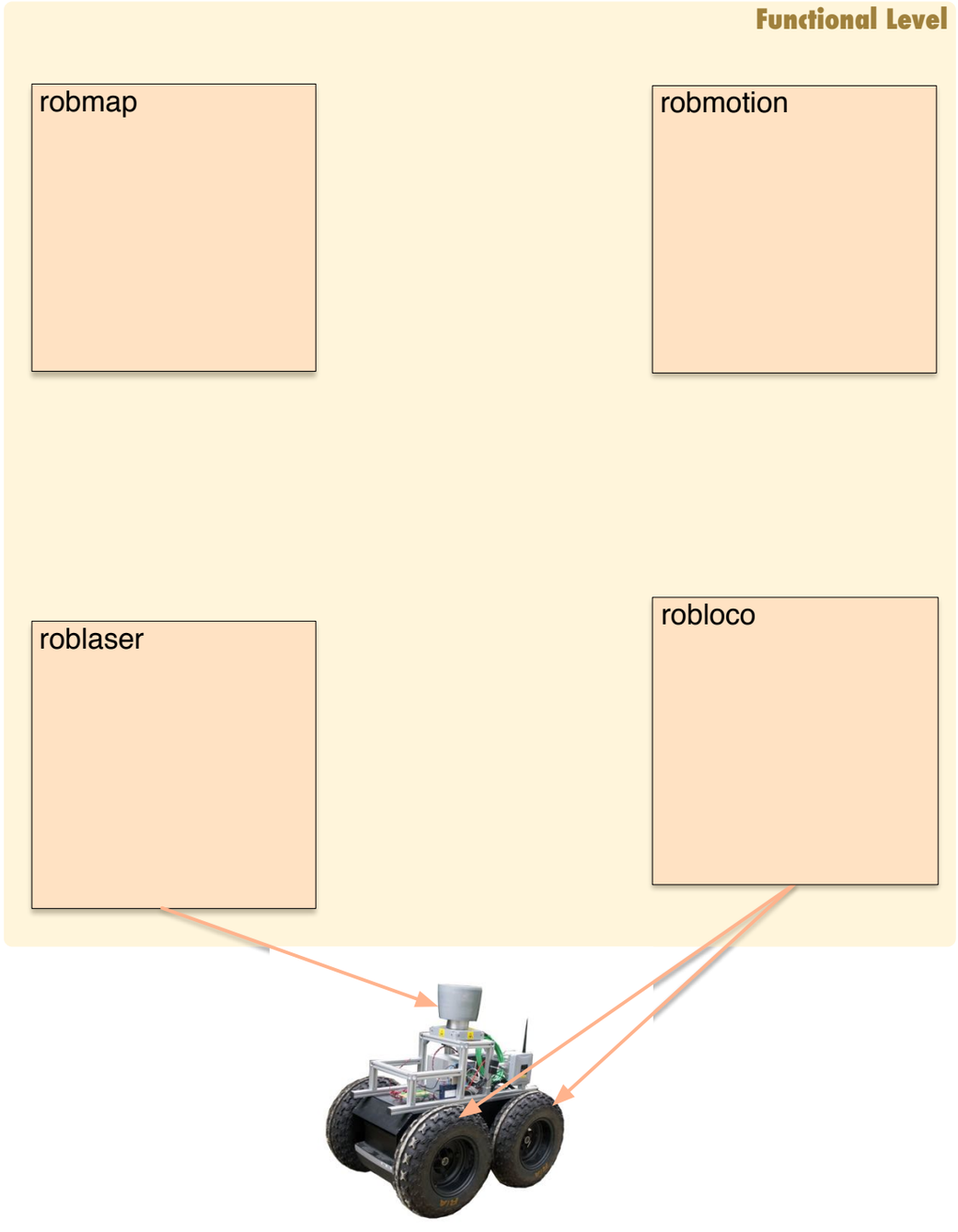
- Anything similar we already have?
- Any existing modules for a particular sensor/effector?
  - laser?
  - this robot HW
- Any existing modules for a particular perception, map building, etc



Modules/Components

How many modules/components do we need?

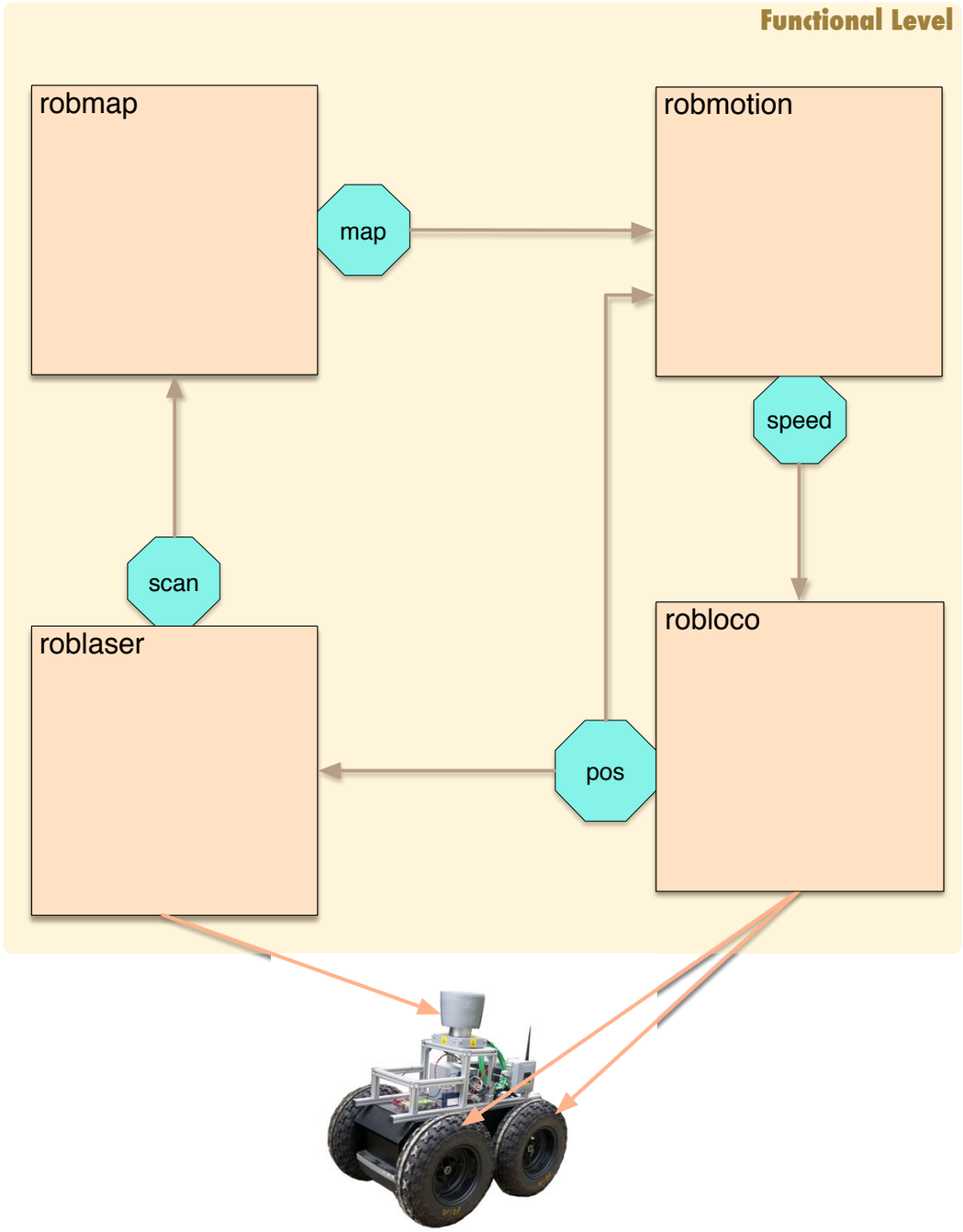
- **robloco** in charge of the wheels (applies a ***speed reference*** to the wheels, looks at the ***wheel encoders*** to estimate its displacement and position)
- **roblaser** in charge of the laser sensor, scans and returns a ***local scan*** of the free space in its range
- **robmap** builds a global ***map*** of the perceived env so far
- **robmotion** given a ***goal position***, produces a ***speed reference*** which lead the robot to the goal while avoiding obstacles in the ***map***



Modules produce and use data

Which data need to be shared?

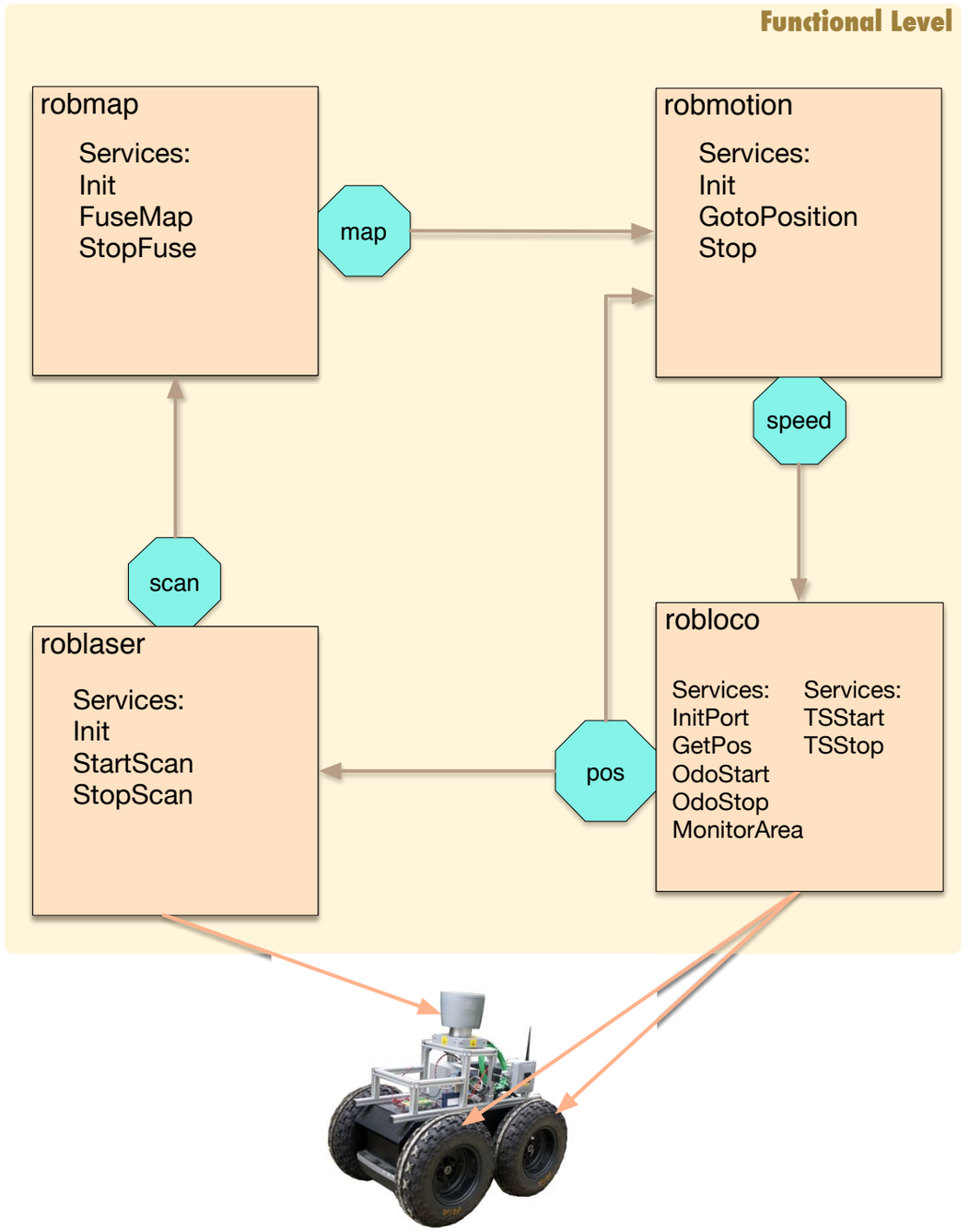
- **robloco** produces the current position *(pos)*, and uses the speed reference *(speed)*
- **roblaser** produces the laser scan *(scan)*, and uses the current position *(pos)* (to tag the laser scan)
- **robmap** produces the explored map *(map)* and uses the laser scan *(scan)*
- **robmotion** produces the speed_reference *(speed)* and uses the current_position *(pos)*



Modules provide services to “client”

Which services do we need?

- **robloco** track_speed_ref, track_odo, stop_robot, monitor_area, etc
- **roblaser** start_laser_scan, stop_laser_scan,
- **robmap** start_map_fuse, stop_map_fuse, print_map
- **robmotion** goto_position, stop

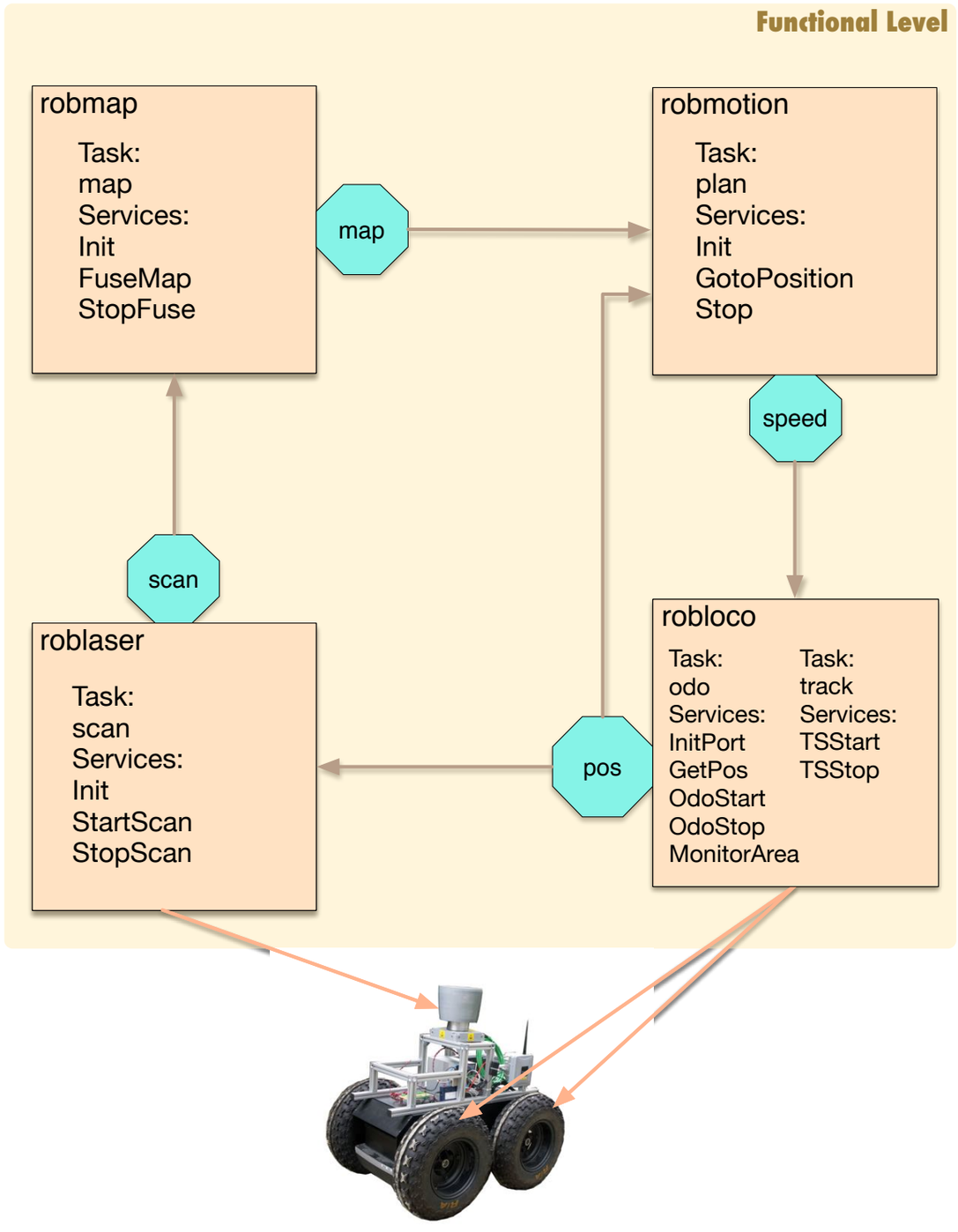


Services execute in tasks

Consider the required parallelism

Tasks (threads) per module (consider the required parallelism)

- **robloco** has two tasks, ***track*** (periodic) in charge of managing the *speed_reference*; ***odo*** (aper) in charge of odometer (read from the encoder) and the *current_position*
- **roblaser** has one periodic task, ***scan*** which handles the *laser_scan*
- **robmap** has one periodic task, ***map***
- **robmotion** has one periodic task, ***motion***

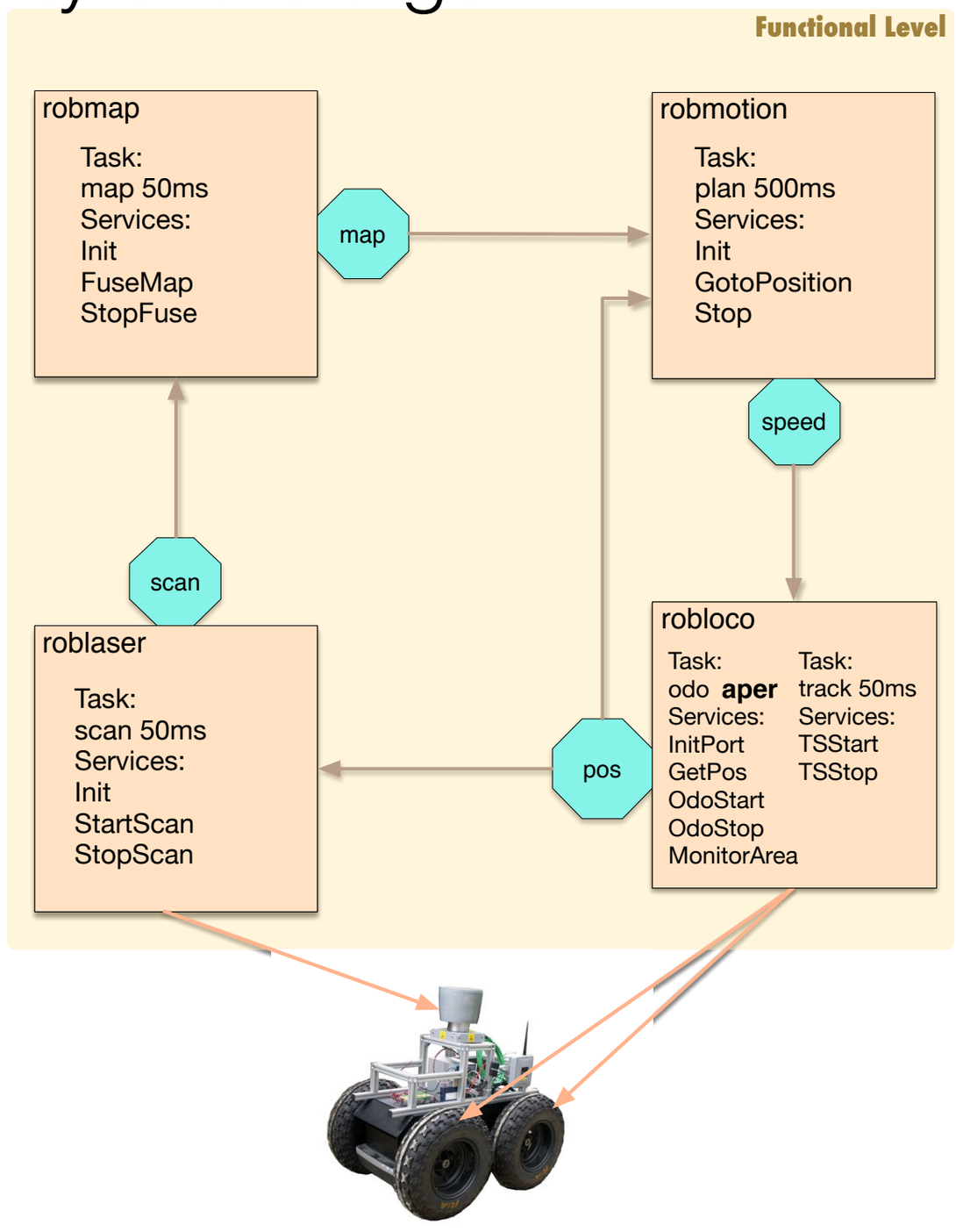


Periodic tasks period

Consider how fast these data are or need to be produced, but also the complexity of the algo.

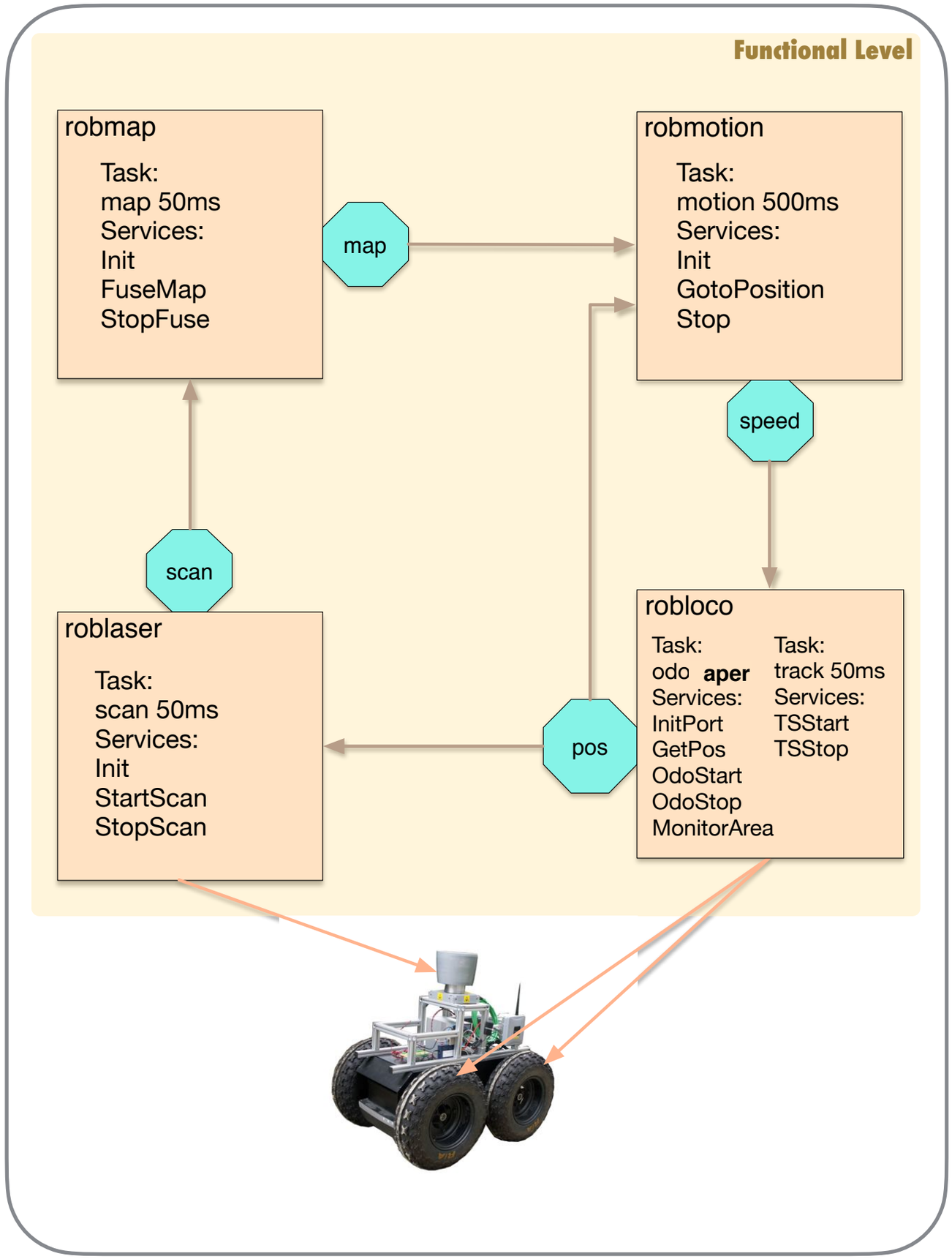
Tasks (threads) period.
Values are “application/hardware” defined.
This is “presumably” the period at which they will read and write external data.

- **robloco**, ***track*** 0.05s; ***odo*** aper
- **roblaser**, ***scan*** 0.05s
- **robmap**, ***map*** 0.05s
- **robmotion**, ***motion*** 0.5s



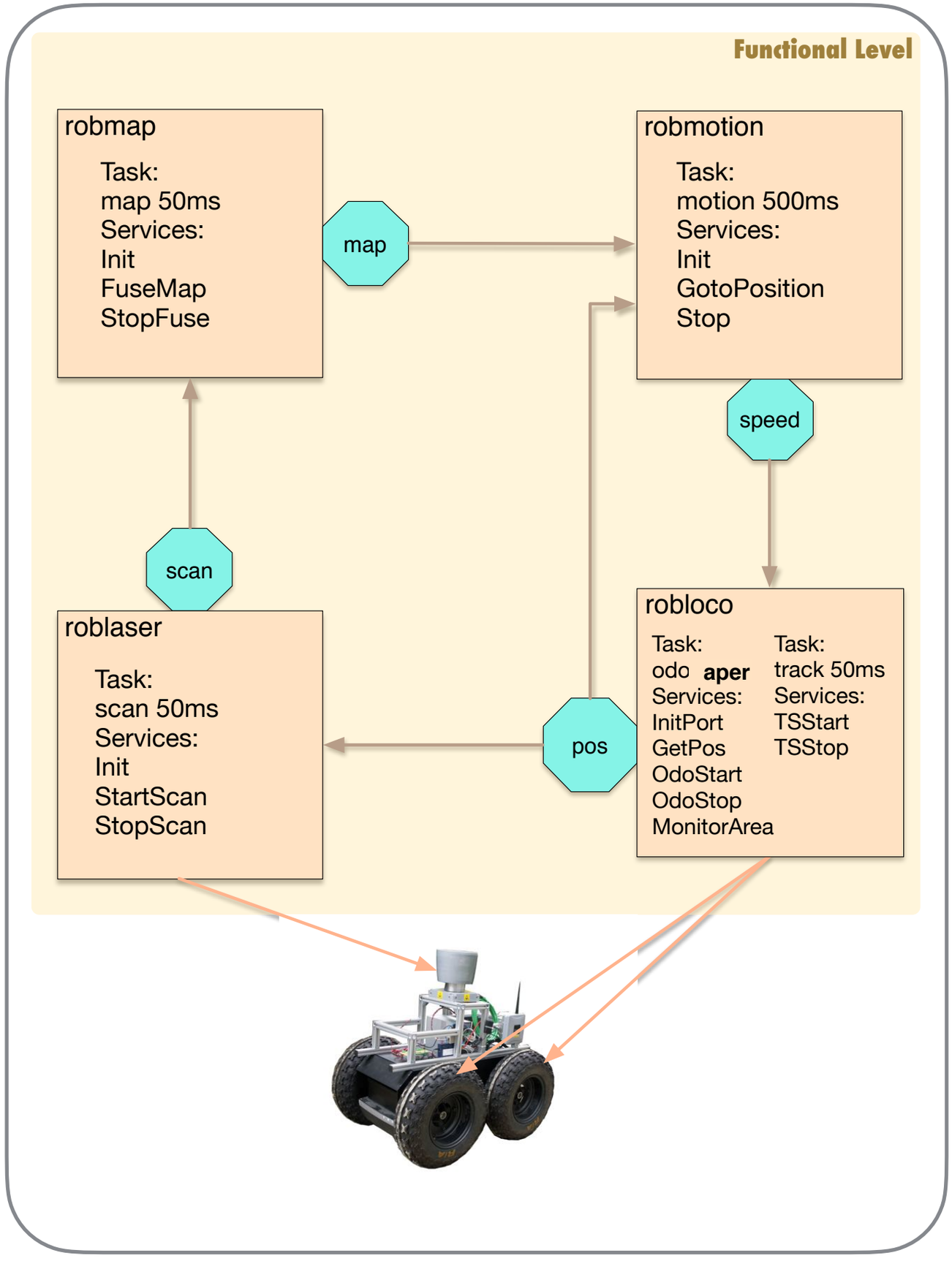
RobNav

- 4 modules for robot navigation
- 4 shared “data”
- 4 control task and 5 execution tasks
- >16 services



RobNav

- How do we implement this?
- Which OS?
- Which OS primitives for RT programming?
- What communication?
- Which supervisor?



A more realistic example

Inspired from a drone example

A drone with a camera for sheep flock monitoring (e.g. to prevent bear attacks):

- propellers speed controller (1KHz)
- IMU (1KHz)
- external GPS (10Hz)
- camera

- 2 positions “providers” (IMU & GPS)
  - different frequency
  - different accuracy (or drift)
- need to merge them

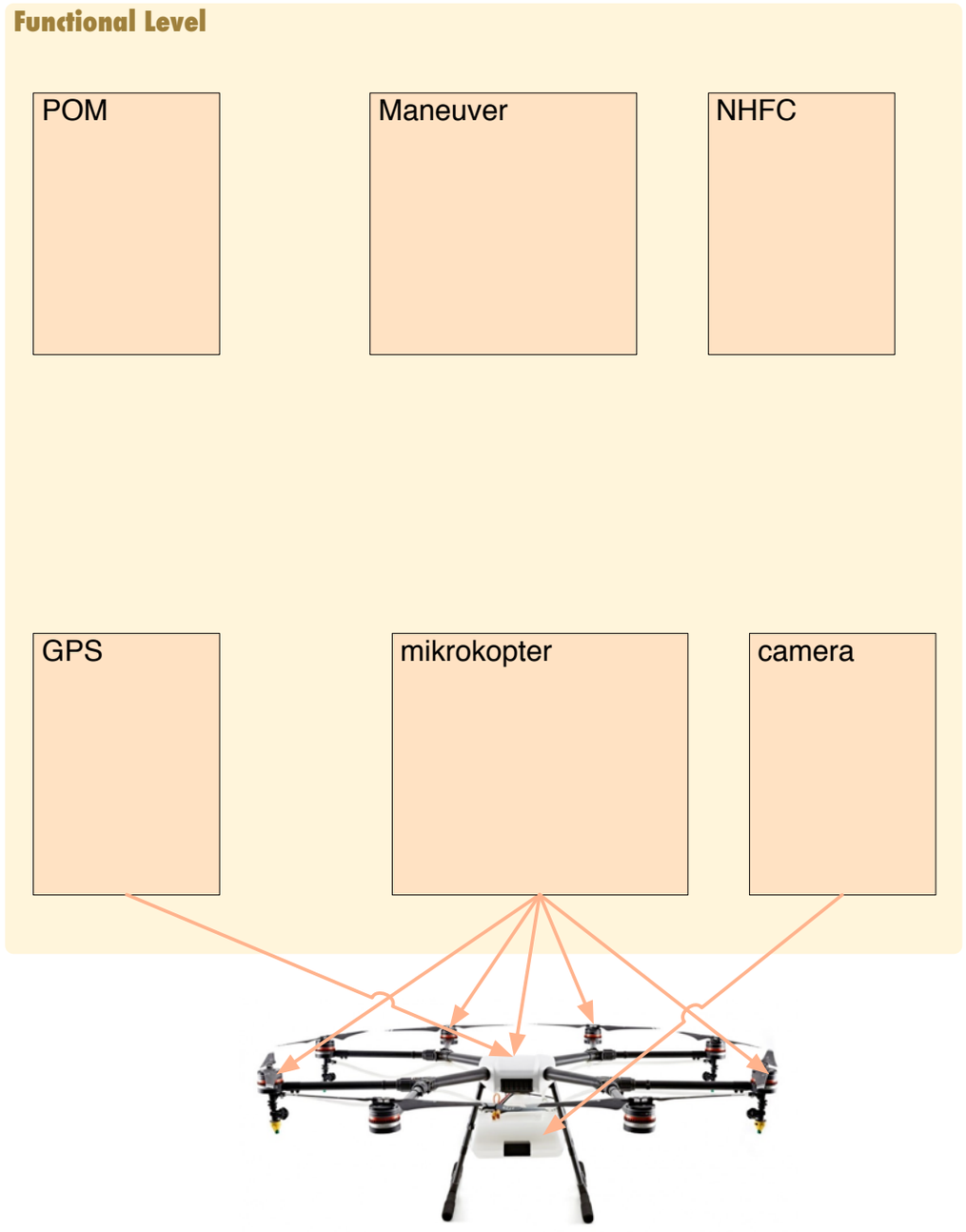
A more realistic example

- Propeller velocity is also measured and available
- motion control law needed
- moving and hovering
- rectangle spanning (e.g. to survey the flock)

Modules/Components

How many modules/components do we need?

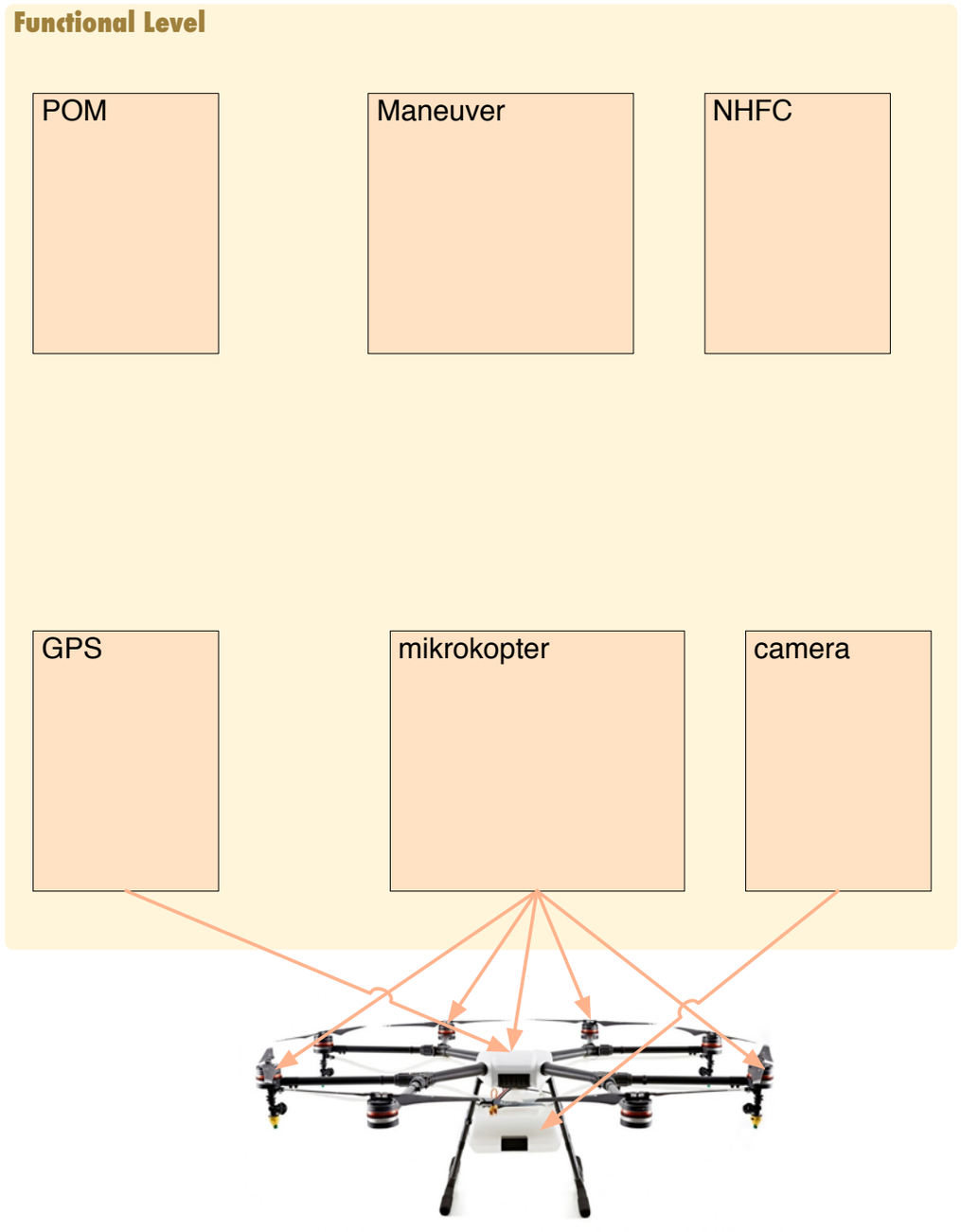
- Anything similar we already have?
- Any existing modules for a particular sensor/ effector?
  - GPS, IMU?
  - camera?
- Any existing modules for a position merging, trap planning, etc



Modules/Components

How many modules/components do we need?

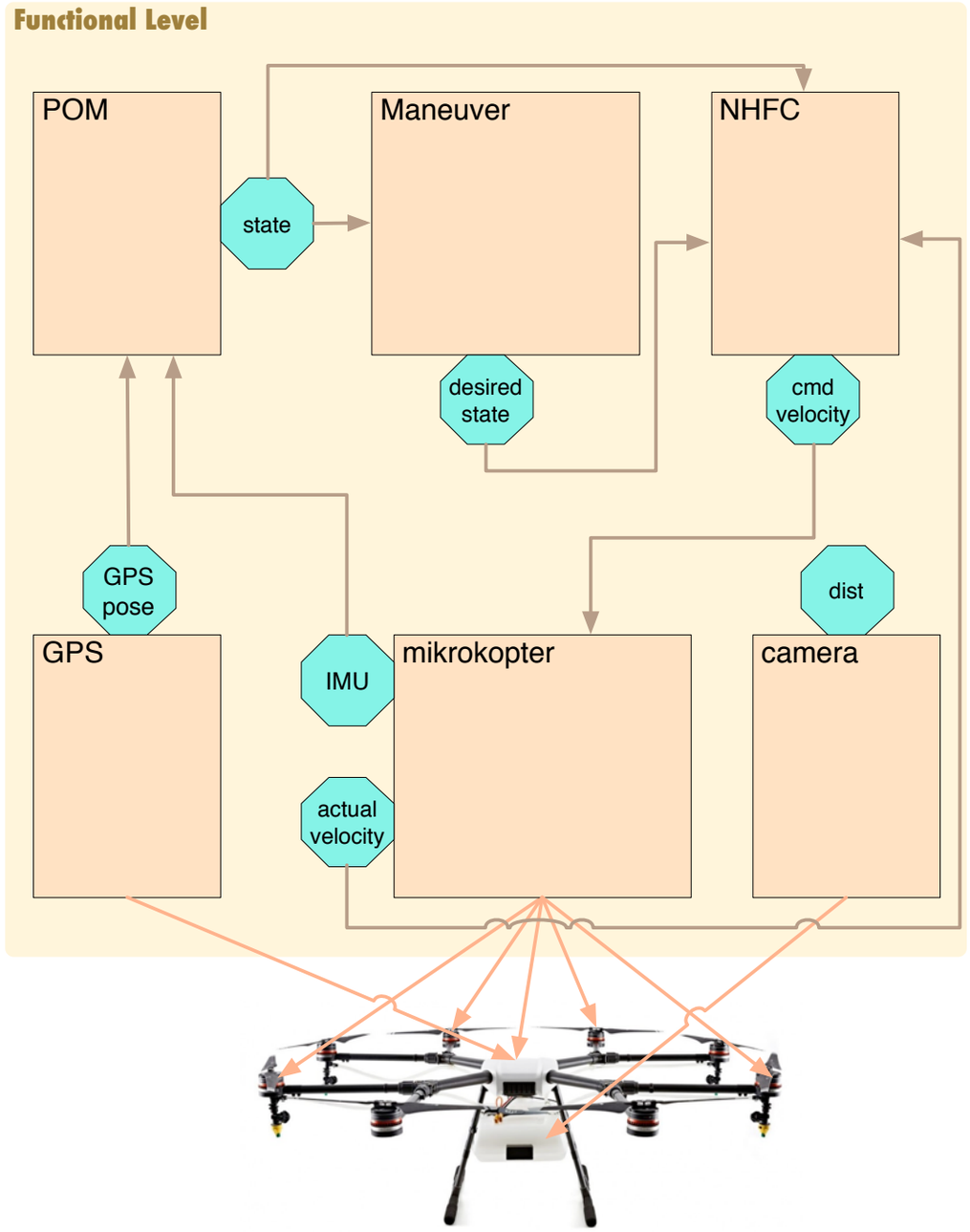
- **mikrokopter** in charge of the propellers (applies a ***velocity reference*** to the propellers, and gives the ***actual velocity*** and manages the ***IMU*** to estimate its displacement and position)
- **GPS** in charge of the ***pos***
- **POM** merges ***IMU*** and GPS ***pos*** Kalman filter
- **NHFC** given a position, produces a ***velocity*** to move to and hoover above this position
- **Maneuver** given a sequence of positions, produces a trajectory and intermediate positions to navigate to each of them
- **camera** in charge of surveying the flock for bear



Modules produce and use data

Which data need to be shared?

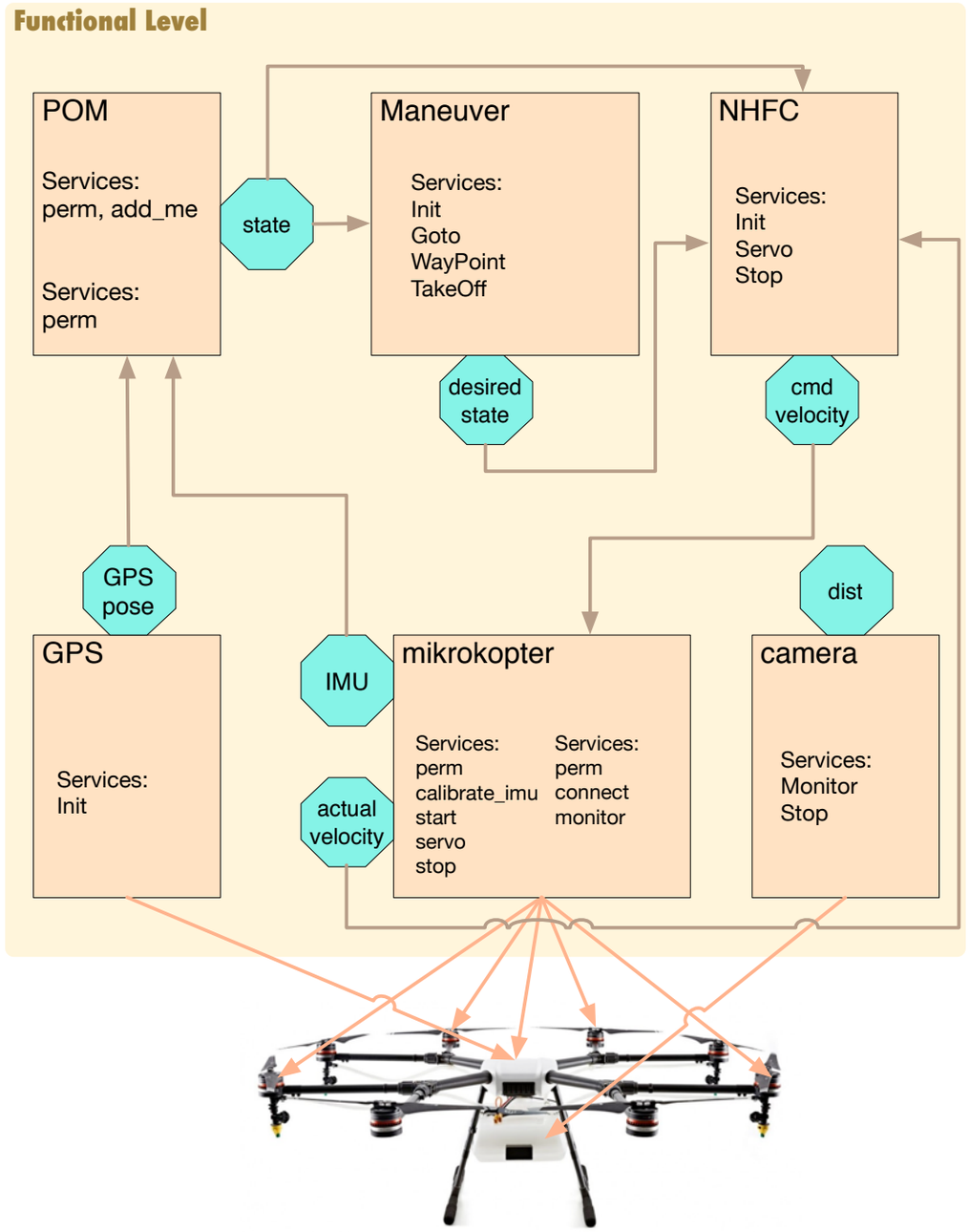
- **mikrokopter** reads a *cmd velocity* reference from **NHFC** or **traj_planner** and produces the *IMU* and also the *actual velocity*
- **GPS** produces *pos*
- **POM** reads *IMU* and *pos* and produces a merged *pos*
- **NHFC** reads the current *pos* and the *actual velocity* and produces a *velocity*
- **Maneuver** reads the current *pos* and the produces *intermediate pose*
- **camera** produces image and detect bear (distance)



Modules provide services to “client”

Which services do we need?

- **mikrokopter** perm, calibrate_imu, start, servo, stop, connect, monitor
- **GPS** init
- **POM** perm, add_me
- **NHFC** servo, init, stop
- **Maneuver** Init, GotoPositions, Stop
- **camera** Monitor, Stop

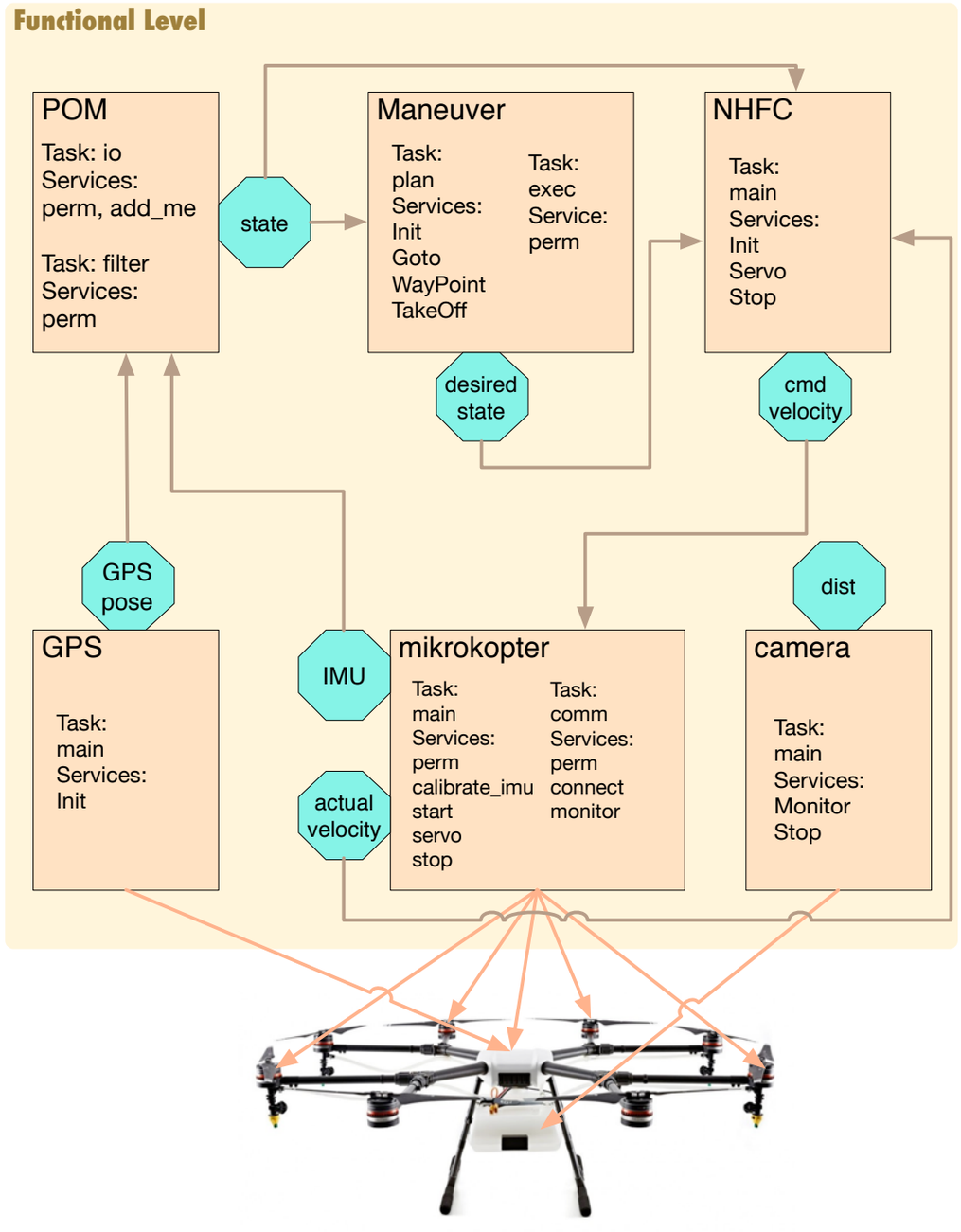


Services execute in tasks

Consider the required parallelism

Tasks (threads) per module (consider the required parallelism)

- **mikrokopter** has 2 tasks, ***main*** and ***comm*** (to communicate with the low level controller)
- **GPS** has 1 task, ***main***
- **POM** has 2 tasks, ***io*** and ***filter*** (Kalman)
- **NHFC** has 1 task, ***main***
- **Maneuver** has 1 task, ***plan***
- **camera** has 1 task, ***main***

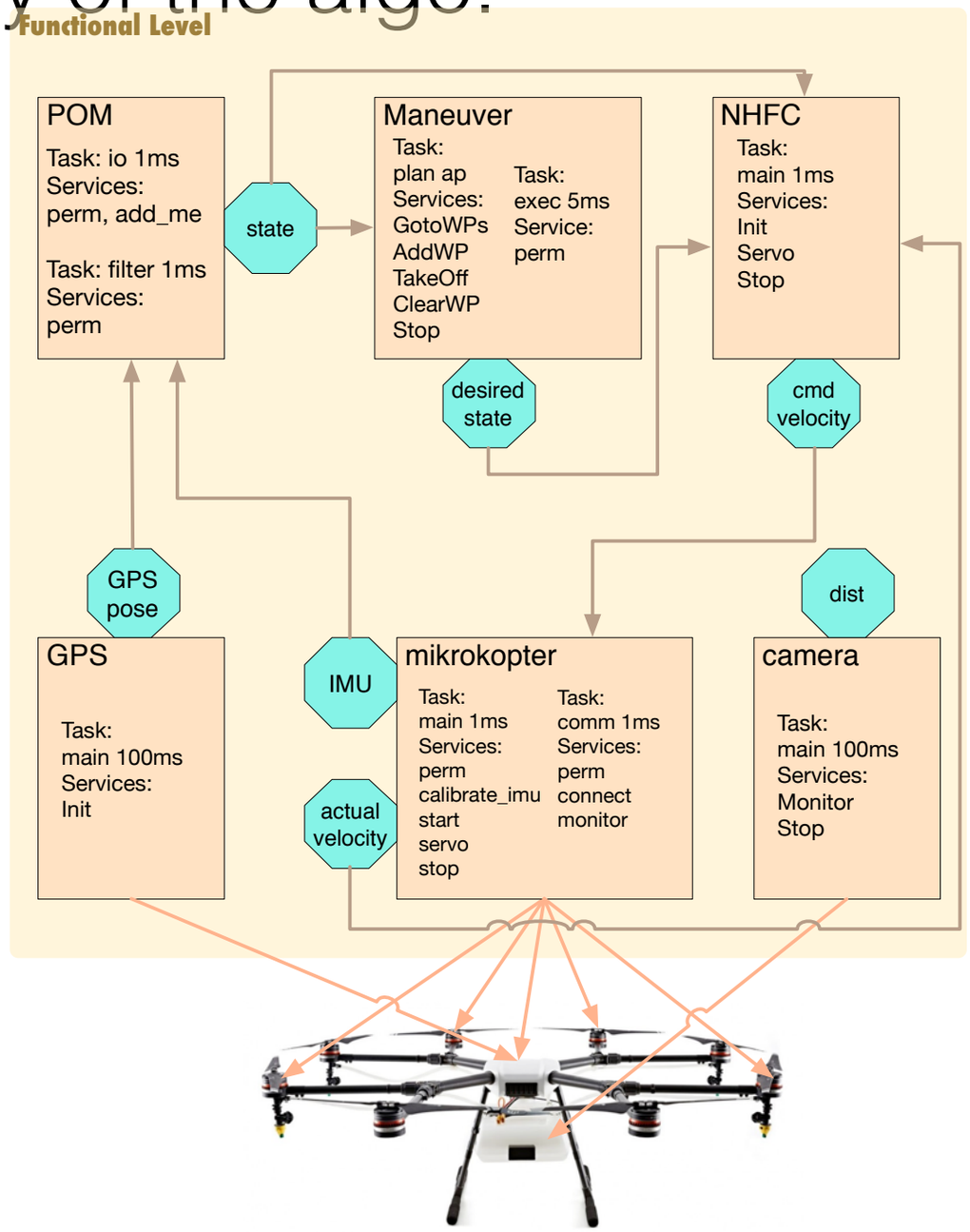


Periodic tasks period

Consider how fast these data are or need to be produced, but also the complexity of the algo.

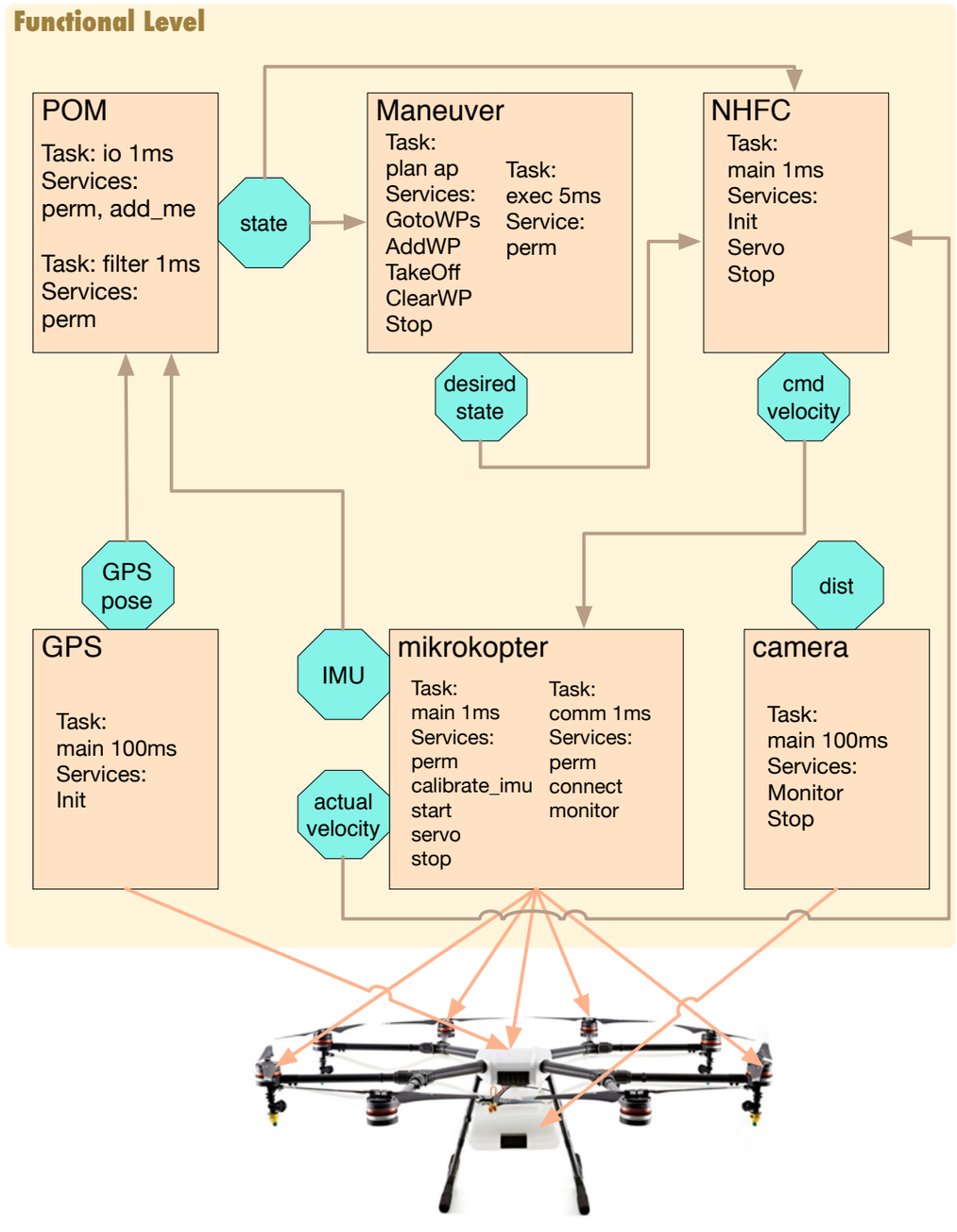
Tasks (threads) period.
Values are “application/hardware” defined.
This is “presumably” the period at which they will read and write external data.

- **mikrokopter** ***main*** and ***comm*** at 1KHz
- **GPS** ***main*** at 10 Hz
- **POM** ***io*** and ***filter*** at 1KHz
- **NHFC** ***main*** at 1KHz
- **Maneuver** ***plan*** at 1KHz
- **camera** ***main*** at 10 Hz



MikroKopter

- 6 modules for this drone
- 7 shared “data”
- 6 control task and 8 execution tasks
- >20 services



Real Example from LAAS

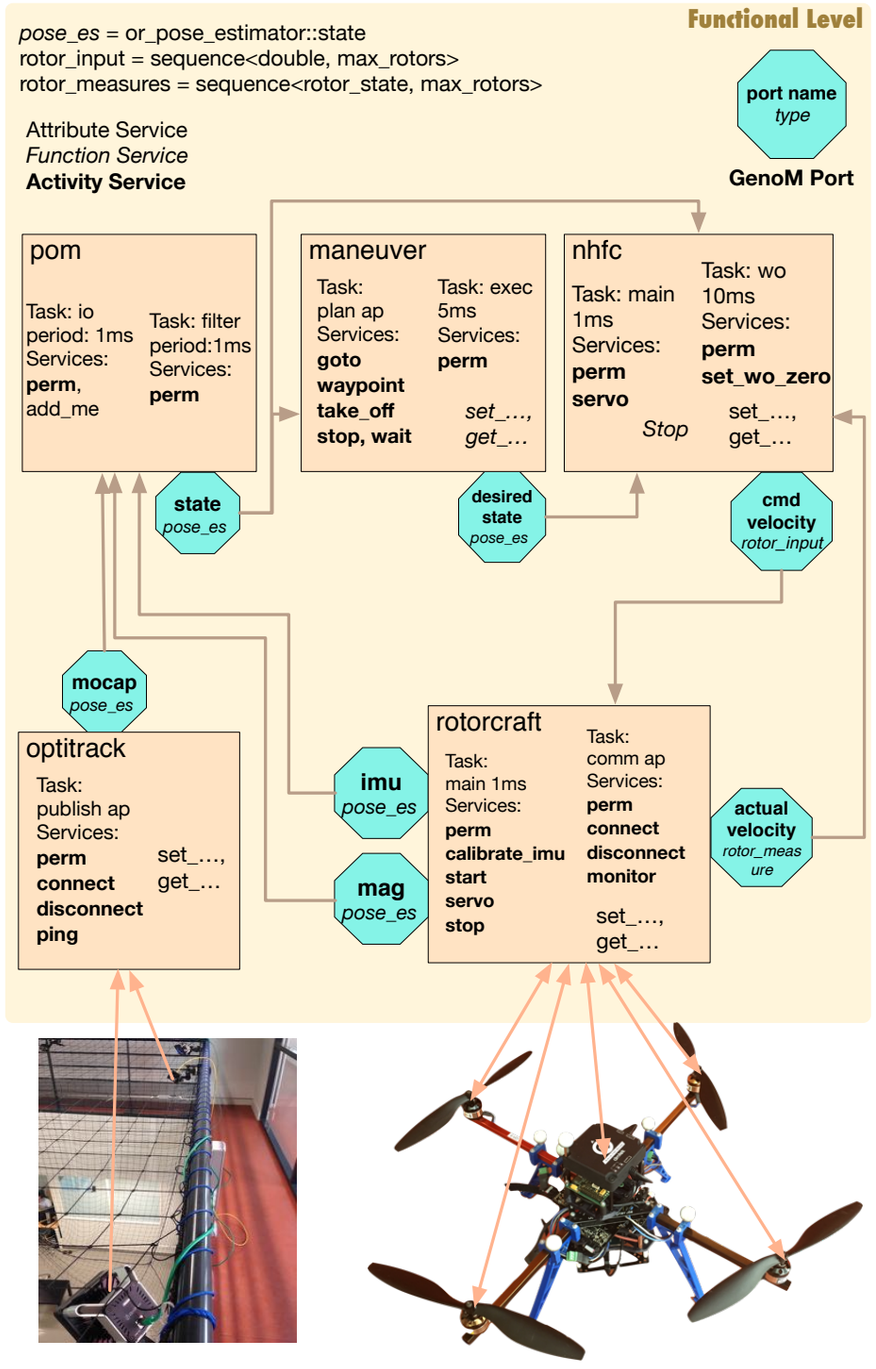
Drone @ LAAS

- propellers speed controller (1KHz)
- IMU (1KHz)
- external motion capture or GPS (10Hz)
- flight controller (NHFC) 1KHz
- path/traj planner (maneuver) 200Hz

- 2 positions “providers” (MC & IMU)
  - different frequency
  - different accuracy (or drift)
- need to merge them

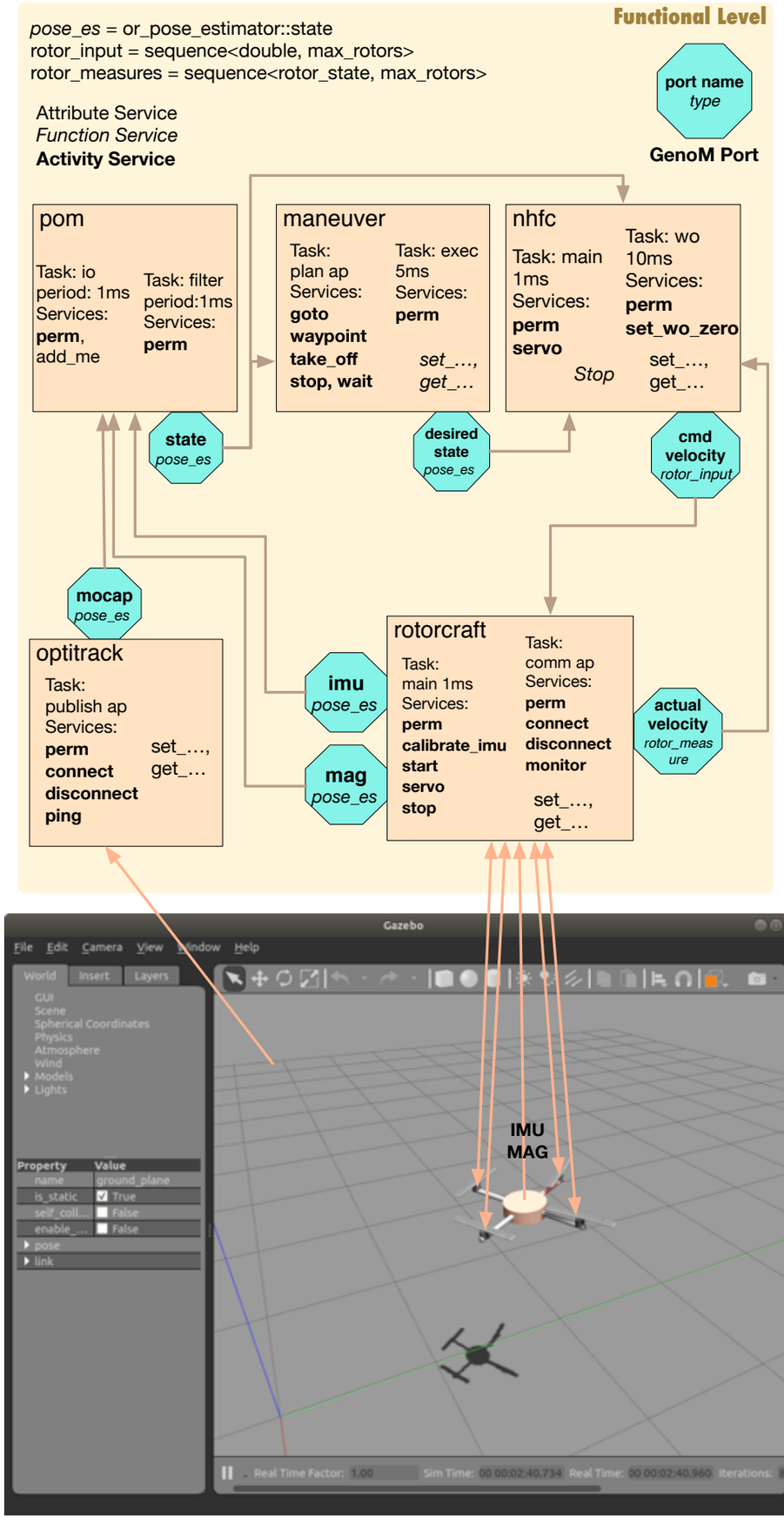
Drone

- 5 modules for this drone
- 8 shared “data”
- 5 control task and 8 execution tasks
- >20 services
- Full source available



Gazebo/MRSim

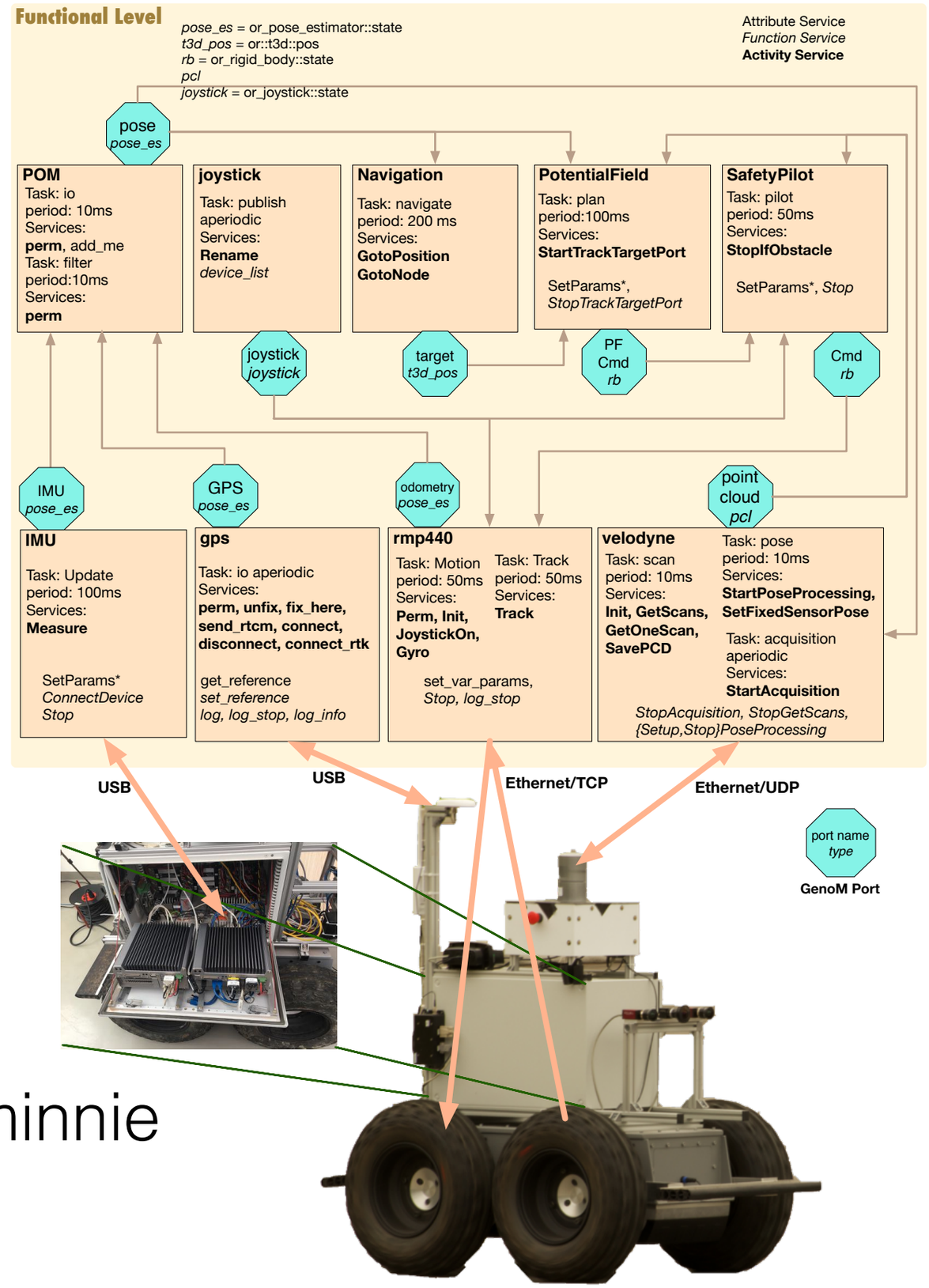
- 5 modules for this simulated drone
- 8 shared “data”
- 5 control task and 8 execution tasks
- >20 services
- Physical simulation in Gazebo (propellers velocity)
- Full source available
- Demo



Another real example from LAAS

- Minnie
- 9 components
- 9 ports
- >20 services

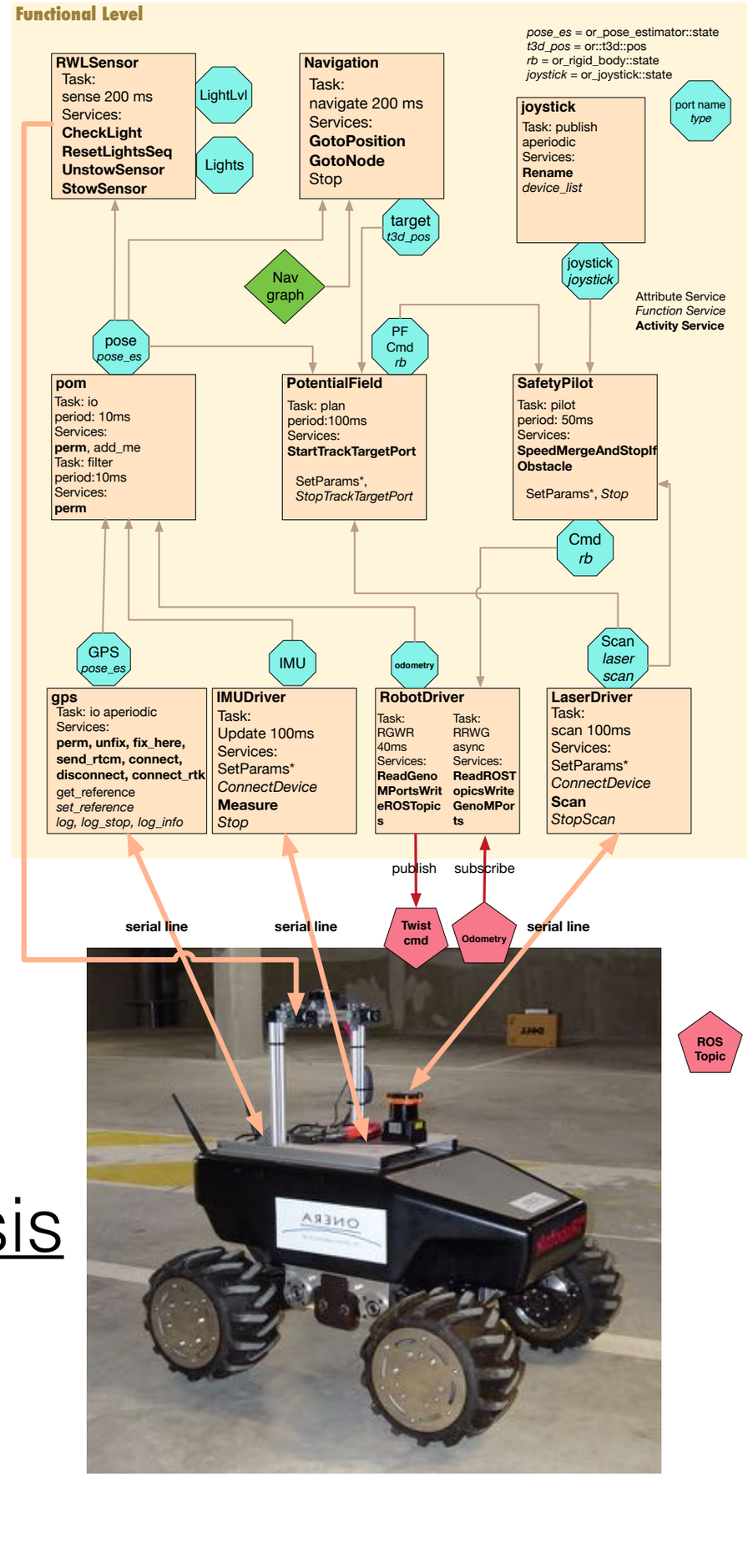
- Full source available https://redmine.laas.fr/projects/minnie



One more…

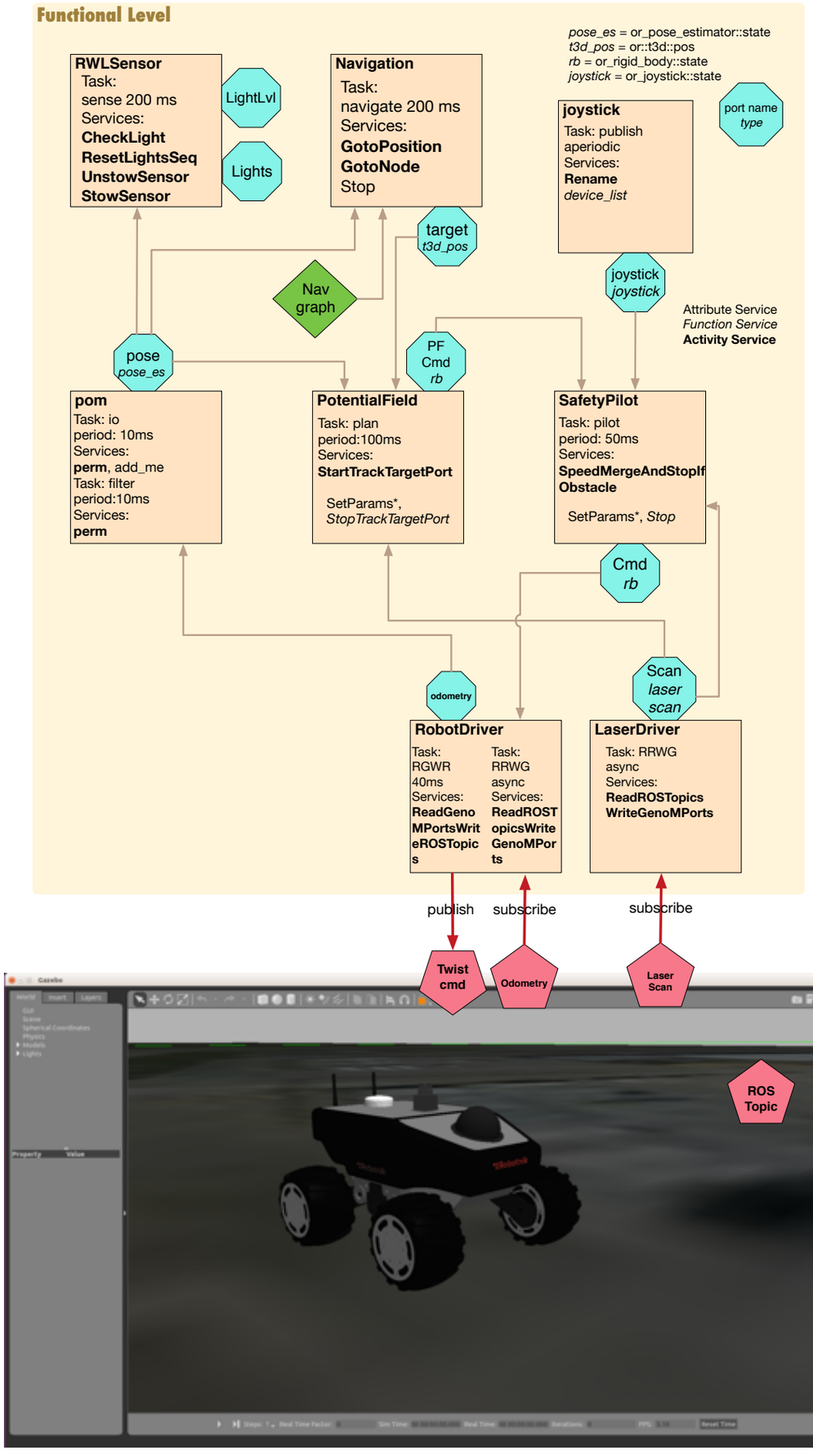
- Osmosis
- 10 component
- 12 ports
- >20 services

- Full source available
  https://redmine.laas.fr/projects/osmosis



… simulation.

- Osmosis
- simulation with Gazebo
- Démo
- Full source simulator https://osmosis.gitlab.io/experiment/simulation.html



MiddleWare Requirements

- Offer some OS and hardware abstraction
- Some inter process communication means
  - publish-subscribe or client-server
  - one CPU, multiple CPU
  - multiple CPU, manage endianness
- Process/thread management
- Timers
- Lock, shared resources

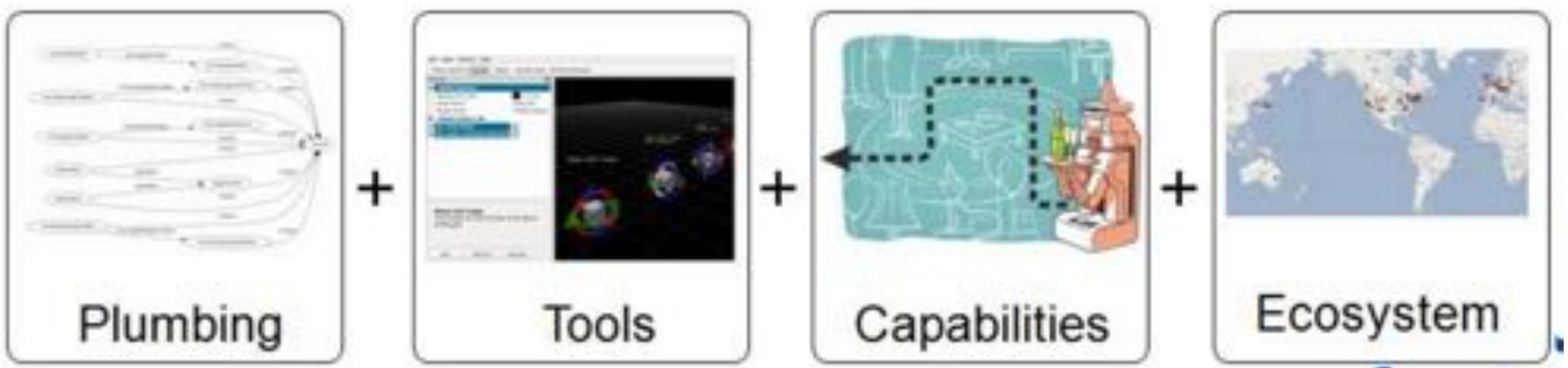
What is ROS?
(Robot Operating System)

- A “meta” operating system for robots (but despite the name it is **not** a real OS, it relies on other OSes, mostly Linux)
- A collection of packaging, software building tools
- **An architecture for distributed* inter-process/inter-machine communication and configuration** (ROS Comm)
- Development tools for system runtime and data analysis Open-source under permissive BSD licenses (ros core libraries)
- A language-independent architecture (c++, python, lisp, java, and more)
- A scalable platform (ARM CPUs to Xeon Clusters)

With the intent to enable researchers to rapidly develop new robotic systems without having to “reinvent the wheel” through use of standard tools and interfaces.

What is ROS?
(Robot Operating System)

- Open source (BSD)
- Created by Willow Garage
- Maintained by Open Source Robotics Foundation (OSRF)



from ISAE - Pascal CHAUVIN

ROS Comm

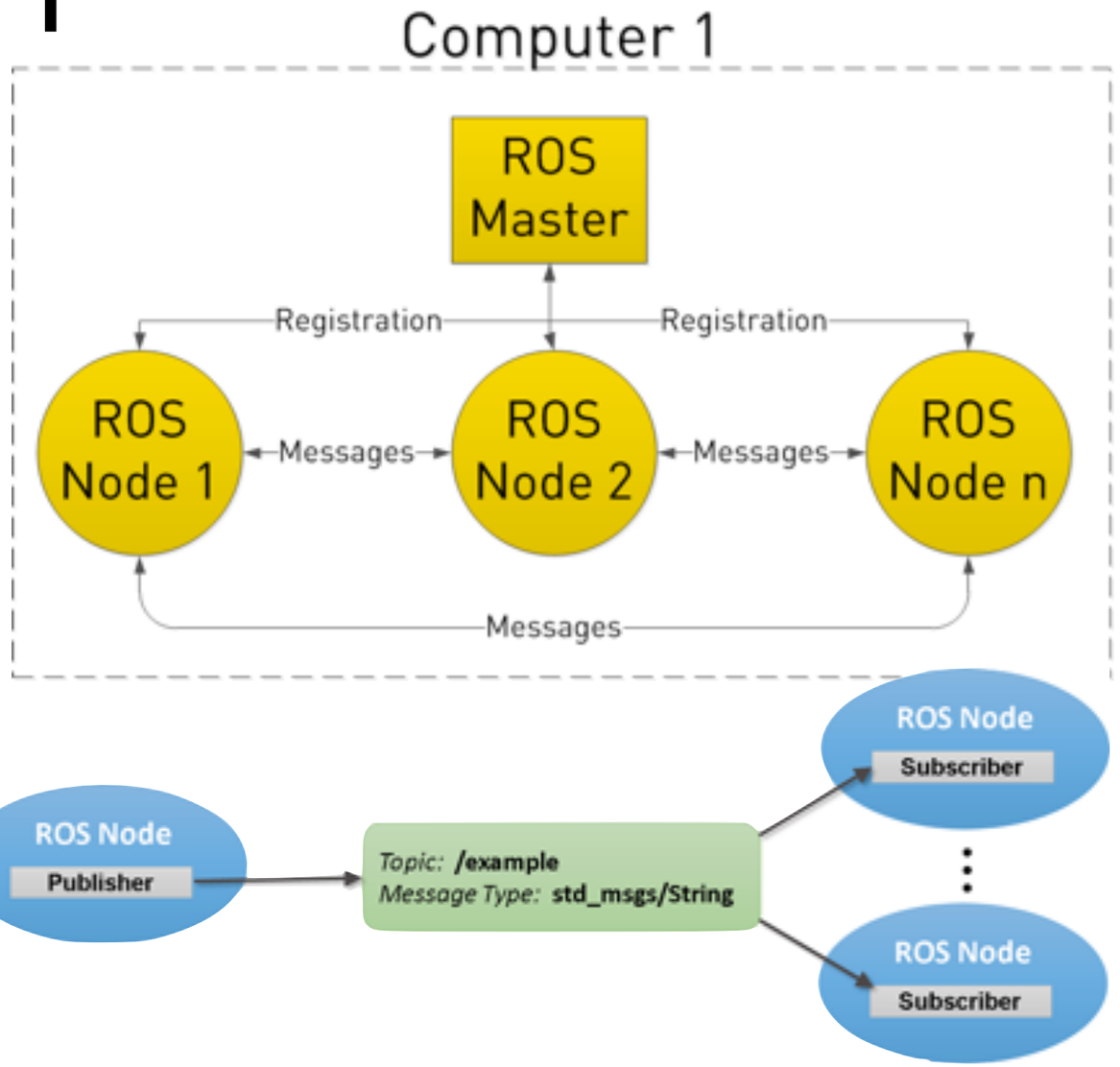
- ROS Communication Protocols helps in Connecting **nodes** (programs) over the network
- These capabilities are currently built entirely on two high-level communication APIs

- ROS **Topics**
  - A **node** can publish a data of interest to other nodes
  - Another **node** interested in this data will subscribe to it.
- ROS **Services**
  - A **node**, specialized in a particular service (computation), advertises it.
  - Another **node** may use this service.

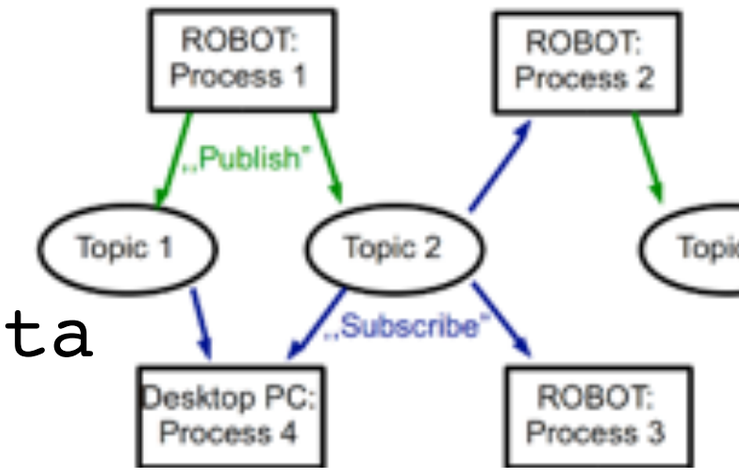
- ROS **Master** (unique for an experiment) handles the networking mess…aging…

ROS Topics

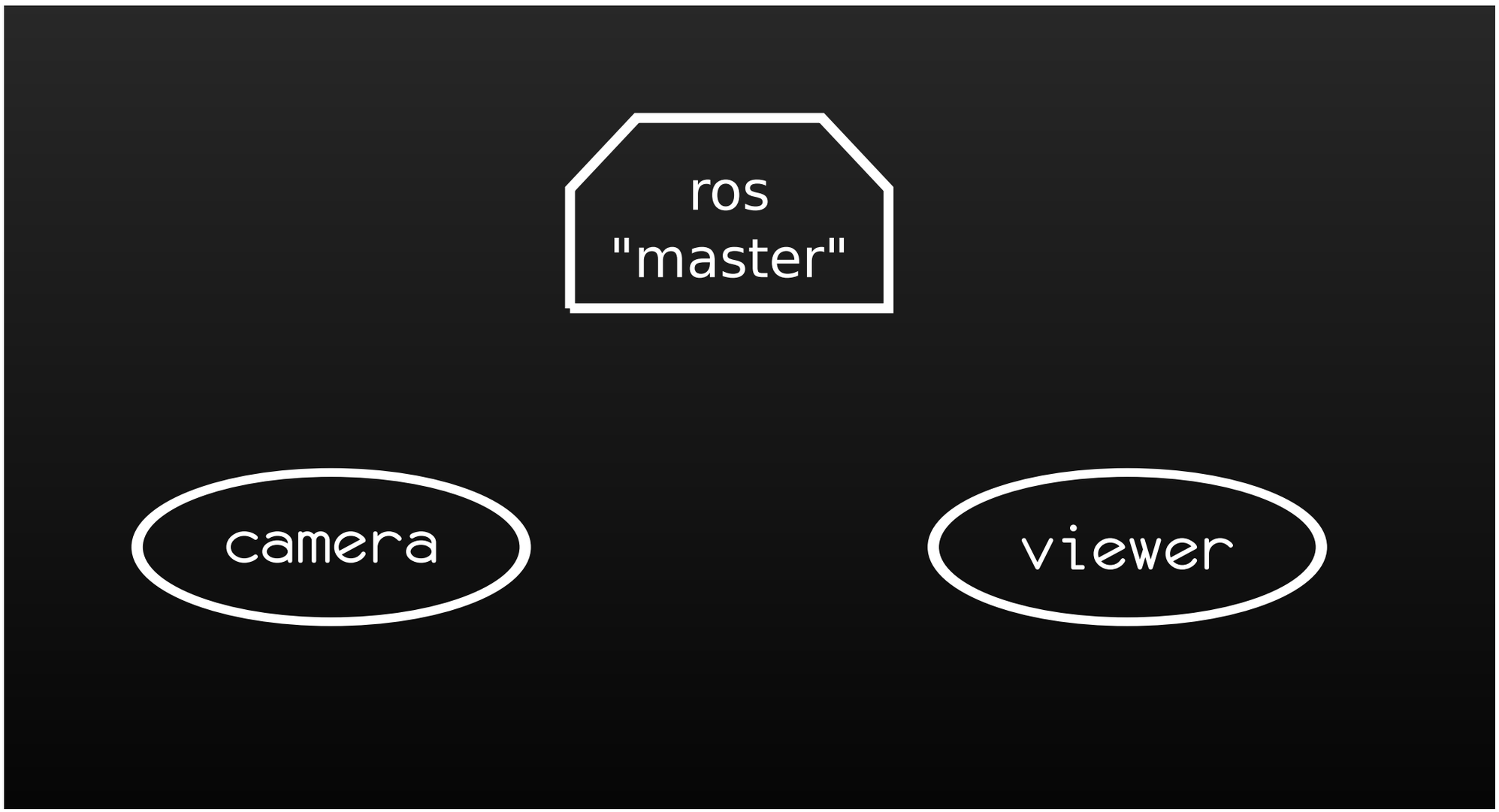
- ROS Topics
  - **Asynchronous** “stream-like” communication
  - TCP/IP or UDP Transport
  - Strongly-typed (ROS .msg spec)
  - Can have one or more *publishers*
  - Can have one or more *subscribers*
  - publish & subscribe (**callback**)




```
MyTopic.msg
  float64 x
  float64 y
  float64 theta
```

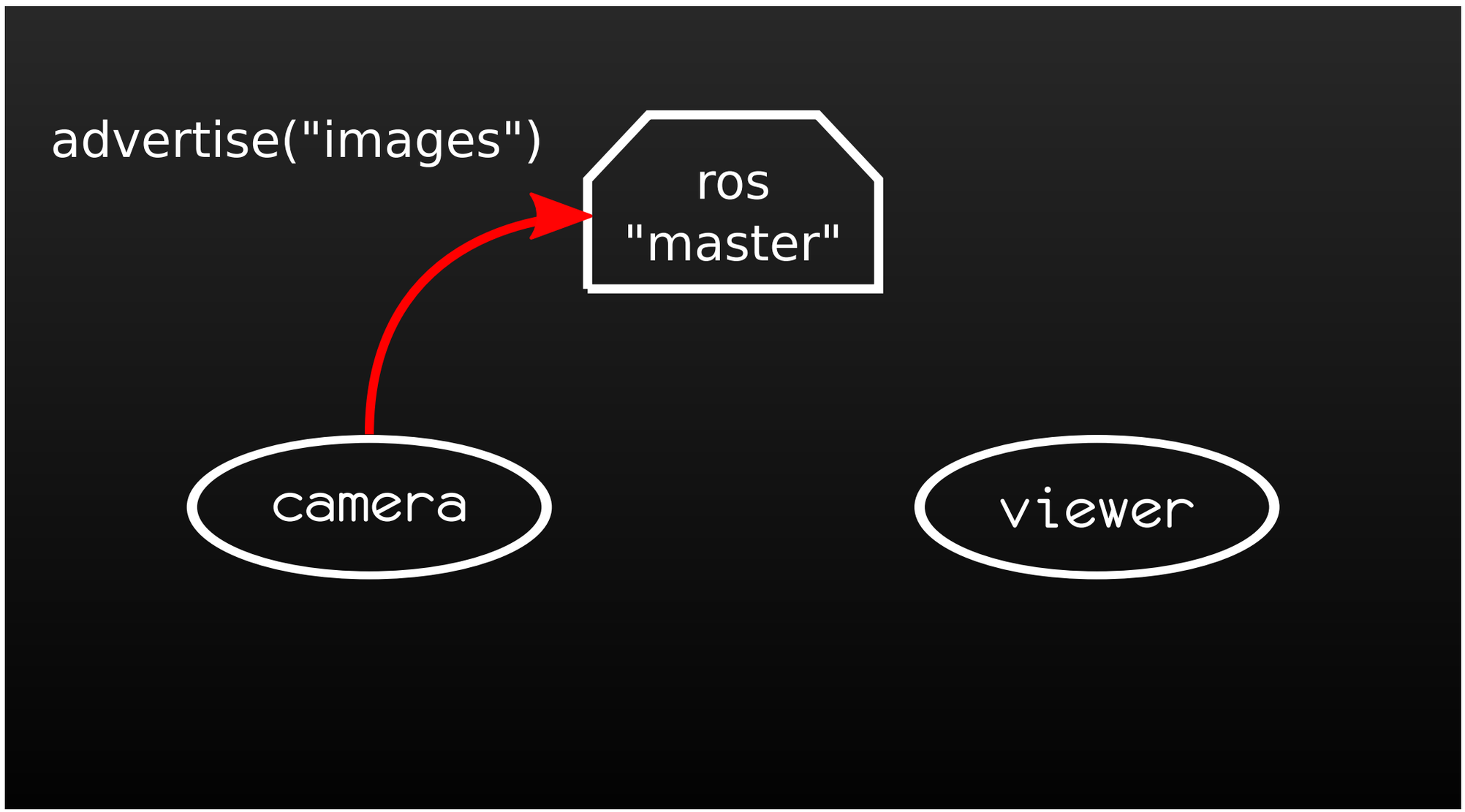


ROS Topics

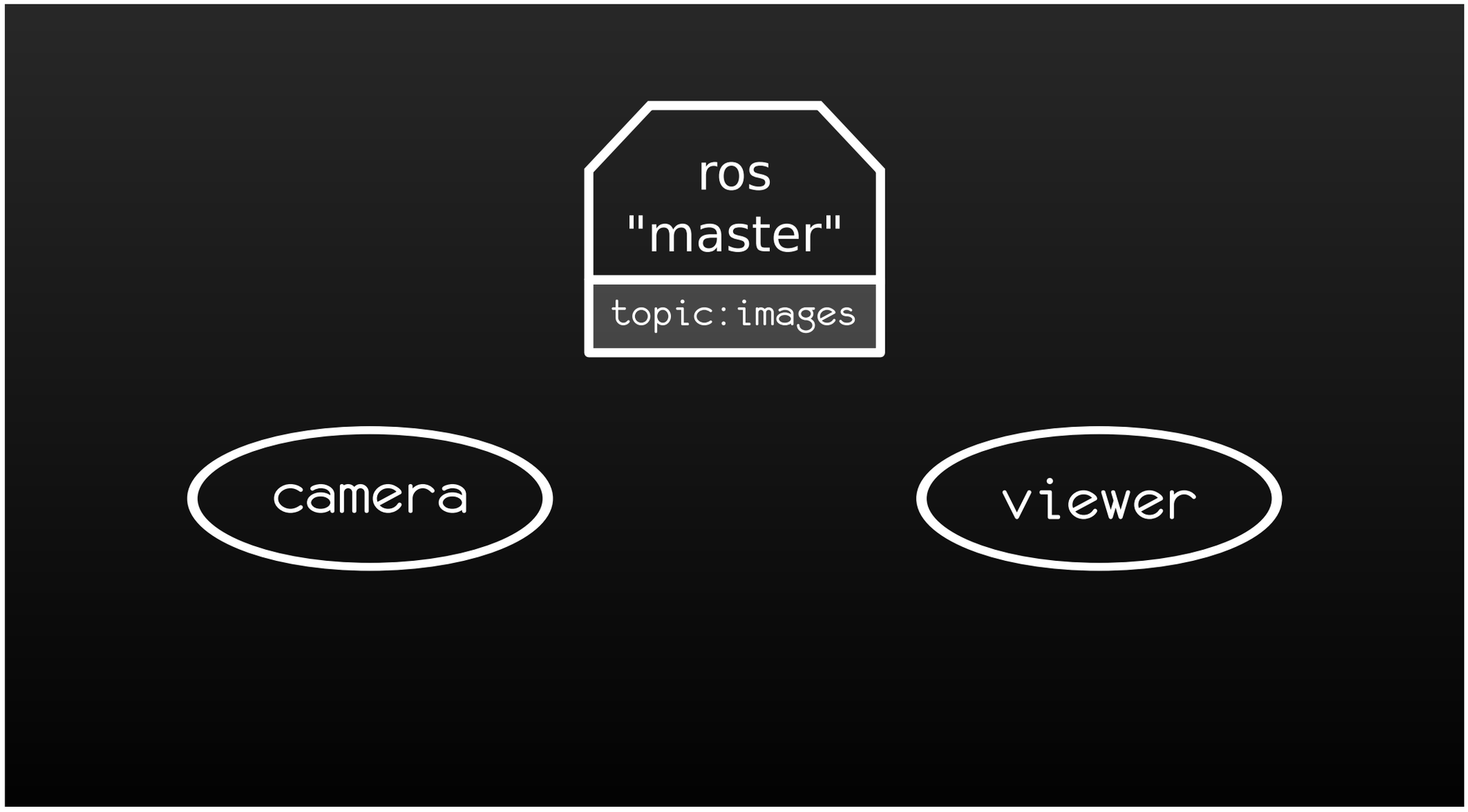


From Jonathan Bohren (JHU LCSR)

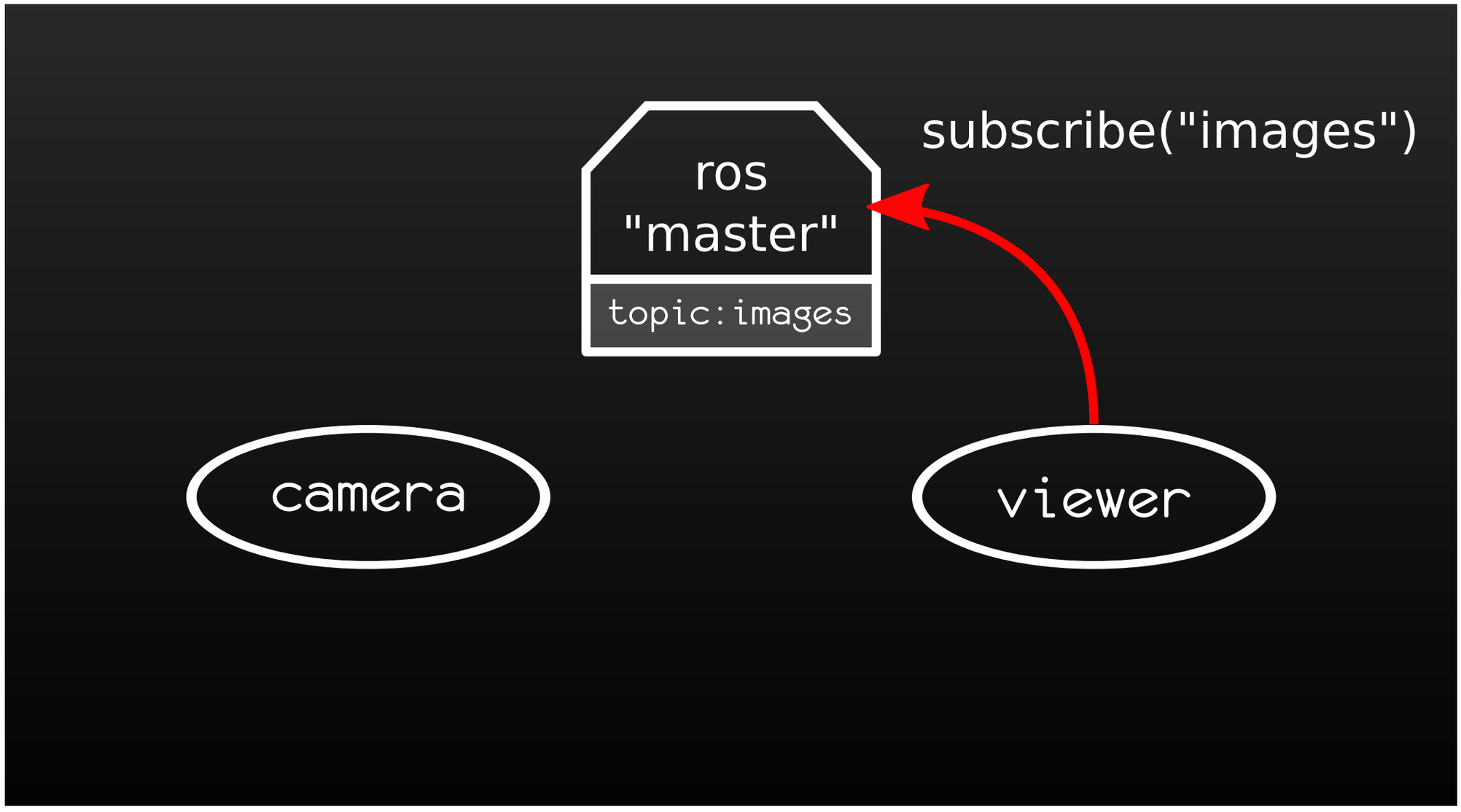
ROS Topics



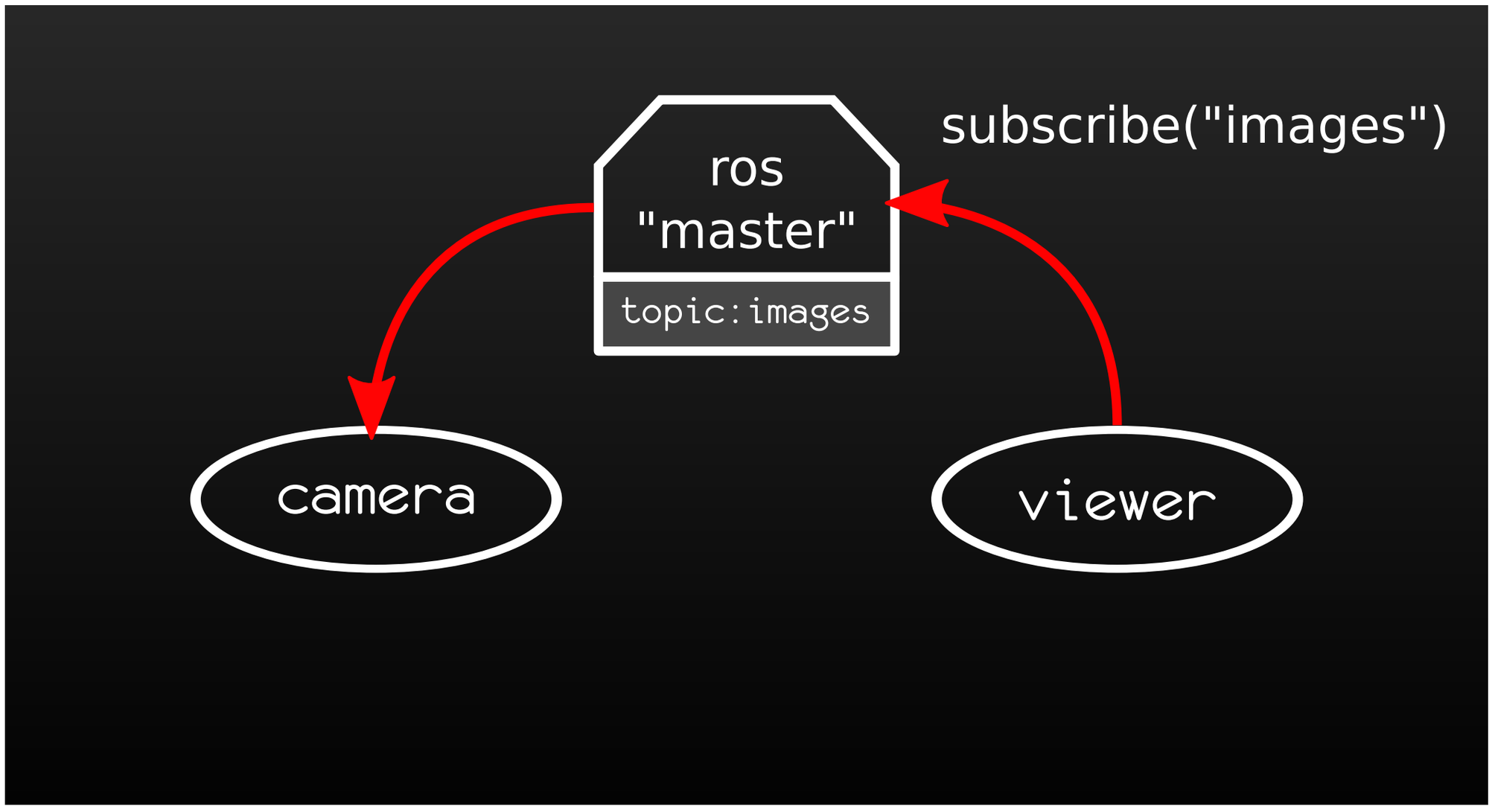
ROS Topics



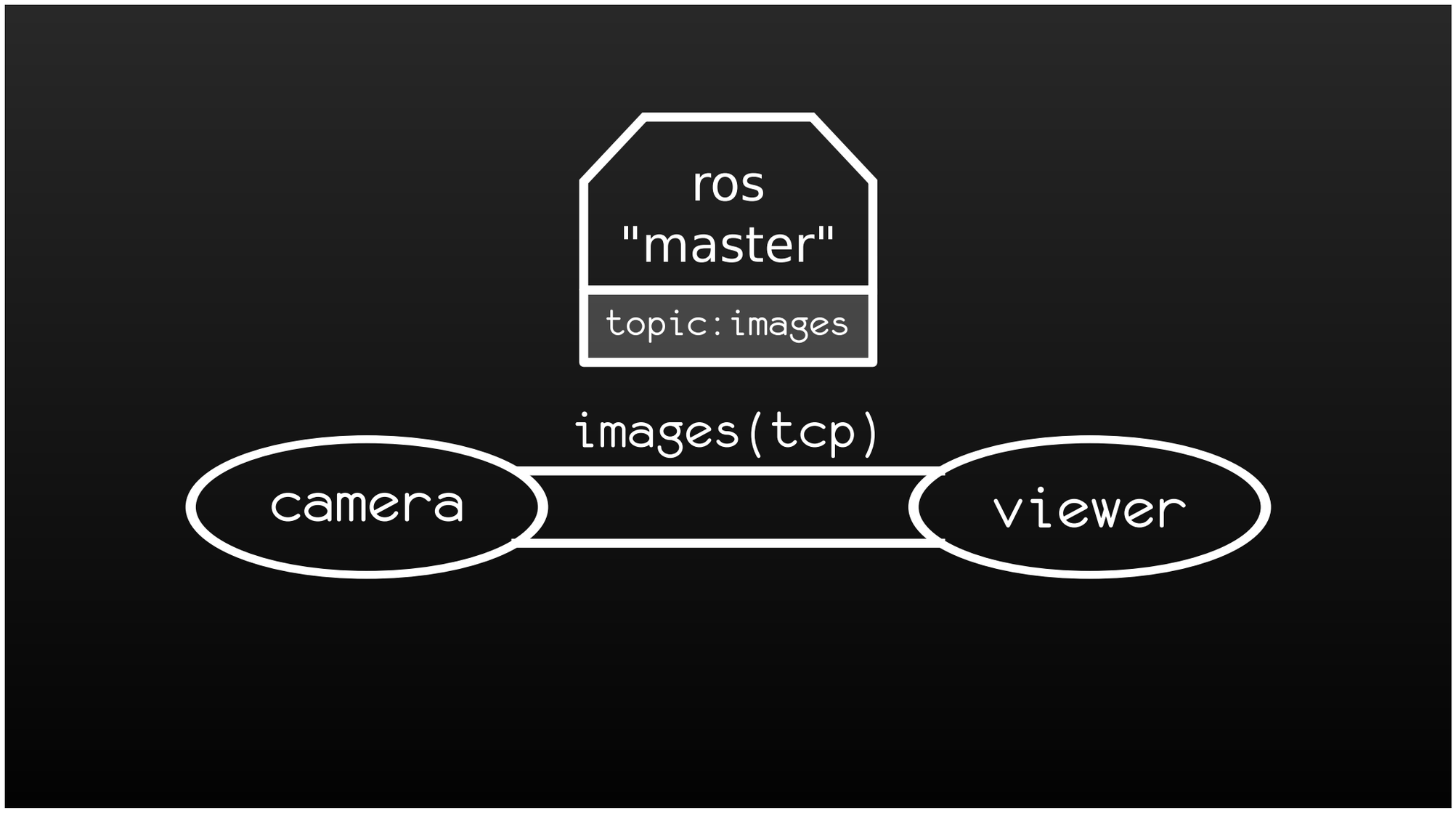
ROS Topics



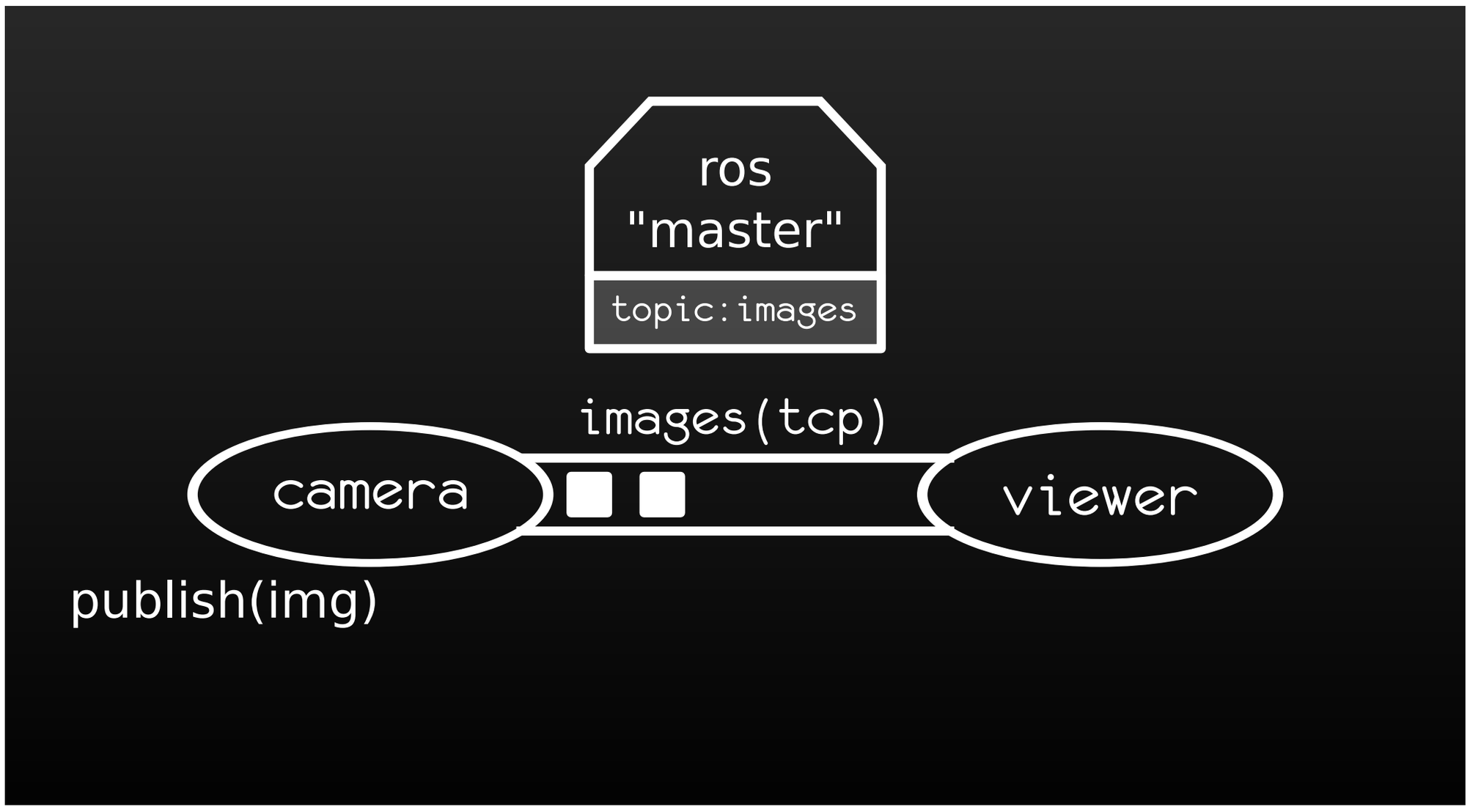
ROS Topics



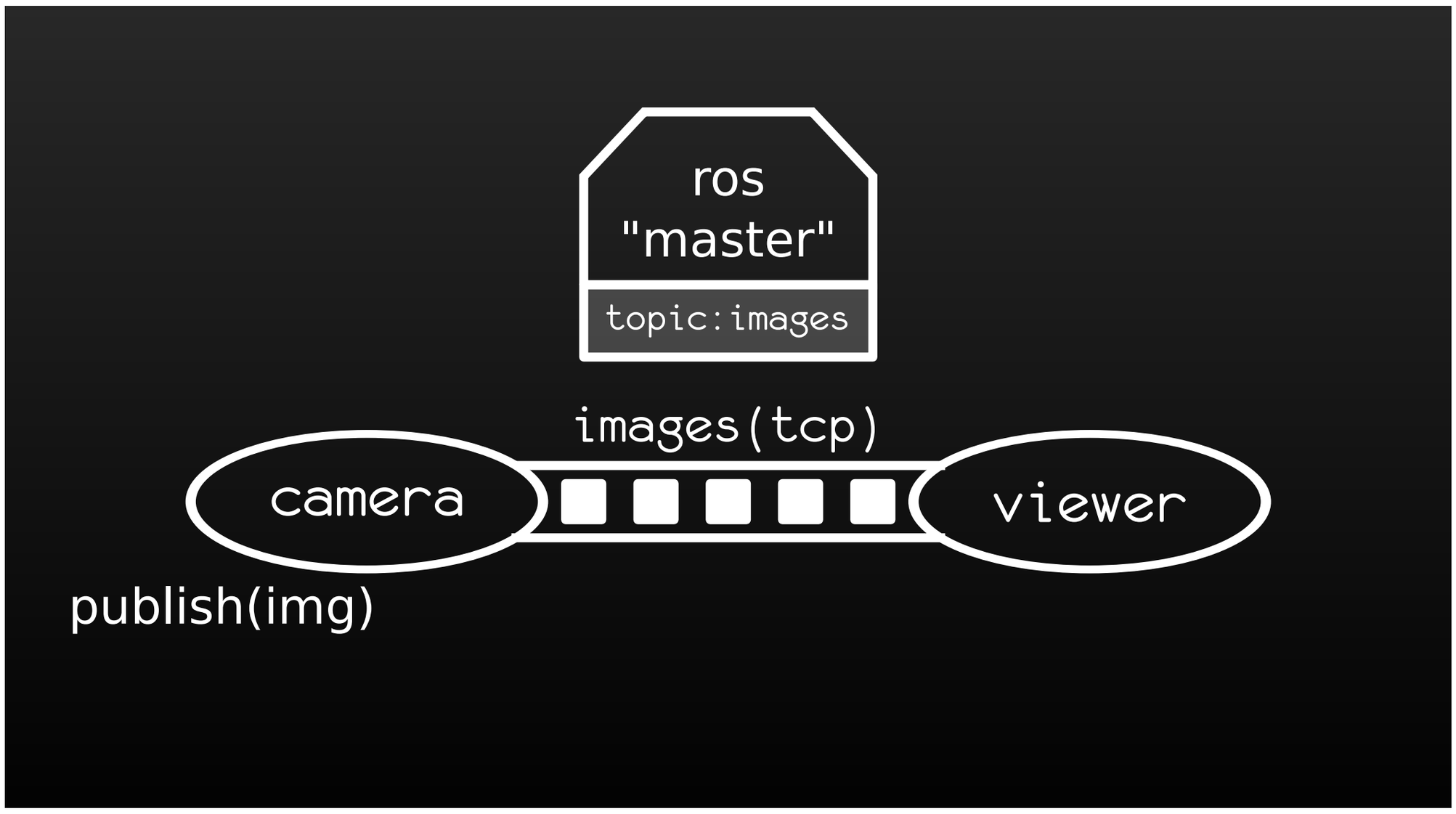
ROS Topics



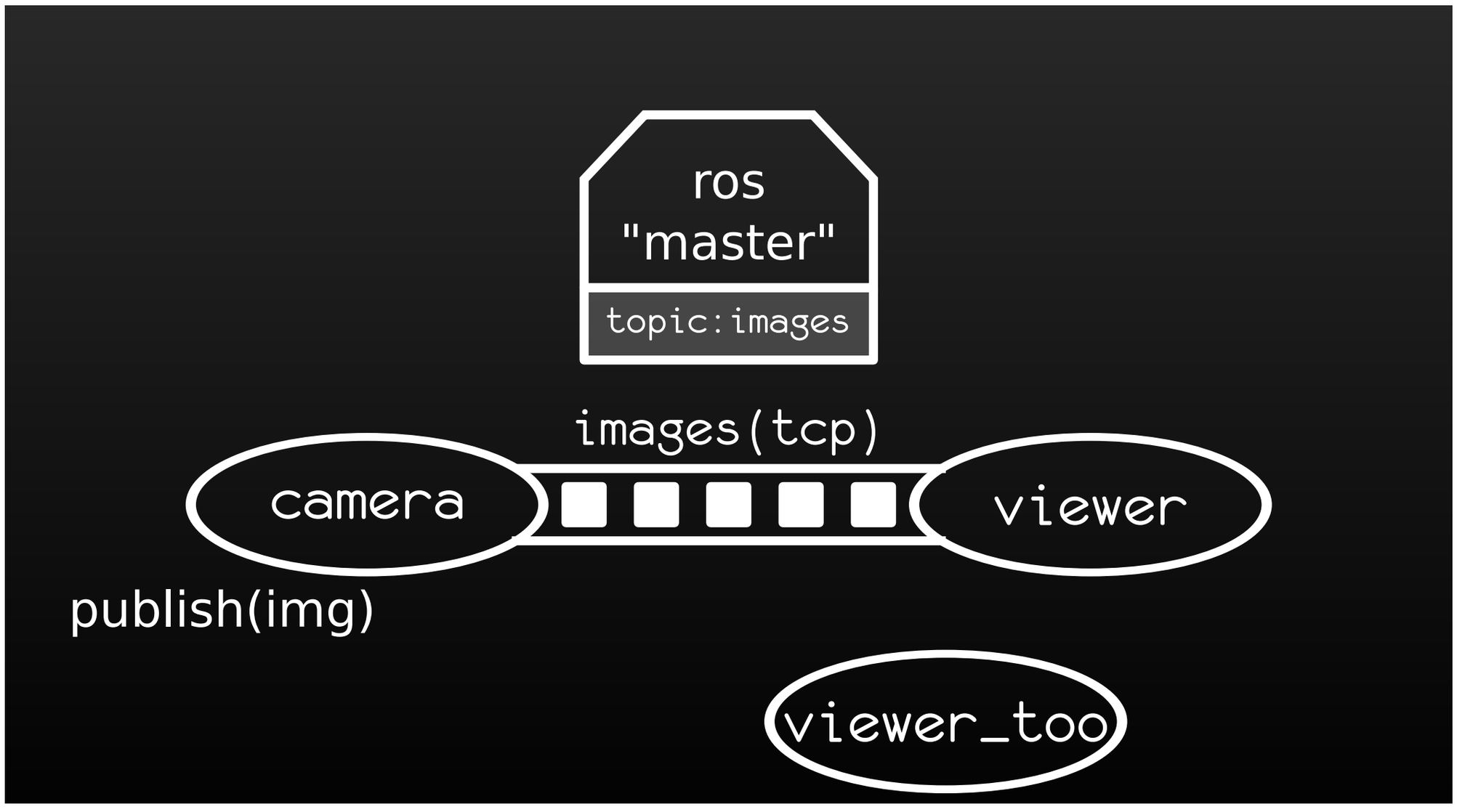
ROS Topics


ros
"master"
topic:images
images(tcp)
camera
viewer
publish(img)

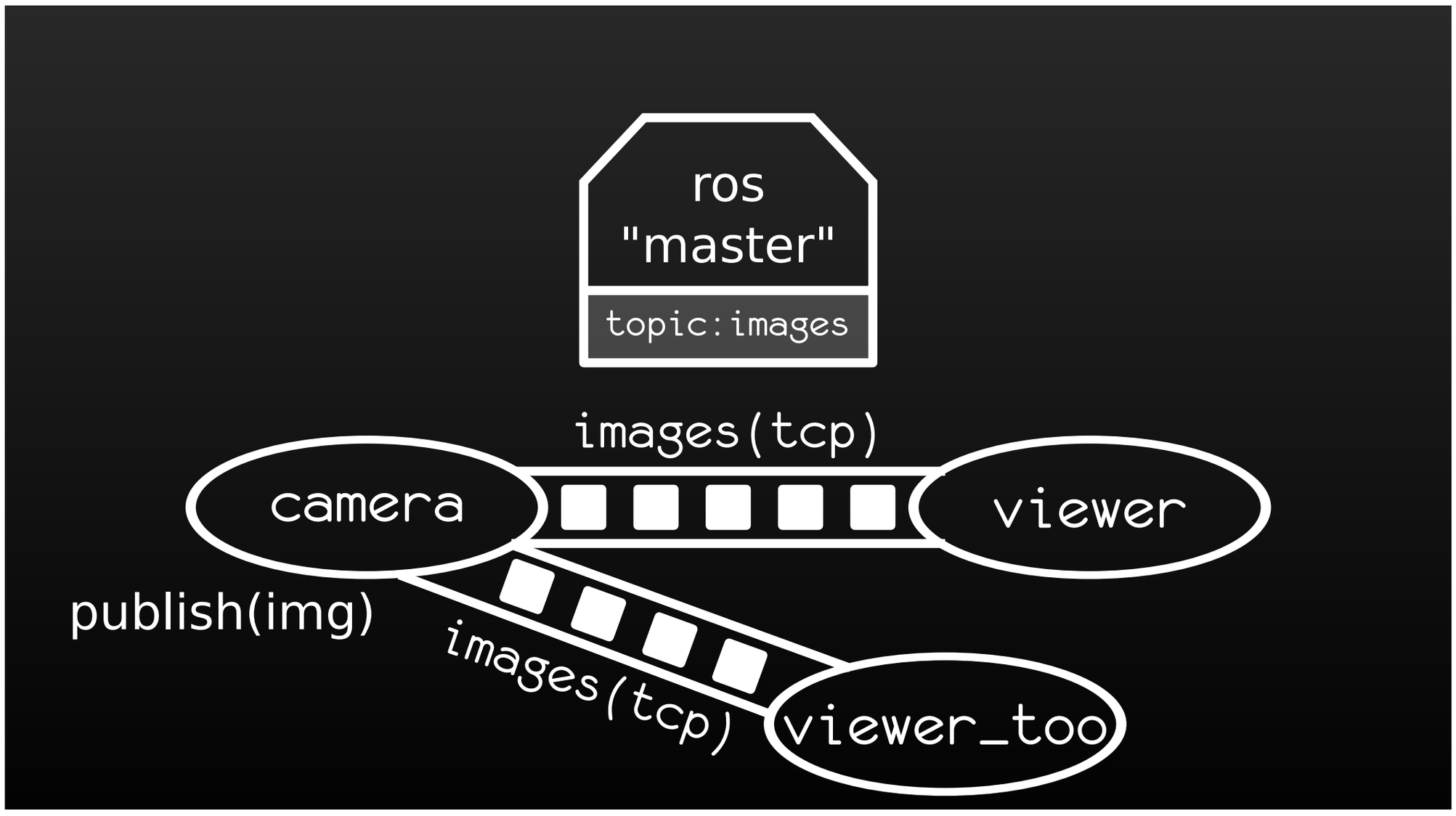
ROS Topics


ros
"master"
topic:images
images(tcp)
camera
viewer
publish(img)

ROS Topics

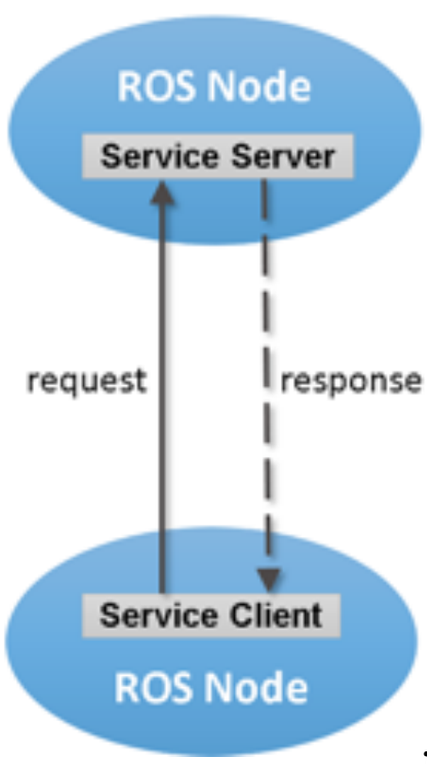

ros
"master"
topic:images
images(tcp)
camera
viewer
publish(img)
viewer_too

ROS Topics


ros
"master"
topic:images
images(tcp)
camera
viewer
publish(img)
images(tcp)
viewer_too

ROS Services

- ROS Services
  - **Synchronous** “function-call-like” communication
  - TCP/IP or UDP Transport
  - Strongly-typed (ROS .srv spec)
  - Can have only one *server*
  - Can have one or more *clients*



Service Name: **/example_service**
Service Type: **roscpp_tutorials/TwoInts**

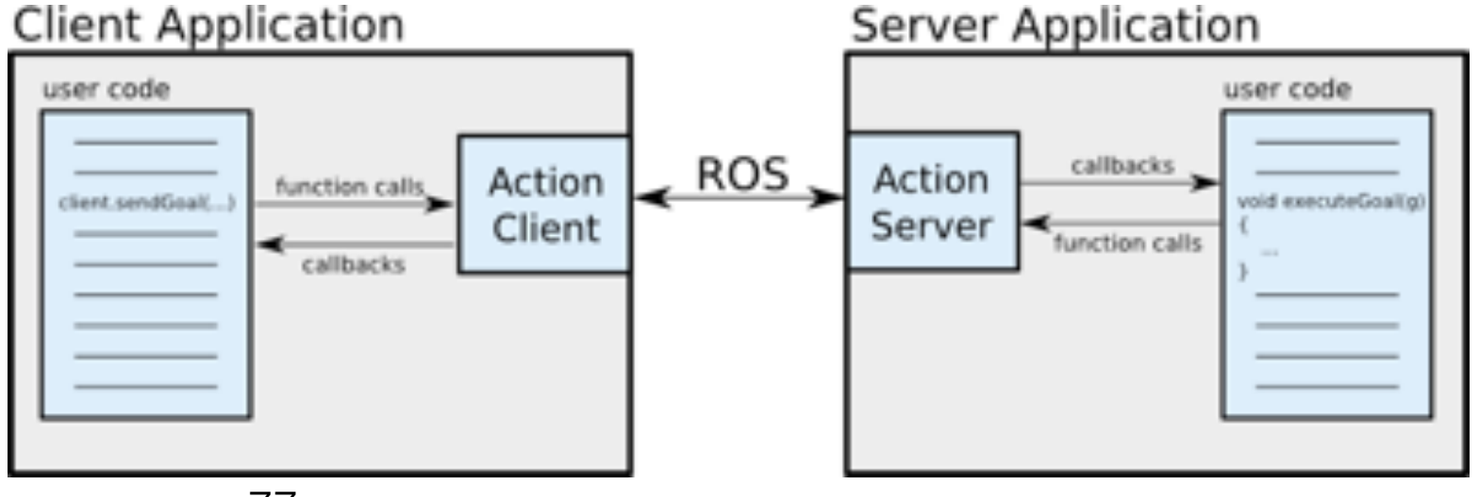
Request Type: **roscpp_tutorials/TwoIntsRequest**
Response Type: **roscpp_tutorials/TwoIntsResponse**

```
MyService.srv
 float64 x
 float64 y
 ---
 float64 result
```

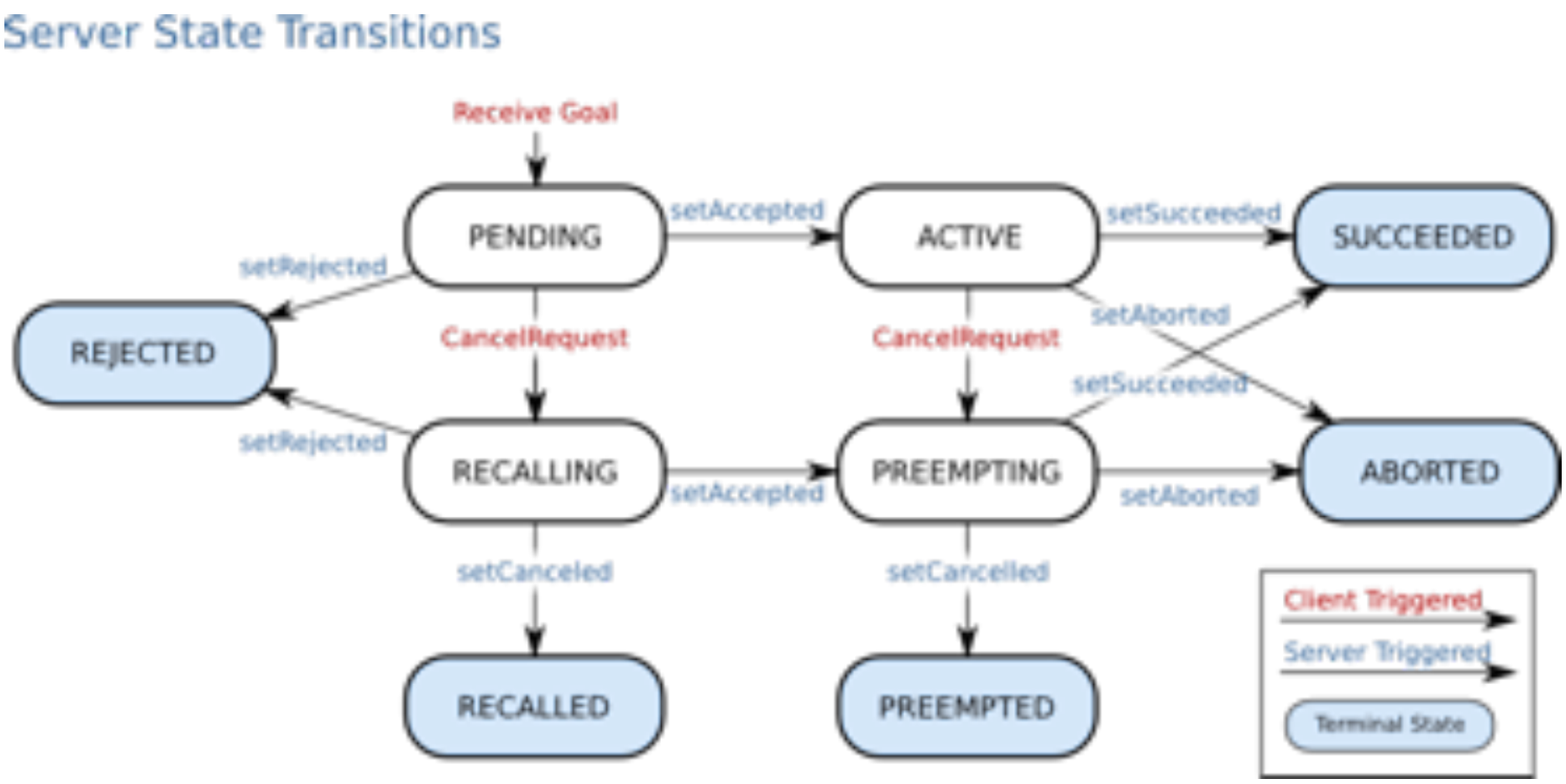
ROS Actions

- Library which offers a higher level of interaction between components
  - Client: call sendGoal, and define callbacks for feedback* and result
  - Server: executeGoal (can send feedback, and send result)
  - Built on top of topics
  - **Asynchronous**, use callback
    - Action Specification: Goal, Feedback, & Result

```
MyAction.act
 #goal
 int32 order
 ---
 #result
 int32[] sequence
 ---
 #feedback
 int32[] sequence
```



ROS Actions


Server State Transitions
Receive Goal
PENDING
setAccepted
ACTIVE
setSucceeded
SUCCEEDED
setRejected
REJECTED
CancelRequest
CancelRequest
setAborted
setSucceeded
setRejected
RECALLING
setAccepted
PREEMPTING
setAborted
ABORTED
setCanceled
setCancelled
RECALLED
PREEMPTED
Client Triggered
Server Triggered
Terminal State

ROS tools

- There are a lot of them…
- GUI to inspect nodes, topics, traces, etc
- RVIZ to visualize robots and environment (most of the time, display topics)
- Simulator Gazebo, Morse, etc.
- Check the page (slides and course) of Olivier Stasse: https://homepages.laas.fr/ostasse/drupal/content/enseignements-ros
  or the ROS course by Pascal Chauvin (ISAE)

roscore

- Handle the registration of **topics**/**services** provided by **nodes**:
  - who advertise/publish
  - who subscribe
- **One** roscore per experiment
- One **ROS_MASTER_URI** environment variable for all nodes. Example:
  `ROS_MASTER_URI=http://hostname:11311/`
- One must start one **roscore** before anything else (you cannot run any other command without it)

from ISAE - Pascal CHAUVIN

rosnode

Command-line tool for printing information about ROS Nodes.

- Commands:
  - rosnode ping
    - test connectivity to node
  - rosnode list
    - list actives notes
  - rosnode info
    - print information on the node
  - rosnode machine
    - list nodes running on a particular machine or list machines
  - rosnode kill
    - kill a running node
  - rosnode cleanup
    - purge registration information of unreachable nodes

rosnode <command> -h for more detailed usage, e.g. 'rosnode ping -h'

rostopic

Command-line tool for printing information about ROS Topics.

- Commands:
  - rostopic bw
    - display bandwidth used by topic
  - rostopic delay
    - display delay of topic from timestamp in header
  - **rostopic echo**
    - print messages to screen
  - rostopic find
    - find topics by type
  - **rostopic hz**
    - display publishing rate of topic
  - **rostopic info**
    - print information about active topic
  - **rostopic list**
    - list active topics
  - **rostopic pub**
    - publish data to topic
  - rostopic type
    - print topic or field type

rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'

rosmsg/rossrv

Command-line tool for printing information about **msg/srv**.

- Commands:
  - **rosmsg show**
  - rosmsg list
  - rosmsg md5
  - rosmsg package
  - rosmsg packages

rosmsg <command> -h for more detailed usage, e.g. 'rosmsg show -h'

rosparam

A command-line tool for getting, setting, and deleting parameters from the **ROS Parameter Server**

- Commands:
  - rosparam set
    - set parameter
  - rosparam get
    - get parameter
  - rosparam load
    - load parameters from file
  - rosparam dump
    - dump parameters to file
  - rosparam delete
    - delete parameter
  - rosparam list
    - list parameter names

rosservice

- Commands:
  - rosservice args
    - print service arguments
  - **rosservice call**
    - call the service with the provided args
  - rosservice find
    - find services by service type
  - rosservice info
    - print information about service
  - **rosservice list**
    - list active services
  - rosservice type
    - print service type
  - rosservice uri
    - print service ROSRPC uri

rosservice <command> -h for more detailed usage, e.g. 'rosservice call -h'

rosbag

A bag is a file format in ROS for storing ROS message data.
The rosbag command can **record**, **replay** and manipulate bags.

- Usage: rosbag <subcommand> [options] [args]
  - Check filter ... Compress ... Decompress …
- filter
  - Filter the contents of the bag
- fix
  - Repair the messages in a bag file so that it can be played in the current system.
- help
- info
  - Summarize the contents of one or more bag files.
- **play**
  - Play back the contents of one or more bag files in a time-synchronized fashion.
- **record**
  - Record a bag file with the contents of specified topics.
- reindex
  - Reindexes one or more bag files.

from ISAE - Pascal CHAUVIN

(c++) Talker node (python)

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;

  ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);
  while (ros::ok())
  {
    std_msgs::String msg;
    msg.data = "hello world";

    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
  }

  return 0;
}
```

```python
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

from ISAE - Pascal CHAUVIN

(c++) Listener node (python)

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle nh;

  ros::Subscriber sub = nh.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

from ISAE - Pascal CHAUVIN

Why use/not use ROS?

Pros

- Large base of users, programs, web site, even company using ROS on their robots
- Tutorials and courses
- Learning curve is reasonable
- multi-cpu
- Rich environment & ecosystem

Cons

- roscore single point of failure
- Performance issues
- Not hard real-time
- Hardly model based software development
- Some high level paradigm (actions), smach, or even T-ReX (but hardly used and now obsolete)

ROS 2

- No more Master (**roscore**) (single point of “failure”)
- ROS Comm is replaced with **DDS** (Data Distribution Service)
  - DDS, better implementation? (some commercial)
- The concepts remain the same
- Move from multi platform to one platform automatically (former nodelet shared mem)
- **ACTION**s are now “first class citizen” (like **topics** and **services**)

- But a lot of legacy software now…

Some ROS 2 commands

- For developper: **catkin** replaced by **colcon**
- All ros2 command are prefixed with "`ros2`"
  - `ros2 {node|topic|interface|service|action|…}`
- <tab> is your friend… `-h or --help also`
- Env variables to close and limit your deployment setup:
  - `ROS_LOCALHOST_ONLY` and `ROS_DOMAIN_ID`

PoCoLibs (LAAS)

POrtability and COmmunication LIBraries

- Initially developed for VxWorks, excellent real-time performance and memory footprint (portLib ensures compatibility of other Unixes with the VxWorks primitives)
- data sharing with shared memory (strongly typed)
  - proper lock mechanism
- **client/server** communication with mailbox
  - each component has an in/out mailbox (message queue)
  - receive request, reply with reports

https://git.openrobots.org/projects/pocolibs/gollum/index

PoCoLibs (LAAS)

PortLib (portability using Posix.4, (see your RT programming courses))

- Tasks (threads) management
- Semaphores (P & V)
- Watchdogs

PoCoLibs (LAAS)

ComLib

- Shared Memory (Poster)
- MailBox and csMailBox (cs Client Server)
  - attach callbacks to the Server side
  - send/receive request
  - send/receive reply
- Timers
- Posters, find, read, write, etc (properly locked)
- Events (inter process event com), wait on an event, etc

Why use/not use PoCoLibs

Pros

- Good real-time capabilities (first developed for VxWorks a real time OS)
- No memory allocation at run-time
- Good memory footprint
- It just does what is expected from a middleware, nothing more
- hardly used directly

Cons

- support for multi-cpu of posters require an additional component (PosterServ)
- used at LAAS and a few other places…

Other middleware

- DDS (used in ROS2)
- ZeroMQ
- Protocol Buffer / gRPC (used by Boston Dynamics)
- YARP
- RT Middleware (OpenRTM AIST)
- Orocos RTT (used by Orocos)

Above middleware

- Modeling tools relying on middleware
  - ROS Action lib (already presented)
  - GenoM (to be presented)
  - Orocos (real time, arms control)
  - BRICS
  - ROCK (based on Orocos RTT)
  - MARIE
  - MAUVE (ONERA)
  - etc

All have pros and cons. We focus on GenoM -> V&V