Artificial Intelligence: Methods and applications

Lecture 5: Hybrid robot architectures

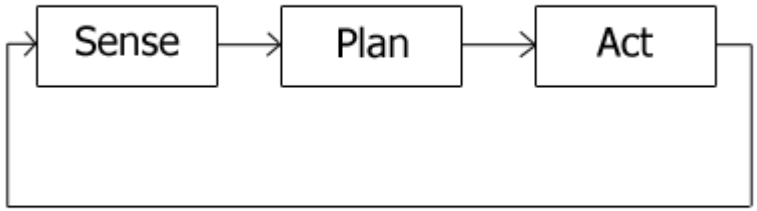
Ola Ringdahl
Umeå University
November 18, 2014

Contents

- Hybrid (reactive/deliberative) paradigm
- Examples of different hybrid architectures
- Robotic software frameworks (middleware)
  - ROS

Deliberative paradigm

- ca 1967 - ca 1990
- AI inspired
- Represent every relevant aspect of the world explicitly
- Interpret sensor data: make it a part of the world model
- Use classical planning to decide what to do



Deliberative paradigm

Pro:

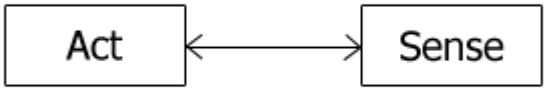
- Hierarchical (top-down) structure allow for the planning module to focus all behaviors towards a single set of goals.

Con:

- Frame problem: Updating and maintaining a sufficiently detailed world model can be too computationally expensive.
- Requires exact knowledge of the world
- Closed world assumption: static world model cause poor performance in dynamic environments

Reactive paradigm

- ca 1988 - ca 1992
- A reaction to classical AI
- Less knowledge representation and planning
- More concrete responses to the environment
- Decompose complex actions into behaviors



Reactive paradigm

Pro:

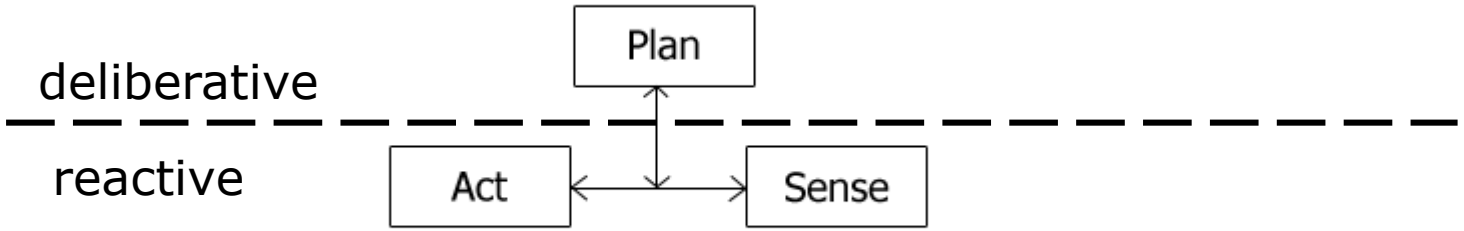
- Short reaction times
- Needs less computational resources
- Easy to implement and expand
- Emergent behavior
- Open world assumption

Con:

- Difficult to design for emergent behaviors
  - No selection of behaviors
- No planning
- No monitoring of performance
- No world representation (no internal map)

Hybrid paradigm

- Combines advantages of previous paradigms
- How to reintroduce planning into the robot architectures without running into the problems that faced deliberative robots?
  - Use reactive functions for low level control
  - Use deliberation for higher level tasks



Combining deliberative and reactive functions

- Deliberative:
  - Long time horizon
  - Global knowledge
  - Works with symbols
- Reaction:
  - Short time horizon
  - No global knowledge
  - Works with sensors and actuators
- Multi-tasking:
  - Deliberative and reactive functions execute in parallel

What should the planning component do?

- Manage behaviors
  - What is the current state of the world?
  - What is the goal state?
  - Which (combinations of / sequences of) behaviors will achieve the goal?
- Monitor performance
  - Was the latest sub-plan successful?
  - Are sensors and actuators working properly?
  - Is the sensor data compatible with my view of the world?
  - If sensors are giving contradictory data, what to do?

Common components

Most hybrid architectures incorporate (variants of) the following components:

- Mission planner
  - Interpret commands, create a high-level plan and divide it into subtasks
- Sequencer
  - Given a subtask, generate a sequence of behaviors to solve it
- Resource manager
  - Allocate resources to behaviors
- Performance monitor
  - Determine if the robot is functioning properly and making progress towards the goals
- Cartographer
  - Create, store, and maintain map information

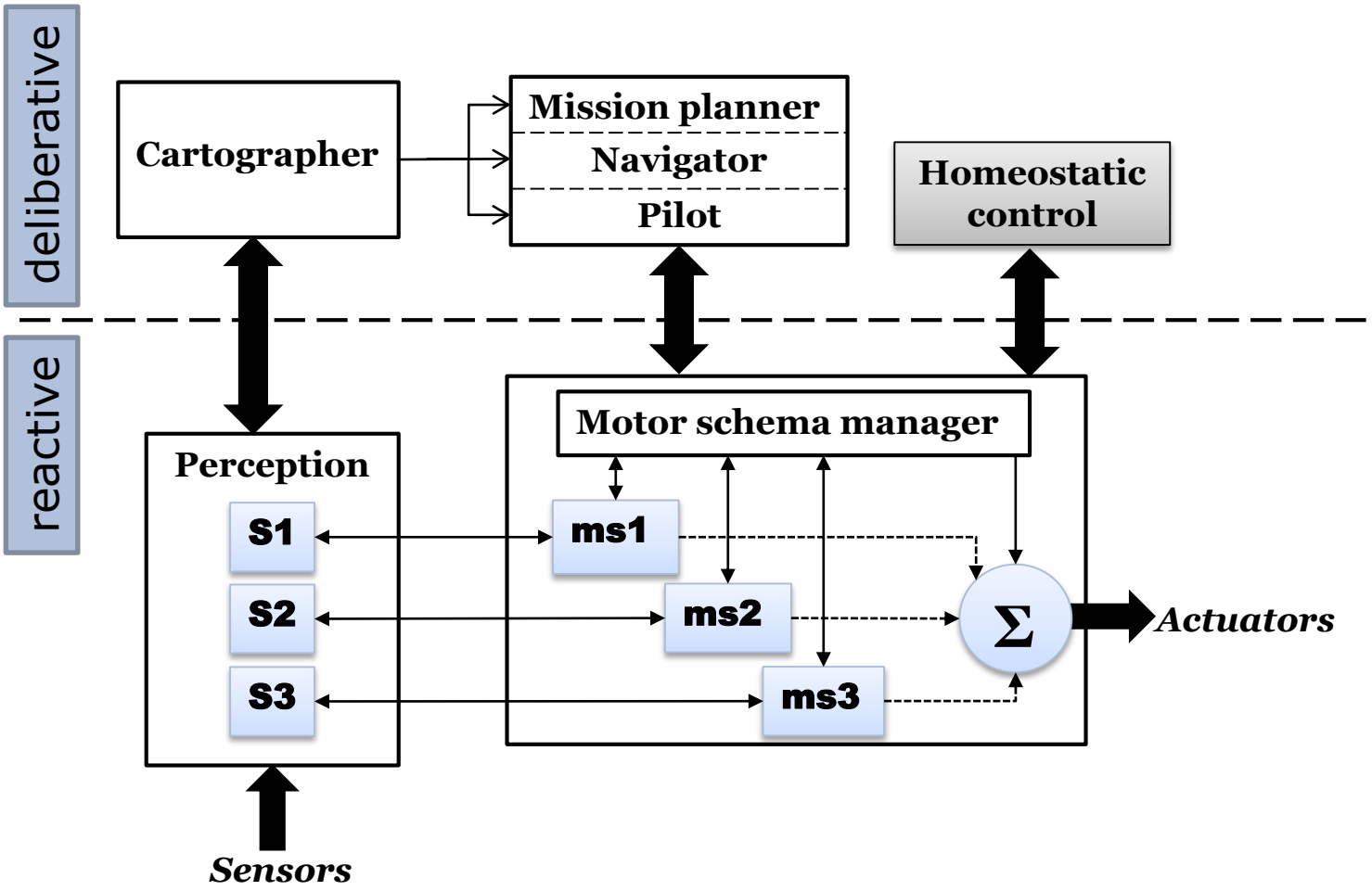
Architecture Styles

- Managerial: Divide responsibility as in business (lower levels refine the plan, can ask for help from a higher level ("boss"))
  - AuRA : Autonomous Robot Architecture
  - SFX : Sensor Fusion Effects
- State Hierarchies: Organize activities by scope of time knowledge. Three layers: past, present and future
  - 3T : 3-Tiered
- Model-Oriented: Global world model serve as virtual sensors. Similar to the hierarchical paradigm
  - Saphira
  - TCA : Task Control Architecture

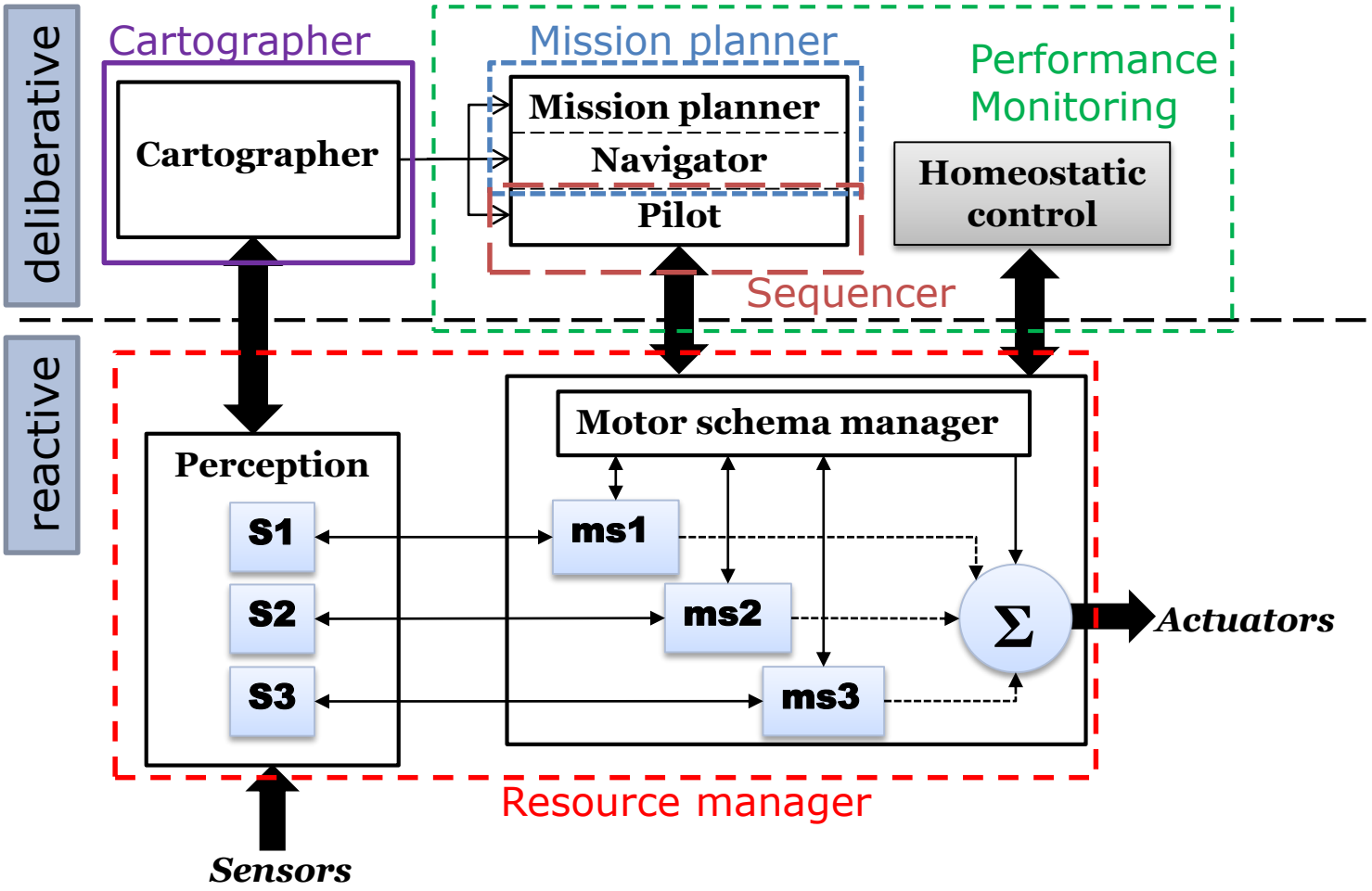
Autonomous Robot Architecture (AuRA)

- Suggested in the mid 1980s
- Ronald C. Arkin
- First hybrid architecture
- Based on schema theory
- Nested Hierarchical Controller
- Potential fields for motor schemas

AuRA architectural layout



AuRA architectural layout



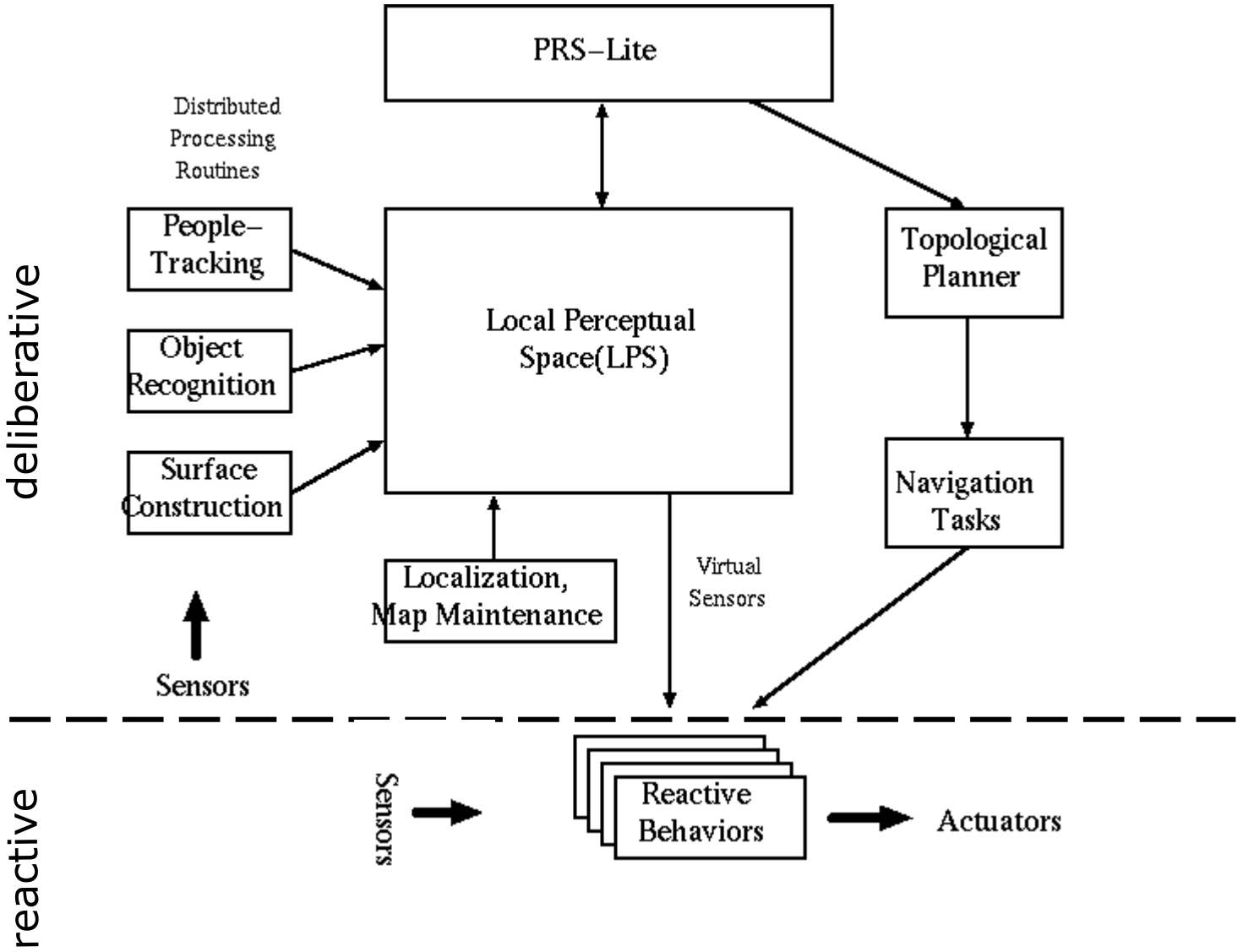
Saphira

- Ca 1997-onwards
- A model-oriented architecture
- Kurt Konolige et al.
- SRI International
- Used on Flakey and Erratic robots

Motivation:

- Coordination
- Coherence
- Communication

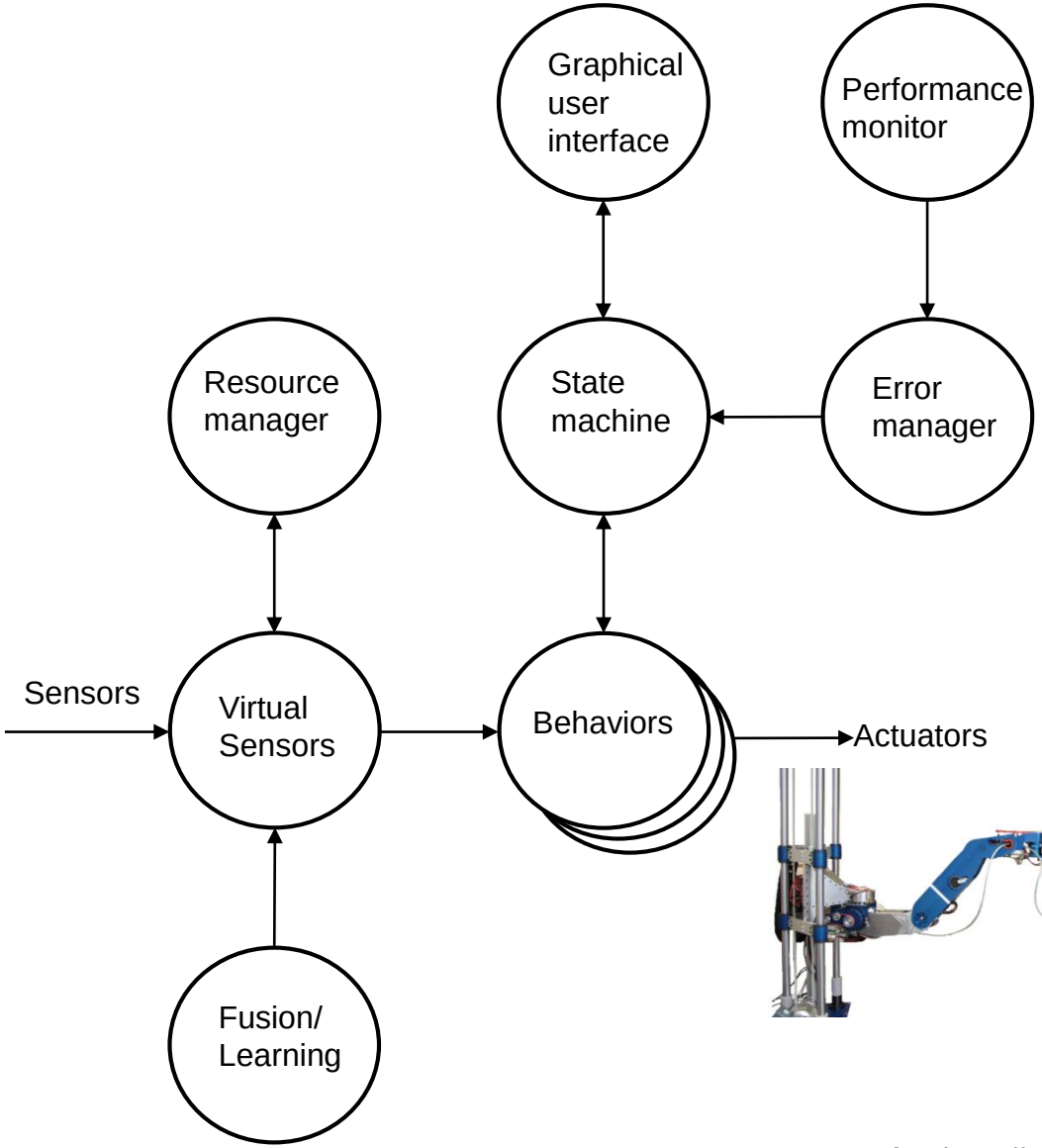
Saphira Architecture



Architecture from UMU

- In the EU project *CROPS* we developed a new architecture
  - Goal of the project: harvest fruits with a manipulator
- Uses a state machine instead of traditional planner
- Implemented in ROS (*Robotic Operating System*)
  - A robotic framework/middleware similar to MRDS that you are using in Assignment 2

Robot architecture

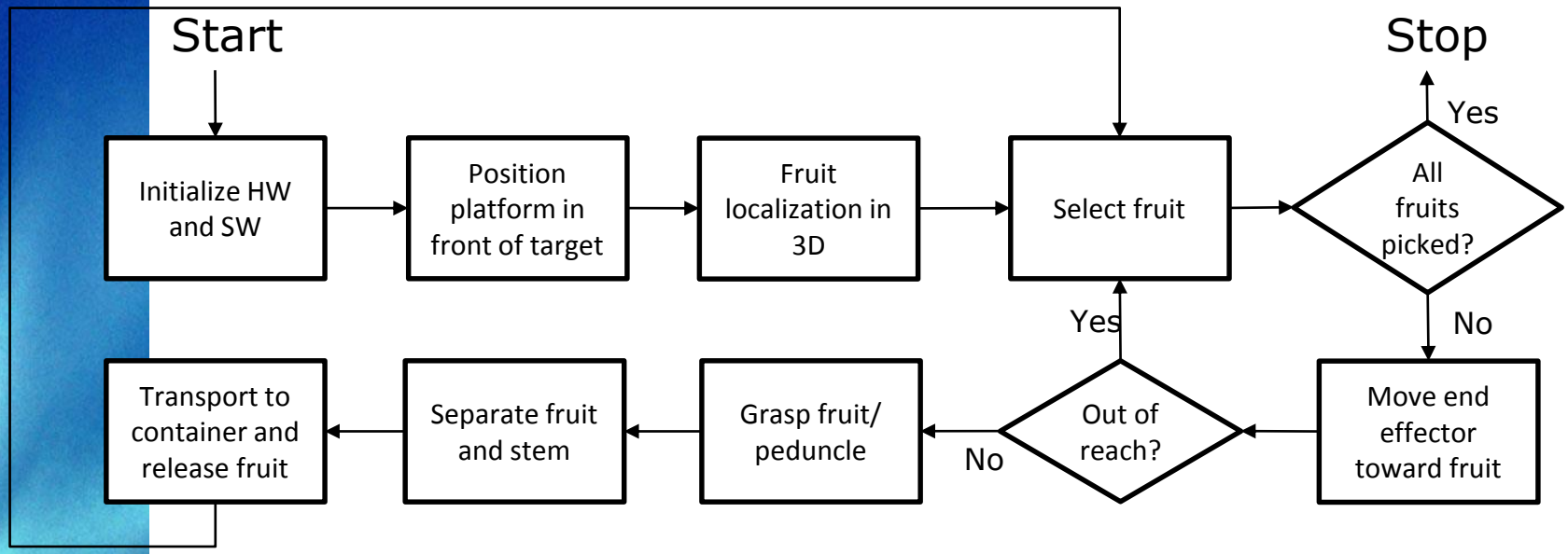


- The Main control program is implemented as a State machine that runs the main loop
- Each sub task is implemented as Behaviors
- Computational behaviors derive information like fruit location
- Acting behaviors control actuators (arm, gripper, cutter)
- Each behavior is typically implemented as a ROS node

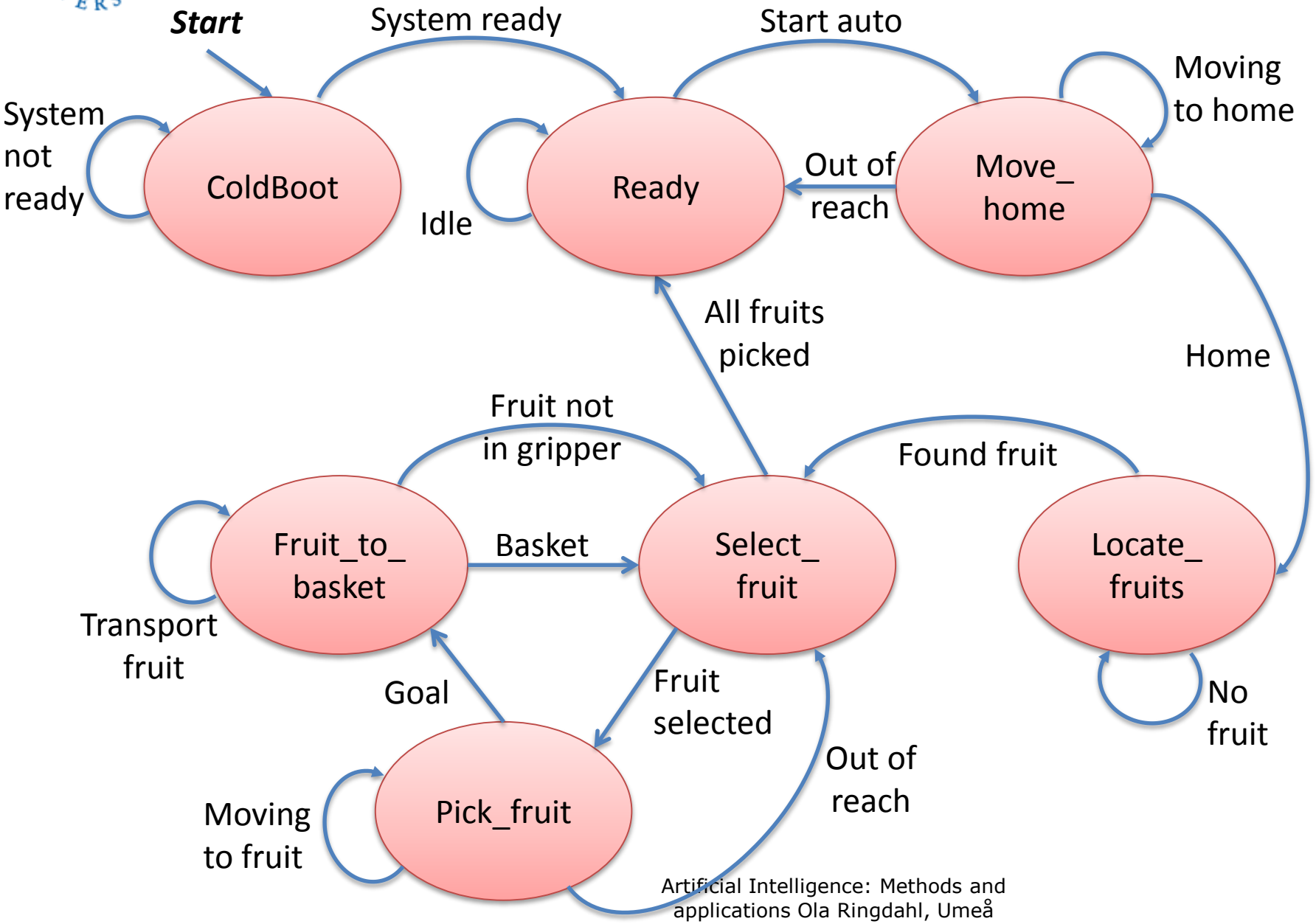
Using the framework

The developed framework may be used for several different applications.

- Let's begin with a (simplified) flowchart for a fruit picking robot:

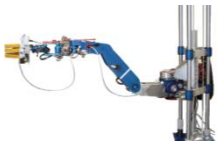


State machine



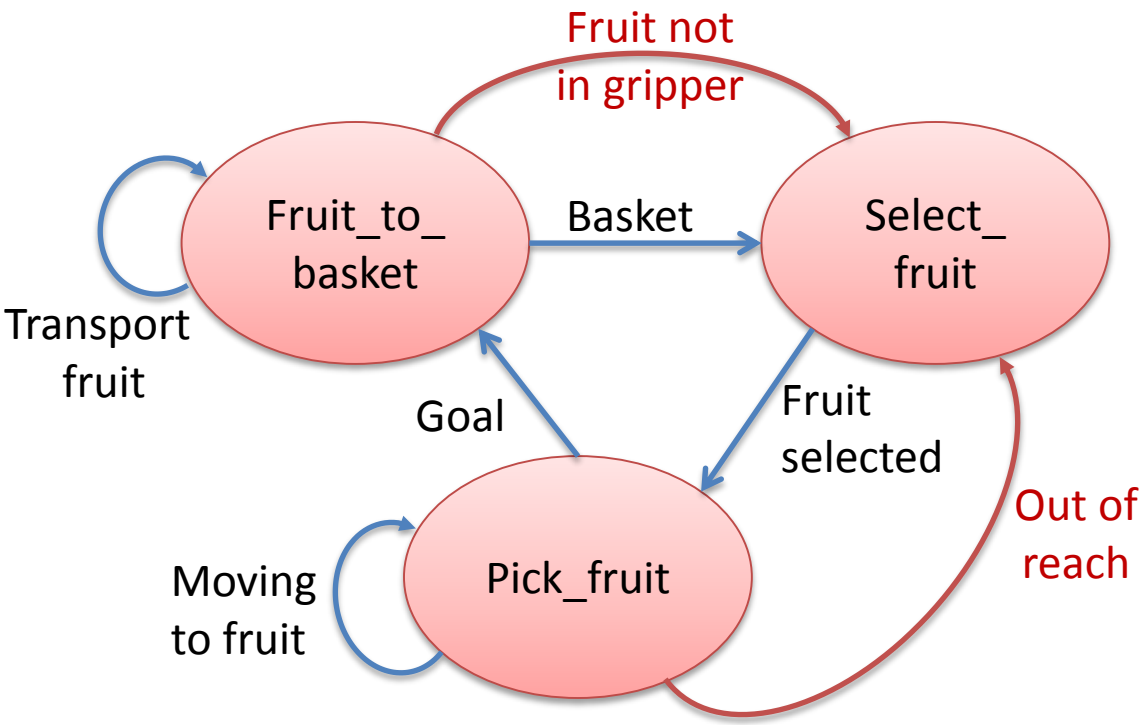
States versus behaviours

- Each state is normally conected to a behaviour that does the actual job

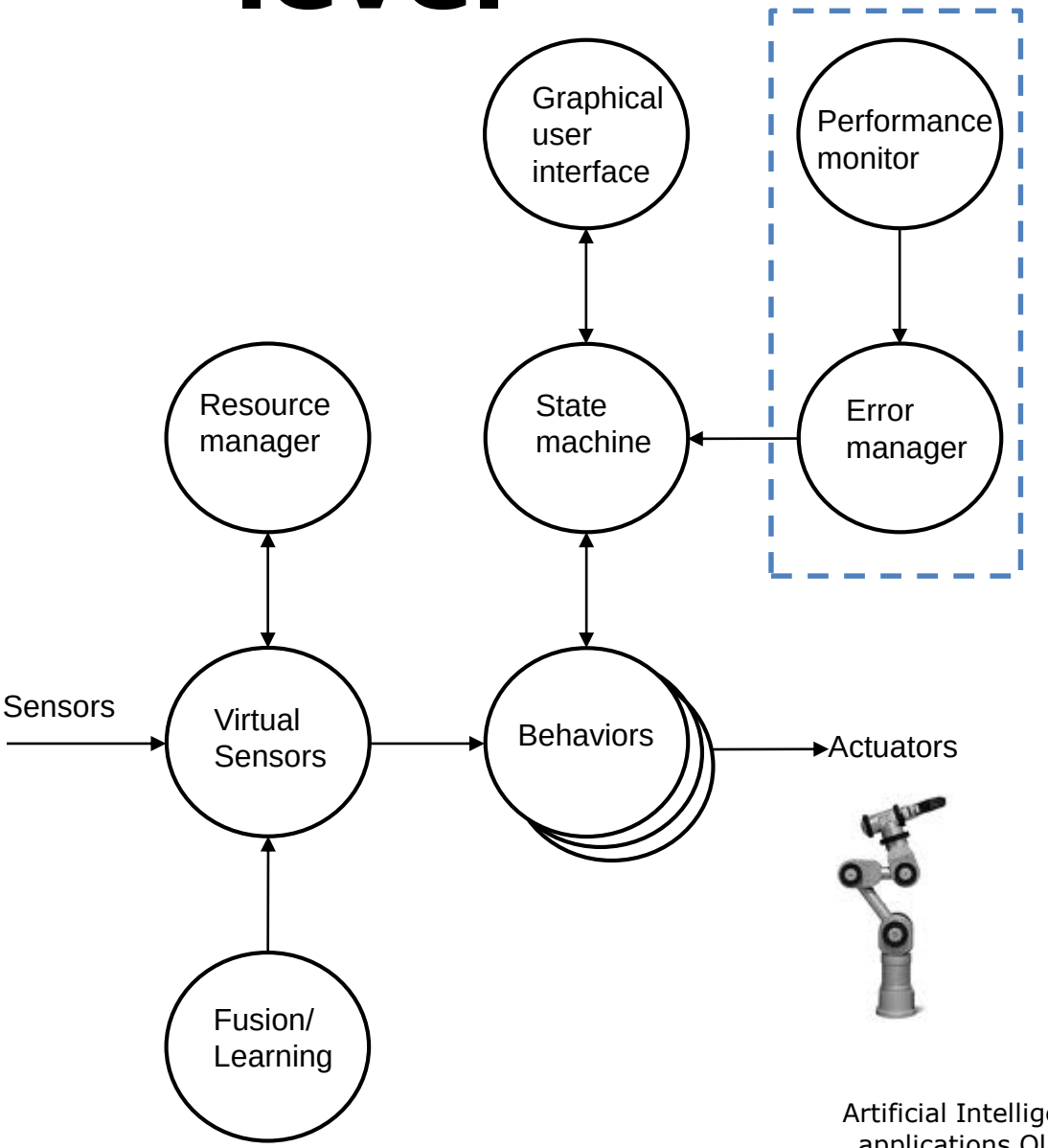


Local error handling

- Detected and dealt with in the State machine



Error handling at system level



- Errors are detected by the Performance monitor and dealt with in the Error manager
- Example:
  The camera stops working

Deliberative vs. Hybrid

Do deliberative and hybrid architectures simply come to the same conclusions in different ways?

- Hybrids are closer to software engineering principles (modularity, coherence, reuse…)
- In hybrid architectures, the world model is only used on a high level
  - Use symbolic representation for high-level ”thinking”
- The frame problem is not much of a problem for hybrids
  - Think in terms of a closed world
  - Act and sense in an open world
- Deliberative functions in hybrid architectures don’t have to do detailed planning
- Hybrids can be relevant for cognitive science

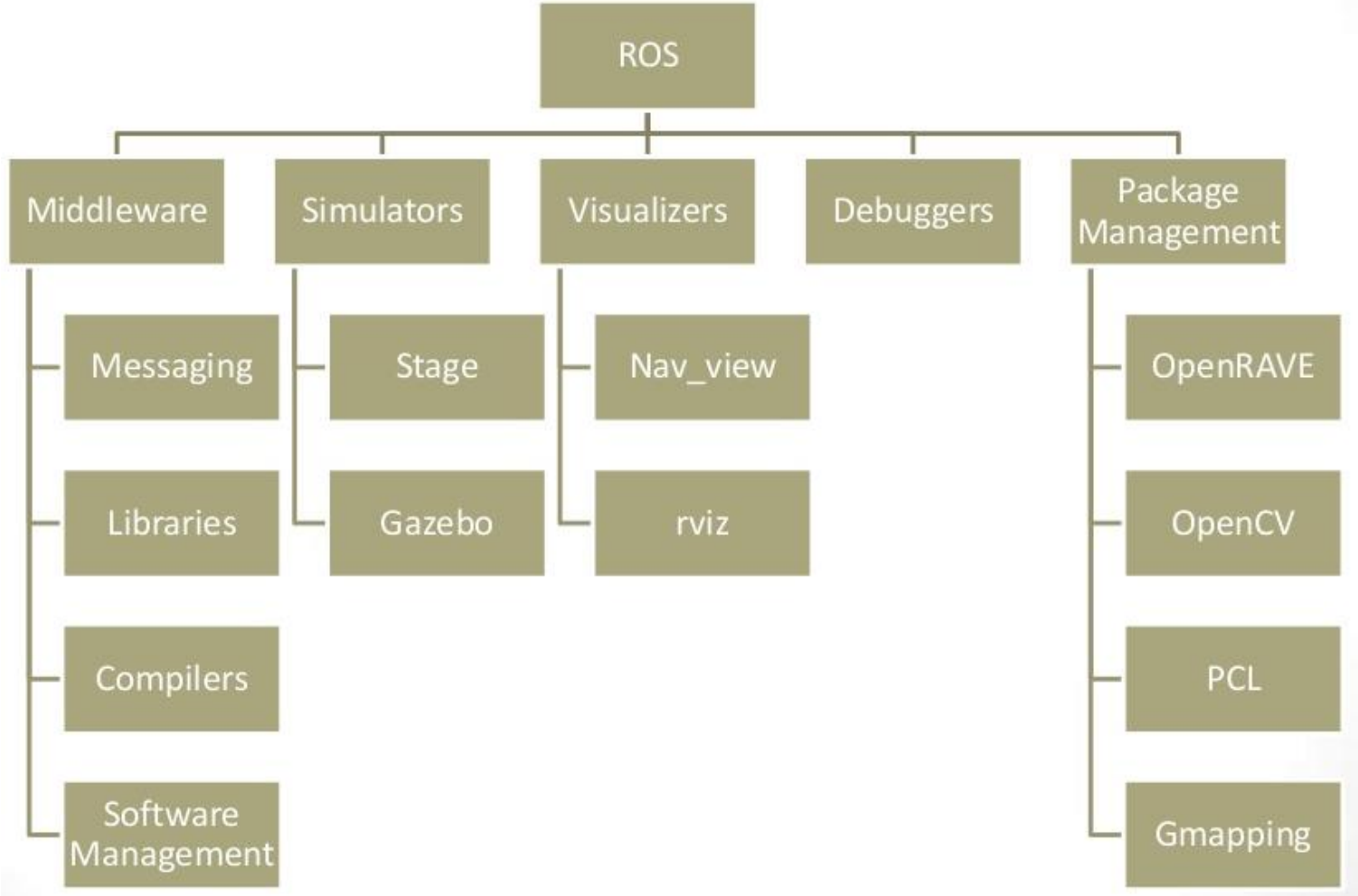
Robotic software frameworks

- A.k.a. "*middleware*"
- Creating truly robust, general-purpose robot software is *hard*.
  - Is there anything that can help us?
- Robotic framework to the rescue!
  - Collection of tools for communication, computation, configuration, coordination, …
  - Simplifies the development of robotic applications
- A lot of frameworks have been developed, but ROS and MRDS are most popular today
  - Let's look at ROS as an example of what a modern framework looks like
  - MRDS (assignment 2) is very similar

What is ROS?

- ROS is a collection of tools to help software developers create robot applications
  - First developed 2007 at Stanford. Since 2008 by Willow Garage, a robotics research institute/incubator
- Example of what is provided:
  - hardware abstraction, device drivers, visualizers, message-passing, package management, logging, 2300+ libraries
- Distributed: support for running on multiple computers and robots simultaneously
- Many user-contributed packages containing functions such as SLAM, planning, perception, simulation, etc.

Components of ROS



ROS applications

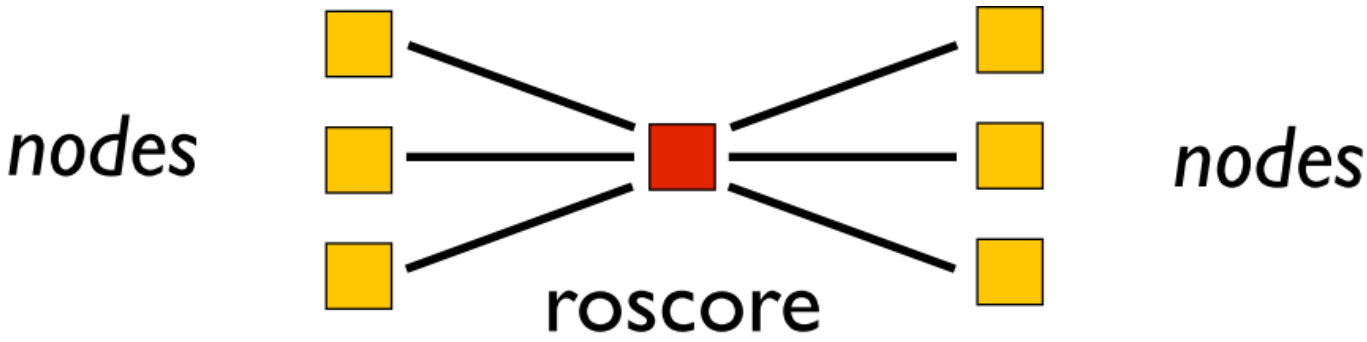
- ROS comes with many built in libraries, e.g:
  - Perception
  - Object identification
  - Face recognition
  - Gesture recognition
  - Motion tracking
  - Stereo vision
  - Mobile robotics
  - Planning and control
  - Grasping
  - …
- Idea: don’t invent the wheel again!
  - Concentrate on the task that you want to do

Client libraries

- **roscpp:** C++ implementation. The most widely used library and is designed to be the high performance library for ROS.
- **rospy**: Python. Good where performance is not important, e.g. GUI, configuration and init. code
- **roslisp**: LISP. Currently used for the development of planning libraries. It supports both standalone node creation and interactive use in a running ROS system.
- **rosjava** (experimental): an implementation of ROS in pure-Java with Android support
- As of January 2014, MathWorks officially supports ROS in **Matlab**
  - Create ROS nodes in MATLAB and exchange messages with other nodes on the network.
  - Provides a MATLAB API interface based on rosjava
  - http://www.mathworks.se/hardware-support/robot-operating-system.html

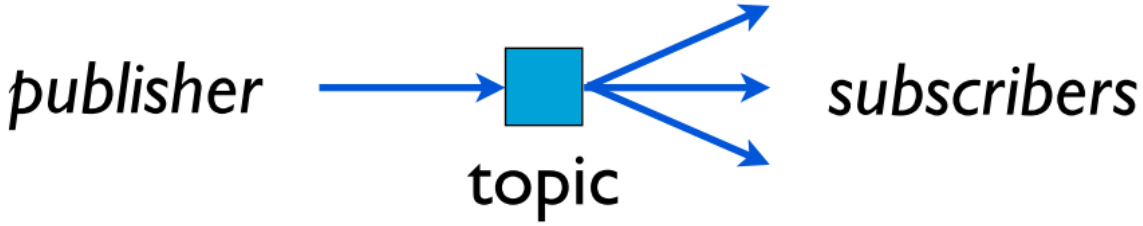
Basic concept: Node

- Modularization in ROS is achieved by separated operating system processes
- Node = a process that uses ROS framework
- Nodes may reside in different machines transparently
- Nodes get to know one another via roscore
  - roscore acts primarily as a name server



Basic concept: Topic

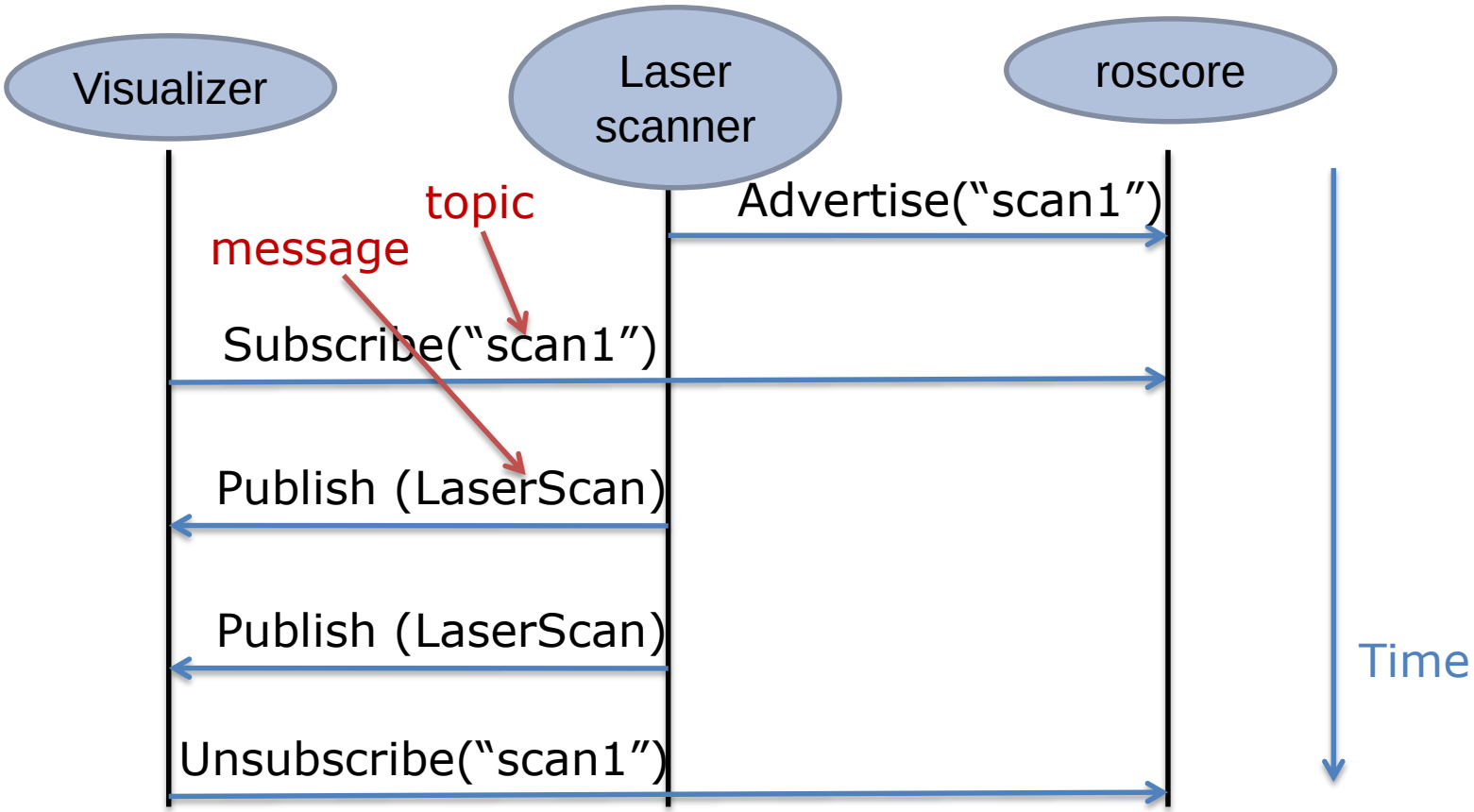
- *Topic* is a mechanism to send messages from a node to one or more nodes
- Follows a publisher-subscriber design pattern



- *Publish* = to send a message to a topic
- *Subscribe* = get called whenever a message is published
- Published messages are broadcast to all Subscribers
- Messages are defined in .msg files
- Example: Laser scanner publishing scan data

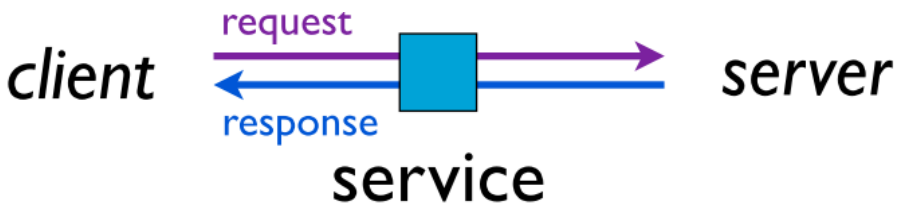
Example: Topic

Publish/subscribe (push)



Basic concept: Service

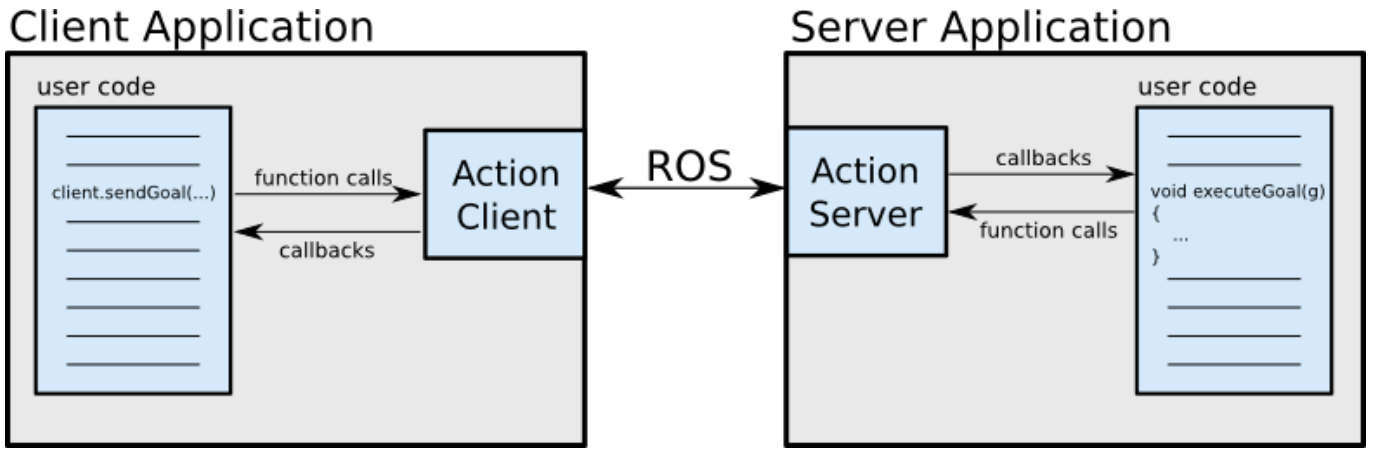
- Service is a mechanism for a node to send a request to another node and receive a response in return
- Follows a request-response design pattern



- A service is called with a request message, and in return, a response message is returned
- Example: take a picture, return the image
- Services are defined in .srv-files
  - Basically contains two message definitions

Basic concept: Actions

- In some cases, if a *service* takes a long time to execute, the user might want to cancel the request or get periodic feedback
- actionlib provides tools to create servers that execute goals that can be preempted. It also provides a client interface in order to send requests to the server.



Basic concept: Actions (2)

Example: controlling a robot arm

- Action client: A node sending commands to the arm (e.g. moveToFruit).
  - Can anytime cancel the goal or set a new one
- Action server: Manipulator motion controller
  - Move the arm towards the goal
  - Sends continuous feedback about the progress to the client
  - When goal is reached, the result is sent to the client

- Action message contains goal, feedback, and result and is specified in a .action file

ROS messages

- All messages (*topic*, *service* and *actions*) are defined in text files
- Messages can include:
  - Primitive types (integer, float, Boolean, etc.)
  - Names of other message descriptions
  - Arrays of the above (e.g. float32[] ranges)
  - The special Header type (contains timestamps and some other stuff)
- Example:

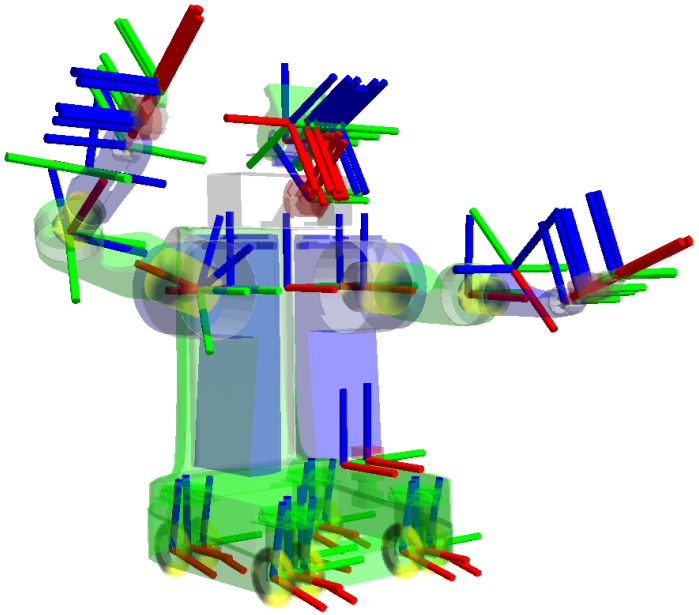
```
# This represents an orientation with reference coordinate
frame and timestamp.
Header header
Quaternion orientation
```

Parameter server

- Can be used by nodes to store and retrieve parameters at runtime
- Best used for static, non-binary data such as configuration parameters
- Meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

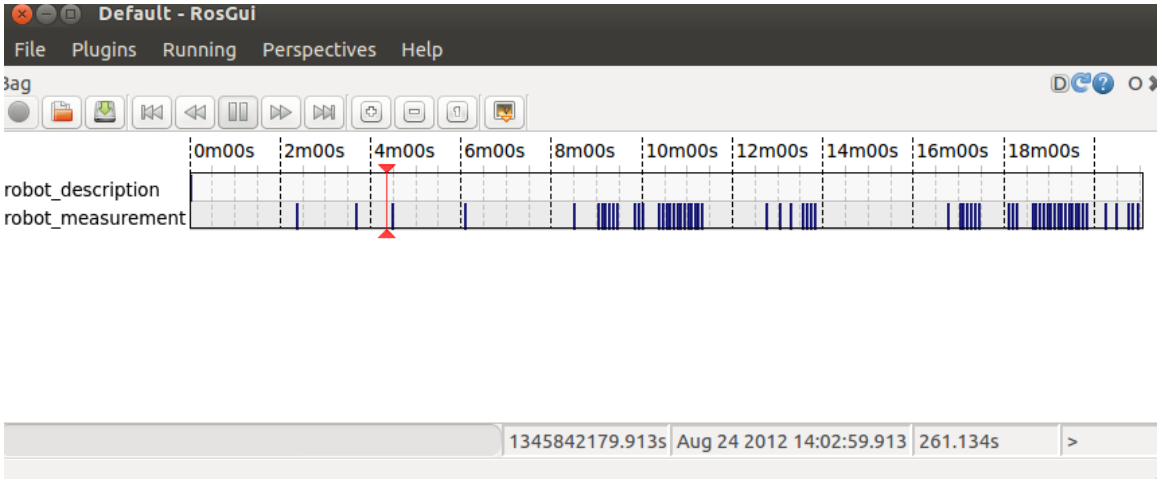
Coordinate frames (tf)

- tf keeps track of multiple coordinate frames over time
- Let the user transform points, vectors, etc between any two coordinate frames at any desired point in time.



Recording and playback of messages

- ROS contains utilities for recording messages to file and play them back later
  - rqt_bag provides a GUI plugin for recording, displaying, and replaying ROS bag files.
  - rosbag provides command line interface
  - C++ and Python APIs also available
- Replaying a file is the same as having nodes sending the same data



External libraries

ROS provides integration with several large external libraries:

- **OpenCV** – Computer Vision
- **Gazebo** – 3D physics simulator
- **PCL** (Point Cloud Library) – Working with point clouds (depth images)
- **MoveIt!** – Motion planning for mobile manipulators

Robots using ROS

Complete list on http://wiki.ros.org/Robots

Fraunhofer IPA Care-O-bot

Videre Erratic

TurtleBot

Aldebaran Nao

Lego NXT

Shadow Hand

Willow Garage PR2

iRobot Roomba

Robotnik Guardian

Merlin miabotPro

AscTec Quadrotor

CoroWare Corobot

Clearpath Robotics Husky

Clearpath Robotics Kingfisher

Festo Didactic Robotino

Gostai Jazz

Neobotix mp-500

Neobotix mpo-500

DARPA Grand Challenge

- The first long distance competition for driverless cars in the world
  - Goal: autonomous military vehicles
- 2004: No one finished the race.
  - Best: 12 of 240 km before getting stuck
- 2005: 5 of 23 vehicles completed the course
  - Winner: *Stanley* from Stanford University
- 2007: *Urban Challenge*; 96 km urban area course to be completed in < 6 hours
  - Obey traffic laws, deal with other traffic (autonomous and human operated)
  - Winner: Tartan Racing, CMU