Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Institute for AI and Autonomous Systems

Tools for Robot Software Development

Dr. Alex Mitrevski
Master of Autonomous Systems

Structure

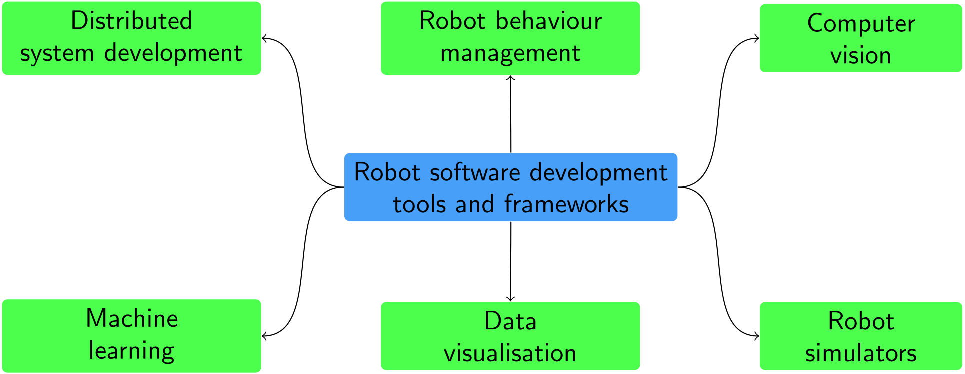
- Preliminaries
- Distributed software development
- Behaviour management: State machines and behaviour trees
- A bag of (other) tools

Preliminaries

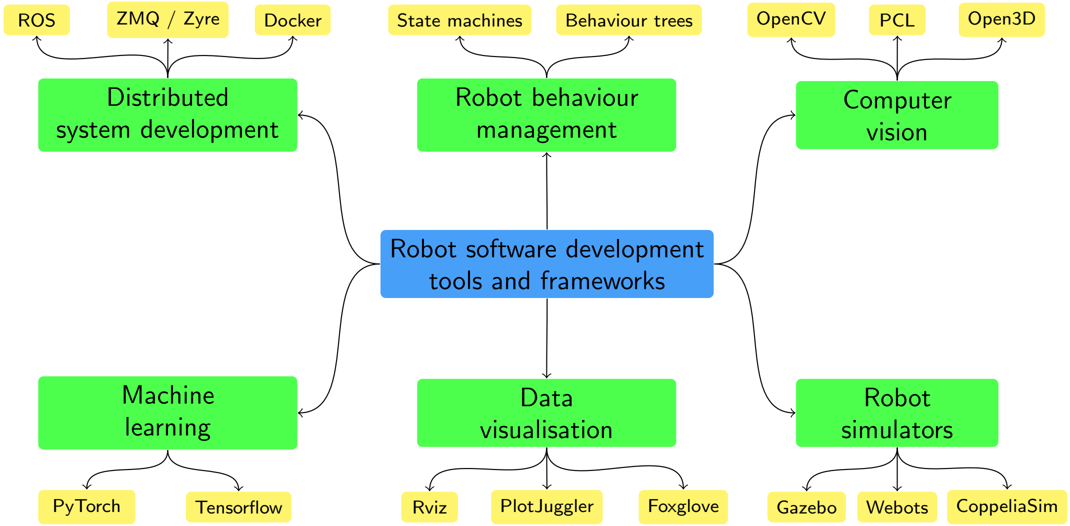
Robot Software Development as a Complex, Diverse Process

- Robotics software development is **quite diverse** and **involves a large variety of tools and frameworks**
  - Diversity exists in terms of the type of software and the purpose for which it is developed
- In this lecture, we will briefly introduce various relevant tools that are commonly used in practice
  - **Frameworks and tools evolve** or **get replaced by new ones over time** — robot software development is a dynamic process, so it is important to keep up with new developments
  - We will focus on frameworks that have either been in use for a prolonged period and have thus stood the test of time, or are becoming more important due to recent research advances

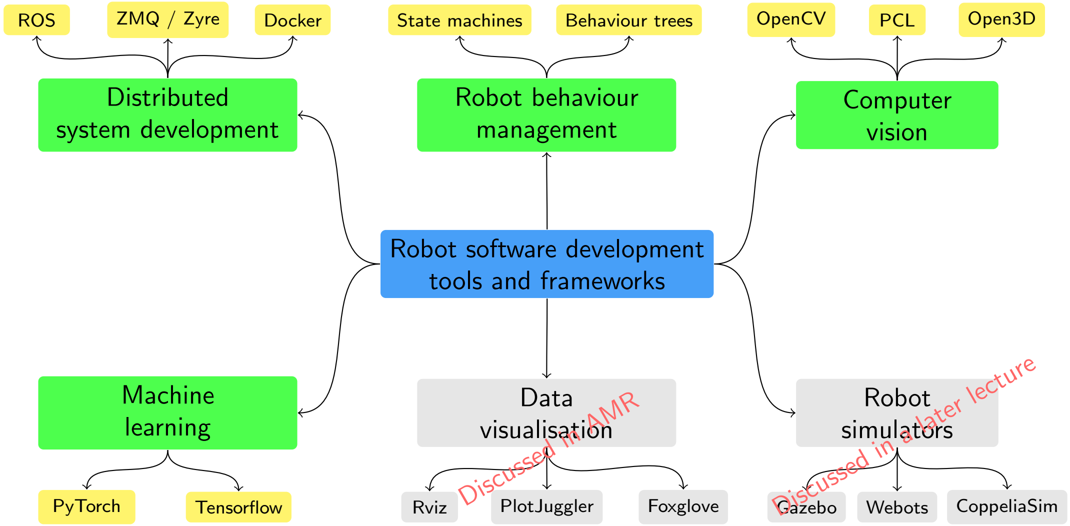
Overview of Robot Development Frameworks



Overview of Robot Development Frameworks



Overview of Robot Development Frameworks



Distributed Software Development

Distributed Development

- As discussed in the previous lecture, **a robot is a distributed system**, with components running over multiple machines on the same network or potentially even online

Distributed Development

- As discussed in the previous lecture, **a robot is a distributed system**, with components running over multiple machines on the same network or potentially even online
- A variety of distributed development tools have been used in robotics over the years
  - Some prominent examples are the **Common Object Request Broker Architecture (CORBA)** and **Internet Communications Engine (Ice)**

Distributed Development

- As discussed in the previous lecture, **a robot is a distributed system**, with components running over multiple machines on the same network or potentially even online
- A variety of distributed development tools have been used in robotics over the years
  - Some prominent examples are the **Common Object Request Broker Architecture (CORBA)** and **Internet Communications Engine (Ice)**
- The **Robot Operating System (ROS) has evolved into a de facto standard for robot software development**
  - ROS is standard at least in the academic setting — essentially all research robot platforms provide a ROS interface and most popular sensors have a ROS driver
  - There are, however, other frameworks that can be used to achieve similar goals and are sometimes more suitable

Publish-Subscribe vs. Service-Client

As discussed in the previous lecture, distributed systems can use publish-subscribe or service-client communication

Publish-Subscribe vs. Service-Client

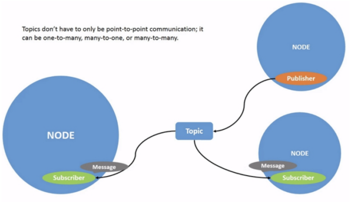
As discussed in the previous lecture, distributed systems can use publish-subscribe or service-client communication — **when do you use which pattern?**

Publish-Subscribe vs. Service-Client

As discussed in the previous lecture, distributed systems can use publish-subscribe or service-client communication — **when do you use which pattern?**

Publish-subscribe is an **asynchronous pattern**, where **the arrival and processing of data does not need to immediately trigger a subsequent execution**

- Enables **many components to receive the same message**
- **The publisher is not blocked after publishing a message**
- Very useful for data arriving at high frequencies (e.g. sensor data)



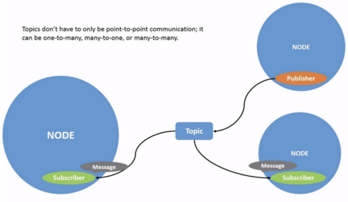
http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

Publish-Subscribe vs. Service-Client

As discussed in the previous lecture, distributed systems can use publish-subscribe or service-client communication — **when do you use which pattern?**

Publish-subscribe is an **asynchronous pattern**, where **the arrival and processing of data does not need to immediately trigger a subsequent execution**

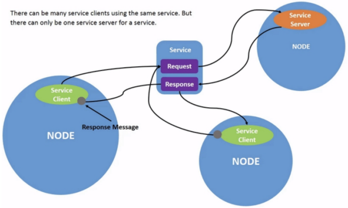
- Enables **many components to receive the same message**
- **The publisher is not blocked after publishing a message**
- Very useful for data arriving at high frequencies (e.g. sensor data)



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

Service-client is a **synchronous pattern**, where **a request from a client triggers an immediate execution from the server**, which then **sends a response back to the client**

- Enables **peer-to-peer communication between components**
- **The client is blocked after calling the server** and waits until the server responds back or times out — but ROS2 allows asynchronous requests
- Useful when the execution of the caller depends on something provided by the server (e.g. retrieving data from a robot's knowledge base)



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html

Robot Operating System (ROS)[1]

- ROS is a **middleware that enables developing complex applications for robots**

[1] https://www.ros.org

Robot Operating System (ROS)[1]

- ROS is a **middleware that enables developing complex applications for robots**
- The major driver for ROS is its **open source nature**, which invites community contributions

[1] https://www.ros.org

Robot Operating System (ROS)[1]

- ROS is a **middleware that enables developing complex applications for robots**
- The major driver for ROS is its **open source nature**, which invites community contributions
- ROS has **traditionally been dominated by two programming languages — Python and C++** — although other languages are supported as well

[1] https://www.ros.org

Robot Operating System (ROS)[1]

- ROS is a **middleware that enables developing complex applications for robots**
- The major driver for ROS is its **open source nature**, which invites community contributions
- ROS has **traditionally been dominated by two programming languages — Python and C++** — although other languages are supported as well
- **The original ROS was superceded by ROS2 a few years ago**; new Ubuntu distributions (since 22.04) only support ROS2

[1] https://www.ros.org

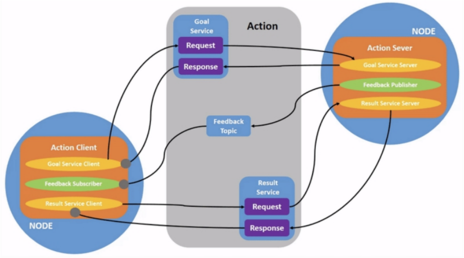
Robot Operating System (ROS)[1]

- ROS is a **middleware that enables developing complex applications for robots**
- The major driver for ROS is its **open source nature**, which invites community contributions
- ROS has **traditionally been dominated by two programming languages — Python and C++** — although other languages are supported as well
- **The original ROS was superceded by ROS2 a few years ago**; new Ubuntu distributions (since 22.04) only support ROS2

You were already introduced to ROS in the MAS Foundations Course — we will not repeat how it works in this lecture

[1] https://www.ros.org

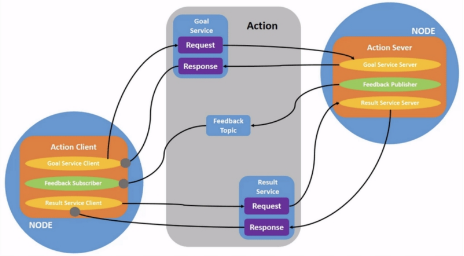
ROS Services vs. Actions



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

- In addition to services, ROS also includes the concept of actions — as in the case of services, **the provider of an action is called an action server** and **the caller is an action client**

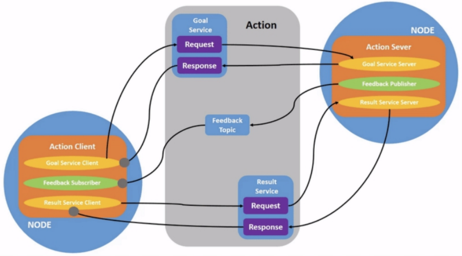
ROS Services vs. Actions



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

- In addition to services, ROS also includes the concept of actions — as in the case of services, **the provider of an action is called an action server** and **the caller is an action client**
- **Actions are meant for long(er)-running operations** during which **feedback on the server's execution can be received**
  - ROS actions are intuitively a suitable concept for managing the execution of robot actions (e.g. picking an object)

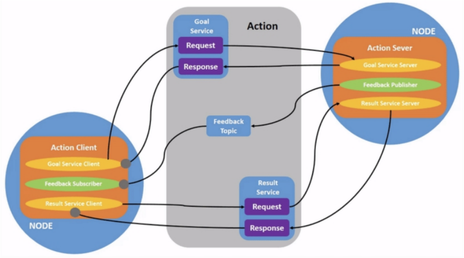
ROS Services vs. Actions



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

- In addition to services, ROS also includes the concept of actions — as in the case of services, **the provider of an action is called an action server** and **the caller is an action client**
- **Actions are meant for long(er)-running operations** during which **feedback on the server's execution can be received**
  - ROS actions are intuitively a suitable concept for managing the execution of robot actions (e.g. picking an object)
- Actions are **executed asynchronously** — the execution of the caller is not blocked while the action server is running

ROS Services vs. Actions



http://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

- In addition to services, ROS also includes the concept of actions — as in the case of services, **the provider of an action is called an action server** and **the caller is an action client**
- **Actions are meant for long(er)-running operations** during which **feedback on the server's execution can be received**
  - ROS actions are intuitively a suitable concept for managing the execution of robot actions (e.g. picking an object)
- Actions are **executed asynchronously** — the execution of the caller is not blocked while the action server is running
- Calling an action server is **not peer-to-peer communication** — actions expose ROS topics

Problems with ROS

- **ROS is a rather large framework** — it has lots of dependencies and components that are part of the framework by default
  - It does not make much sense to install and use ROS in cases where simple communication between components is desired

Problems with ROS

- **ROS is a rather large framework** — it has lots of dependencies and components that are part of the framework by default
  - It does not make much sense to install and use ROS in cases where simple communication between components is desired
- **Overreliance on ROS can encourage "lazy" development** that does not follow good development practices
  - Network communication is slow and not very reliable — ideally, it should be avoided whenever possible, particularly for operations that require high frequency and high reliability

Problems with ROS

- **ROS is a rather large framework** — it has lots of dependencies and components that are part of the framework by default
  - It does not make much sense to install and use ROS in cases where simple communication between components is desired
- **Overreliance on ROS can encourage “lazy” development** that does not follow good development practices
  - Network communication is slow and not very reliable — ideally, it should be avoided whenever possible, particularly for operations that require high frequency and high reliability
- We will now look at a few alternatives to ROS, which can be more suitable to use in some cases

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**

[2] https://zeromq.org

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**
- **Communication in ZMQ is performed over sockets of different types**, which can use different protocols (e.g. TCP)

[2] https://zeromq.org

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**
- **Communication in ZMQ is performed over sockets of different types**, which can use different protocols (e.g. TCP)
- **ZMQ supports the same communication patterns as ROS** — publish-subscribe and service-client can both be implemented

[2] https://zeromq.org

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**
- **Communication in ZMQ is performed over sockets of different types**, which can use different protocols (e.g. TCP)
- **ZMQ supports the same communication patterns as ROS** — publish-subscribe and service-client can both be implemented
- While ROS primarily supports C++ and Python, ZMQ supports a large variety of languages, which provides larger development flexibility

[2] https://zeromq.org

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**
- **Communication in ZMQ is performed over sockets of different types**, which can use different protocols (e.g. TCP)
- **ZMQ supports the same communication patterns as ROS** — publish-subscribe and service-client can both be implemented
- While ROS primarily supports C++ and Python, ZMQ supports a large variety of languages, which provides larger development flexibility

Adapted from https://github.com/alex-mitrevski/action-execution

```
class ExecutionDataLogger(object):
    def __init__(self, port):
        self.context = zmq.Context()
        self.socket = self.context.socket(zmq.PUB)
        self.socket.bind("tcp://*:{0}".format(port))

    def log_model_data(self, action_name, document_data):
        json_data = json.dumps(document_data)
        self.socket.send_multipart([bytearray(action_name, 'utf8'),
                                    bytearray(json_data, 'utf8')])
```

[2] https://zeromq.org

ZeroMQ (ZMQ)[2]

- ZMQ is a more lightweight communication framework, where **message exchange occurs without a broker**
- **Communication in ZMQ is performed over sockets of different types**, which can use different protocols (e.g. TCP)
- **ZMQ supports the same communication patterns as ROS** — publish-subscribe and service-client can both be implemented
- While ROS primarily supports C++ and Python, ZMQ supports a large variety of languages, which provides larger development flexibility

Adapted from https://github.com/alex-mitrevski/action-execution

```
class ExecutionDataLogger(object):
    def __init__(self, port):
        self.context = zmq.Context()
        self.socket = self.context.socket(zmq.PUB)
        self.socket.bind("tcp://*:{0}".format(port))

    def log_model_data(self, action_name, document_data):
        json_data = json.dumps(document_data)
        self.socket.send_multipart([bytearray(action_name, 'utf8'),
                                    bytearray(json_data, 'utf8')])
```

Adapted from https://github.com/ropod-project/black-box

```
class JsonZmqReader(object):
    def __init__(self, url, port, topic_params):
        self.publisher_url = url
        self.port = port
        self.context = zmq.Context()
        self.socket = self.context.socket(zmq.SUB)
        self.topic_names = [topic.name for topic in topic_params]
        for topic in self.topic_names:
            self.socket.setsockopt_string(zmq.SUBSCRIBE, topic)
        self.sub_thread = None

    def start_logging(self):
        self.socket.connect('{0}:{1}'.format(self.publisher_url,
              self.port))
        self.sub_thread = threading.Thread(target=self.log_msg)
        self.sub_thread.start()

    def log_msg(self):
        topic, msg = self.socket.recv_multipart()
        # process the message
```

[2] https://zeromq.org

Zyre[3]

- In some robotics applications, **network communication between components needs to be flexible** and **the network should enable new components to join and leave at any point** (e.g. in a multi-robot system)

[3] https://github.com/zeromq/zyre

Zyre[3]

- In some robotics applications, **network communication between components needs to be flexible** and **the network should enable new components to join and leave at any point** (e.g. in a multi-robot system)
- Zyre is a ZMQ-based library in which **named nodes** send UDP beacons, thereby **allowing automatic discovery of components**

[3] https://github.com/zeromq/zyre

Zyre[3]

- In some robotics applications, **network communication between components needs to be flexible** and **the network should enable new components to join and leave at any point** (e.g. in a multi-robot system)
- Zyre is a ZMQ-based library in which **named nodes** send UDP beacons, thereby **allowing automatic discovery of components**
- In Zyre, communication is organised into **groups**, which nodes can join or leave as necessary

[3] https://github.com/zeromq/zyre

Zyre[3]

- In some robotics applications, **network communication between components needs to be flexible** and **the network should enable new components to join and leave at any point** (e.g. in a multi-robot system)
- Zyre is a ZMQ-based library in which **named nodes** send UDP beacons, thereby **allowing automatic discovery of components**
- In Zyre, communication is organised into **groups**, which nodes can join or leave as necessary
- Communication between Zyre nodes can be performed **by broadcasting messages to all members of a group** (a process known as **shouting**) or **in a peer-to-peer fashion** (a process known as **whispering** to a node)

[3] https://github.com/zeromq/zyre

Zyre[3]

- In some robotics applications, **network communication between components needs to be flexible** and **the network should enable new components to join and leave at any point** (e.g. in a multi-robot system)
- Zyre is a ZMQ-based library in which **named nodes** send UDP beacons, thereby **allowing automatic discovery of components**
- In Zyre, communication is organised into **groups**, which nodes can join or leave as necessary
- Communication between Zyre nodes can be performed **by broadcasting messages to all members of a group** (a process known as **shouting**) or **in a peer-to-peer fashion** (a process known as **whispering** to a node)

Adapted from https://github.com/ropod-project/black-box

```python
class BlackBoxQueryInterface(RopodPyre):
    def __init__(self, data_sources, black_box_id, groups):
        super(BlackBoxQueryInterface, self).__init__({'node_name'
                : black_box_id + '_query_interface', 'groups':
                groups, 'message_types': list()})
        self.data_sources = data_sources
        self.black_box_id = black_box_id
        self.start()

    def zyre_event_cb(self, zyre_msg):
        if zyre_msg.msg_type in ("SHOUT", "WHISPER"):
            response_msg = self.receive_msg_cb(zyre_msg.
                    msg_content)
            if response_msg:
                self.whisper(response_msg, zyre_msg.peer_uuid)

    def receive_msg_cb(self, msg):
        dict_msg = self.convert_zyre_msg_to_dict(msg)
        if dict_msg is None:
            return

        message_type = dict_msg['header']['type']
        variable_data = dict()
        for data_source in self.data_sources:
            variable_data[data_source] = self.db_interface.
                    get_variables(data_source)
        response_msg = self.__get_response_msg_skeleton(
                message_type)
        response_msg['payload']['receiverId'] = dict_msg['payload
                ']['senderId']
        response_msg['payload']['variableList'] = variable_data
        return response_msg
```

[3] https://github.com/zeromq/zyre

Message Structure

- When working with distributed systems, it is essential to **define a standard message structure** so that **all components can send messages based on that structure** and also **know how to process incoming messages**

Message Structure

- When working with distributed systems, it is essential to **define a standard message structure** so that **all components can send messages based on that structure** and also **know how to process incoming messages**
- In ROS, **only registered message types** (predefined or custom) **can be sent between components**
  - This prior definition enables automatic code generation from the message descriptions
  - Filling out such messages minimises the possibility for introducing data errors

Message Structure

- When working with distributed systems, it is essential to **define a standard message structure** so that **all components can send messages based on that structure** and also **know how to process incoming messages**
- In ROS, **only registered message types** (predefined or custom) **can be sent between components**
  - This prior definition enables automatic code generation from the message descriptions
  - Filling out such messages minimises the possibility for introducing data errors
- Frameworks such as ZMQ are not strict in this respect, as **messages are always sent as strings**
  - **Standard data formats (such as JSON) are often used for structuring messages** in this case
  - Defining (general or concrete) **message schemas** is a good idea — such schemas can define the expected fields, their types, or even the allowed values

Behaviour Management: State Machines and Behaviour Trees

Robot Behaviour Management: The Essence of Robot Software Development

- One essential question when developing robot software is **which formalism to use for representing and managing the runtime behaviour of robot operations**
  - Robots are complex systems, but their behaviour can often be decomposed into well-defined functionalities

Robot Behaviour Management: The Essence of Robot Software Development

- One essential question when developing robot software is **which formalism to use for representing and managing the runtime behaviour of robot operations**
  - Robots are complex systems, but their behaviour can often be decomposed into well-defined functionalities
- The standard and most common way of behaviour management is using **finite-state machines**

Robot Behaviour Management: The Essence of Robot Software Development

- One essential question when developing robot software is **which formalism to use for representing and managing the runtime behaviour of robot operations**
  - Robots are complex systems, but their behaviour can often be decomposed into well-defined functionalities
- The standard and most common way of behaviour management is using **finite-state machines**
- In the last few years, **behaviour trees** have become a popular alternative to state machines
  - A prominent example that uses behaviour trees is the navigation stack in ROS2: https://navigation.ros.org/index.html

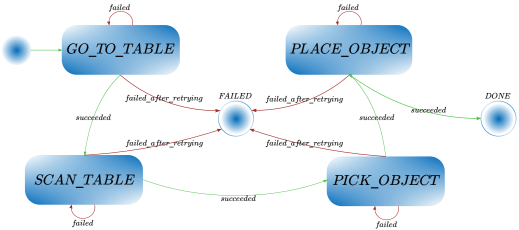
State Machine Definition

- A (finite)-state machine **models a computational process through a finite set of states**
- At each time, **a system is in one of the states** and can **transition to other states** (including self-transitions)
- States typically also **receive inputs** and **produce outputs** (from predefined sets)

"A **finite-state machine** $M = (S, I, O, f, g, s_0)$ consists of a finite set S of states, a finite input alphabet I, a finite output alphabet O, a transition function f that assigns to each state and input pair a new state, an output function g that assigns to each state and input pair an output, and an initial state s_0." (K. H. Rosen, "Discrete Mathematics and Its Applications", McGraw-Hill, 4th ed., 1998, p. 641.)

State Machine Illustration

- An ilustration of a state machine is shown on the right — for a robot task of:
  1. moving to a table
  2. finding an object on it
  3. picking the object, and
  4. immediately placing it back on the table
- In the state machine, rounded rectangles represent states and labelled edges are transitions



A simple SM for a pick-and-place robot task. A. Mitrevski, "Skill generalisation and experience acquisition for predicting and avoiding execution failures," *Ph.D. dissertation*, Department of Computer Science, RWTH Aachen University, 2023, p. 51.

Process Management Using State Machines

- State machines are useful for **managing long-running processes** that can be decomposed into a discrete set of states of interest
- The diagram on the right illustrates one such state machine that manages a process throughout its lifecycle and reacts to faults during the operation
- ROS2 has **managed nodes** whose operation is governed by a similar state machine

A fault-tolerant state machine for managing a long-running process, inspired by the state machine of Linux processes (https://tldp.org/LDP/tlk/kernel/processes.html). A. Mitrevski, "Skill generalisation and experience acquisition for predicting and avoiding execution failures," *Ph.D. dissertation*, Department of Computer Science, RWTH Aachen University, 2023, p. 56.

SMACH

- SMACH (pronounced “smash”) is **a standard library for state machine development in ROS**
- In SMACH, **each state is a separate class** with a method **`execute`**, which is called every time a robot is in that state
- Data sharing within the state machine is made possible by a shared structure called **`userdata`**, which is a **dictionary where user-defined input and output entries are stored**
  - For each state, the input / output userdata keys that are used within the state need to be defined explicitly

SMACH

- SMACH (pronounced "smash") is **a standard library for state machine development in ROS**
- In SMACH, **each state is a separate class** with a method **`execute`**, which is called every time a robot is in that state
- Data sharing within the state machine is made possible by a shared structure called **`userdata`**, which is a **dictionary where user-defined input and output entries are stored**
  - For each state, the input / output userdata keys that are used within the state need to be defined explicitly

```
class PickObject(RosState):
    def __init__(self, node, robot):
        RosState.__init__(self, node, robot,
                          total_retries=3,
                          outcomes=['retry', 'done', 'failed'],
                          input_keys=['object_to_grasp'],
                          output_keys=['grasping_arm'])
        self.robot = robot
        self.number_of_retries = 0
        self.total_retries = total_retries

    def execute(self, userdata):
        object = userdata.object_to_grasp

        ### perform necessary activities for
        ### picking up the object with the robot
        success, grasping_arm = self.robot.grasp(object)

        if success:
            userdata.grasping_arm = grasping_arm
            return 'done'
        else:
            if self.number_of_retries < self.total_retries:
                return 'retry'
            else:
                self.number_of_retries = 0
                return 'failed'
```

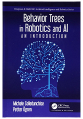
SMACH

- SMACH (pronounced “smash”) is **a standard library for state machine development in ROS**
- In SMACH, **each state is a separate class** with a method **`execute`**, which is called every time a robot is in that state
- Data sharing within the state machine is made possible by a shared structure called **`userdata`**, which is a **dictionary where user-defined input and output entries are stored**
    - For each state, the input / output userdata keys that are used within the state need to be defined explicitly

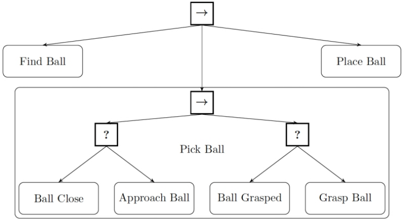
```
class PickObject(RosState):
    def __init__(self, node, robot):
        RosState.__init__(self, node, robot,
                          total_retries=3,
                          outcomes=['retry', 'done', 'failed'],
                          input_keys=['object_to_grasp'],
                          output_keys=['grasping_arm'])
        self.robot = robot
        self.number_of_retries = 0
        self.total_retries = total_retries

    def execute(self, userdata):
        object = userdata.object_to_grasp

        ### perform necessary activities for
        ### picking up the object with the robot
        success, grasping_arm = self.robot.grasp(object)

        if success:
            userdata.grasping_arm = grasping_arm
            return 'done'
        else:
            if self.number_of_retries < self.total_retries:
                return 'retry'
            else:
                self.number_of_retries = 0
                return 'failed'
```

```
sm = StateMachine(['done', 'failed'])
with sm:
    StateMachine.add('PICK_OBJECT', PickObject(node), {'done':'
            PLACE_OBJECT', 'retry': 'PICK_OBJECT', 'failed': '
            failed'})
    StateMachine.add('PLACE_OBJECT', PlaceObject(node), {'done':'
            done', 'retry': 'PLACE_OBJECT', 'failed': 'failed'})
```

Behaviour Tree Definition

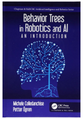


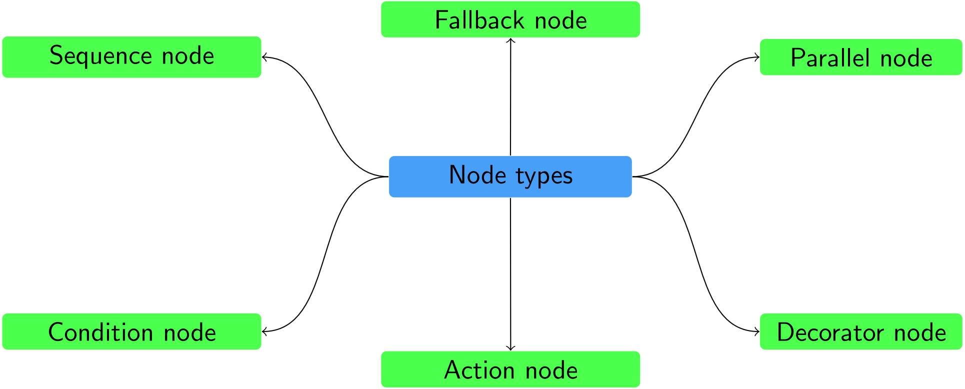
- A behaviour tree organises the behaviour of a robot into **behaviours**, which are nodes that execute based on predefined rules
- The execution of a behaviour tree is coordinated by signals called **ticks**, which are sent from the root node and propagated to the children nodes
- **Nodes in a behaviour tree can be defined hierarchically** — a node can itself be a tree



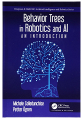
"A behaviour tree is a directed rooted tree where the internal nodes are called **control flow nodes** and leaf nodes are called **execution nodes**... The root is the node without parents; all other nodes have one parent. The control flow nodes have at least one child." (M. Colledanchise and P. Ögren, "Behavior Trees in Robotics and AI: An Introduction," CRC Press - Taylor and Francis Group, 2018, p. 6.)

Behaviour Tree Node Types





Sequence, Fallback, and Parallel Nodes



Sequence node

- Returns `Success` if all children succeed
- Returns `Failure` or `Running` if any of the children (from left to right) return those

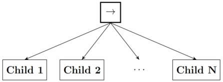
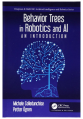


Figure 1.2: Graphical representation of a Sequence node with N children.

Sequence, Fallback, and Parallel Nodes



Sequence node

- Returns `Success` if all children succeed
- Returns `Failure` or `Running` if any of the children (from left to right) return those

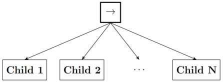


Figure 1.2: Graphical representation of a Sequence node with N children.

Fallback node

- Returns `Failure` if all children return that
- Returns `Success` or `Running` if any of the children (from left to right) return those

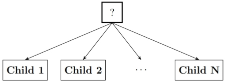
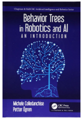


Figure 1.3: Graphical representation of a Fallback node with N children.

Sequence, Fallback, and Parallel Nodes



Sequence node

- Returns `Success` if all children succeed
- Returns `Failure` or `Running` if any of the children (from left to right) return those

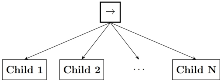


Figure 1.2: Graphical representation of a Sequence node with N children.

Fallback node

- Returns `Failure` if all children return that
- Returns `Success` or `Running` if any of the children (from left to right) return those

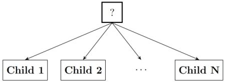


Figure 1.3: Graphical representation of a Fallback node with N children.

Parallel node

- Returns `Success` if $m \leq n$ of its children return that
- Returns `Failure` if $n - m + 1$ children return that
- Returns `Running` Otherwise

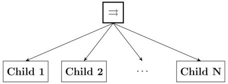
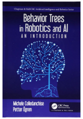
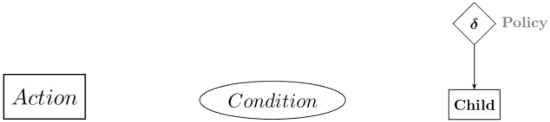


Figure 1.4: Graphical representation of a Parallel node with N children.

Action, Condition, and Decorator Nodes



- An **action node** executes a given operation, such that it returns `Running` if the execution is not complete, and `Success` or `Failure` at the end of the execution depending on the outcome
- A **condition node** returns `Success` or `Failure` depending on the result of a given condition
- A **decorator node** can control the return value of a node or send a tick to a node based on certain predefined conditions



(a) Action node. The label describes the action performed.

(b) Condition node. The label describes the condition verified.

(c) Decorator node. The label describes the user defined policy.

Figure 1.5: Graphical representation of Action (a), Condition (b), and Decorator (c) nodes.

State Machines vs. Behaviour Trees

- **Finite state machines are based on a well-defined formal framework** (automata theory); **behaviour trees are more ad-hoc**

State Machines vs. Behaviour Trees

- **Finite state machines are based on a well-defined formal framework** (automata theory); **behaviour trees are more ad-hoc**
- **Large state machines can be difficult to maintain**; **behaviour trees are supposed to make the maintenance easier** because they (in principle) enable flexible composition of behaviours

State Machines vs. Behaviour Trees

- **Finite state machines are based on a well-defined formal framework** (automata theory); **behaviour trees are more ad-hoc**
- **Large state machines can be difficult to maintain**; **behaviour trees are supposed to make the maintenance easier** because they (in principle) enable flexible composition of behaviours
- **Concurrent execution is supported by default with behaviour trees**; this is not the case with state machines without extra effort

State Machines vs. Behaviour Trees

- **Finite state machines are based on a well-defined formal framework** (automata theory); **behaviour trees are more ad-hoc**
- **Large state machines can be difficult to maintain**; **behaviour trees are supposed to make the maintenance easier** because they (in principle) enable flexible composition of behaviours
- **Concurrent execution is supported by default with behaviour trees**; this is not the case with state machines without extra effort
- In general, **state machines are still more widely accepted and used than behaviour trees**

A Bag of (Other) Tools

Robotics is More Than Communication and Behaviour Management

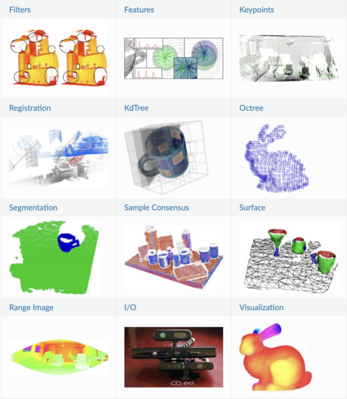
- Frameworks for distributed system development and behaviour management represent just one segment of the robot software development toolbox
- Robot software development relies on a **variety of (open-source) software frameworks that provide dedicated functionalities relevant for robotics**
- **Sensor data processing** is one area where standard frameworks exist, particularly in the context of images and point cloud data
- **Machine learning** is another area where the reliance on open and well-maintained libraries is remarkably obvious

Robotics is More Than Communication and Behaviour Management

- Frameworks for distributed system development and behaviour management represent just one segment of the robot software development toolbox
- Robot software development relies on a **variety of (open-source) software frameworks that provide dedicated functionalities relevant for robotics**
- **Sensor data processing** is one area where standard frameworks exist, particularly in the context of images and point cloud data
- **Machine learning** is another area where the reliance on open and well-maintained libraries is remarkably obvious
- On the following slides, we will briefly introduce a variety of software libraries and frameworks that are commonly used throughout robot software development

PCL[4] for Point Cloud Processing

- As sensors such as RGB-D cameras and 3D lidars produce point cloud data, processing point clouds is important for extracting meaningful information from such data
- The Point Cloud Library (PCL) is a **library that implements a large variety of common point cloud processing algorithms** and **provides standardised interfaces for implementing custom processing functionalities**
- PCL is **compatible with ROS** (through specialised interfaces for dealing with ROS messages), which is one reason for its popularity in robotics applications

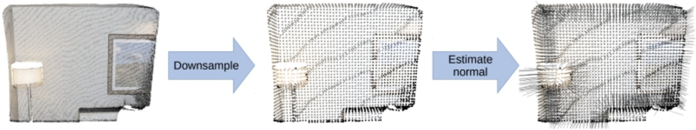


https://pcl.readthedocs.io/projects/tutorials/en/master/walkthrough.html

[4] https://github.com/PointCloudLibrary/pcl

Open3D[5]

- One downside of PCL is that it is (only) a C++ library; using it with Python (a very popular language in robotics) is challenging because there is no officially supported Python interface
- Open3D is an **alternative point cloud processing library** that implements similar functionalities as PCL, but is fully compatible with Python
- Open3D-ML, an extension of Open3D, makes it possible to perform machine learning tasks on 3D point cloud data



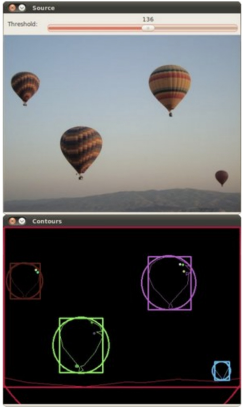
(a) A simple 3D data processing task: load a point cloud, downsample it, and estimate normals.

Q-Y. Zhou, J. Park, and V. Koltun, "Open3D: A Modern Library for 3D Data Processing," arXiv:1801.09847, 2018.

[5] https://github.com/isl-org/Open3D

Computer Vision Using OpenCV[6]

- Most robots need to process visual data in some form, so image processing and, more generally, computer vision tasks need to be done in different contexts
- OpenCV is a **standard framework for performing (classical) image processing tasks**, such as noise removal, morphological transformations, or feature detection
- The results of OpenCV can be **used as a precursor for further processing** — for instance as features for machine learning algorithms



https://docs.opencv.org/4.8.0/da/d0c/tutorial_bounding_rects_circles.html

[6] https://github.com/opencv/opencv

Person (Keypoint) Detection Using OpenPose[7]

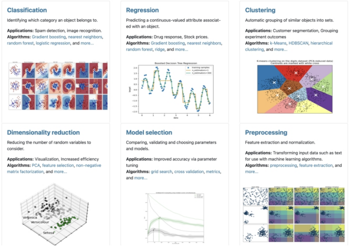
- In human-robot scenarios, detecting and tracking people are essential processes for effective interaction and collaboration
- OpenPose is a **library that detects human skeletons from RGB images by identifying predefined keypoints on the human body**
  - 135 keypoints are detected on the arms, legs, neck, head, face, as well as on the fingers and toes
- Keypoint detection is done by a **pretrained neural network model**
- The ability to perform detection in real time and to handle occlusions rather reliably is one reason for the library's widespread use

Z. Cao et al., "OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 1, pp. 172–186, Jan. 2021.

[7] https://github.com/CMU-Perceptual-Computing-Lab/openpose

scikit-learn[8] for Machine Learning

- Modern robots use learning-based components for a variety of tasks; developing machine learning models is thus an important and common task in contemporary robotics
- scikit-learn is an extensive machine learning library in Python, which **includes implementations of many (classical) learning models and algorithms**
- The library can also be used for learning with neural networks, although more specialised libraries exist for that purpose

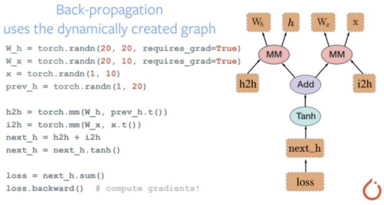


https://scikit-learn.org/stable/

[8] https://github.com/scikit-learn/scikit-learn

Neural Networks With PyTorch[9]

- Neural network-based machine learning has evolved into an important component for many tasks in robotics, ranging from vision to natural language processing
- Various libraries for developing neural networks are available, but **PyTorch is a particularly widely used and supported library**
- PyTorch (and other similar libraries) **represent complex computations into a computational graph and perform automatic differentiation**, which is what makes them suitable for handling deep neural networks

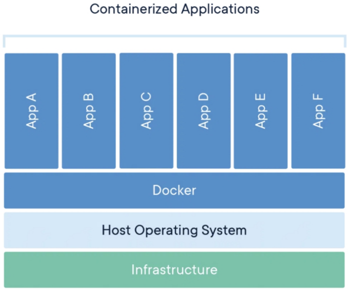


https://github.com/pytorch/pytorch

[9] https://github.com/pytorch/pytorch

Docker

- Docker is a framework that **enables development of containerised applications**, which are processes that can run on any host and are independent of other running processes
- Docker containers are created from **images**, which describe how the environment for a container should be set up
  - Docker images can build on each other — **derived images inherit everything that is included in a base image**
- Containerisation is useful for robot software development because **it simplifies portability of robot software**
  - The host robot that executes a container does not need to have any software setup that is required by a robot — everything can be included in the container
- Docker is not a Swiss knife though — **communication with and between containers is performed over a network**, which is slower than executing everything directly on the host



https://www.docker.com/resources/what-container/

Summary

- Various frameworks can be used for developing robots as distributed systems; ROS is the predominant framework, but others, such as ZMQ and Zyre, can also be useful in certain cases
- Robot behaviour can be implemented by following different formalisms, with finite state machines and behaviour trees being particularly common
- Robot software development benefits from many open-source frameworks that are used for a large number of tasks, such as image and point cloud processing, machine learning, as well as software sharing and deployment