CRTK Overview

Anton Deguet
Johns Hopkins University

- Motivating uses cases
- Main CRTK commands
- Implementation
  - Devices
  - Python client API

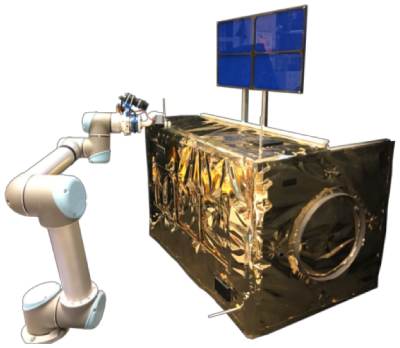
Motivating Use Cases

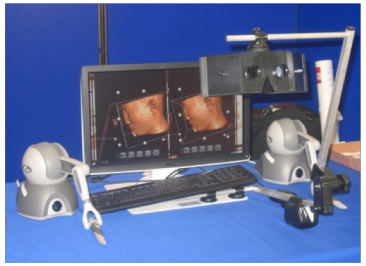
1. Teleoperation (diverse devices, communication channels, sensor feedback)
2. Autonomous motion
3. Custom kinematics/control
4. Cooperative or compliant control
5. Custom instruments

Use Case 1: Teleoperation

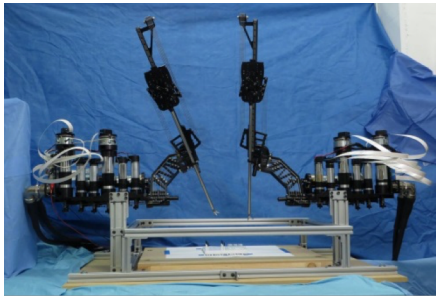
- Diverse master and slave devices
- Different communication channels (performance)
- Bilateral teleoperation, force reflection

Cartesian position, velocity, incremental position, effort (robot and tool)

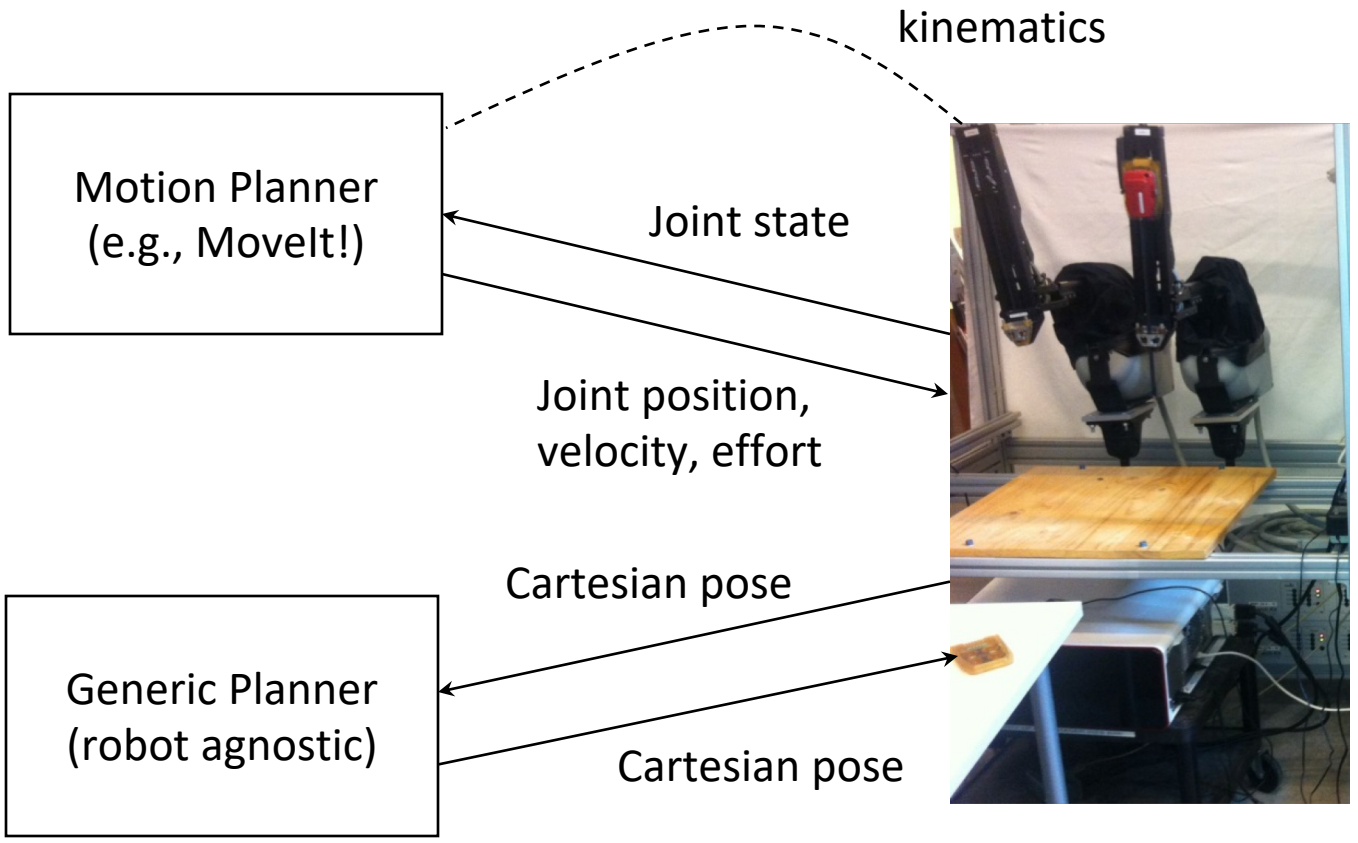




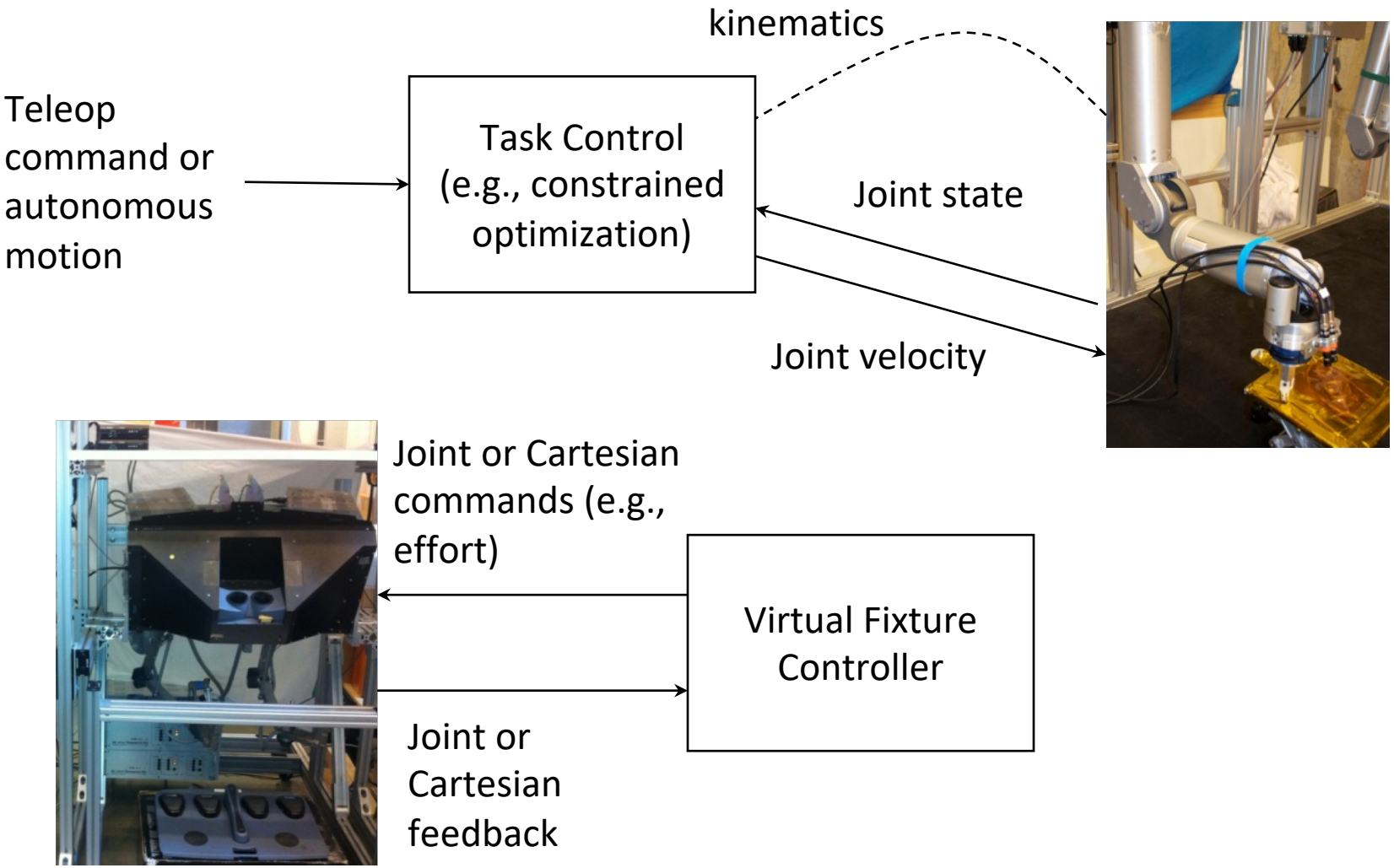
Cartesian state, joint state, generalized forces



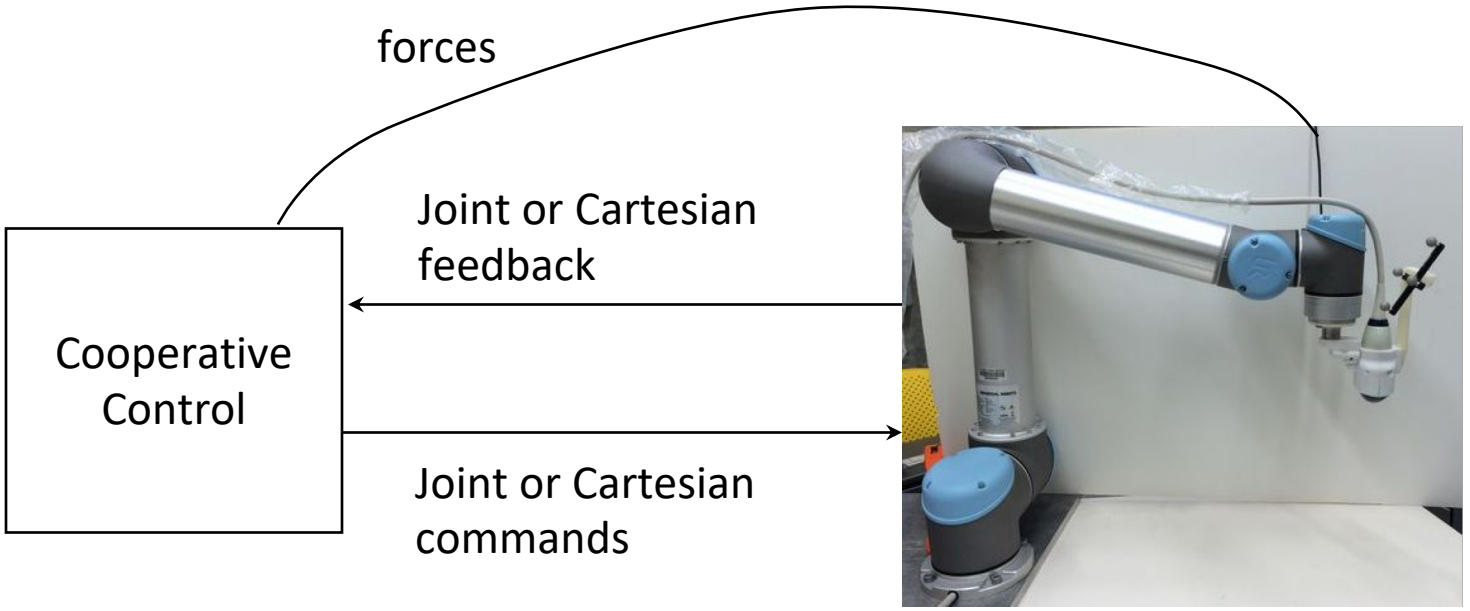
Use Case 2: Autonomous Motion



Use Case 3: Custom Kinematics/Control

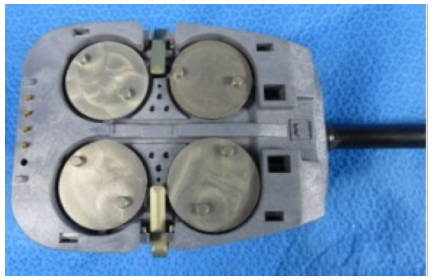


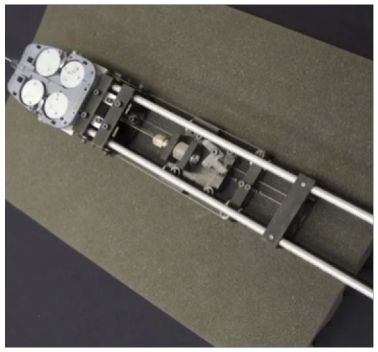
Use Case 4: Cooperative or Compliant Control



Use Case 5: Custom Instruments

- Custom instruments for Raven / dVRK
  - Interface to 4 driving disks





- Powered/sensorized tools/instruments
  - In addition to 4 driving disks
  - Grippers, end-effectors for other robots

Use Case: dVRK API Over Time

We added support for many of the use cases previously mentioned as needed…

First, just PID with joint space:

`GetPositionJoint`, `SetPositionJoint` – so far so good

Needed to distinguish measured from commanded positions

`GetPositionJointDesired` (PID goal)
`GetPositionJoint` (implicitly measured one)

Added kinematic on top of joint PID:

`GetPositionCartesian`, `GetPositionCartesianDesired` , `SetPositionCartesian`

Added more feedback, velocities and efforts:

`GetVelocityJoint`, `GetForceJoint`, `GetVelocityCartesian`, `GetForceCartesian`

Started to use ROS more extensively:

Better timestamps, use strings id in payloads, different names

`Get{Position,Velocity,Force}Joint{,Desired}` grouped in `GetStateJoint{,Desired}`
`Get{Velocity,Force}Cartesian` became `Get{Twist,Wrench}`

Added trajectory generation (using Reflexxes):

`SetPositionJointGoal` and `SetPositionCartesianGoal` – names are getting a bit long…

About to add relative position goal:

`SetPositionCartesianGoalIncrement`. For ROS topics, not using CamelCase:

`set_position_cartesian_goal_increment 37 characters!`

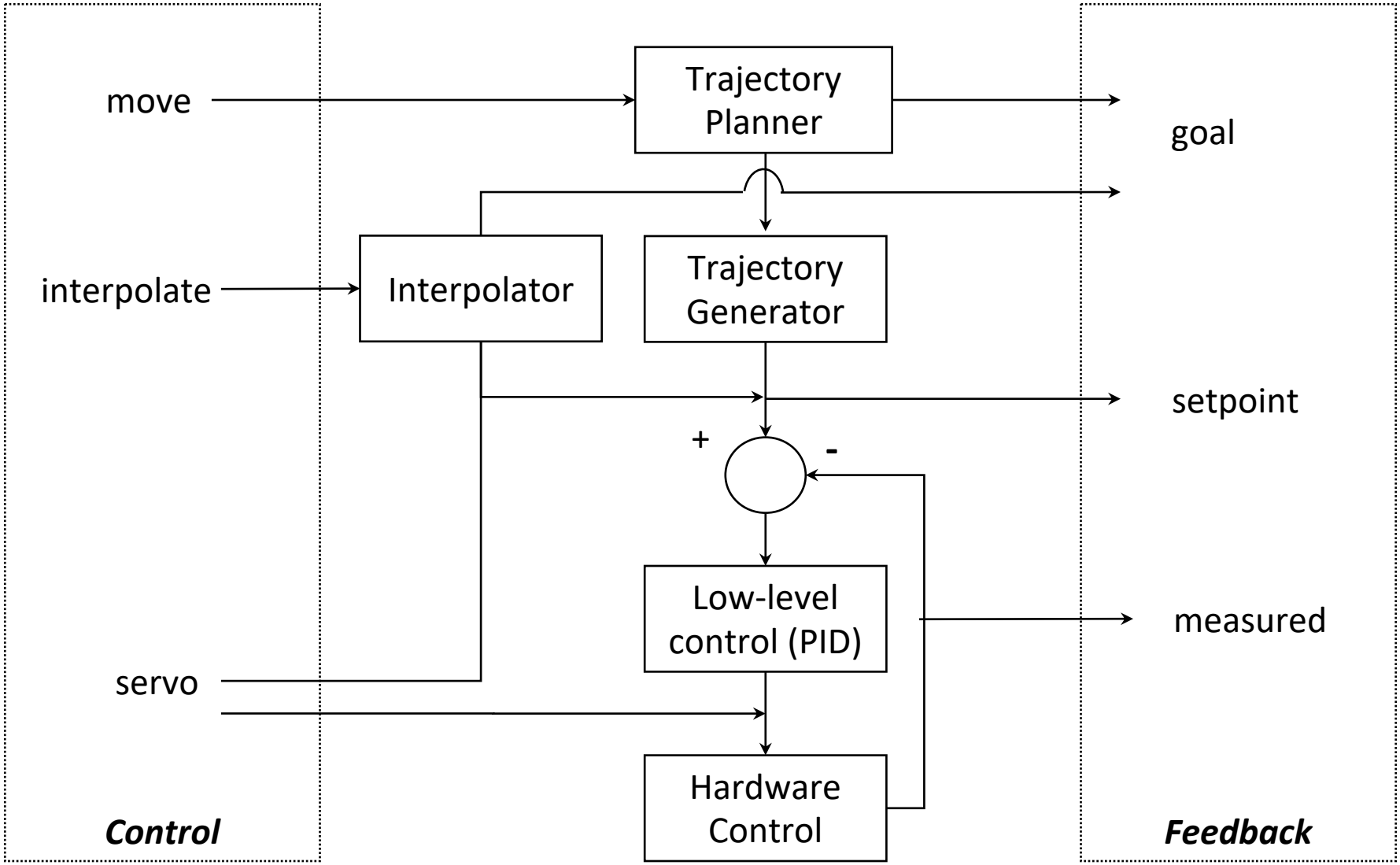
CRTK: Guiding Principles

- Based on identified use cases
- Defines command names and payloads, not restricted to a programming language!
- As simple as possible, few parameters (e.g., stream of positions easier than PVT)
- Logical and consistent naming conventions (somewhat like part-numbering convention)
  - All robots not required to implement all commands, but should use consistent name
- Short enough to type into interpreter (e.g., Python, Matlab) or 80 column text editor
- In most cases, compatible with publish/subscribe (ROS topics) and client/server (ROS services)

Categories of Commands

- Robot state feedback (Joint/Cartesian position, velocity, effort)
- Robot motion control (Joint/Cartesian motions)
- Operating state feedback and control (e.g., homed, enabled, paused, fault, busy)
- Other: configuration, retrieve capabilities, …

Feedback and Control: Overview



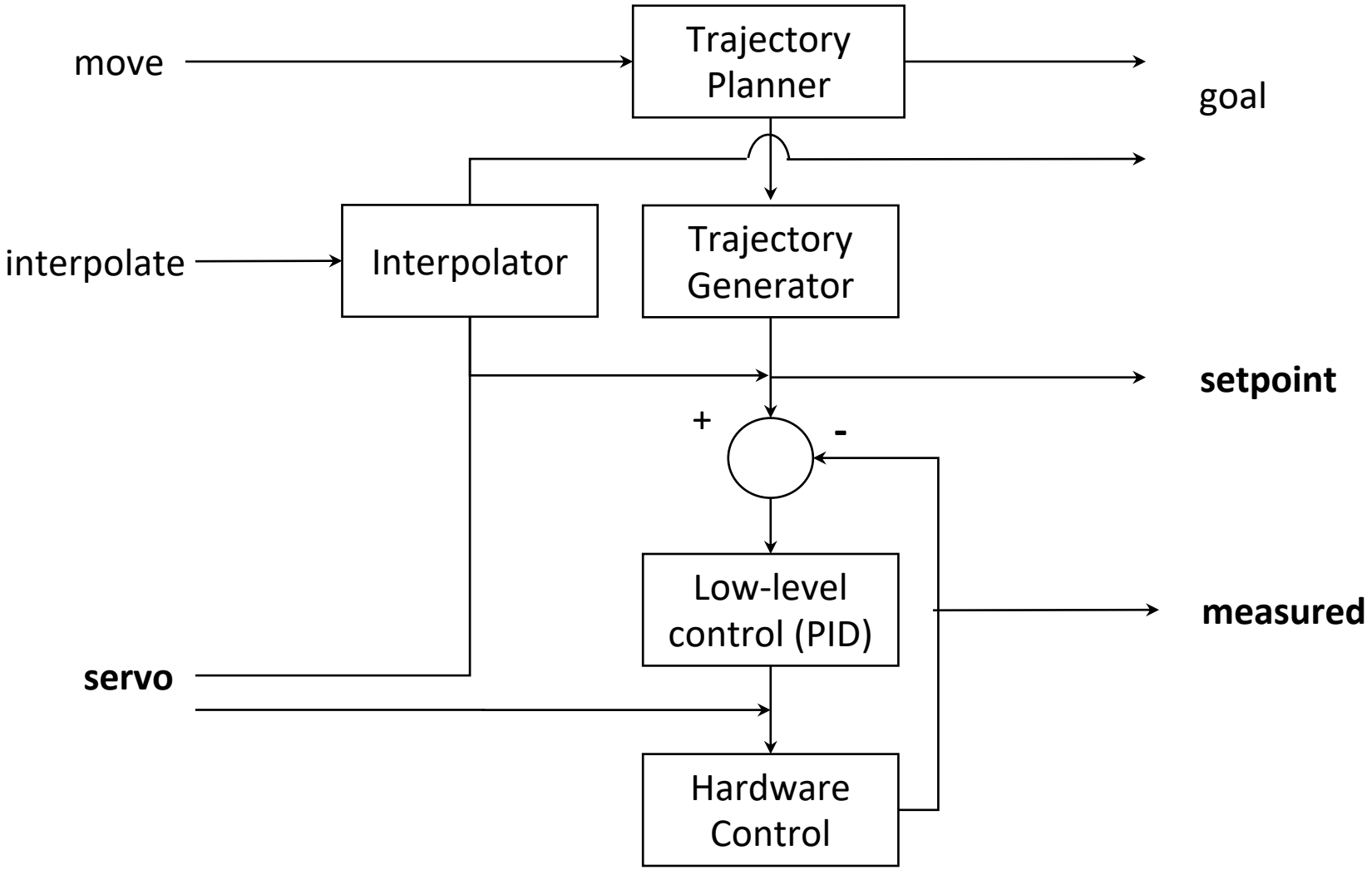
Feedback and Control: Convention

Control level	• *servo*: direct real-time stream (pre-emptive) • *interpolate*: interpolated stream (pre-emptive) • *move*: plan trajectory to goal (not pre-emptive)
Feedback	• *setpoint*: current setpoint to low-level control • *goal*: most recent interpolate or move goal • *measured*: sensor feedback • *measuredN*: redundant sensor feedback (N=2, 3, …)
Space	• *_j*: joint • *_c*: cartesian
Type	• *p*: position • *r*: relative (incremental) position • *v*: velocity; (t: twist) • *f*: generalized force (e: effort, w: wrench) • *s*: state (position, velocity, effort) feedback

Examples: `servo_jf`, `setpoint_cp`, `measured2_js`, etc.

See https://github.com/collaborative-robotics/documentation/wiki/Robot-API-motion

Feedback and Control: `servo`

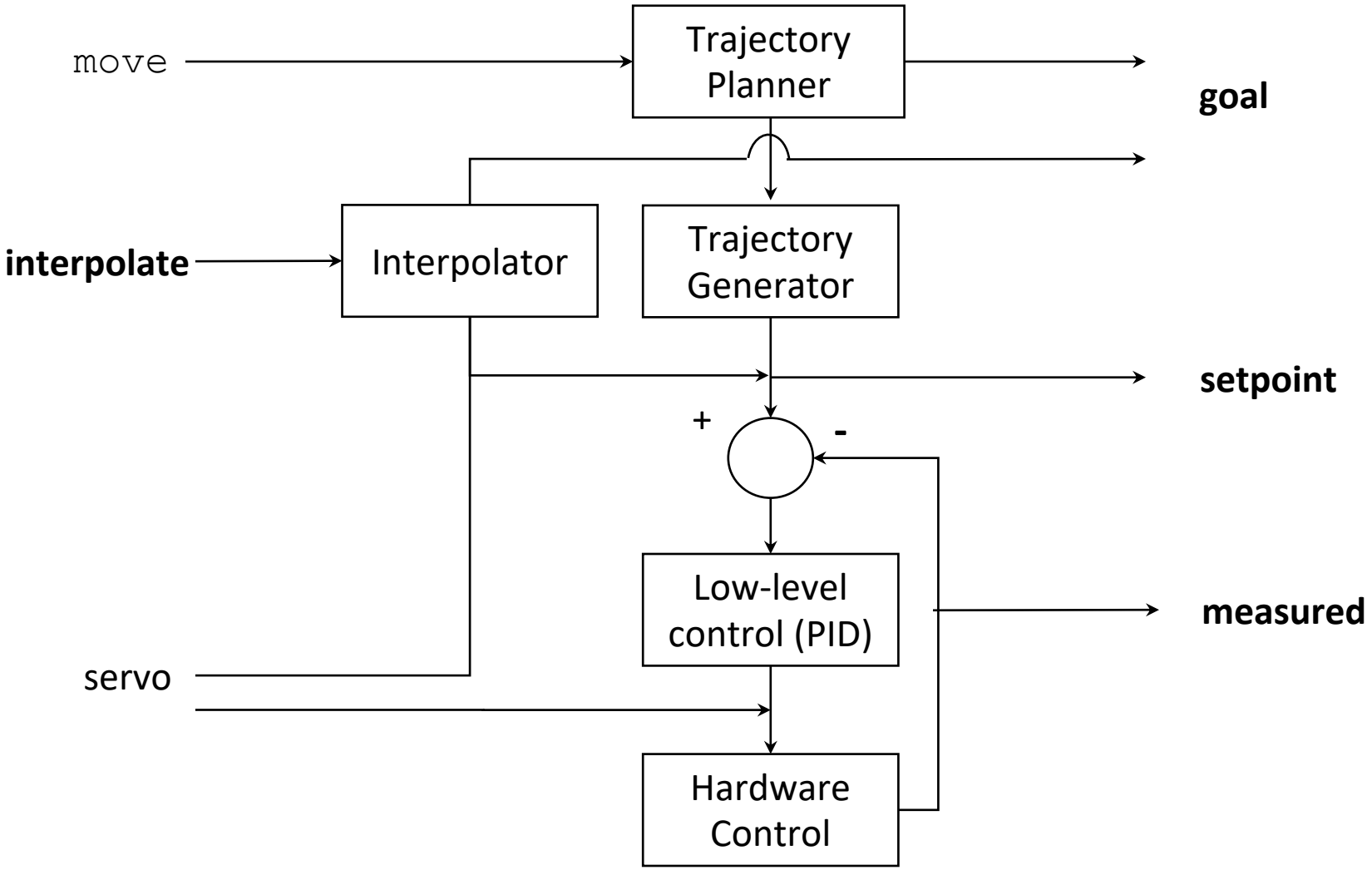


Servo commands

Direct access to the low-level controller

- **Use cases**:
  - `servo_p`: User has a smooth and continuous trajectory coming from a master arm or recorded trajectory and can send commands at a high rate.
  - `servo_v`: Closing the loop in velocity mode using a fast external sensor with a task based Jacobian.
  - `servo_f`: Haptic feedback on master arm.
- **Type**: These commands can be either position, velocity or effort based. Positions can be provided relative from the latest setpoint position.
- **Continuity**: Users should send continuous commands. The low-level controller is not required to generate intermediate setpoints to ensure that the command is feasible (i.e., setpoint should be close to current state). The low-level controller can enforce limits, e.g., reject a command if the difference from the previous command is greater than a defined threshold.
- **Time**: Users are expected to send commands periodically at a rate close to the low-level rate. These commands are pre-emptive.

Feedback and Control: `interpolate`

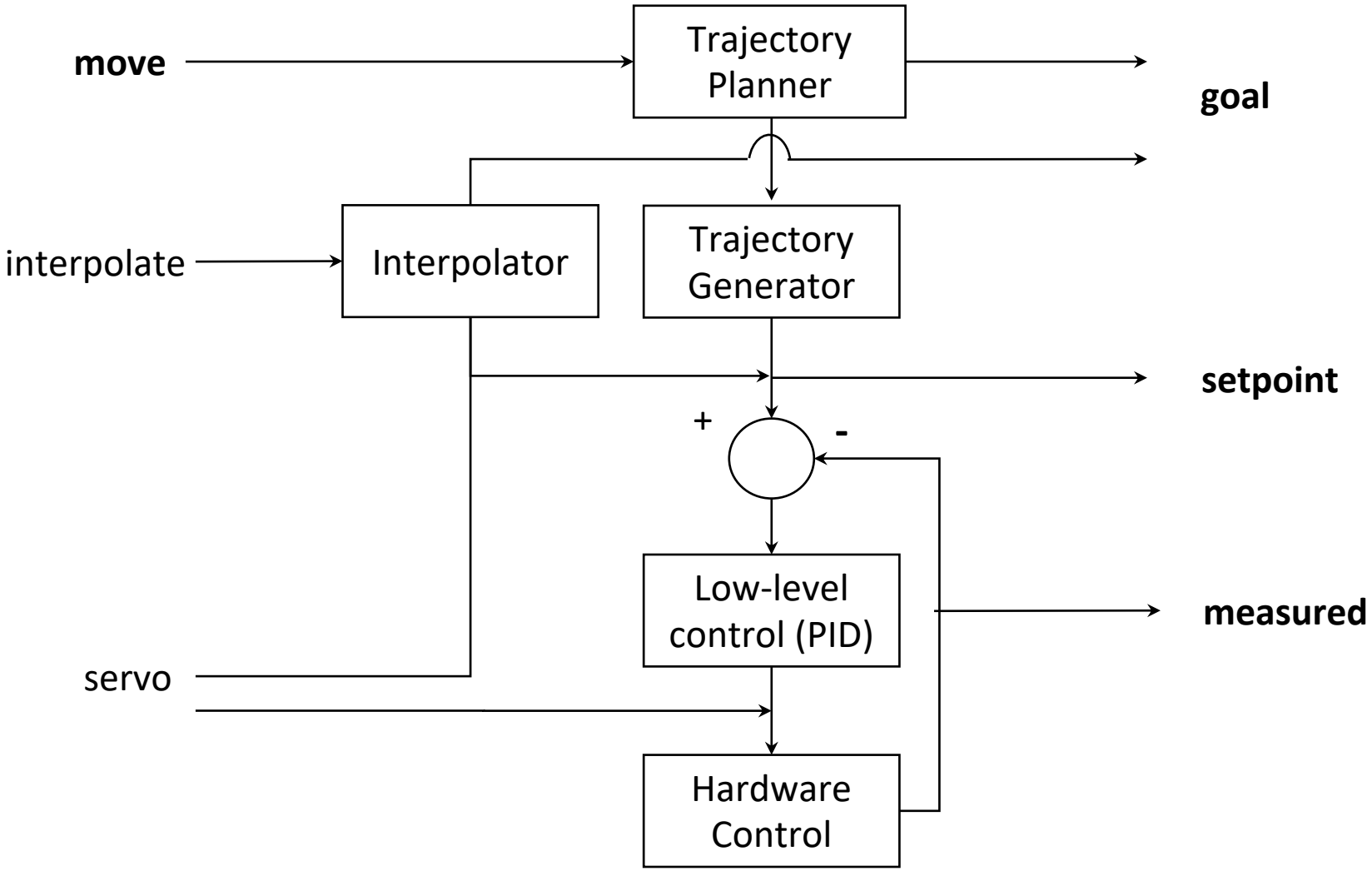


Interpolate commands

Simple interpolation

- **Use cases:**
  - User has a smooth trajectory coming from a master arm or recorded trajectory, but cannot send commands at a high rate (e.g., 50Hz visual tracking, remote tele-operation).
  - In general, similar usage as servo but the application can not send commands fast enough to provide a smooth command so the low-level controller needs to interpolate the user commands (smooths but adds latency).
- **Type:** see *servo*
- **Continuity:** See *servo*
- **Time:** Users are expected to send commands periodically at a rate lower than the low-level controller. The velocity of the motion is defined by the user commands. These commands are preemptive.
- Still a fair amount to specify so we're experimenting different approaches.

Feedback and Control: `move`



Move commands

Move with trajectory generation:

- **Use case**: User wants to move to a given position and stop there (e.g., home position, pick and place).
- **Type**: These commands are position based, either absolute or relative.
- **Continuity**: Users must send feasible commands. The low-level controller will compute a complete trajectory to move from the current state (position and velocity) to the desired goal.
- **Time**: Users are expected to send a single command and wait for completion before sending a new one. Time of execution is defined by the trajectory generation parameters (acceleration and velocity).
- As for the interpolate command, there is still a fair amount to decide. Should we make an attempt to go on straight line in `_cp`, synchronize joints in `_jp`, which parameters to use for trajectory generation?

Feedback

`setpoint_`:

- If the last user command is `servo`, `setpoint` should report the last user `servo`. Example:
  - `servo_jp(j)`
  - `setpoint_jp() == j`
    `setpoint_cp() == ForwardKinematics(j)`
- If the last user command is `interpolate` or `move`, the `setpoint` should report the estimated velocity from the interpolator or trajectory generator
- If the low level controller computes an effort (current controlled PID), `setpoint_{jc}f` can be used to report it

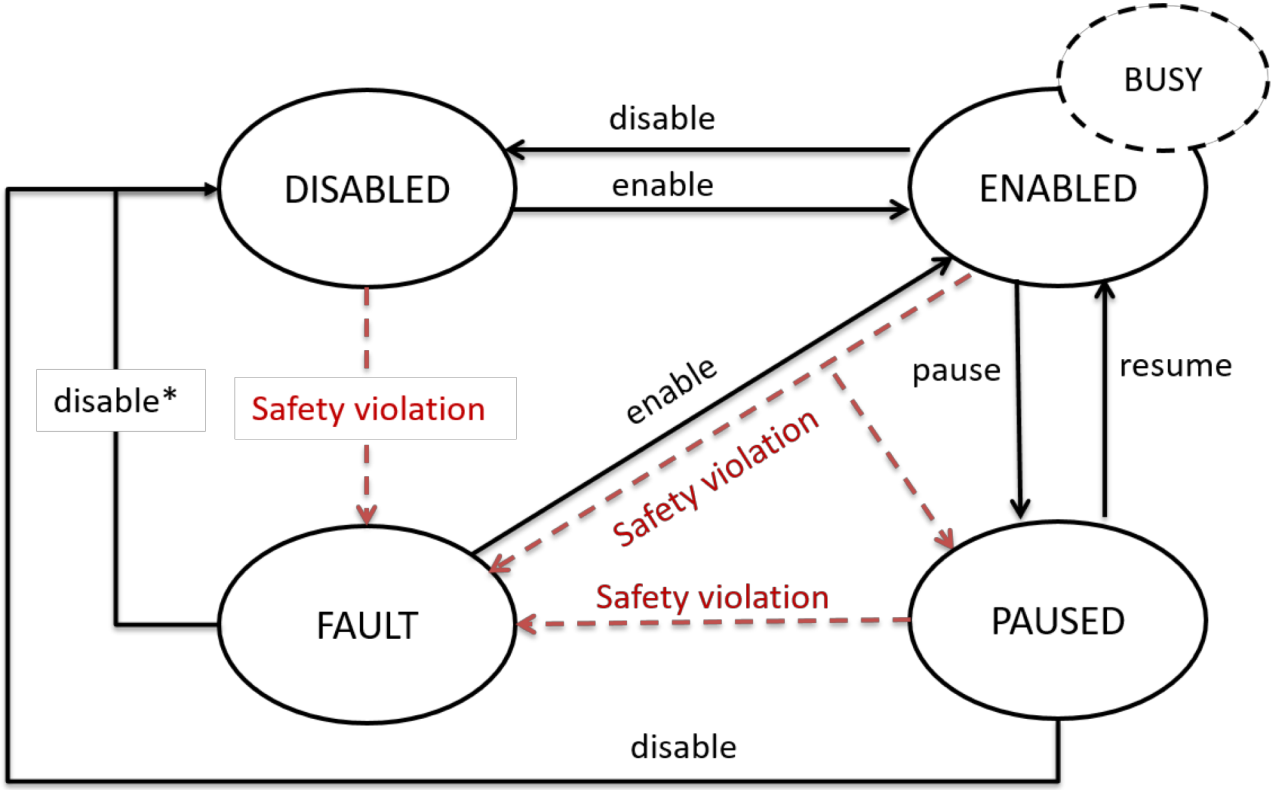
CRTK: Examples

- `move_jp` Plan trajectory and move to joint position
- `servo_jv` Real-time update of joint velocity
- `interpolate_cr` Interpolated relative Cartesian position move
- `servo_jf` Real-time update of joint force (torque)
- `measured_js` Measured position, velocity, force
- `measured_cp` Measured Cartesian pose (position)
- `measured_cv` Measured Cartesian velocity (twist)
- `measured_cf` Measured Cartesian force (wrench)

Robot Status

- Not attempting to standardize robot state machine (too much variability)
- Instead, define meta-states and operating modes
- Define standard commands to change/query

Robot Meta-States and Modes

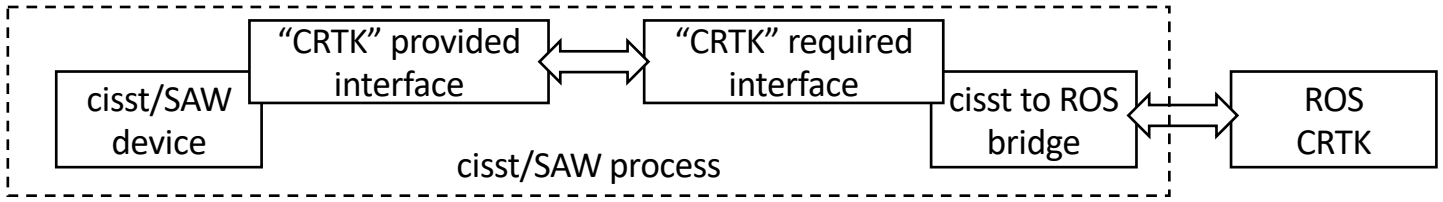


Meta-state queries: is_disabled, is_enabled, is_paused, is_fault
Operating mode queries: is_busy, is_homed

CRTK Implementations

CRTK is not tied to a specific programming language nor a library

- ROS:
  - CRTK ROS topic/service/action names and payloads
  - CRTK compatible devices (C/C++/Python ROS based on developer preference)
  - CRTK Client API (Python, Matlab, C++) – to hide middleware but use CRTK for method names
- "Internal" APIs:
  - cisst/SAW component based multi-tasking framework with dynamic loading: Requires standardized API!



CRTK ROS: Brief ROS introduction

- Communication models
  - *Parameters*: initial configuration and runtime configuration changes
  - *Topics*: asynchronous stream of data, *publishers* and *subscribers*
  - *Services*: synchronous call, sends request and waits for response (RPC like)
  - *Actions*: similar to *Services*, doesn't block and gets periodic feedback
  - *tf2*: transformation
- Payloads
  - *Topics*: `std_msgs`, `sensor_msgs`, `geometry_msgs`, some with header
  - *Services*: `request` & `response`
  - *Actions*: `goal`, `feedback` and

CRTK ROS: Motion Feedback

- Communication model
  - `measured_`, `setpoint_`, `goal_` are non blocking streams
  - Topics work great
- Payloads:
  - `_js`: `sensor_msgs::JointState`
  - `_cp`: `geometry_msgs::TransformStamped`
  - `_cv`: `geometry_msgs::TwistStamped`
  - `_cf`: `geometry_msgs::WrenchStamped`

CRTK ROS: Control `servo`

- Communication model
  - `servo_` are non blocking commands
  - *Topics* work fine
- Payloads:
  - `_jp`, `_jv`, `_jf`: `sensor_msgs::JointState`
  - `_cp`: `geometry_msgs::TransformStamped`
  - `_cv`: `geometry_msgs::TwistStamped`
  - `_cf`: `geometry_msgs::WrenchStamped`

CRTK ROS: Control `interpolate`

- Communication model
  - `interpolate_` are non blocking commands
  - *Topics* work fine
- Payloads:
  - `_jp`, `_jp`, `_jf`: `sensor_msgs::JointState`
  - `_cp`: `geometry_msgs::TransformStamped`
  - `_cv`: `geometry_msgs::TwistStamped`
  - `_cf`: `geometry_msgs::WrenchStamped`
- `setpoint_` can be used as progress *feedback* (like *Actions*)
- Note: `interpolate_` not implemented yet

CRTK ROS: Control `move`

- Communication model
  - `move_` are pre-emptible commands but caller might want to wait for end of move
  - *Actions* could be used (see MoveIt! *FollowJointTrajectory Action*), or combination of two *Topics*, `move_` and `is_busy` ("lightweight *Action"*)
- Payloads:
  - `_jp`, `_jp`, `_jf`: `sensor_msgs::JointState`
  - `_cp`: `geometry_msgs::TransformStamped`
  - `_cv`: `geometry_msgs::TwistStamped`
  - `_cf`: `geometry_msgs::WrenchStamped`
- `setpoint_` might be provided as progress *feedback* (like *Actions*)
  Note: Some vendor APIs don't provide this feature

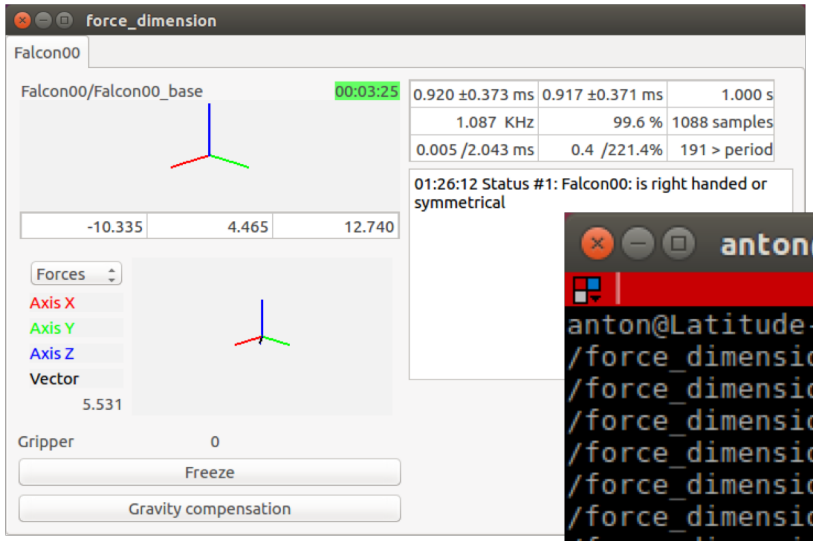
CRTK ROS: Operating State

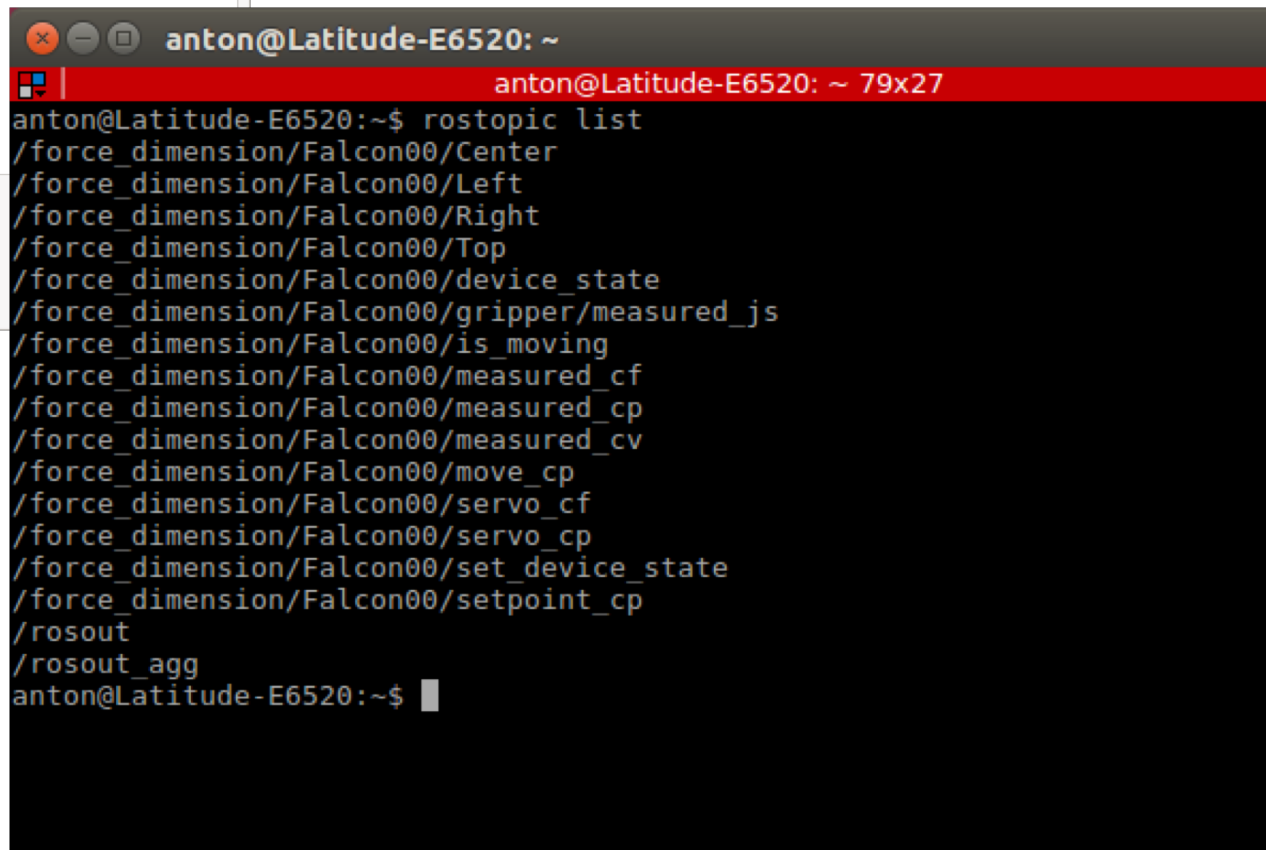
- For queries, first CRTK specific messages in ROS on topic `operating_state`:
  - `state` as a string, must be one of: `DISABLED`, `ENABLED`, `PAUSED` and `FAULT`
  - Booleans for `is_homed`, `is_busy`,
  - Published on topic when changed but can also be queried using ROS service
- For requests, use ROS `std_msgs::String` on topic `state_command`. Examples:
  - `enable`, `disable`
  - `pause`, `resume`
  - `home`, `unhome`

CRTK ROS: To Be Specified

- ROS Param
  - Likely using a format close to ROS control/MoveIt
  - Device features (e.g. does it support `servo_cf`)
  - Publish rates
  - …
- tf2
  - Open question
  - Maybe CRTK could use tf2 to compute relative cartesian goals

CRTK: What a device looks like?





CRTK ROS: Devices

- Devices used to prototype the CRTK API:
  - Sensable Omni, Novint Falcon and Force Dimension, dVRK, Raven2
- Devices/platforms we plan to support:
  - Python and Matlab client APIs
  - Robots: Universal Robot, Galil based JHU Robots
  - 3D Tracking Devices: NDI trackers, Atracsys, Micron Tracker
  - Force Sensors: ATINetFT, OptoForce, JR3
- Maybe also:
  - Dynamic simulation: Chai3D, H3D, Gazebo, V-REP, PhysX, iMSTK, SOFA or …
  - Unity
  - OpenIGTLink
  - …

CRTK: ROS Client APIs

Motivation:

1. Simplify application development!
   - Hide most the ROS communication layer (callbacks, topic settings such as queue size and latching)
   - Provide conversion methods from ROS messages to data types with proper algebra, e.g. apply a rotation to a vector
2. Add “high level” features
   - Wait on events
   - Operating state logic
   - …

CRTK: ROS Python Client API

Design decisions (1/2)

- Hide all `rospy.Publisher`, `rospy.Subscribers`…
- Use convenient data types instead of ROS messages
  - PyKDL for cartesian position:
    ```
    f = PyKDL.Frame()
    f.p = 0.5 * measured_cp().p
    ```
  - Numpy arrays for joints, twist, jacobian…
- Use Python threads to provide blocking waits on events:
  - Device state management: `enable(timeout)`, `disable(timeout)`…
  - Motion status: `home(timeout)`, wait until robot is done moving…

CRTK: ROS Python Client API

Design decisions (2/2)

- Populate the Python Client dynamically
  - Add only the publishers and subscribers one needs
  - Motivation:
    - The CRTK device might not support all the features
      (This could be solved using ROS parameters)
    - Subscribing to all features, including unused ones will likely degrade performances
      This is based on our previous experience on the dVRK
  - Features can be added and removed anytime

Client APIs can be time consuming to implement but are now shared across all CRTK compatible devices!

CRTK ROS: Python Client Example Configure

```python
# example of application using device.py
class crtk_move_cp_example:

    # configuration
    def configure(self, device_namespace):
        # ROS initialization
        if not rospy.get_node_uri():
            rospy.init_node('crtk_move_cp_example', anonymous = True, log_level = rospy.WARN)

        print(rospy.get_caller_id() + ' -> configuring crtk_device_test for: ' + device_namespace)
        # populate this class with all the ROS topics we need
        self.crtk_utils = crtk.utils(self, device_namespace)
        self.crtk_utils.add_operating_state()
        self.crtk_utils.add_measured_cp()
        self.crtk_utils.add_move_cp()
```

CRTK ROS: Python Client Example Utils

```
# internal methods for move_cp
def __move_cp(self, goal, blocking = False, timeout = 300.0):
    # convert to ROS msg and publish
    msg = TransformToMsg(goal)
    self.__move_cp_publisher.publish(msg)
    if blocking:
        self.__wait_while_busy(timeout)

def add_move_cp(self):
    # throw a warning if this has alread been added to the class,
    # using the callback name to test
    if hasattr(self.__class_instance, 'move_cp'):
        raise RuntimeWarning('move_cp already exists')
    # create the subscriber and keep in list
    self.__move_cp_publisher = rospy.Publisher(self.__ros_namespace + '/move_cp',
                                               geometry_msgs.msg.TransformStamped,
                                               latch = True, queue_size = 1)
    self.__publishers.append(self.__move_cp_publisher)
    # add attributes to class instance
    self.__class_instance.move_cp = self.__move_cp
```

CRTK ROS: Python Client Example Move

```python
def run_move_cp(self):
    if not self.enable(60):
        print("Unable to enable the device, make sure it is connected.")
        return

    if not self.measured_cp_valid(60):
        print("Waited 60 seconds and didn't receive a measure_cp message, make sure it is connected.")
        return

    # create a new goal starting with current position
    start_cp = PyKDL.Frame()
    start_cp.p = self.measured_cp().p
    start_cp.M = self.measured_cp().M
    goal = PyKDL.Frame()
    goal.p = self.measured_cp().p
    goal.M = self.measured_cp().M
    amplitude = 0.01 # 2 centimeters

    # first move
    goal.p[0] = start_cp.p[0] + amplitude
    goal.p[1] = start_cp.p[1] + amplitude
    goal.p[2] = start_cp.p[2]
    self.move_cp(goal)
    self.wait_while_busy(20)
    # second move
    goal.p[0] = start_cp.p[0] - amplitude
    goal.p[1] = start_cp.p[1] - amplitude
    self.move_cp(goal)
    self.wait_while_busy(20)
```

CRTK ROS: Matlab Client API

- Matlab supports ROS with the Robotics Toolbox
- It uses the ROS Java stack so it runs on Linux, Windows and MacOS
- CRTK Matlab Client API doesn't exist… yet!
- But there's a dVRK ROS Matlab client!
  - API is messy, names are not following a strict convention
  - Performances are not optimized
    - Relies on callbacks which seems to have low priority in Matlab scheduler
    - Subscribes to all the dVRK topics (pre-CRTK)
- So we know we could provide it if there's a need

~~CRTK~~ ROS: Matlab Client API

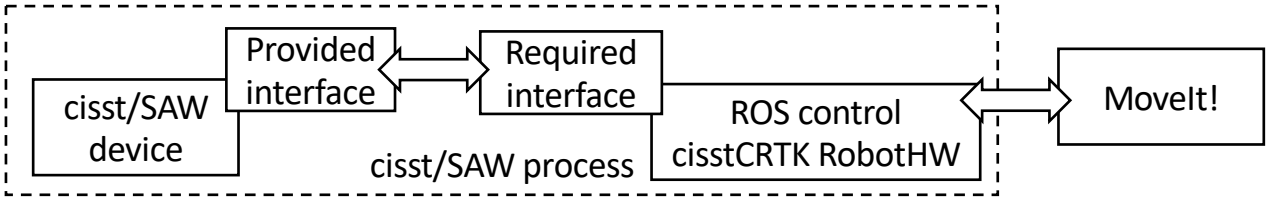

```
cartesian_home = r.get_position_desired();
amplitude = 0.03; % 3 cm
% first move
goal = cartesian_home;
goal(1:2, 4) = goal(1:2, 4) + amplitude;
r.move(goal);
% second move
goal = cartesian_home;
goal(1:2, 4) = goal(1:2, 4) - amplitude;
r.move(goal);
% back home
r.move(cartesian_home);
```

Who would like to see a CRTK ROS Matlab Client API?

CRTK: One Last Thing

- Using existing controllers from ROS-control, work in progress!
  - Why? Any CRTK device could have a MoveIt! Interface for "free"
  - How? Implement at ROS control `hardware_interface::RobotHW` with:
    - Read joint state: `measured_js`
    - Control joints in:
      - Position: `servo_jp`
      - Velocity: `servo_jv`
  - API level?
    - ROS middleware:

    - cisst/SAW:



CRTK: One Last Thing

- Using ROS control future controllers
  - *JointCartesianTrajectory* is clearly designed for MoveIt! joint trajectories, MoveIt! community is now interested in cartesian trajectories
- Using ROS control for new CRTK controllers
  - Take advantage of existing code base
  - Increase potential user base (anyone who has a *RobotHW* implemented)
  - Prime candidate would be `interpolate_jp`
  - `interpolate_cp` might be harder to implement until ROS control supports cartesian space

Links (repositories and branches)

Base libraries

- cisstNetlib: https://github.com/jhu-cisst/cisstNetlib.git
- cisst: https://github.com/jhu-cisst/cisst.git [devel]
- crtk-msgs: https://github.com/collaborative-robotics/crtk-msgs.git
- cisst-ros: https://github.com/jhu-cisst/cisst-ros.git [feature-crtk]

Devices

- https://github.com/jhu-saw/sawSensablePhantom.git [feature-crtk]
- https://github.com/jhu-saw/sawForceDimensionSDK [feature-crtk]
- https://github.com/collaborative-robotics/crtk-python-client.git
- https://github.com/jhu-dvrk/sawIntuitiveResearchKit/wiki [feature-crtk]

All should compile using catkin build tools with ROS kinetic or melodic