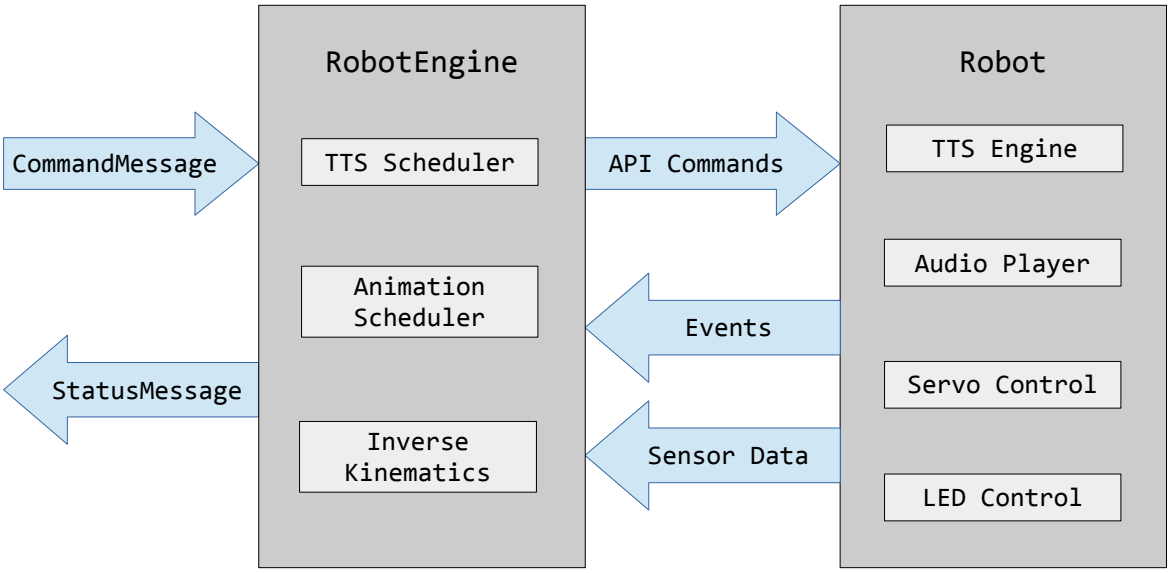
General Principles

This part of the documentation applies to the entire RobotEngine framework. It explains the different components of an application setup, the messages exchanged between those components and different approaches for running each one of them.

Reasons For Using This Framework

- modularity
  - the same application logic can be used with different robotic or virtual agents
  - different applications can be used to control the same agent
- focusing on the high-level interaction
  - control application is mostly independent of the specific agent's capabilities
  - control application is completely independent of the programming language used for the specific agent
- less implementation effort
  - only the interface between the RobotEngine and the given agent is adapted
  - helper functionality such as FIFO scheduling can be re-used with similar agents

How Is The Messaging Protocol Structured?



Basic Principles

Components

- Control Application
  - user interface
  - high-level reasoning, e.g. dialogue management or gaze target selection
  - as little robot-specific code as possible
- RobotEngine
  - translation of high-level commands to robot-specific API calls
  - helper functionality, e.g. FIFO command scheduling or rotation angles for gaze
  - no application logic
- Robot Software
  - usually proprietary “blackbox” software
  - provides API for low-level commands, e.g. text-to-speech output or servo motion
  - no application logic

Data Flow

- *RobotEngine* receives an action command from outside
- *RobotEngine* translates the command to API calls for the specific robot
- *RobotEngine* monitors the command execution on the robot
- *RobotEngine* sends out status reports in order to inform the control application about the execution progress

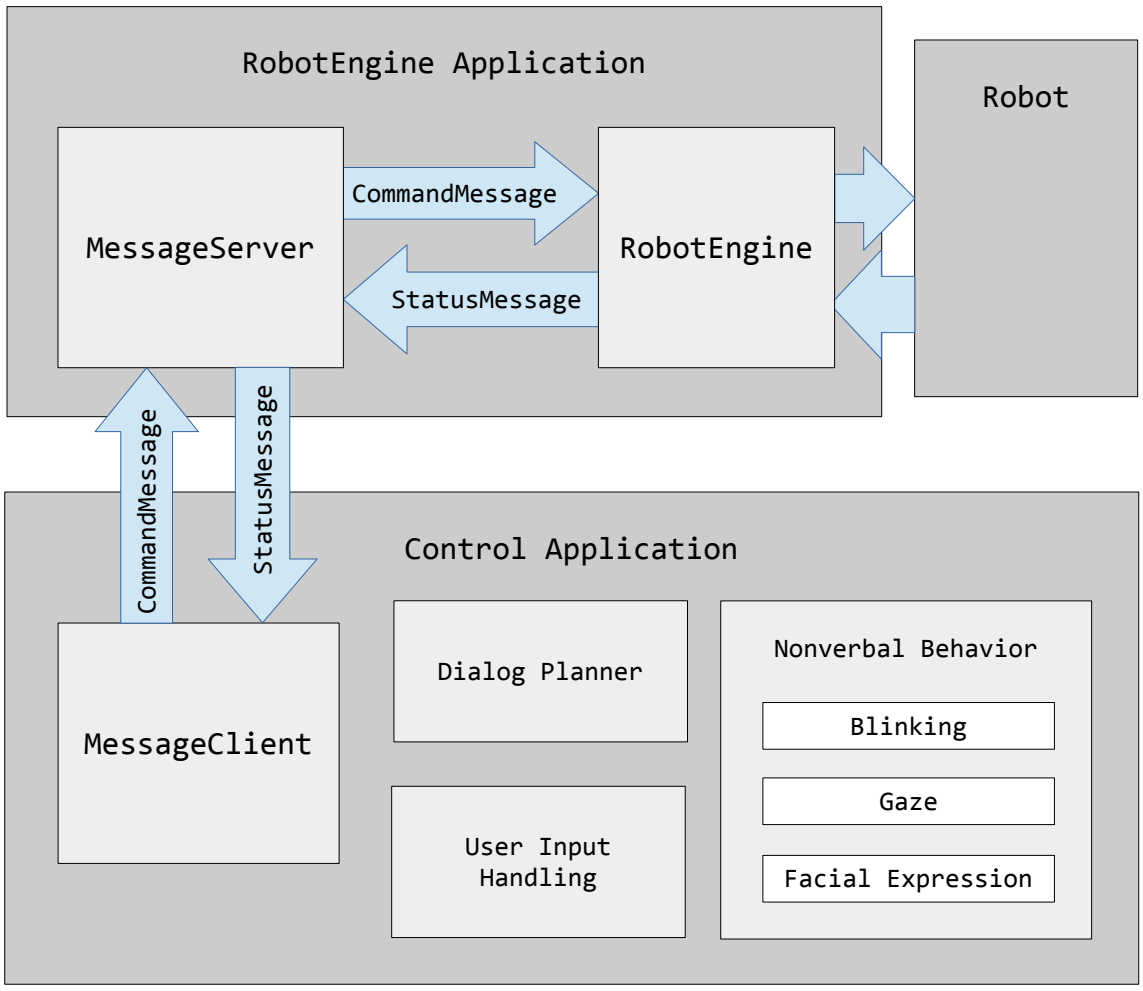
Messaging Classes

- general
  - task ID allows for unambiguous mapping between the command and the associated status messages
  - canonical XML representation
- CommandMessage
  - contains an incoming action command
  - attributes
    - unique task ID
    - action type, e.g. “anim” or “speech”
    - arbitrary list of parameters for executing the action
  - examples
    - <command task="N6::Sample_Anim::t0::u0" type="speech" text="Now I am $42 smiling."/>
    - <command task="N6::Sample_Anim::t0::u0::b0" type="anim" name="Face/happy"/>
    - <command task="N6::Sample_LED::t1::u0::b0" type="led" color="yellow" side="left"/>
- StatusMessage
  - contains an outgoing status report
  - attributes
    - task ID of the associated command
    - current statur of the command execution, e.g. “finished” or “bookmark”
    - arbitrary list of details regarding the execution progress
  - examples
    - <status task="N6::Sample_Anim::t0::u0" status="bookmark" id="0"/>
    - <status task="N6::Sample_Anim::t0::u0::b0" status="finished"/>
    - <status task="N2::Sample_Reset::t1::u0::b0" status="rejected" reason="file not found"/>

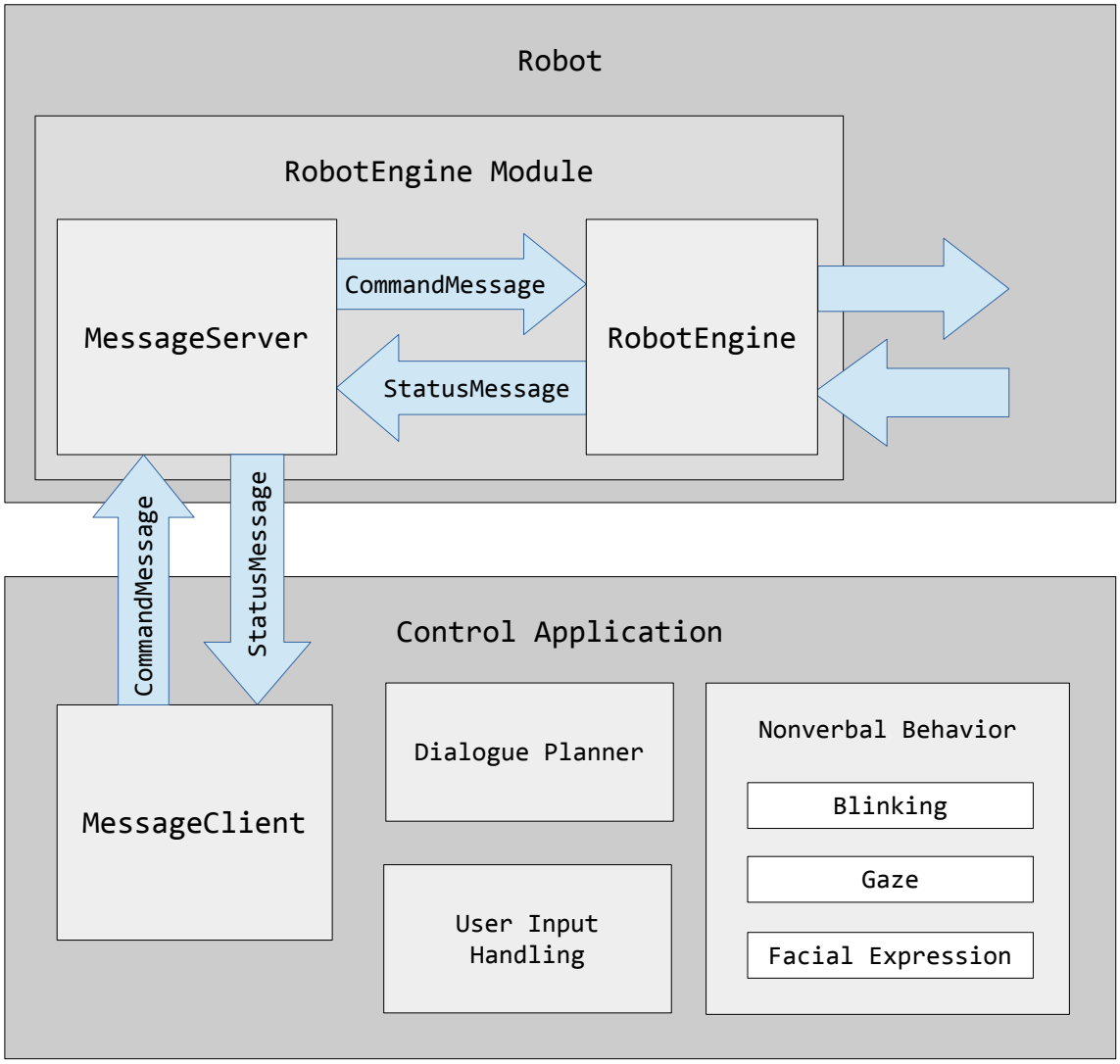
How Is The RobotEngine Used?

Variant A: UDP Connection To An External Control Application

- sample RobotEngine application provided for most programming languages
  - starts a *MessageServer* for receiving commands
  - usually also accepts XML commands via console input
- application configuration file
  - provided to the RobotEngine application as a starting argument
  - general parameters
    - *engine config*: path to a file with robot-specific configuration parameters
    - *local IP:* address of the machine running the RobotEngine application
    - *local port*: port on which the udp server listens for commands
    - *buffer size:* number of bytes for receiving and sending UDP messages
  - implementation-specific parameters
    - *engine class*: name of the RobotEngine child class for the robot in question, e.g. for use in a Java application
    - *window size:* size of the window displaying a virtual agent, e.g. the Klappmaul



- can sometimes be installed directly on the robot
  - e.g. a Python script that is launched automatically or via SSH
  - e.g. an Urbi module loaded when the robot starts



- can connect to any control application which uses the same messaging protocol
  - e.g. the Java class *de.kmj.robots.controlApp.DefaultControlApplication*
  - e.g. a matching Executor loaded into Visual SceneMaker

Variant B: embedded directly into the control application

- possible use cases
  - self-contained Wizard-of-Oz interface
  - animations editor
- RobotEngine API
  - command execution
    - create a CommandMessage
      - unique task ID
      - appropriate action type
    - add parameters
    - call execute() method of the RobotEngine
  - status handling
    - e.g. make the surrounding application implement StatusMessageHandler interface and register it as a listener