Actionlib

⇒ The actionlib stack provides a standardized interface for interfacing with preemptable tasks.

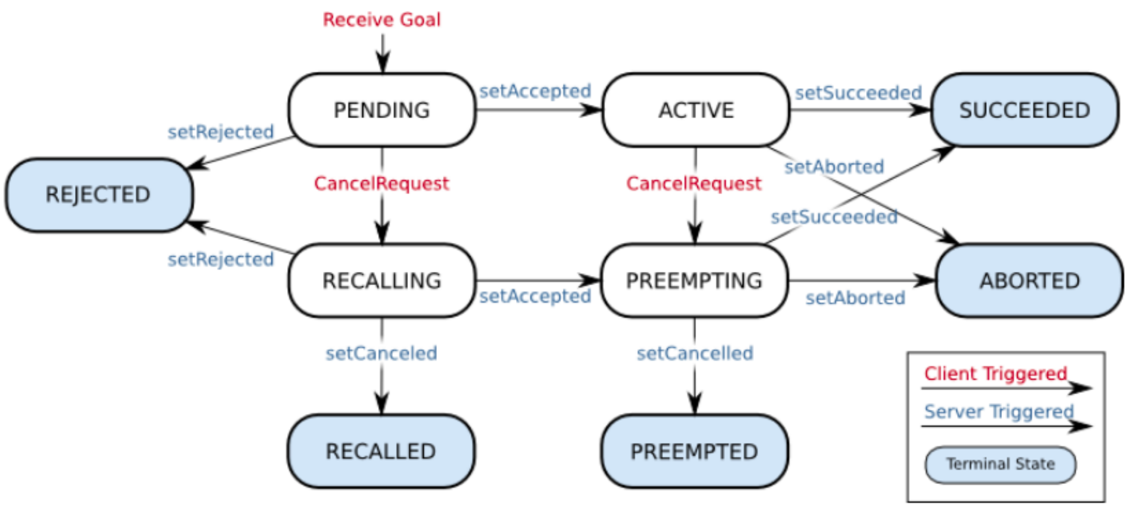
In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time.

★ High level Client/Server Interaction

① Server Description

⇒ Goals are initiated by an ActionClient.

⇒ Once a goal is received by an ActionServer, the ActionServer creates a state machine to track the status of the goal.



⇒ Note that this state machine tracks an individual goal, and not the ActionServer itself.

↳ Thus, there is a state machine for each goal in the system.

⇒ The majority of these state transitions are triggered by the server implementer, using a small set of possible commands:

- **setAccepted** - After inspecting a goal, decide to start processing it
- **setRejected** - After inspecting a goal, decide to never process it because it is an invalid request (out of bounds, resources not available, invalid, etc)
- **setSucceeded** - Notify that goal has been successfully processed
- **setAborted** - Notify that goal encountered an error during processsing, and had to be aborted
- **setCanceled** - Notify that goal is no longer being processed, due to a cancel request

⇒ The action client can also asynchronously trigger state transitions:

- **CancelRequest**: The client notifies the action server that it wants the server to stop processing the goal.

⇒ Server States:

Intermediate States

- **Pending** - The goal has yet to be processed by the action server
- **Active** - The goal is currently being processed by the action server
- **Recalling** - The goal has not been processed and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled
- **Preempting** - The goal is being processed, and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled

Terminal States

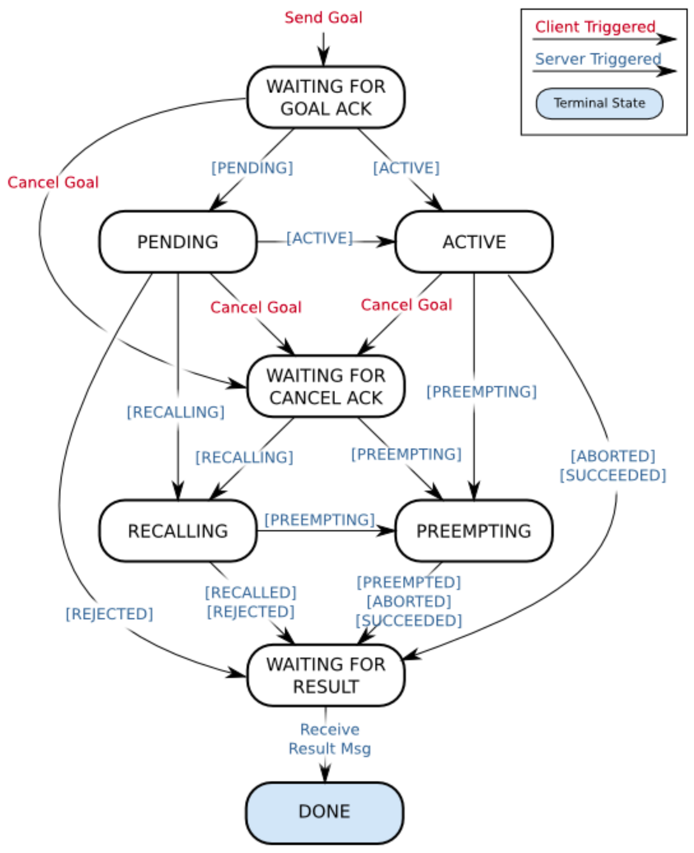
- **Rejected** - The goal was rejected by the action server without being processed and without a request from the action client to cancel
- **Succeeded** - The goal was achieved successfully by the action server
- **Aborted** - The goal was terminated by the action server without an external request from the action client to cancel
- **Recalled** - The goal was canceled by either another goal, or a cancel request, before the action server began processing the goal
- **Preempted** - Processing of the goal was canceled by either another goal, or a cancel request sent to the action server

⇛ It is somewhat unintuitive that an action server can setAccepted a goal even after it has received a CancelRequest.

↳ This is because of a race condition with a CancelRequest being processed asynchronously.

② Client Description

⇛ In actionlib, we treat the server state machine as the primary machine, and then treat the client state machine as a secondary/coupled state machine that tries to track the server's state

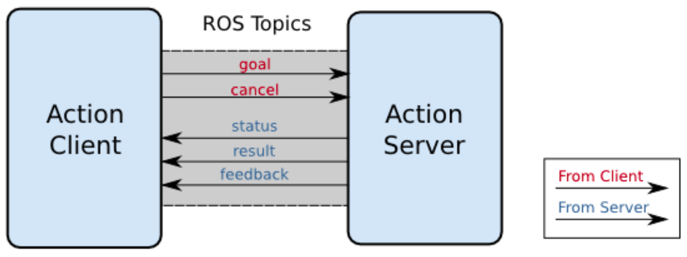


⇛ Since the client is trying to track the server's state, most transitions are triggered by the server reporting its state to the ActionClient.

★ Action Interface & Transport Layer

⇛ The action client and server communicate with each other using a predefined action protocol.

⇛ This action protocol relies on ROS topics in a specified ROS namespace in order to transport messages.



ROS Messages

- **goal** - Used to send new goals to servers
- **cancel** - Used to send cancel requests to servers
- **status** - Used to notify clients on the current state of every goal in the system.
- **feedback** - Used to send clients periodic auxiliary information for a goal.
- **result** - Used to send clients one-time auxiliary information upon completion of a goal

● Data Association and Goal IDs

A Goal ID is a string field that is used in all the messages in the action interface.

This provides the action server and client a robust way to associate messages being transported over ROS with the specific goals being processed.

★ goal topic : Sending Goals

⇛ The goal topic uses an autogenerated ActionGoal message, and is used to send new goals to the action server.

★ cancel topic : Cancelling Goals

⇛ The cancel topic uses actionlib_msgs/GoalID messages, and lets action clients send cancel requests to an action server.

⇛ Each cancel message has a timestamp and goal ID, and how these message fields are populated will affect which goals are canceled.

Cancel Request Policy

		stamp: empty	stamp: filled
ID	empty	cancel all goals	cancel all goals before *stamp*
ID	filled	cancel goal *Goal ID*	cancel goal *Goal ID* + cancel all goals before *stamp*

★ status topic : Server goal state update

⇛ The status topic uses actionlib_msgs/GoalStatusArray, and gives action clients server goal status information about every goal currently being tracked by the action server.

⇛ This is sent by the action server at some fixed rate (generally 10 Hz), and is also sent asynchronously on any server goal state transition.

⇛ A goal is tracked by the action server until it reaches a terminal state.

★ feedback topic: Asynchronous goal information

⇛ The feedback topic uses an autogenerated ActionFeedback message, and provides server implementers a way to send periodic updates to action clients during the processing of a goal.

★ result topic: Goal information upon completion

⇛ The result topic uses an autogenerated ActionResult message, and provides server implementers a way to send information to action clients upon completion of a goal.