Design for a Robot Evaluation System

Overview

The main function of the system is to enable requesting robot evaluation runs and returning the results of these tests. There are 4 main components: a central Server controlling a Database of evaluation Tasks and two types of client applications: ML Client and Operator Client.

Components in Detail

1. Database

The Database stores records of all previous, scheduled and running evaluations. Let's assume this is going to be a SQL database, such as MariaDB, running its own server process on a dedicated server machine. The Server application will work on the same machine and communicate with this Database process using MariaDB Python Connector.

Each robot evaluation session is called a "Task" for short, since in essence it is a task to be performed by a robot operator. Data is organized into 2 tables: Tasks and Robots.

The Tasks table holds all recorded evaluations. Each Task is described by the following fields:
- task_id – unique identifier, assigned by the Server;
- creation_time – Server timestamp representing the time and date of the Tasks creation;
- robot_id – identifier of the robot on which the task was/is to be executed;
- runs – the requested number of runs;
- branch – the name of the Git branch containing robot configuration for the task;
- status = [waiting, running, finished] – the current status of the task;
- attempts – number of attempts performed while running the task;
- successes – number of successful attempts.

The Robots table stores the information about robots recognized by the system. Robot fields:
- robot_id – unique identifier;
- robot_name – display name for client applications.

2. Server

The Server is the central component, connecting to all other parts of the system. It exposes a REST API, that may be used by client applications to issue requests for reading from or writing to the Database. The Server API is detailed in a following section of this document. It enables querying for single task entries as well as returning several task records at a time, with filtering.

The Server is responsible for creating the Task and Robot entries in the Database in response to client requests. Client authentication is also performed by the server. The Server also tracks the timestamp of the last Database modification, which is used for client auto-updates.

The Server is a Python console application with a simple command-line interface used for launching and configuration. It connects to the Database using a Python Connector library. It runs on a dedicated server machine (which also hosts the Databae) and connects to client applications via local network.

3. ML Client

The Machine Learning Client is a GUI application used by ML Engineers to create and review Tasks. It runs on the ML Engineer workstations, which are connected to the Server via a local netwrok. It can be implemented as a desktop GUI application (e.g. in the Qt Framework) or a web application running in a browser.

Its main feature is the Recent Tasks window which shows a subset of existing Tasks, based on their creation time, e.g. all Tasks created in the past 7 days. The Tasks are displayed in a table, divided by Task status (waiting, running, finished) columns target robot rows. The table is automatically updated using a timer – on each tick the client checks the Server's last modification timestamp – if it changed since the last tick, the Task table data is queried and updated. Manual updates may also be triggered by the user.

The Tasks are represented by their unique ids and sorted by time of creation within their table cells (e.g. when there are several waiting tasks for the same robot). Clicking a table entry opens a Task detail view, displaying all fields of that Task. This enables checking the evaluation results of a single Task.

New Tasks may be created by clicking an appropriate button and filling in the required data in adialog: target robot name, number of runs and the name of the Git branch containing robot configuration.

Another button enables downloading a (optionally) filtered list of all recorded tasks and saving it to a file (CSV, JSON etc.). This is useful for analyzing groups of tasks in an external application.

4. Operator Client

The Operator Client is a GUI application very similar to the ML Client, but offering slightly different functions. It runs on the computers controlling the robots and is used by the robot Operators to receive evaluation requests and publish their results. Its implementation would share a substantial amount of code with ML Client. Alternatively is could be a different run mode of the same application.

On startup, Operator Client requires specifying the target robot and the directory containing the robot control script, used to run the robot.

It also has the Recent Tasks window, which works and looks more or less the same as in ML Client. It doesn't have the New Task function however. In the Recent Tasks table, the row representing the connected robot is highlighted. Clicking a Task from that row displays an extended Task Detail window, containing a Run Now and Send Results buttons.

Clicking Run Now will execute the robot control script the number of times specified in the Task's "runs" field, providing it the name of the configurations branch stored in the "branch" field. This action will also send a request to the Server to update the Task's status to "running".

The script is expected to store its results in a log file, at a predefined location relative to the script directory provided to Operator Client at startup. Using the Send Results function in the Task Detail window will cause Operator Client to look for the log file, extract the number of attempts and successes and send this information to the Server via an appropriate PATCH request. Failing to get the expected results exposes the Run Again option, which can be used to retry the entire evaluation from scratch.

Server API

The primary role of the server is providing a REST API for the connected clients. It is described in the table below. Request URLs are given relative to Server address. All Server responses communicate their outcome using status codes.

Operation	**Method**	**Request URL***	**Request Data**	**Response Data**
Any client requests Server's last modification time	GET	/info?q=mod_time	None	Text: Server last modification time as Unix timestamp
Any client requests the Robot list	GET	/robots	None	JSON: array of {robot_id:robot_name}
ML Client adds a new Task	POST	/tasks	Text: robot_name, branch, runs	Text: task_id of created Task
Operator Client modifies Task state and/or adds evaluation results	PATCH	/tasks?id=task_id	JSON with keys: state, attempts, successes	None
Any client requests recent Task list	GET	/tasks?q=recent	None	JSON: array of JSONs representing recent Tasks, based on creation_time value
Any client requests a single Task	GET	/tasks?id=task_id	None	JSON representing the Task
Any client requests a filtered list of multiple Tasks	GET	/tasks?q=all	None; request headers contain filter parameters	JSON: array of JSONs representing filtered Tasks, based on request headers
ML Client deletes a waiting Task; operation will fail if Task state != waiting	DELETE	/tasks?id=task_id	None	None