Design Doc:
Collaborative Online Judge System

Overview

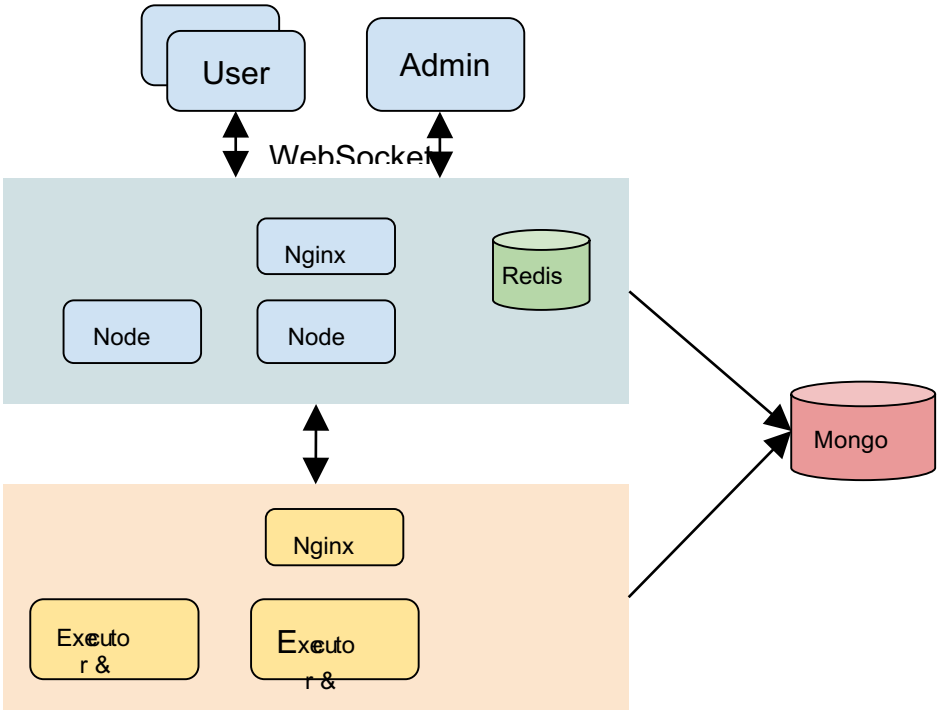
Collaborative Online Judge System (COJ) is a full-stack system supporting collaborative code editing, compiling, execution and result judgement. This document covers the details of the implementation of **Collaborative Editor** and **User Code Executor** from an engineering perspective.

Major Use Cases

1. User can use interactive code editor to edit code. Supported languages are Java, C++ and Python. In addition, we need to keep the capacity for new languages.
2. Multiple users can edit the same piece of code simultaneously. Each user's change can be seen and applied to every other user's code immediately.
3. User can compile the code by clicking 'compile' button. The compile result will be displayed to user.
4. User can run the code by clicking 'run' button. The execution result will be displayed to user.
5. User can browse pre-stored coding problem list.
6. User can get details of a specific coding problem by clicking the problem in the list.
7. User can submit the code through 'submit' button to submit the code to solve the chosen question. The result, including compiling, correctness and running time, will be displayed to user.
8. User's submissions will be recorded for reference.
9. User can check his progress / statistics for questions.
10. Admin can manually add new problem.

High-level Stack Diagram

Stack	Technologies
Frontend - client	Angular.js, Socket.io
Frontend - server	Node.js, Socket.io, Redis, MongoDB, Nginx
Backend (executor)	Nginx, Flask, Docker



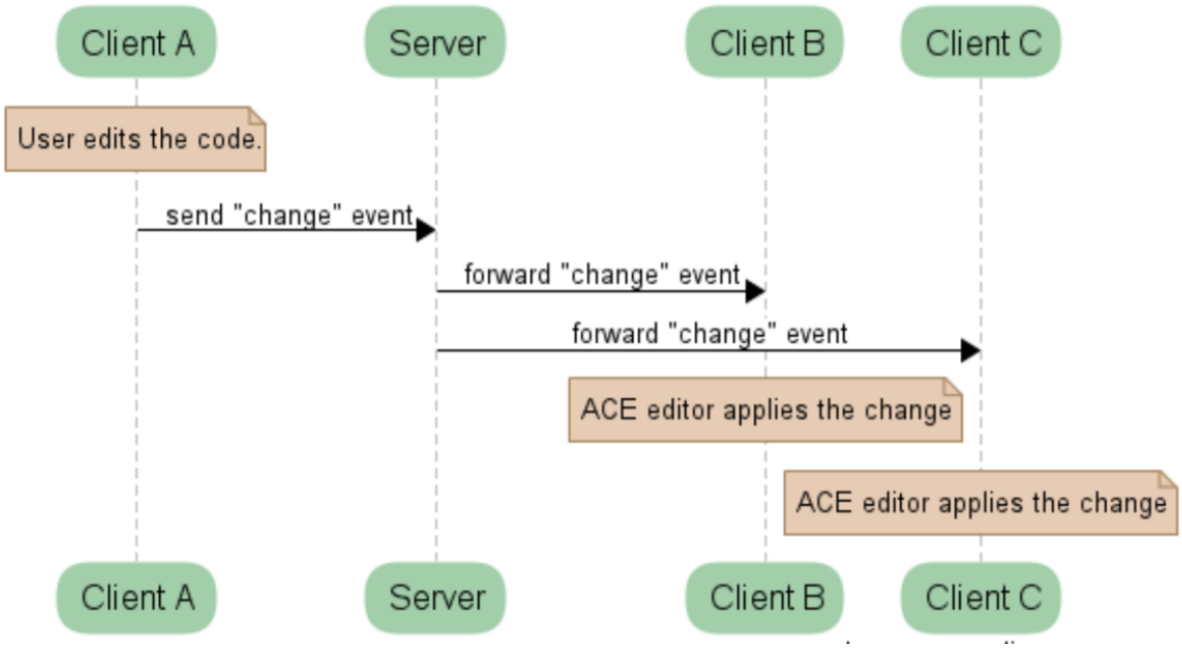
Detailed Design

Collaborative Editor

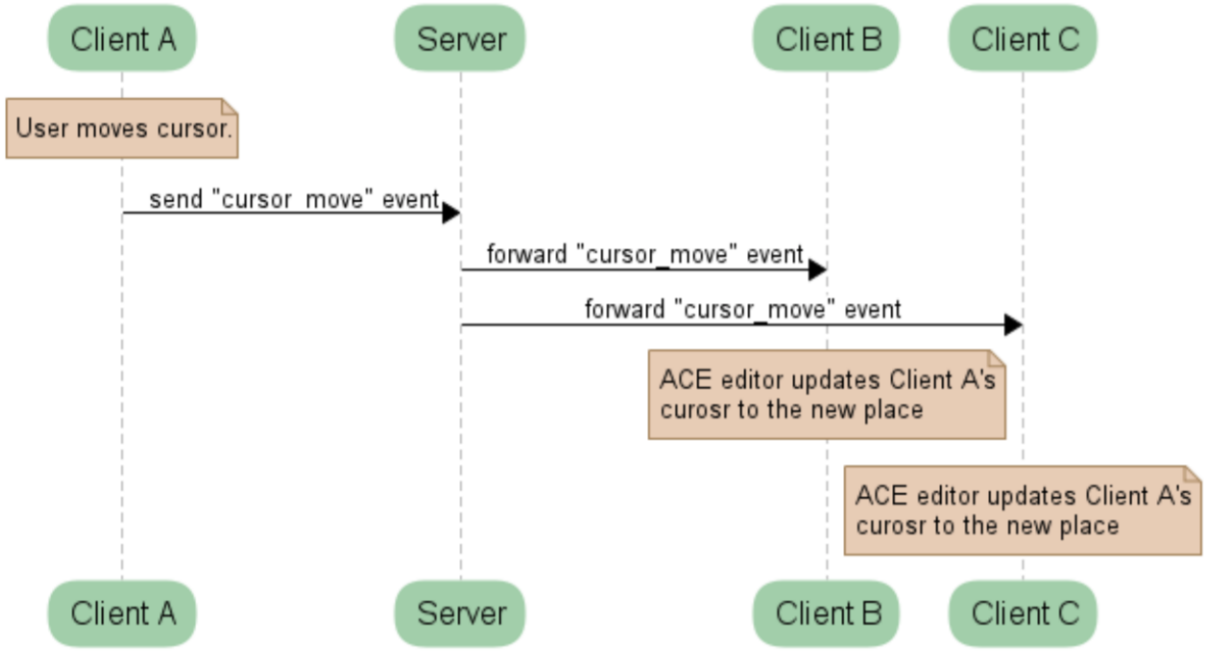
We are using socket.io as the communication protocol between client and server. The reasons are:

- Client-server communication is heavy;
- Full-duplex asynchronous messaging is prefered;
- WebSockets pass through most firewalls without any reconfiguration.

Sequence of Editing



Sequence of Cursor Moving



Client-side Editor

ACE Editor

Here we have two options to choose an editor for browser: ACE and CodeMirror. They are both Javascript-based editor for browser and support source code editing. They both support multiple languages, color themes, programing APIs for advanced usage. (Complete feature comparison can be found here.)
Programing API is the top-1 feature we should consider. We need to dynamically get and change the status of the editor. These include getting the change of the content, applying the change to the current content, and so on. Both ACE and CodeMirror expose a good set of APIs.

We chose ACE as it has been proven to be a stable editor by adopting by Cloud9 IDE. It is easier to get help from community considering the number of users.

ACE Editor API

ACE editor provides APIs to get and set editor status. The APIs we will use are:

API	Description
Editor.on("change", function(Object e))	Emitted whenever the document is changed. e - required. Contains a single property, data, which has the delta of changes
Document.applyDeltas(Object deltas)	Applies all the changes previously accumulated. These can be either 'includeText', 'insertLines', 'removeText', and 'removeLines'.
Selection.on("changeCursor", function())	Emitted when the cursor position changes.
Selection.getCursor() -> Number	Gets the current position of the cursor.
EditSession.addMarker(Range range, String clazz, Function | String type, Boolean inFront) -> Number	Adds a new marker to the given Range. If inFront is true, a front marker is defined, and the 'changeFrontMarker' event fires; otherwise, the 'changeBackMarker' event fires.

Server-side

Editing Session

Editing session is the concept similar to file. It keeps file content, list of participants, editing history and metadata. Multiple users can be in the same editing session, in which case, they work on the same source file simultaneously.

Users in the same editing session will be synced whenever the source file has been changed. In addition, users can see everyone's cursor position in real time.

Editing session will be kept in memory (M) temporarily and stored in Redis (R).

The schema of an editing session:

Name	Type	Location	Description
session_id	int	M, R	UUID to identify an editing session.
type	enum	M, R	[PUBLIC | PRIVATE | SHARED].
owner	int	M, R	The owner of this editing session.
participants	int[]	M, R	List of users who can access this session. If type is PUBLIC or PRIVATE, this field will be ignored; If type is SHARED, this field keeps the users who can access this editing session (other than the owner).
snapshot	string	M, R	Latest snapshot of content.
change_history	string[]	M	Changes after latest snapshot.
last_updated	long	M, R	Timestamp of last change.
created	long	M, R	Timestamp of creation.

Type
An editing session type can be one of PUBLIC, PRIVATE and SHARED. If the type is PUBLIC, anyone with the session_id can access and edit the source file. If the type is PRIVATE, only the owner can access and edit. If the type is SHARED, the users listed in **participants** are allowed to access and edit.

Type is first set to PRIVATE at the time an editing session is created. The owner can make the editing session public or shared with other users later.

Participants
Participants is a list of users the owner shared with. If the session type is PUBLIC or PRIVATE, this field will be ignored. If the type is SHARED, we need to check participants list before we accept an access request (web socket request).

Participants can be modified by owner.

Snapshot
This field keeps the latest snapshot (in string) of the content. The idea of snapshot is we don't update content in memory/storage every time it changes. We just store the list of changes since the last snapshot.

Change_history
When a user makes a change (add an character etc.), the change will be sent to server from web browser through web socket. Then, the server broadcasts the change to all active sockets in the same editing session. Everyone else then updates the content on the browser by applying the change. We will keep the changes in a list.

If a user joins an editing session, the server sends the latest snapshot with the change list. The user's browser will apply changes on the latest snapshot to get the latest content.

If the last user in an editing session exits or after a period of time or the size of change_hisotry exceeds a certain threshold (whichever comes first). The server will update the snapshot by applying all changes in **change_history** and update the entry in Redis. This update prevents the size the of change_history become too large to transfer and sync.

* Change history is kept in memory only. It acts as buffer before flushing to Redis.

Fast Forwarding Restore

It is very natural and common that a new user jumps into an existing session. Or, the existing user may leave the editing page then come back later. In this scenario, we should resume user's editing session by restoring the editor content and fast forward to the latest point.

We have three options here:

- **Keep All Change Events**

This solution is straightforward: server stores all change events it receives in an ordered list: *[event_0, event_1, event_2 … event_n]*. When a new user joins the editing session, server sends the list of all change events to user. User then applies all changes locally to catch up. However, this solution is not optimal as the size of events list increases rapidly. It will consume a lot of bandwidth and memory.

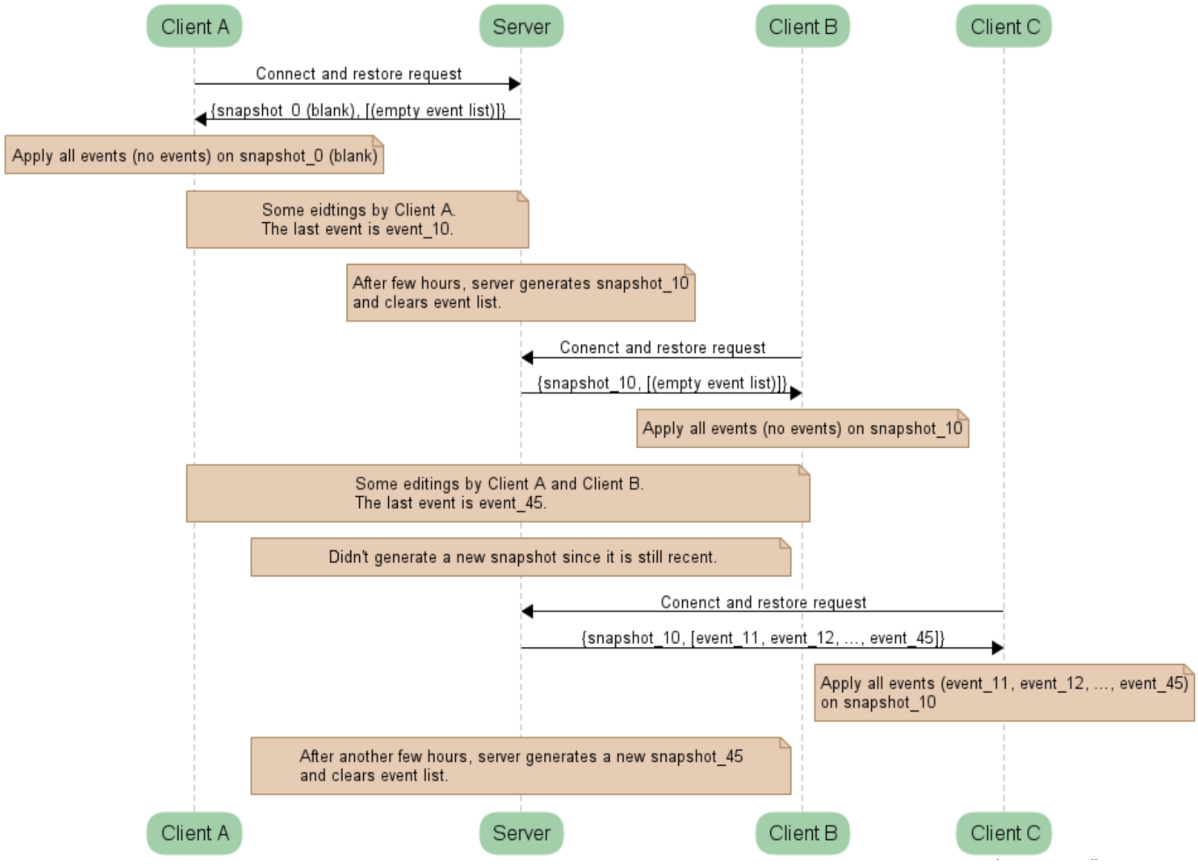
- **Keep Latest Snapshot**

In this solution, server will not keep all events. Instead, it keeps a latest snapshot of editor content. Behaving like an ACE editor, the server will keep a local copy of editor content and apply changes every time it gets a change event. This solution is fast and memory efficient when restoring content for user - just send the snapshot. However, it loses the ability to roll back to an old point or "undo" some operations on server side.

- **Combine Snapshot and Change Events (Adopted)**

This solution combines the above two. Server keeps a snapshot before a certain point (e.g. 1 hour before), and list of change events since that point: *{snapshot_n, [event_n, event_n+1,*

event_n+2 ...]}. This solution limits the size of event list, as well as keep the ability for rolling back.



Snapshots and change events are stored granularly based on the frequency of access.

Location	Content
Memory	List of change events since last snapshot.
Redis	Latest **few (TBD)** snapshots.
Disk / Database	Old snapshots.

User Code Executor

We allow users to submit their code through web UI. We will try building and running code behalf of user. For security reason, we cannot execute user code directly on server. We can utilize

1. language specific security tool/package: SecurityManager in Java, Pypy in Python etc.

2. Container technology: Docker etc.
3. Virtual machine: VirtualBox, Vagrant etc.

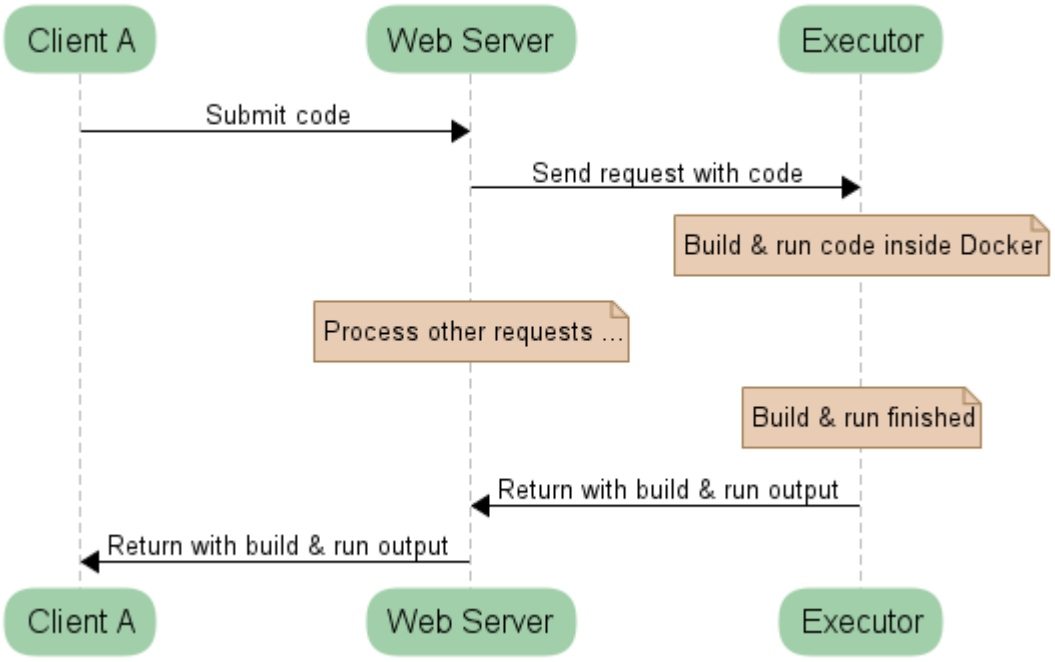
Here we compare pros /cons of different approaches:

Options	Pros	Cons
Language Specific Tool/Package	Small overhead;	Need configuration for each language; Need clean up work after the execution;.
Container	Lightweight; Quick to initialize;	Weaker OS isolation;
Virtual Machine	Complete isolation;	Slow to initialize;

Container is an obvious winner if we want to support multiple languages and don't worry about performance too much.

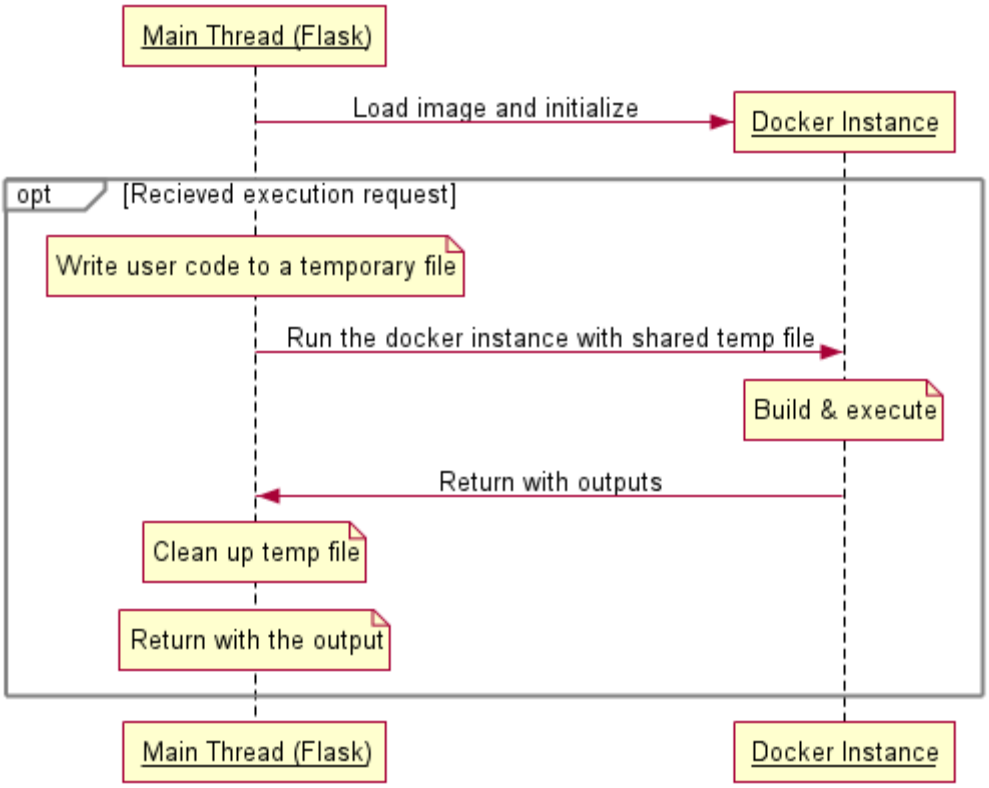
Executor Server

We are going to use Docker container to execute user-submitted code on server. In order not to slow down the frontend server (Node.js server), we should deploy Docker container on backend server and make it accept execution requests coming from frontend server.



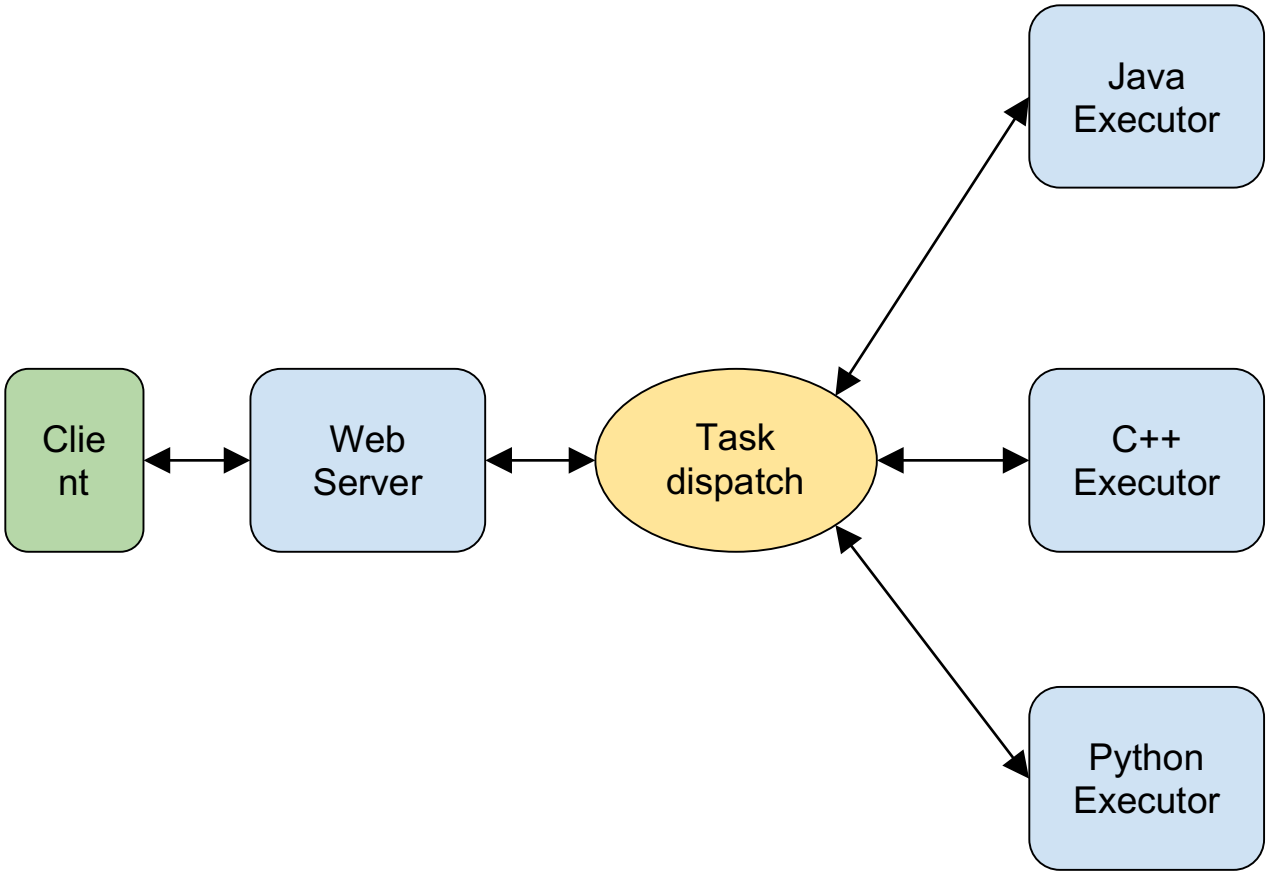
Docker

With Docker Hub, we are able to pre-create a container image with all necessary environment & tools ready, then use it on all execution instances. This approach needs one-time image download and initialization every time it executes code. Considering the fast initialization and loose time constraints, it is OK to accept the initialization time.



Task Dispatcher

In order to make the system scalable and easy to maintain, for each language, we can set up individual Docker image. In this structure, we need a dispatcher to dispatch execution tasks based on language type.



Task Dispatcher can be implemented using Node.js as it is perfect for I/O heavy scenario (which is our case).

Scalability (TBD)

Test Plan (TBD)

Launch Plan (TBD)

Future Work (TBD)

Reference (TBD)