Specification

Introduction

Context

A few lectures of our university program are using tournaments as edu-candies for spicing up the labs, e.g., competitions organized during Artificial Intelligence 1, Multi-agent Systems or Human-like Artificial Agents. However, running tournaments by hand is time consuming as well as developing new semi-automated solutions for them. This project aims to provide a standardized environment for conducting and judging of these tournaments. On a broader scale, such a software can be employed in the context of AI competitions held at various academic conferences (such as CIG, AIIDE or IJCAI). Our primary competitor is http://theaigames.com/ webpage, whose solution is proprietary and therefore it cannot be adapted for our purposes.

Example use-case

To bring a better overview on what the goals of this project are, we present an example use-case in the following section. This use-case includes the main pipeline of the stages that happen during creating/competing in/evaluating the tournament. The text is split into three parts corresponding to these stages.

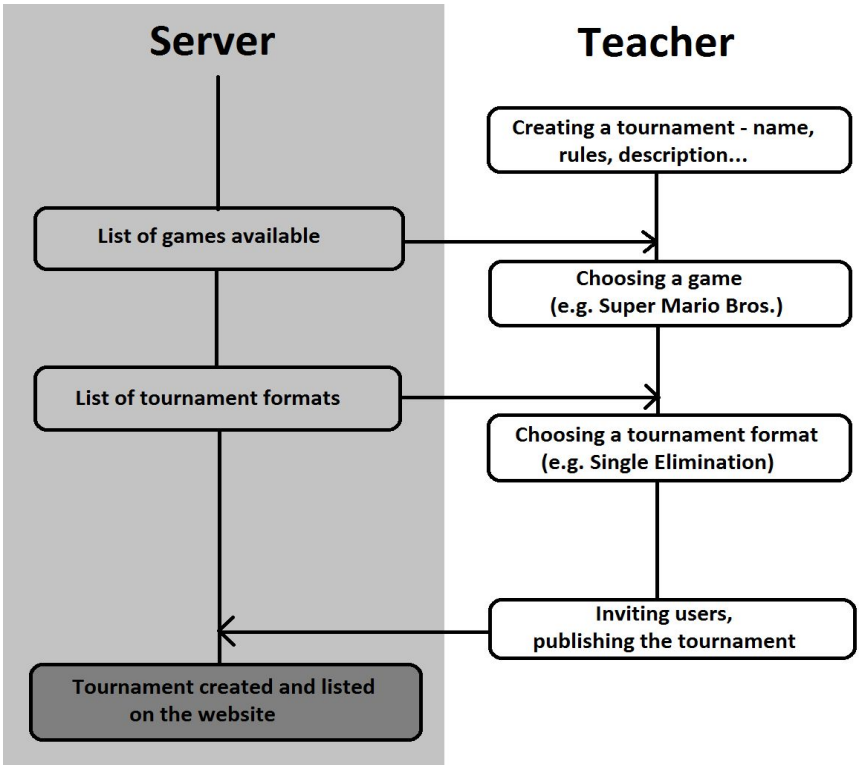


Fig. 1: Creating a tournament

Creating a tournament

To provide one of the desired use-cases, we will provide an example from our university. At the beginning of the process, there is a teacher, leading a course which uses tournaments as a part of rating the students' solutions. Schema of creating a tournament is presented in figure 1.

When the teacher wants to create a tournament, first he has to fill in some basic information about it, such as name, description, rules etc. Then, the server provides him a list of games and tournament formats currently supported. After choosing the desired game and format, the teacher can finally edit the tournament, eventually invite users to it (students of his course, for example) and publish it. After publishing, the tournament is created and listed on the website, and is available for submitting the solutions. The tournament will be visible either to all users (if it was created as public), or those that the teacher invited (in case of private tournaments).

Submitting a solution

The second stage of our pipeline is submitting a solution. The diagram of this stage is given in figure 2.

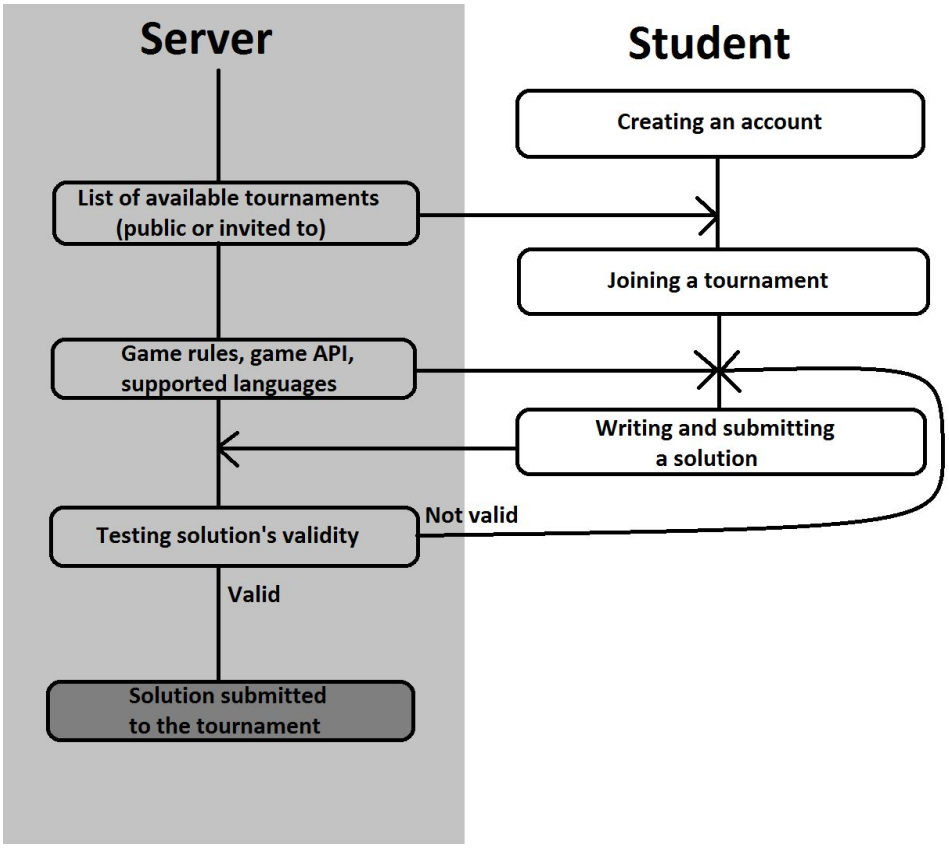


Fig. 2: Submitting a solution

To join a tournament, a user (student) has to create an account first. When logged in, he can choose from a list of tournaments available to him (that is, public or private ones that he is invited to). After choosing a tournament, he can write a solution and submit it on the tournament's page. The solution must follow game rules and API, and must be written in one of the supported programming languages. If everything is ok, the solution is submitted successfully and the user has joined the tournament.

Evaluating the tournament

The last stage, the evaluation of the tournament, happens entirely on the server's side of the pipeline. The final diagram is presented in figure 3.

After the tournament's evaluation starts, the server starts to plan matches depending on tournament's chosen game and format. When the match is planned, the solutions (e.g. players in that match) are chosen from the currently available pool of players. The server then runs the match, and evaluates its results. Alongside with evaluating the results, they also become available to the users that created solutions participating in that match. If the tournament is not finished, the whole cycle happens again, otherwise the tournament is evaluated completely, and its results are published on the website.

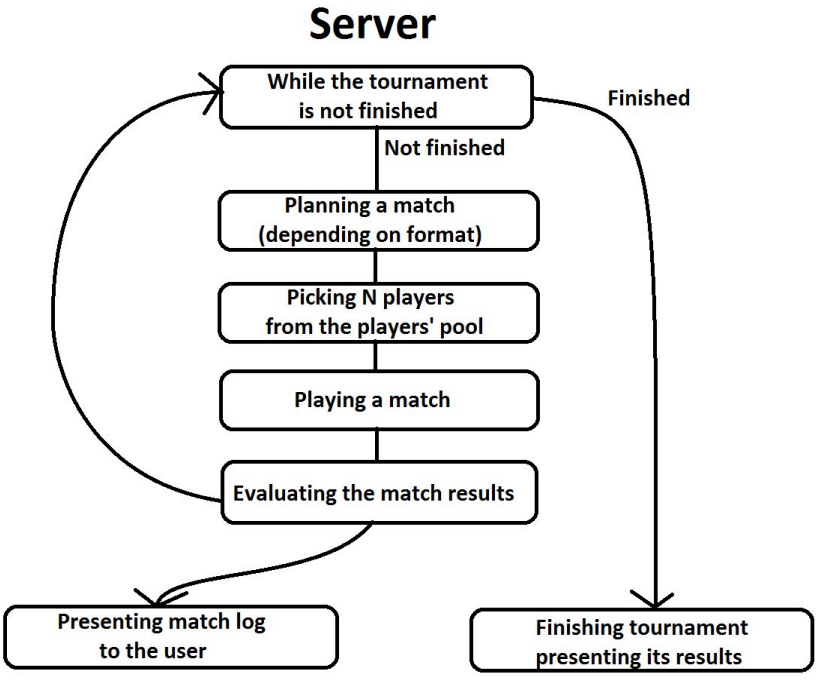


Fig. 3: Evaluating a tournament

Terminology

In this section, we will list terminology used in the following text alongside with a brief description of these terms.

Game - a game/puzzle to be played
Match - a match/round of game, played by submitted bots
Bot - any bot/solution participating in the matches
Submission - a source code with the bot submitted by the competitor
Tournament - a contest of bots with a specific format in a specific game
Guest - any guest of our site, that is not logged in
Competitor - a logged in user, someone who contests in the tournaments
Organizer - user that creates tournaments for competitors
Admin - administrator of the web, manages the website/app/servers
Game module - organizer-provided “blackbox” that runs the game and the bots

Functional requirements

Conceptual model

Let us first briefly summarize the conceptual model of our platform. The central notion is a **tournament**, which is carried out in individual **matches**. Depending on the **game** in which the tournament is held, there may be one or more **bots** in a single match. These bots will be provided by individual **competitors**. The number of matches, their order and selection of participating bots is determined by the given **tournament format** (e.g. single elimination). Each tournament also has a **tournament scope**, which divides the tournaments into **ongoing tournaments** and **deadline tournaments**.

For each game, there will also exist a corresponding **game module**, which is a part of the platform responsible for the actual match execution. Extending the platform to support a new game would be done by implementing and installing a new game module. Execution of a match by the game module should be parameterizable to allow different tournament types to be held. Such parametrization is achieved by associating a **game configuration** to the tournament. This configuration determines game specific conditions such as time per turn for individual players (like in normal vs speed/rapid chess).

Tournaments

There are many ways in which a tournament in games in general can be organized. Our platform will provide a basic set of tournament formats which should be general enough for most use cases. These tournament formats are listed below, grouped by the number of players competing in an individual match.

Single player games (puzzles)

The most simple type of a game (or a puzzle) is a single player game. For such games, all the supported tournament formats are simple ranking. Each match in a single player game yields a result in points which then contributes to its overall ranking that is simply added to the total number of points obtained by the bot.
The following formats are supported for single player games:

Minimal/Average/Maximal performance ranking

Submissions in these formats are ranked based on the minimal/average/maximal number of points obtained in their matches.

Two player games

The most usual type of AI games in tournaments is a two-player game. For this category of games, our platform offers four most usual tournament formats:

Elo ranking

First type is an Elo ranking tournament. Players of such a tournament are ranked based on their Elo value (https://en.wikipedia.org/wiki/Elo_rating_system), which should express how good they are in the game compared to the other players in the tournament. Competitors entering the tournament will get some basic Elo value, and after each match, the values are updated for both players. The new value is computed as the old elo value added to the difference between expected and actual performance of a player, multiplied by a suitable constant. For example, players get more ELO points if they defeat better opponents and less ELO points when they defeat lower-ranked players.

Table tournament

Another popular format of tournament is a table tournament (the tournament used by most sport leagues). In those tournaments, competitors play matches against each other in a specified number of rounds (one or two are the usual values). The results (“result points”) of all matches are summed and based on that sum, the competitors are ranked in a table. The specific game modules determine the actual values of those resulting points. The points can be distributed in any way, however, the usual requirements are: a) constant number of points awarded per match; b) sum of points of both players in match is equal to that constant. An usual example of such a point system is “three points for a win system”, used in most sport competitions (3 points for win, 0 for loss, eventually 2/1 points for close win/close loss, 1.5 points for draw if the game allows it).

Single/Double elimination brackets

Single and double elimination brackets are two other very popular formats of tournaments, even outside the (PC-)games scope. As the name suggests, these tournaments generate an elimination bracket for N players when created, and the players are automatically assigned to their matches. The number of players required for the tournament can be specified. The “single/double elimination” keyword signifies how many matches a player must lose to get out of the tournament (one or two). After all the matches are finished, the tournaments’ ranking is determined by the completely filled brackets.

Multiplayer games

Finally, our platform will also support games for more than two players (possibly “any number”). We consider all games for N players to be played in a free-for-all fashion. In a free-for-all match, all the bots compete against each other and are rewarded with some points (for example, in a deathmatch, the points could correspond directly to the order in which they were eliminated). Our platform will provide only one tournament format for such games:

Multiplayer Elo ranking

Based on the resulting points from the matches, the submissions are ordered in a ranking system based on a multiplayer version of ELO.

Tournament Scopes

A tournament scope defines the persistency of the tournament, there will be two tournament scopes available.

Ongoing tournament

In an ongoing tournament, the bots can be submitted at any time. The matches between submitted bots are played “infinitely” in regular intervals, and the ranking of the players is continually updated as the matches are executed. All of the ranking formats support the *ongoing tournament* scope, that is:

- **Minimal/Average/Best performance rankings**
- **Elo ranking**
- **Multiplayer Elo ranking**

Deadline tournament

In a deadline tournament, the submissions for such a tournament must be submitted before some deadline, after which the matches (or the rest of the matches, depending on the format) will be played. The results of the tournament are available after all the matches are finished, and thus the rankings are final.

- **All the tournament formats support this type of scope**

Users

The goals of this project that were set out in the introduction suggest different functional requirements for multiple categories of users. These categories correspond to different roles users can play in tournaments held on our website. In the following sections, we will focus on those different roles and briefly describe their available features.

All of the users of our web application belong to one of the four categories listed below.

- **Guest**
- **Competitor**
- **Organizer**
- **Administrator**

The categories are ordered hierarchically according to their permissions, meaning that e.g. Organizers are allowed to do everything that Competitors can do.

Guest

A **guest** is essentially a user who is not logged in. Such users only have access to information about games and public tournaments. Guests may become competitors by creating an account and logging in. On our website, the guests are able to:

- View some general static pages (Frontpage, Terms of use, etc.)
- View a list of supported games and their info
- View a list of public tournaments

- Login/Sign up

Competitor

A **competitor** is anyone who wants to take part in the tournaments. They have access to detailed information about available tournaments and are able to submit their bots to particular tournaments and view results of individual matches. Specifically, in addition to the guests' abilities, competitors are able to:

- View detailed information about tournaments
- Submit a new bot to a particular tournament
- Manage and view information about their submissions
- See matchlogs (results of matches) of all bots
- See detailed logs of their bots (such as match details, compiler/checker logs…)
- See detailed results of tournaments they participate in

Organizer

An **organizer** is the highest publicly available category. As the name suggests, the category is meant for users who want to create their own tournaments. Users can be upgraded to organizers by administrators. Apart from competitors' abilities, the organizers can also:

- Create a new tournament
- Manage settings and information about their tournaments
- View detailed info about all submissions to their tournaments, including all matchlogs

Administrator

This category is meant only for people managing the platform and therefore such users have highest privileges. These privileges include:

- View and edit user information, including their category/privileges
- View and edit all tournaments
- Manage the list of installed game modules (and therefore list of available games)
- View system status, such as available disk space and currently executing matches

Web application

All users will communicate with the platform via a web frontend, the following sections describe the individual pages and their function. The pages are logically grouped by the user categories with enough privileges to view them.

Public pages

Public pages are those which can be viewed by any user.

Static pages

The web contains several static pages available to everyone, such as the **Frontpage** (with general info about the project), Privacy Policy, Terms of usage, Rules/How to use, ... All of these pages are available directly from the **Frontpage**.

Games

The **Games** page is a list of currently supported games, and is available directly from the **Frontpage**. Each game in the list has some basic info about itself and a title photo. The page also optionally features some info about running the bot and the game itself - such as what exactly is a bot/submission for that game, link to the source code of the game, what dependencies it has etc. Finally, there is also a link to all the tournaments filtered by the chosen game. That link will take the guests to the **Tournaments** page.

Tournaments

The **tournaments** are the core of this application, and so is the page. This page contains a list of all tournaments that are publicly available. Some basic info about the tournament, such as the game the tournament is held in, tournament type, time to the end of tournament etc. are to be seen directly from the list. That list can also be filtered in many ways, such as filtering by game, tournament type, time to go etc. Tournaments in the list also serve as a link to their own detailed page.

The detailed page of the tournament contains several publicly available tabs. On the main tab (**Overview***)*, the user can view general information about the tournament. **Leaderboard** tab is dedicated to the tournament final/ongoing results. The format of these results depends on the tournament format - it could be for example a visualised bracket, or an Elo ranking table. Next, there is a tab called **Rules** which is dedicated to detailed description of specific game rules (if there are any), as well as the tournament format rules (such as brackets format, ranking type and so on). A link called **API** redirects guests to the wiki of the game API/environment, as the name suggests.

Sign up/Log in

Finally, there is also a **Sign up/Log in** page, whose meaning is pretty self-explanatory. Any guest can create an account by entering the usual credentials - email address, full name, affiliation (optionally), nickname and password. Once the account is confirmed via the mail sent to the given address, the user can log in to our web page.

Competitor Pages

Pages/tabs available to all logged in users are listed below.

Competitor dashboard

After logging in, the competitors can view most of their user-related information on the **Competitor dashboard**. First of all, they can see and eventually change their credentials there (password or email), and also optionally add some more information about themselves, such as real name, country, school/company and so on.

The main purpose of the dashboard is a list of the tournaments the competitor participates in. This includes tournaments that the user has already submitted a solution to, as well as the tournaments the competitors were invited to (by the tournaments' organizers).

Tournament - tabs Match Log and Submissions

On the tournament page, registered users (competitors) have access to two additional tabs. First of them is called **Match log**, and it contains performance of the player's bots/solutions for the chosen tournament. The format of this log is visualised depending on the tournament type.

The last tab, **Submissions**, is the most important. Upon opening, it lists all the submissions the user has sent for the tournament, alongside with their validity and tournament results (match logs), which could also be viewed directly through here. Contestants can also choose which of their valid submissions will be evaluated further in the tournament, if the tournament format and scope allows it. Finally, the users can also submit new bots/solutions to the chosen tournament. Once submitted, the submission is passed to the game module for validation. Results of the validation process will be available in the submission tab. If validation succeeds, the bot is considered valid and will take part in the tournament.

Organizer Pages

Users belonging to Organizers and Administrators category have access to the administration area of the web application.

Tournaments section

This section allows organizers to add, manage and list tournaments that are available on the platform. In order to edit a tournament, the organizer needs necessary permissions for that particular tournament.

Tournaments are created in two separate steps. In the first step, only basic information is filled in. After that, a tournament is created and the organizer is redirected to a page where they have complete control over all the settings of the tournament.

The following list contains the most important settings of individual tournaments.

- **Name and description**
- **Choosing a game:**
  The organizer has to decide which game will the tournament be held in, that is choosing from the list of games supported by the website. Each game on our website has a **game type** (determined by the number of players playing one match of the game), so the choice of the game also influences the possible formats of the tournament.
  The following game types are supported:
    - **Single player**
    - **Two players**
    - **N players**

  After choosing a game, the organizer has to specify the **configuration** of the game (of its game module). Every game in our app has its configuration following its own configuration format. Such configuration is basically just a .json file that could

determine for example the map/plan of the game (if there are more than one), time for players' turns and so on. The schema for the configuration file can be different for each game.

- **Tournament format:**
  Our web supports majority of the most popular tournament formats, both ranking and bracket types. These formats are divided into groups by the **game type** of the game used in the tournament, and they were described in detail in the **Tournaments** section.
- **Tournament scope**
  After choosing the tournament format, the organizer also has to specify the scope (or the "persistency") of the tournament. Basically, there are two types of those scopes: ongoing tournament and deadline tournament, which were described above in their respective section. However, not all of the tournaments' formats support both scopes. Regardless of the scope chosen, the organizers can also choose a starting date for the tournament, after which it becomes available to competitors.
- **Rules**
  Organizers can provide rules of the tournament. These rules include details about the tournament format and scope (as they were described above). The organizers can change the rules description manually, or even add some not-game-specific rules (for example determined by the configuration for the tournament, or information about the bots' compilation).
- **Wiki**
  The organizers are also able to create simple documents (using markdown) associated with the tournament/game. These documents can serve for example as a detailed description of the rules of the game, or as an API documentation of the game (or both).
- **Availability**
  The tournaments on our web can be created either as **private** or **public**. **Private** tournaments won't be available to the public, that is, they will not be listed on tournaments page. To make these tournaments available, the organizers can invite users using their usernames or emails. The invited users will get an email notification, and they can also see all the tournaments that they were invited to in the dashboard. After getting to the private tournament's page, the competitors can freely submit their solutions as usual. The organizers also can invite some more users later through the Manage tournament page.

  The second type of tournament, the **public** one is an ordinary tournament available to anyone.

The organizers also have the option to invite other organizers to manage the tournament they created. These invited organizes will have the same privileges in terms of managing the tournament as the "main" organizer.

After a tournament is created, it is in a state of concept. Its organizer (organizers) are further able to view and edit everything. After the organizer is satisfied with the tournamen's form, they can **publish** the tournament. After publishing, some of the sections of the tournament

can be edited further, mainly the ones that do not violate the tournament's run/submissions. For example, the organizers can still edit information like Rules and Wiki. The organizers can also manually start/stop the tournament, are able to view all the match logs in their tournament and see all the submissions and their results.

Admin pages

Finally, there will be some pages that will be available only to the most privileged users.

Games section

This section allows administrators to add, manage and list games that are available on the platform.

Create a game

The process of adding a new game to the platform has two steps. The first step consists of creating the game through the website and filling some basic information about the new game, such as name, title photo, optionally links to the source code etc.

The second step - adding the game module - is more complicated and can possibly involve installing the actual executable and other key components on the target machine. Thus the second step cannot be done through the website. Instead, the game module has to be manually installed on all worker machines.

System dashboard

The system dashboard page displays the overall health and status of the platform. The information displayed will fall mainly into following categories

- **Errors and notifications** - e.g. messages about failed match executions, notifications about depleted disk space.
- **Currently executing matches**
- **User management** - editing user information, including privileges

Tournament Execution Backend

Running Games with Bots

The ultimate goal of our platform is the ability to host tournaments in the "big games" such as Unreal Tournament (through Pogamut middleware), StarCraft etc. This poses nontrivial requirements on the platform because each game may use different means to interoperate the AI bot into the game itself. For illustrations, we consider the following cases:

1. The bot is started as a separate process which communicates with the actual game instance via e.g. TCP socket
2. The bot implementation resides in a dll and is loaded into the actual game process

Both these approaches require different handling in terms of the actual starting/monitoring the match between multiple bots. There can be even further differences between handling games from the same “group”.

Another matter is extracting the match result after the match execution finishes. For each game, there are different attributes which determine the winner of the match. First-person shooters in death-match mode may order players based on their kills to deaths ratios, winners in real-time strategies are those who survive the longest etc.

Our platform therefore should aim to be flexible enough to accommodate the specific needs of these games. The game-specific functionality would be therefore separated into individual **Game modules**. These modules will be provided by the tournament organizers and our platform would use them on black-box basis to execute the matches.

Interface of Game modules

The interface of the game modules will consist of several entry points for individual stages of the submission evaluation. The entry points are the following:

- **Checker** - Does preliminary checks on the competitor submissions, such as that the submission contains all required files.
- **Compiler** - Compiles the solution submitted by the student. In case of complex games, this step will also make sure that all appropriate libraries are supplied for the compilation
- **Validator** - Performs further validation on the compiled submission. It may for example start a simple match to check that the provided bot actually connects and properly communicates with the game
- **Executor** - Performs the actual execution of a match and summarizes the match results in a format which will be further processed by the OPCAIC platform. This step may also produce additional data files e.g. recordings of the match which could be then replayed to closely observe the AI behaviors.
- **Cleanup** - Allows the module to recover from failure, e.g. kill hanging processes.

All these entry points except cleanup will accept two paths: a directory containing the input files for the entry point, and the output path for storing the results and logs.

Workers

Executing many tournaments could potentially consume too much resources and degrade the performance of the web server. In order to scale well, our platform should distribute the tournament execution to a pool of dedicated workers.

Since some games we would like our platform to eventually support do not run flawlessly on all operating systems, our platform should support workers with potentially different operating systems. This implies that not all workers would be able to run the same set of games. The workers will therefore expose a list of games they are able to run based on the

list of the installed Game modules. The server will then distribute the matches according to the workers' capabilities.

Because the worker machine may occasionally fail, the work distribution mechanism should be able to recover from crashing worker. This will be achieved by implementing a heartbeat between broker and worker machines. While the server receives heartbeat messages from a worker, the worker is considered alive. If heartbeat messages stop coming (e.g. due to worker machine crashing or being forcibly restarted) then the worker is considered dead and the broker should reschedule the work to another available worker, or wait for the worker reconnection.

The rescheduling of a work to different worker should be tried only limited number of times in case the specific work item was the cause of the worker crash.

Scenarios

Verifying a submission

1. User submits his solution via a Web app
2. Server saves the submission
3. Server queues the submission for (asynchronous) validation on a worker
4. Worker downloads the submission files
5. Worker finds the appropriate Game module and invokes its entry points
    a. Checker
    b. Compiler
    c. Validator
6. Worker uploads the results back to the server
7. Server saves the results in database

Executing a match

1. Server schedules a match to be executed on a worker
2. Worker finds the appropriate Game module for the given match
3. Worker for each bot participating in the match
    a. Downloads the files submitted for the bot
    b. Invokes the Compiler entry point for downloaded files
    c. Validates the compiler output by invoking the Validator entry point
4. Worker invokes the Executor entry point
    a. Executor produces results to a known location
5. Worker uploads the results back to the server
6. Server saves the match results to the database

Deploying a new game on a worker

1. Organizer implements the Game module for the chosen game
2. Admin installs the game on the worker machine
3. Admin adds the game module to the worker
4. Admin restarts the worker process

5. Worker finds the new module among the others and reports it among its capabilities when connecting to the server
6. Server now can schedule matches in the new game on the worker

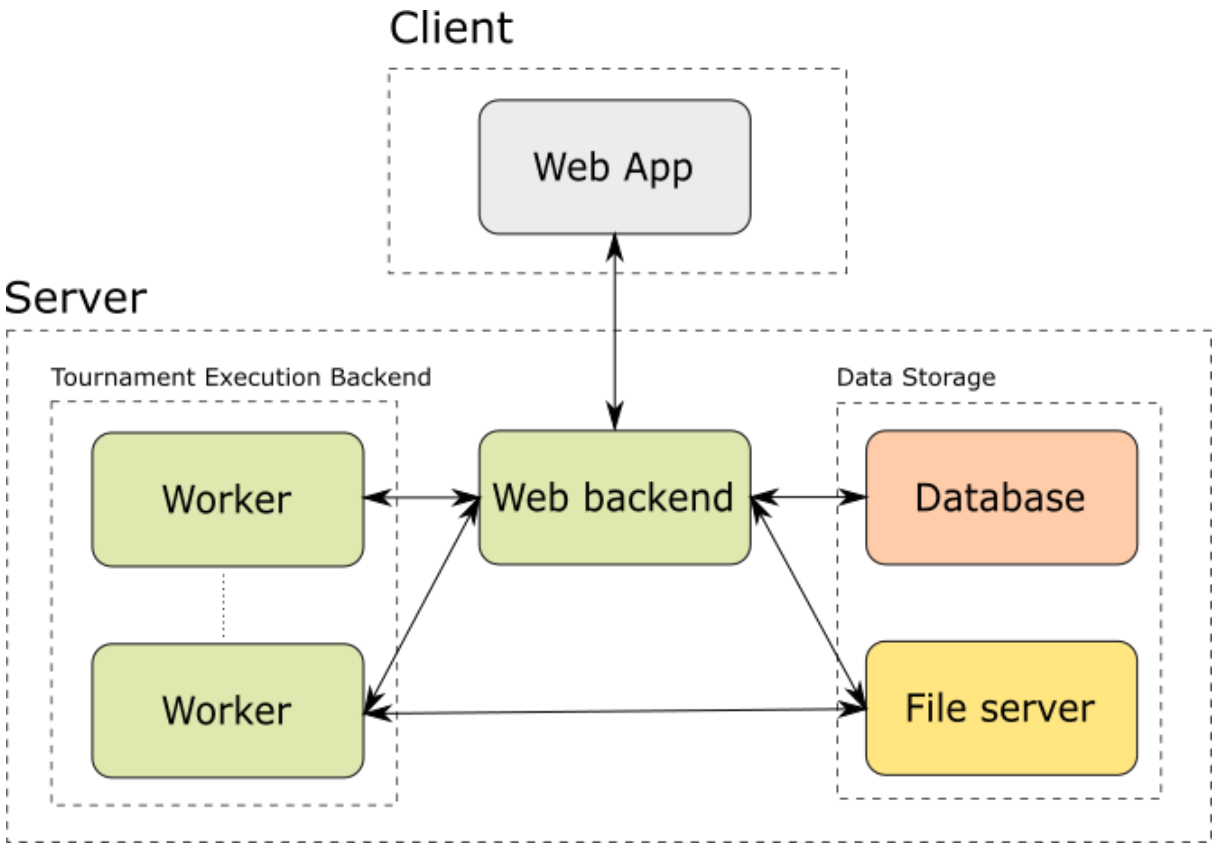
Recovering from crash on worker

1. Worker stops sending heartbeat messages
2. After several missed heartbeats, server considers the Worker unavailable and removes it from the set of available workers
    a. If worker was executing a match, server reschedules it to another worker
    b. If the situation repeats then given match is no longer scheduled, the error is logged appropriately
3. Worker is restarted and reconnects to the server
4. Server adds the worker back to the set of available workers and once again schedules matches for this worker

Analysis

Architecture

The general requirements suggest using a standard three-tier architecture, with javascript frontend app running in a web browser, server backend for serving the data to frontend and database for data persistence. Additionally, the requirement that the tournament execution should be distributed implies that the server backend will contain also a pool of workers for tournament execution. Our platform will also need some sort of a file server to save all the competitor bot submissions, logs, and tournament results. The following figure illustrates the big picture of the platform architecture



Web - backend

Web server will be based on ASP.NET Core technology, version 2.2. We chose this technology because of its simplicity in building such applications - it is easily configurable and contains built-in HTTP pipeline. This technology allows us to build RESTful API, which can then be used by frontend application.

Architecture of this API application will be based on three-tier architecture design pattern, which physically separates presentation, business and data layer. Class on any level can only use the services of classes, which are on the same or lower layer.
We will also use Inversion of Control principle, which is used to create modular and extensible applications by binding of interfaces rather than actual implementations. We will achieve this with the built-in dependency injection library.

Presentation layer is separated into controllers, which will define methods of communication with the web application and all data models to be sent or received. API created by server will be described by Swagger OpenAPI documentation, which is a common way to create documentation of ASP.NET Core-based APIs.

Business layer is divided into services. This layer will be responsible for scheduling of tournaments and for communication with tournament execution backend. This layer will also solve other issues like sending mails, creating replays, etc.

Data layer is represented by repositories. These repositories provide only methods for data querying and modification and do not contain any logic. They are also the only place of knowledge of real database structures. Interfaces of these repositories will provide only basic operations on data entities. We decided to use EntityFrameworkCore as the library for ORM (object-relational mapping), because of its level of abstraction between application and database. Programmer then does not need to have any knowledge of the underlying database, basic knowledge of collections is sufficient.

We would also like to use open source third-party libraries/services, so that we will not need to write every utility from scratch. We certainly want to use some client for sending emails, for this purpose we consider SendGrid as good option, we will also need some tool to create mails from templates, which can be achieved by known tool Handlebars. Another issue, which can be easily solved by external library is mapping between data entities and domain classes, we will use AutoMapper, which provides simple and powerful solution for this problem.

Web - frontend

There are currently two popular ways of writing web applications:

- **Server-side approach** - The actions of the user are processed on a web server that returns HTML responses which are then rendered in the browser. The client itself contains only very little logic. With the backend based on the ASP.NET Core, technology, it is popular to use Razor syntax for generating HTML pages.
- **Client-side approach** - The code (usually written in JavaScript) runs in the browser and communicates with the API server to display appropriate content to the user. There are several popular frameworks and libraries for developing client-side applications, including React, Vue, Angular 2, just to name a few.

We decided to use the client-side approach because the resulting applications are usually more responsive and user-friendly. We decided to use the combination of React and Redux libraries because it is an approach which is currently very popular, well-documented and we already have experience in writing such applications.

Tournament Execution Backend

As to our knowledge, there does not exist a ready-to-use implementation of workload distribution which would satisfy our needs. There is a possibility of using parts of the ReCodEx project developed at our university. ReCodEx is a platform designed to be used in programming courses for evaluating students homework submissions in a sandboxed environment. However, the backend part of ReCodEx is written in C++ and currently does not support multiple communicating processes with individual sandboxing limits (available memory etc.). Only way of reusing the ReCodEx backend is by adapting it’s source code. It would be necessary to make nontrivial adaptation to the runtime and structure of the so called job configuration files. Having parts of the platform implemented in different languages would also add further complexity to the project. This and the fact that not all team members

are versed in C++ led us to the decision to implement our own evaluation backend in pure C#.

Workload Distribution

In order to distribute individual game matches for execution on the workers, we need to implement some kind of load balancer component. Which we will from now on call Broker. The broker will hide the complexity of distributed match execution from the rest of the platform.

The Broker could be implemented in two ways, either as a standalone node with a defined interface which the server backend would use for requests, or as a service integrated inside the server backend itself. The first approach would be beneficial only if we were to have multiple server applications which would independently use the worker pool to execute some tasks. Since our evaluation backend will be quite specialized, it does not make sense to allow other systems to use it. We will therefore integrate the Broker into the server backend to keep the complexity of the project low.

Communication between Broker and Workers

Our implementation of the workers will require some kind of communication with the server backend (more specifically, the broker). There are various possibilities for implementing inter-machine communication available in dotnet, examples include using REST API, messaging queues, remote procedure calls (RPC) and other paradigms. Since the tournament evaluation may take very long time, the communication should be made asynchronous. This rules out using RPCs. Between the other possibilities, we have decided to favor messaging queues for their inherent asynchronicity and lower communication overhead when compared to HTTP requests.

There are multiple messaging libraries available for .NET Core. Many of these libraries come with a message broker and provide features such as message persistency and automatic load-balancing. Since in the future we would like to support heterogenous workers (Linux vs. Windows etc.), we probably would not be able to directly use the load-balancers provided by these enterprise messaging systems. We have therefore decided to use more lightweight NetMQ library, which is native implementation of ZeroMQ messaging library for .NET.

Sandboxing

Because our platform will execute third-party code (the submitted bots), it is necessary to provide sufficient sandboxing capabilities in order to prevent possible attacks. One solution to this would be running the potentially malicious code in a virtual machine. This approach, however, would require implementing virtual machine management in dotnet (by using APIs exposed by virtualization software such as QEMU or VirtualBox), which we find to be too difficult.

Another option would be using some sort of sandboxing tool. To our knowledge, there is no sandboxing tool that would satisfy our needs and would work on both Linux and Windows

operating systems. We have therefore decided to favour only the Linux operating system in our initial release. As for selected tool for process isolation, we will be using Isolate (https://github.com/ioi/isolate), which provides a simple command line interface and internally uses the user namespaces feature of the modern Linux kernel. Isolate sandbox was designed for running code submitted in programming competitions and allows configuring restrictions such as maximum available memory, access to filesystem, access to network interfaces etc., which provides sufficient levels of isolation for our purposes.

Database and File Server

Our application will need to store a lot of data, ranging from regular/structured data about users, submission, tournaments, etc. to unstructured data like actual source code files (user submissions), log files from matches, replay files etc. We have therefore decided to use two types of storage: SQL database for the more structured data, and plain file system storage for the rest.

Since we use Entity Framework as ORM framework, we can use any common SQL database implementation, and because the framework's interface provides sufficient abstraction, the actual choice of particular SQL database is not important.

As for storing and accessing the unstructured part of stored data, we have decided to reuse file server from the ReCodEx project (https://github.com/ReCodEx/fileserver). This component provides access to stored files via HTTP api which can be used from both web backend and worker machines. However, if this solution becomes unsatisfactory, we might have to implement similar functionality ourselves.

Proof of concept

Super Mario Bros.

As a proof of concept, we decided to implement a game being currently used in the Artificial Intelligence I labs at our university - Super Mario Bros. Super Mario Bros. is a simple open-source arcade game based on the legendary Super Mario franchise. In our labs, we use an open-source Java project edited to the needs of the labs, whose source code is freely available on Github (https://github.com/medovina/MarioAI). Students are asked to implement a basic agent controlling Mario through a few basic levels.

There is also a tournament being held, which uses a slightly more advanced level. All the bots compete separately in the level, and they are ordered in the tournament first by their success rate and then by the time spent to finish the levels. We will implement that tournament using our web application and we want to test it in the labs being taught next semester.

If there will be some time left, we will also try to include more of the Artificial Intelligence I labs tournaments on our website, since the automatic evaluation of them would be much

easier for the teachers. As we also have an easy access to the code of the games and direct contacts to its authors, these tournaments/games seem to be an ideal proof-of-concept for our website.