Software agents in support of scheduling group training

Giorgi Mamatsashvili[1], Konrad Gancarz[1], Weronika Łajewska[1], Maria Ganzha[2], and Marcin Paprzycki[3]

[1] Faculty of Mathematics and Information Sciences, Warsaw University of Technology, Warsaw, Poland
[2] Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland
[3] Faculty of Management and Technical Sciences, Warsaw Management Academy, Warsaw, Poland

Abstract. Nowadays, being fit is becoming more and more popular. This includes eating habits – with, for instance, companies offering personalized box-diets – as well as exercising. Obviously, one can go to a fitness club alone, but it is much more fun to go together with friends. Here, it should be obvious that reaching an agreement, between two persons, on the time and location of a fitness club, may be relatively easy. However, it becomes more complex with each additional person that would like to join the group. The aim of this contribution is to show how an agent-based application can be used to negotiate schedule of group training sessions. As a matter of fact, it will be shown that, under a limited number of "good-will assumptions", the proposed application can fully eliminate human involvement in the scheduling process, as it can find the best suitable place and time for a training session, considering individual preferences.

Keywords: Software agents · scheduling · individual preferences.

1 Introduction

The idea of scheduling something, as straightforward as a training session, isn't very complicated. Therefore, one may question the need for a "software support" in a seemingly trivial task. As long as one has a training place, time for said physical activity, and a means to arrive there, there isn't much work to be done. However, with fitness becoming so popular, forming groups to exercise together becomes more and more popular as well. Hence, the real problem arises when the number of potential participants increases. Finding an appropriate time and place to train becomes exponentially difficult with every new member interested in joining the group. This can be the case, due to multiple reasons. In modern days, most people have multiple things, "to which they must attend". Moreover, these events are, often, daily occurrences. From educational institutes to jobs, whether it's a typical nine to five job, or something with a more unorthodox schedule, it is highly unlikely that ones personal schedule matches well with schedules of friends, with whom (s)he may want to work out. This does not even include situations that occur unexpectedly.

This trend has contributed to the rise in popularity of scheduling applications such as *Doodle* or *Appointy*. To schedule an event "is first and foremost about finding a date and/or time. Doodle supports this by recognizing times and time spans, when creating a poll and by calendar import/export features, time zones, etc. when participating in one" [1]. Although Doodle allows users to vote on presented options, it does not account for preferences each user might have, nor does it have the important information about the user, or the event. Instead, it leaves the user no choice but to do the research her(him)self. Appointy ([2]), is a more business-oriented application, which allows users to schedule and to accept appointments with ease. It also offers many other services, such as sending reminders through email or SMS and its integration with social media makes it a very flexible tool. However, similarly to Doodle, it is not designed for unpredictable scenarios, and it does not have the "intelligence" to carry out the decision for the user.

Here, let us note, that people have different preferences, e.g. time of the day when they would like to exercise, or whether the place should carry a late-evening football broadcasts. Attempting at satisfying such preferences can lead to direct conflicts. The aforementioned applications, and many similar ones, try to resolve these problems by having a vote – where individual preferences can be "represented through casting a vote". Hence, they try to reach consensus by using a "democratic approach". However, they require that users actively take place in "negotiations" (at least, by actually placing a vote). Here, let us note that daily schedules are more chaotic than well-structured (unlike meetings in a business environment), which can make involvement of democracy somewhat tedious and inefficient, when the number of preferences to be taken into account increases. The question thus arises – is it possible to reduce, if not eliminate, direct human participation in scheduling fitness session for a group of friends? The aim of our work was to show that this can be done (at least to some extent). Specifically, with enough preferences given, the proposed approach is expected to arrive at a conclusion that can be interpreted as a *compromise* between the preferences of various people without leaving anyone *too upset*.

Let us now assume that the main use case that we are interested in, is as follows. A group of "busy friends" likes to exercise together. As many people today, they live their lives "according to a calendaring application". In other words, we assume that, outside of extraordinary circumstances, all of their upcoming appointments / meetings are stored in such an application. However, we do not assume that the same calendaring application has to be used by each of them. Separately, we assume that they agree to trust the fitness scheduling application (that we describe in what follows). In other words, they let it facilitate group negotiations and specify place and time of an upcoming fitness session. More specifically, let us imagine that a group of three friends decides to have a training session together. They all work from Monday to Friday until 4 p.m., and one of them has a boxing session every Tuesday evening. Then, the program will try to book a session sometime after 4 p.m. on Monday, Wednesday, Thursday or Friday as, obviously, one of the members will not be able to join on Tuesday, due to the boxing class. Here, the exact time is to be decided by finding one matching preferred training time of all users. Moreover, if the application fails to find a suitable time during the week, in a more trivial case, it will attempt to book a training session during the weekend when the users, most-likely, have more free time.

Because of the nature of the problem, *software agents* seem to be the preferable resolution. Here, note that, according to classical references (e.g. [3, 4]) software agents are very good in negotiating and re-negotiating contracts / schedules / agreements / etc. Furthermore, they can act independently [5], while representing their "owners". Our work is meant to illustrate, in the specific context, usability of software agents in finding consensus, while solving a, relatively simple, scheduling problem. Here, agents representing users need to be provided with the needed information (representing user preferences), and then they will communicate to establish the "consensus representing place and time". Furthermore, it is assumed that if any of the preferences change, agents will adapt to the situation, and renegotiate the schedule, to reach the desirable outcome.

To this effect we proceed as follows. In Section 2, we summarize some related work. We follow, in Section 3 with outline of the proposed approach. Next, in Section 4 we show how the implemented application works to solve the, above outlined, problem.

2 Related work

Agents are very versatile and can have many different applications. They can be trusted with different tasks, if they're configured properly and have enough information, based on which they can represent interests of the user. If an agent has enough information, it can be expected to make a decision that is very similar to the one that would be made by the person it represents. Agents, of course, don't necessarily have to represent people; given enough data, they will make choices that are appropriate in the given scenario [6]. Due to this fact, agents are often tasked with decision-making [7]. In this context, due to their independence and adaptiveness, they are capable of helping with the scheduling and rescheduling of certain events.

It is not an uncommon practice to use agents for scheduling [8]. Agents are expected not only to formulate the original schedule, but also to adapt to the changes, depending on what information is available to them, reevaluate the situation, and adjust the schedule based on the newly obtained data. For example, system mentioned in [9] is used by Taxi companies for real-time vehicle scheduling. Multiagent system, developed by Magenta Corporation for Addison Lee, is one of many examples of this specific type of application for agent systems, in the real world. As stated in the document, with the advancements of the modern age, transportation networks have grown exponentially. The growth was accompanied by its difficulties, new problems that required new solutions. Industry giants such as DHL, UPS, TNT, DPD, and many others, have very large and complex networks, which the enterprise resource planning systems that are in use cannot handle well. This is particularly the case when response is needed to dynamically changing situations. The article claims, that the systems have failed to be very efficient due to the fact, that they've failed to keep up with the ever-evolving technological world and stuck to their older technologies. Solutions to these problems are outdated and limited. Multiagent systems can provide solutions to the problems that may arise with networks such as these. In the case of taxi companies, an agent can consider an order of a taxi at a specific time with a specific request. For example, a group of 7 people with luggage may not fit in a normal taxi car, and the current city traffic might make it difficult for certain drivers to reach the customer. The handling the order will need to consider all these requirements and match the customer with the most appropriate driver. This can be done by comparing what information the agent holds, concerning both the customer and the driver (and his vehicle). The solution to the problem is very practical as it can help save many resources for the company as well as it is scalable, which can make the concept future-proof in the commercial world.

As mentioned before, the multi-agent system solutions appear in multiple contexts of scheduling. For instance, HOLOS multi-agent scheduling system is discussed in [8]. HOLOS was designed specifically for manufacturing enterprises. The system works by configuring a group of agents for a specific shop floor. These agents are tasked with exchanging information about the relative production orders; this information is used by the system to generate schedules. What is interesting about this technology, is that the scheduling system is derived from HOLOS-GA, is not fully automated, but rather semi-automated. Due to many challenges in the field of manufacturing, the system is interactive, it is implemented to assist the expert by automating most of the steps of the HOLOS methodology used by the system. This allows enterprises to tailor the system to their specific needs. Although, the system does not aim to eliminate the human involvement in the scheduling process, it still does a great deal of minimizing the amount of work an expert needs to do, which in turn can be both time and resource efficient.

In [10], problems that manufacturing enterprises have faced as the world moves, more and more, towards a global economy are discussed. Due to the high competition in the field, for the enterprises to survive, they have to be flexible and agile. Similar to what was described in [9], many existing systems are outdated and can no longer be considered efficient. They lack many features a modern market requires such as flexibility and re-configurability. Although the systems are explicitly build to optimize production, they fail to respond to change. For that exact reason, the author suggests usage of software agents in the world of manufacturing. Agents are completely autonomous and can be tasked with dealing with the ever changing demands of the industry [11]. Their intelligence, paired with independence, can create an system that is resource-efficient. The author shows an example of an architecture that looks as follows. The agent tasked with supervision of the process, queries for an available agents that represent factory resources. The supervisor agent looks for an agent who possesses the skill that is required to complete the task. Here, each and every resource agent has to verify its skills and answer whether or not they're capable of performing the task. In the specific example given in the paper, we have agent #1 refusing to do the task as it is out of service, and agent #2 being unable to accept the task as it is overloaded. The task, then, is given to agent #3 who needs to negotiate process of transportation with the agent that is tasked with transportation. In this paper, we can see an example of an multi-agent based infrastructure which schedules and negotiates tasks in a business environment. This system can help with the full automation of the tasks and doesn't require any human-involvement.

All these papers, have not only confirmed correctness of our decision to use software agents, but also provided valuable insights to the system that we have undertaken to develop.

3 Proposed approach

Taking into account what has been discussed thus far, let us now outline the proposed approach. We will start form the requirements for the considered application.

3.1 Top-level requirements and architecture

As indicated above, we propose an agent-based architecture, in which each user will have a *personal agent* representing her/his interests. Moreover, each user will have all of her/his upcoming appointments / meetings / commitments stored in a *calendaring application*. In this way the proposed system will be able to know when the user is busy, and thus unable to exercise. Since we plan to use software agents, each user can easily use a different calendaring application. Each of them will be interfaced with her/his *personal agent*. In this way, while the calendaring applications will differ, *personal agent*s, which belong to the same *agent platform*, will be able to communicate and negotiate meeting time.

When the *system is initialized* (for the first time), after authenticating, users will be presented with a *user interface* where they will be able to specify their preferences. The *interface* will be split into two main sections, concerning *time* and the *place* (the *gym*). In each of these sections, user will be able to specify her/his ideal training preferences, so that the program will attempt to schedule a training session that isn't too different from them. The first section, concerning time, will ask for preferred training hours and duration of the session. The second section will focus on the gym itself. The user will have to specify what is important in the gym; e.g. what "equipment" is necessary for them for their workout. They will be also able to suggest what they'd want to see in a gym (e.g. TV's). Each agent will store this information and use it in the negotiations. This information will be stored within the application, for future use, and will be editable (in case if the preferences change). For a depiction of the interface, see figure 1.

In the system, one of *personal agent*s will be acting as *group leader* agent. *Group leader* is the *personal agent* that starts negotiations, and knows which other *personal agent*s are to be involved in them. It represents a person who is "organizing the training session". This person (her/his agent) is the one that starts the "scheduling process". The role of *group leader* can be assumed by any *personal agent*.

Negotiations are to be orchestrated by a separate agent. We will refer to it as the *central agent*. The negotiation process starts when the *group leader* agent instance is created by one of the users. This user also states who else should be in the group that will go out. This will result in instantiation of *personal agent*s representing each user. The *personal agent*s, after their creation, send, as a message, "their" preferences (retrieved from the preference repository) to the *central agent*. The *central agent* receives preferences from all *personal agent*s representing users that are expected to exercise together. It also knows, from the *group leader*, how many / which *personal agent*s are to send such preferences. The *central agent* waits for a limited time (which is a parameter of the system) for preferences to arrive. In case when some agents do not send preferences (e.g. when they are placed within mobile devices that are turned off / not connected to the Internet) in time, they are eliminated from the pool and will not take part in scheduling.

Afterwards, the *central agent* will make decision, based on received preferences and sends messages back to the *personal agents*s so that they can add the event to calendars of their users. The decision the *central agent* makes, is to represent a compromise between what all the users want. The decision making process is described in some detail in sections 3.2.2 and 3.2.1.

3.2 Technical aspects

Let us now briefly describe key technical aspects of the developed application. Let us start from the way of dealing with time and activities stored in a calendaring application. Here, the schedule of the user has to be extracted from her/his calendar. As mentioned, thanks to use of agent infrastructure, we can "hide" calendaring applications behind *personal agent*s, which have to "know" how to interface with their calendars. However, for the initial prototype we have selected the Google Calendar. Hence, we access user data using the Google Calendar API [12]. Note that, in case of other calendaring applications, we would use their respective interfaces. For instance One Calendar [13], Fantastical 2 [14], Lightning [15] (and many others) provide interfaces that allow easy integration with external software. In our program, we use the following method, which facilitates access all the user-events, in a specified time period (period that is pertinent to scheduling joint exercise).

```
public List<Event> getEventsBetween(
            DateTime from, DateTime to, String userId)
            throws IOException {
        CalendarConsumer consumer = (entry, calendar, results)
        -> {Events events = calendar.events()
                .list(entry.getId())
                .setDefaultBetweenCriteria(from, to).execute();
            results.addAll(events.getItems());
        };
        return abstractGet(consumer, userId);
    }
```

Obviously, data can be read only after the user gives the program permission to access her/his calendar data. This method utilizes Google Calendar API to fetch the events from the user's calendar. More specifically, we are explicitly interested in the upcoming week, when the program will attempts at scheduling a session and needs to know exactly what events the user might have within this time frame. We have limited our time-horizon to only one week as the purpose of the program is to attempt to find free time for a training session in the "near future". As it can be seen in the code snippet, it is possible to fetch events from the user's calendar from any specified time period. Although, the program, in its current form, is interested in the upcoming week only, this method is flexible enough to support different needs and ideas concerning calendar access.

Once the application has access to the events from the calendar, it is time for the agents communicate. In our application, we have selected JADE (Java Agent Development Framework), developed by Telecom Italia. JADE is considered to be one of the most advanced open source agent frameworks available. Apart from agent abstraction, JADE provides powerful task execution, peer to peer agent communication via asynchronous messages and many other features [16]. With the help of JADE, we start by creating the *central agent*, tasked with collecting users preferences. The *central agent* is the first agent created during the scheduling process. To create the agent, a *createNewAgent* method, found in the *AgentController* interface is called. When the agent has started, its state transitions from "INITIATED" to "ACTIVE", and the controller is returned. This method is used to create every agent in the application. For every user, a *personal agent* is created. When one of the users is ready to start forming a group, its agent becomes a *group leader* and invites selected *personal agent*s to participate in training scheduling. It also informs the *central agent*, which agents are in the group that is to be formed.

While this may seem somewhat strange, as one could suggest that the *group leader* should be the one to play the role of *central agent*, this approach was not selected. Here, this would mean that each *personal agent* would have to have code for schedule negotiations. Furthermore, each one of then could have had "access" to other persons calendars (via messages that it would receive from the other *personal agent*s). The latter is a very dubious choice (from the privacy perspective); even if the access would be only to a limited data-set. On the other hand, running a *central agent* in a secure and trusted environment and assuring users that their data will be deleted after being used, is more palatable.

Here, we also have to distinguish between the current application setup, applied in the initial prototype, and the way that the program would have worked in real-life mobile scenario. Currently, all agents are created within a computer that emulates multi-user application. This was done to test the basic logic of preference representation, calendaring application access and, finally, the process of scheduling. In the case when the application was to run in a mobile world, agents would have been instantiated on smart phones (mobile devices, in general). This can be done, as in the most recent version of JADE it is possible to instantiate separate platforms on mobile devices and facilitate their communication. Hence, while currently running only a limited emulation, choice of agent platform assures that real-world deployment is possible.

*Personal agent*s provide the *central agent* with preferences of users they represent. To store user preferences, we use JpaRespository [17]. With the help of H2 database [**?**], we can persist the data in a local file stored on the machine (in the case of a mobile application, each agent would store data in a local instance of the database). Pertinent user preferences (we assume that there may be user preferences that are not related to a given scheduling session) are extracted form the database, and turned into string (by software called by the *personal agent*). If the agent fails to start for whatever reason, the error is logged. Next an ACL REQUEST message (see, [18]) is sent to the *central agent*. This message contains user preferences (in JSON format).

When initiated, the *central agent* starts and is continuously, through CyclicBehaviour, waiting for messages from *personal agent*s. For each REQUEST message, it tries to extract its content (with errors being logged). Once all expected messages are received (or the time-out happens), preferences extracted form JSON content are stored as objects and the scheduling algorithm is called. Let us now consider, in some detail, the way that user preferences are taken into account.

3.2.1 Gym Preferences Initially, during the selection process, the program tries to select the gym, based on user preferences. All gyms are stored in the application as objects and the information regarding them are stored as fields. Specifically, this is a single list, which is constant. Although this isn't an efficient way of providing the application with information about gyms, this is a temporary solution. For the time being, data is simply mocked, while in the future stages, we could expect different solutions to the problem, such as a database containing all the necessary information about gyms. In the latter case, a separate interface to add / remove / modify gym information (manually or by extracting information from the Web) would have to be provided.

During the gym selection process, based on user preferences, which are represented in form of weighs, gyms are scored depending on what they have to offer. There's a minimum amount of points that is required to score by a given gym, for it to be considered further. This is a system parameter and it allows to reduce the number of gyms that are going to be taken into account. Currently, if any of the gyms scores below two points for any user, it will no longer be considered by the algorithm regardless of how high it's score by other users might be. Currently available preferences can be seen on the right side in figure 1. Each preference is worth a specific number of points depending on their importance (again, a system parameter that can be adjusted). In the current version, the user is unable to express the importance of each preference and, instead, they are hardcoded in the program (obviously, this is only a temporary limitation to simplify the prototype). To determine the score of the gym, the program checks whether or not the gym can offer what the user is looking for (expressed in preferences). For every preference the gym can satisfy, the number of points that this preference was worth is added to the gym score, which is initially zero. Eventually, all the individual scored are summed up and the gym with the highest overall score is chosen. Currently, all the gyms. along with the information regarding them, are also hardcoded in the program. This is subject to change as the program evolves.

3.2.2 Time Preferences Afterwards, the algorithm for the time selection begins. Firstly, all available times, during the upcoming week, are extracted from the calendar. After acquiring such data, the algorithm starts working to find the shared free time. For each user, the algorithm focuses on the preferred starting time of the workout and the preferred ending time of the workout. This is taken from the user inputs "Workout start lower bound" and "Workout start upper bound" that can be seen in figure 1.

The algorithm then tries to choose a time for the workout from the shared preferred hours it just obtained, and if the chosen time happens to be available in all users' schedules, the session can be scheduled. In other words, the program will only attempt to schedule the session in the shared preferred time. This way, the chance of the session ending up scheduled either too early or too late is eliminated. From this time interval, the program checks different hours and compares it to the calendars of the users. If the program can find a time period where every user is free, then it will select that time for scheduling. In summary, the program ends up booking the session at a time which is relatively close to every user's preference. Hence, it can be claimed that it arrives at a compromise.

Afterwards, the *central agent* sends the message with performative AGREE, meaning it has agreed to do the request, to *personal agent*s and it will send back time of the training session as a content of the message. In the final step, each *personal agent* calls a method, which adds the event to the calendar of the its, respective, user.

4 Experimental verification

We have implemented the initial prototype of the proposed application and tested it on a number of scenarios. As noted, the application has been run on a single computer, to test the main mechanisms (data representation, access, communication, etc.). Furthermore, a number of aspects of preference representation have been hard-coded to simplify the prototype. Here, let us report on two basic use cases. First, what happens when we try to schedule a simple training session and, second, what will happen when a problem occurs.

4.1 Successful training session scheduling

In figure 1, we can see how the preference selection looks like. The preferences concerning time occupy the left half of the interface, while the ones concerning the gym can be seen on the right. The preferences shown in the picture are used as an example and are likely to be adjusted, as the application is being further developed. In the presented scenario, we have two users looking to book a training session (which is much easier to describe in a legible way). However, we have tested the code for multiple users, as well.

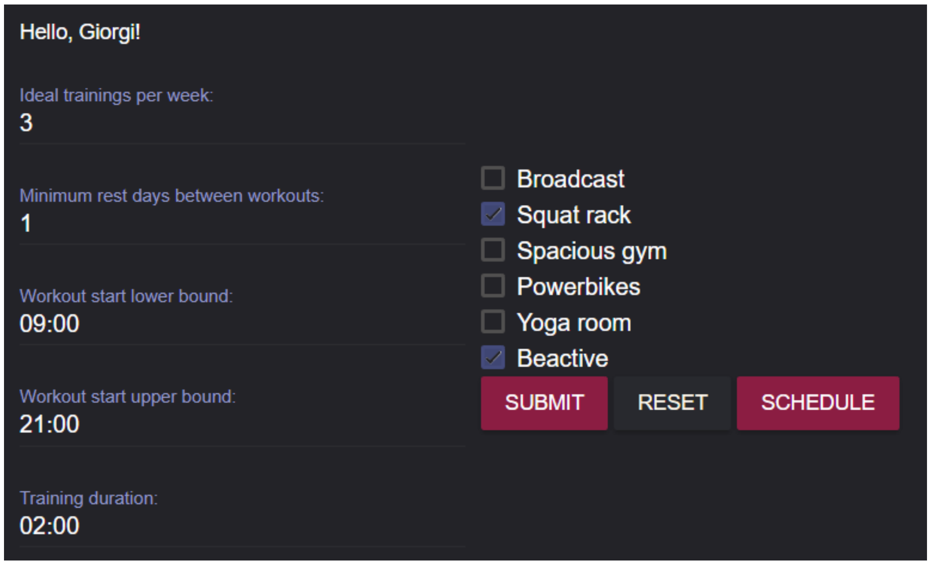
Each user sets up a their preferences for the program. The first user chooses squat rack, yoga room and beactive membership support, as the gym preferences. The second user chooses squat rack and power bikes. All the gyms that are stored (hard-coded) in the application are considered. Here, three gyms that stand out the most and they are “Ancient”, ”Olympus” and “Palace”, since the three of them satisfy more of the user-specified preferences than others. Olympus offers squat rack and a yoga room, and even supports the beactive membership, which leaves it with a score of 6 for the first user, and a score of 3 for the second user, therefore the overall score of the Olympus is 9. Palace, on the other hand, does not have a yoga room, but instead has a room with power bikes, which leaves it with an overall score of 9. Ancient offers everything users specified in their preferences and with the overall score of 10, it is chosen. It is worth noting that upcoming schedules of both users are relatively free, which gives the program a lot of different possibilities, as to when the training session can be scheduled. The program always chooses time period that is the closest to the “current moment”. In this particular case, in figure 2, we can see the upcoming gym event added from 18:00 to 20:00 as both involved users had set 2 hours as a preferred training duration. The fact that the session will be held in the “Ancient” gym is also included in the description of the event.

4.2 Scheduling failure

Recall that the program makes a decision based on what can be found in user’s calendars. If we have a scenario where one or more users’ schedule is full, the program may fail to schedule a training session. Unfortunately, the program cannot take care of such a problem as it tries to consider schedules of all the users, and every time period where there is an event scheduled automatically gets disqualified and will not be used for scheduling the training session. Although a possibility of something like this happening isn’t very high, it becomes more likely as the groups get bigger.

Let us show a specific example and see how the failure is illustrated. In this experiment, we will, once again, have two users attempt to book a training schedule. The first user specifies that the time he’s willing to workout is from 9:00 to 12:00, while the second user’s choice is from 13:00 to 18:00. Since there is absolutely no overlapping between the two preferences, the program fails to schedule an event and the error is logged.

```
2018−10−21 19:32:56.618 ERROR 10896 −−−
```


Hello, Giorgi!

Ideal trainings per week:
3

Minimum rest days between workouts:
1

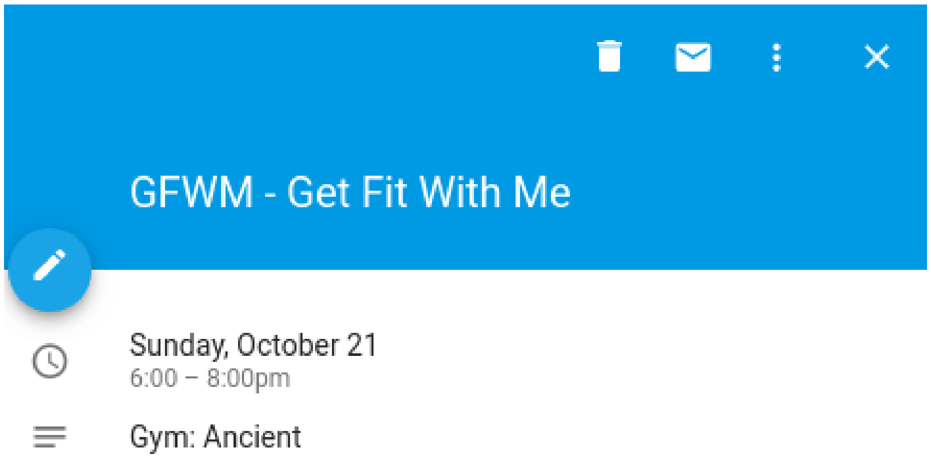
Workout start lower bound:
09:00

Workout start upper bound:
21:00

Training duration:
02:00

- [ ] Broadcast
- [✓] Squat rack
- [ ] Spacious gym
- [ ] Powerbikes
- [ ] Yoga room
- [✓] Beactive

Fig. 1. Example of an user going through their setup


GFWM - Get Fit With Me

Sunday, October 21
6:00 – 8:00pm

Gym: Ancient

Fig. 2. The program is done scheduling the session, and the event is added to the calendars of the users

```
[central] p.g.jacked.services.CalendarService          :
Scheduling Failure - Free time not found
```

The central agent sends back the message of performative FAILURE to individual agents to notify them, that the action failed.

Note that for a group of, let us say, 9 people with busy schedules, it is possible for them not to have an overlapping time period when they all happen to be free in the upcoming week. Moreover, users may not be willing to wait an entire week for the workout. Here a number of scenarios is possible. (1) Program may fail and report this to the users (via their *personal agents*s). In this case, which is the one currently implemented, it is advised that the program is only used by small groups of people, no more than 3 or 4 persons in each group. This way, we can maximize the chance of a desirable outcome. (2) Program may try to eliminate the most “problematic user(s)” and schedule the largest number of them during the closest time. Here, a multicriterial decision making process is needed. In this process number of people that can exercise together will be considered vis-a-vis other criteria. We plan to investigate this approach in the near future.

Overall, it should also be noted that the decision carried out by the program isn’t perfect. The application considers preferences of all users, which might cause it to make a decision that is more likable to some than others. This will depend entirely on the algorithm in place.

5 Concluding remarks

In this work we have considered use of software agent infrastructure to schedule group exercises in a gym. Based on requirements analysis we have implemented an initial prototype and tested it in a number of scenarios. The developed application schedules a training session based on the needs and availability of the user. If the user is willing to trust the decision-making process of the program, then the human involvement can be minimized, when trying to find time for joint activities.

In the text, above, we have indicated a number of immediate shortcomings of the developed prototype. These will be dealt with, first. Furthermore, in order to achieve better quality of scheduling, it is planned to perform an additional research concerning what is available in gyms and what can / should be represented in user preferences. Moreover, possibilities of, for instance: (1) migrating the application to mobile devices, (2) integrating with personal assistants like Alexa, Cortana or Google Assistant, (3) avoiding scheduling a session at a specific time period due to blacklisted events (see, [19]), (4) use of ontologies to represent gyms and user preferences, and semantic technologies, in general, (5) inclusion of diet control, are going to be considered to develop a complete fitness assistant. We will report on our progress in subsequent publications.

References

1. Doodle. `https://help.doodle.com/customer/portal/articles/645363`. Accessed: 2018-09-24.
2. Appointy. `https://www.appointy.com`. Accessed: 2018-09-24.
3. Alex L. G. Hayzelden and John Bigham. *Software Agents for Future Communication Systems*. Springer-Verlag Berlin Heidelberg, 1999.
4. Michael N Huhns and Munindar P Singh. *Readings in Agents*. Morgan Kaufmann, 1998.
5. Frank L. Lewis, Hongwei Zhang, Kristian Hengster-Movric, and Abhijit Das. *Cooperative Control of Multi-Agent Systems: Optimal and Adaptive Design Approaches*. Springer-Verlag London, 2014.
6. Kornelije Rabuzin, Mirko Malekovic, and Miroslav Baca. A survey of the properties of agents, Dec 2005.
7. Yanqing Duan, Vincent Koon Ong, Mark Xu, and Brian Mathews. Supporting decision making process with "ideal" software agents – what do business executives want? *Expert Systems with Applications*, 39(5):5534 – 5547, 2012.
8. R.J. Rabelo, L.M. Camarinha-Matos, and H. Afsarmanesh. Multi-agent-based agile scheduling. *Robotics and Autonomous Systems*, 1999.
9. Andrey Glaschenko, Anton Ivaschenko, George Rzevski, and Petr Skobelev. Multi-agent real time scheduling system for taxi companies. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 29–36, May 2009.
10. Virendra Kumar Verma. Multi-agent based scheduling in manufacturing system. In *National Conference on Futuristic Approaches in Civil & Mechanical Engineering*, March 2015.
11. Paulo Leitao and Stamatis Karnouskos. *Industrial Agents: Emerging Applications of Software Agents in Industry*. Elsevier, 2015.
12. Google Calendar API. `https://developers.google.com/calendar/`. Accessed: 2018-09-02.
13. OneCalendar. `https://www.onecalendar.nl/onecalendar/overview`. Accessed: 2018-10-29.
14. Fantastical 2. `https://flexibits.com/fantastical`. Accessed: 2018-10-29.
15. Lightning Calendar. `https://www.thunderbird.net/en-US/calendar/`. Accessed: 2018-10-29.
16. JADE: Java agent development framework. `http://jade.tilab.com/`. Accessed: 2018-09-05.
17. JPA repository. `https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html`. Accessed: 2018-11-05.
18. FIPA Peformatives. `http://jmvidal.cse.sc.edu/talks/agentcommunication/performatives.html?style=White`. Accessed: 2018-09-05.
19. Wordnet Database. `https://wordnet.princeton.edu/`. Accessed: 2018-09-24.