Kronos: Calendar Management System

Ricardo André Vicente Costa Laranjeiro
ricardo.laranjeiro@tecnico.ulisboa.pt

MSc in Information Systems and Computer Engineering
Instituto Superior Técnico, Lisbon, Portugal

Abstract

Members of the Instituto Superior Técnico community often find it difficult to schedule activities, such as meetings, workshops, events, evaluations and classes, since they lack the means to easily obtain the information necessary for finding the best dates. However, most of that information already exists or can be inferred from the data available in the Fénix system.

Therefore, we present Kronos: a platform that allows users registered in Fénix to estimate and visualize the joint availability of a set of people with whom they intend to schedule an activity. Users can indicate their personal activities, besides the ones imported from Fénix, to be considered in the computation of their availability. The activities contemplated by Kronos can have a fixed schedule (e.g. tests) or not (e.g. projects).

From the tests we conducted, we concluded that Kronos is able to deliver all necessary information in a short time, proportional to the time span and number of schedules being crossed.

Keywords: Activity scheduling, availability estimation, diffuse occupation, workload distribution, schedule crossing.

1. Introduction

A problem often faced by Instituto Superior Técnico (IST) students, teachers and faculty staff is the difficulty in scheduling meetings and events, specially when they are targeted at a large groups of persons. Consequently, the overload of evaluations, project deliveries, events and workshops during some periods of the semester results in unwanted situations such as overlapped evaluations, unfinished projects and unattended workshops and events.

It is essential for student organizations to schedule their activities to dates and times when their target students are available to attend, and for teachers to find the best schedules for lessons and evaluations according to the courses their students are enrolled at. The task of scheduling a meeting could also be less time consuming than it currently is, as a common procedure to schedule a meeting usually consists in multiple e-mail exchanges.

The root of the problem seems to be the lack of enough information to make better scheduling decisions. Examples of such information would be the average availability of the students of a course over the semester, the impact of each project’s workload on students’ availability or the impact of evaluations on the days preceding them.

Almost all of the students’ and teachers’ academic schedules are already stored in the Fénix system. However, Fénix does not provide a feature to cross the schedules of multiple persons to find common available periods. It also does not provide the means to view the impact of activities with an undefined schedule, such as projects. From this arises the need for a system that can gather and process all the necessary data to provide users with the information they need to overcome their scheduling difficulties within a reasonable time.

Our goal is to address the problem by creating a platform for scheduling an activity considering the availability of its intended participants. This platform should make it possible to collect the workload of the intended participants, compute their availability and display it, taking into consideration not only the time spent in activities with a well-defined schedule, such as classes, but also the time spent in long duration activities, such as course projects or study sessions.

In order to reach these goals, we developed a platform that collects information from Fénix, for scheduling activities with one or multiple potential participants.

2. Related Work

2.1. Existing Scheduling Systems

We started by searching existing scheduling solutions that could potentially be used to solve our problem. The Web Appointment Scheduling System (WASS) [11] is a system for scheduling meetings and similar activities over a web interface, where a user may create a calendar in which other users may specify the periods when they are available and are otherwise assumed to be busy on all the remaining time. It also allows to use an academic calendar upon which they may indicate recurring available periods. Doodle [7] allows to schedule an event by specifying the possible time periods at which the activity may occur and letting the invited users vote on the periods they prefer. If the poll owner allows it, an invited user may specify “if-need-be” availability, which means the user has limited availability at that period, but is not completely unavailable. Registered users may also store a personal calendar, in which they may view their activities and even accept or reject activity invitations and it also features the MeetMe service, in which a user may make available a schedule with his availabilities so that other users may propose meetings and other activities. Systems such as BookFresh [2] and BookingBug [3] allow the management of schedules for publishers and to deal with groups of subscribers. Google Calendar [13]...
allows users to register personal activities and activities they wish to invite other users to, and allows to define an activity’s periodicity, duration and privacy definitions as well as other details.

Upon review of these systems, it is possible to conclude that, while they all include desirable features, each fails to provide an adequate solution for our problem. For example, it is necessary to intersect multiple schedules in order to get the availability of a group of users, yet only Google Calendar allows this and only for a limited number of users at a time. While WASS may allow to group user schedules, it unifies them instead of intersecting them, which means it will show when at least one member of a group of users is available, instead of when all or most are available. Systems like BookFresh and BookingBug do not make it possible for publishers to know the availabilities of potential subscribers beforehand, as they may only view the activities they published and who subscribed to them. A limitation common to all the systems mentioned is that they provide no means to consider diffuse activities due to the fact that, except for Doodle, all these systems provide binary scheduling, in which a user either is fully available or not available at all and Doodle’s “if-needed-be” feature does not offer the granularity needed to handle multiple diffuse activities and uneven workloads.

2.2. External Data Sources: Fénix
Fénix is the main information management system used at IST, as well as in other universities. It holds information about almost everything in IST, from staff management to student enrolments. On the matter of schedules, Fénix holds vast information, like scheduled classes, projects, the courses those classes and projects belong to, which students are enrolled to which courses and project groups, which teachers will lecture a given class, which degrees have a given course, which departments are responsible for a degree, etc. In short, and as previously mentioned, Fénix contains most of the data needed to solve the problem at hand.

In order to collect this data from Fénix, one may use the FénixEDU Representational State Transfer (REST) Application Programming Interface (API) [6] or, alternatively, the necessary data from Fénix can be obtained by a direct access to the system’s domain. However, due to the security risks, giving direct access to a prototype in development is not a reasonable approach, but for development and testing purposes a file containing all the necessary data may be used.

2.3. Underlying technology
To develop our solution we used the Fénix Framework [5], which is an open-source Object-Relational Mapping (ORM) framework developed at IST, that provides a transactional and persistent domain model, specified with its Domain Modelling Language (DML), from which it generates all domain objects and relationships. For our prototype we used the Spring Framework [17] for deploying the server applications and implementing the web REST API. We chose AngularJS [12] to develop the client application and Apache JMeter [4] for testing the performance of our solution, as it allows testing the server’s performance and efficiency by simulating the web client application, with several simultaneous instances for simulating concurrent users.

3. Kronos: scheduling activities considering its participants’ availability
3.1. Requirements and use cases
In order for the system to work according to the users’ needs it is necessary to take into account a number of requirements. To facilitate the requirements’ elicitation, students, teachers and faculty staff were asked to fill a questionnaire where they indicated how important they consider a number of features. The questionnaire was answered by 443 students, 9 teachers and 2 faculty staff members. After comparing the results between students and teachers, we concluded the most wanted functional features are viewing the availability of a group of persons, conflict detection and notifications. As for privacy features, they were all considered highly important by both groups, although teachers considered choosing who can view their activities to be of the highest importance, where students gave all privacy features a similar level of importance.

From the problem described and the questionnaire results, it is possible to derive the following functional requirements:

- Create activities with a specified fixed schedule;
- Create activities with a flexible schedule within a specified time interval;
- Select the intended participants, individually or entire groups;
- Choose which of the selected participants are essential for the activity to take place;
- Select the time interval in which the activity might take place;
- Display the availabilities of the selected participants within the selected time interval;
- Send invitations for the invited persons;
- Upon acceptance of an invitation, consider the activity for future availability computations;
- Allow the creation of personal activities with the creator as the sole participant;
- Allow deletion and updating of user activities created via the system interface;
- Increase, decrease or disable the impact of diffuse activities when viewing availabilities;
- View activities with conflicting schedules.

Data from Fénix may be public or private and all security requirements they had originally in the Fénix system must be preserved. Thus, the confidentiality requirements are as follows:

- Access to non-public data must be authenticated and secure;
- Access restrictions to collected data must be greater or equal to its restrictions on the source;
- Access restrictions to inferred data must be greater or equal to its restrictions on the source, because this data may be used to guess the source data;
- Access restrictions to user data should be explicit to its owner;
• Access permissions to another user’s personal data should either be settable by the data owner or restricted by default.

The system may be set up in different environments and the other systems it interacts with may change. Therefore, the following modifiability requirements apply:

• Modules that collect the source data from other systems should be easily replaceable;
• System functionality should be exposed by a service layer, both for user interfaces and external systems;
• It should be possible to replace the inference module without requiring major changes to other parts of the system;
• It should be easy to switch between different implementations of the workload estimation algorithms.

The user interface of the client application is not the main focus of this project, so the user interface requirements are not covered. However, in order to ensure the core of the system does not cripple the usability of the system as a whole, these usability requirements are considered:

• For all operations, even operations that do not return any result, the system must provide feedback on whether an operation was successful or not;
• The system should allow clients for both desktop and mobile devices.

In order to be a solution for the problem, the system must perform all requests within a reasonable time. These are the system’s performance requirements:

• In normal load and networking conditions, after the request is submitted, the system must provide immediate feedback that it is processing it;
• The time necessary to receive the results of a query should be proportional to the number and size of the schedules requested.

The system employs the following use cases:

• Login: The user is authenticated in the system, enabling access to the system’s features;
• View availabilities: The user specifies the type and duration of the desired new activity and selects its intended participants, indicating those who are to be considered essential. When the user configures the search conditions, the system shall return a calendar with the estimated availabilities of the selected participants, marking as unavailable any periods that do not comply with the specified conditions;
• Create activity: When viewing the intended participants’ availabilities, the user may choose a valid period on which to schedule the activity. Having chosen the period, the user will then have the possibility of filling out the activity’s details, such as its name. Finally, the user may confirm the activity, at which point an invitation will be sent to the selected participants;
• Accept invitation: An invited user may accept an invitation, which will then be registered in his schedule as a participation;
• Cancel activity: The owner of an activity may cancel it at any moment. All participants should be notified and their respective participation removed from their schedules;
• View activities: The user may view all activities that he owns or in which he has an invitation or participation;
• Set an activity as private: When creating an activity, the user may mark it as a personal activity, in which he will be the sole participant and no other user will have access to its details, even though it will still be considered for the availability estimation.

3.2. Domain Model

While the Kronos domain model was highly influenced by the Fénix domain model [4], the system needs to easily adapt to any organizational changes and usage scenarios, such as including students groups or using internal department structures. For this reason, the Accountability pattern [5] was used to model relationships between parties and the types of activities and participations, allowing it to easily represent almost any organization.

The Domain Model is composed of the following entities:

• Party: Represents each entity in the system and may either be a Person or an Organization.
  – Person: Represents each user in the system.
  – Organization: Represents organizational entities, such as a department or a degree.
• Association: Represents a parent-child relationship between two Party instances, most often having an Organization as the parent member.
• AssociationType: The role of the child in an Association, such as “Coordinator” or “Student”.
• Activity: Represents each activity in the system. Each Activity has assigned a set of Period instances and a set of Participation instances. Each Activity also has a Person as the owner, a Person as the creator and a Party as the host.
  – SimpleActivity: Activities with a fixed, well defined schedule, such as classes or meetings, fully occupying its assigned Periods, that represent the Simple Activity’s duration.
  – DiffuseActivity: Activities with a duration but without a defined schedule, such as projects, affecting availability at periods not occupied with any SimpleActivity. Diffuse Activities have a workload, representing the number of Periods necessary for completing them, that is to be distributed by its Periods, which represent their time span.
• ActivityType: The type of an Activity, such as “Test” or “Class”.
• Period: The basic time unit in the system’s calendar, corresponding to a block of time.
**Participation**: Represents each Person’s participation in an Activity and may either be an Invitation or a ConfirmedParticipation.

- **Invitation**: A Participation that has not yet been confirmed and will not be considered in the user’s schedule. When confirmed it is replaced by a ConfirmedParticipation.
- **ConfirmedParticipation**: A Participation that has been confirmed and will be considered in the user’s schedule.

**ParticipationType**: The role of a Participation’s participating Person, such as “Teacher” or “Student”.

The correlation between Kronos’ and Fenix’s domains is not direct and data from Fenix must be mapped to Kronos’ domain model. Each Person in Kronos corresponds directly to a Person in the Fenix domain. Departments, Degrees and Courses are all mapped to Organizations in Kronos. Lessons, WrittenTests and Exams are mapped to SimpleActivities and Projects correspond to DiffuseActivities. All relationships between objects mapped to subclasses of Party are mapped to Associations, with their roles in Fenix mapped to AssociationTypes in Kronos. Relationships between Persons and objects mapped to Activities are mapped to ConfirmedParticipations, with the Persons’ roles mapped to ParticipationType.

### 3.3. Architecture

Kronos has a layered architecture, composed of a Controllers layer, a Presentation layer, a Processors layer, a Business layer and a Data Access layer.

The Controllers layer is where the REST API controllers are implemented, as well as where the web client application is made available. The REST controllers validate and handle requests from the REST API and serialize response data in JavaScript Object Notation (JSON), exposing Kronos’ services to client applications.

The Presentation layer is responsible for converting Kronos’ data, obtained from the Processors, into structures modeled according to the concepts in which the data will be made available to the users. As an example, if availability is requested and exposed according to days and periods of a calendar, then this layer structures data that must be accessed as if it were indeed stored according to a calendar. This is also the layer where user sessions and authentication are handled.

The Processors layer has the purpose of collecting and processing data from the Business layer to make it available to the Presentation layer. This is where queries are handled, triggering updates on the Knowledge Base when necessary.

The Business layer is split between two major modules: The Domain module and the Knowledge module. It also exposes a request-oriented API that abstracts the underlying calls to both the domain and knowledge modules. The Domain module deals with all domain objects and domain logic. It calls the Data Access layer both for reading and writing all domain data. Domain data is boxed into Data Transfer Object (DTO)s whenever it must be shared with other modules. The Knowledge module is where the availability data is generated. From the domain data, this module infers all user’s schedules according to their activities. The workload estimation algorithms are also in this module. In order to improve performance, this module may keep a cache of the inferred data, thus computing each schedule only once. While it may be cached as it is generated, knowledge data is never persisted, since it depends directly on the domain data and must change according to any changes in domain data.

The system has the ORM and Connector modules in the Data Access layer. The ORM module connects to the system’s database and is responsible for loading and persisting all domain objects as well as handling relationships and domain restrictions. The Connector is the module responsible for collecting all necessary data from source systems, such as Fenix, and mapping it to Kronos’ domain objects. Each Connector implementation acts as a Mapper and is specifically tailored to the system it will collect data from and, in order to make the system adaptable to different external systems, this module has an interface that is intended to allow changing Connector implementations easily.

### 3.4. Implementation

At the core of the system is the Kronos Engine, which is responsible for starting and stopping the entire system. The Kronos Engine possesses references to the Domain Manager, the Knowledge Manager and the Feature Toggler and is responsible for initializing these components and the Kronos Logger when the system starts. It is also responsible for cleaning up these components when shutting down. The Kronos Logger is responsible for writing the system’s log messages to its configured output streams. The Feature Toggler is where the system’s configuration is held and its settings can be changed at any time during execution.

External data is loaded and mapped to the Kronos domain by modular Connectors. Initially, we implemented a Connector to fetch data from the FenixEDU REST API but, due to the FenixEDU API not being able to deliver all the data required by Kronos at the time, a new Connector was implemented to read a JSON file exported directly from the Fenix system, which contained all the necessary data. The Fenix data is loaded only once at the first initialisation of the system, due to the fact that the data does not change. It is also during this initialisation that the Periods are generated. In order to map the loaded data into the Kronos persistent domain, the Connectors are passed with a reference to a Domain Factory, which is responsible for handling the logic behind the creation of new instances of domain elements, along with all necessary relationships.

Domain objects and relationships are persisted by the FenixFramework and are specified in its DML. The domain logic is implemented within each of these domain objects and is called upon by a Domain Manager object, that abstracts all domain functionality and exposes it to other modules. This makes it possible to have an improved control over the Fenix Framework transactions, allowing all transactions to be opened in the faster
their average Period workload \((W_{avg})\) and each Period’s occupation rate \((R(p))\) of a Diffuse Activity there is still overflow caused by dropping pebbles on a pond. It runs over all Periods from the later to the earlier in order to make sure no overflow remains. If at the first Period of a Diffuse Activity there is still overflow caused by that Diffuse Activity, that overflow gets destroyed, even if there were free Periods preceding the start of that Activity.

- **Ghost Distribution:** This algorithm simply destroys all overflow and was used mostly for testing purposes;

- **Wave Distribution:** This algorithm passes all overflow of a Period’s Occupation to the preceding Period’s Occupation. Since only Periods that are already fully occupied will have overflow, this algorithm results in an effect reminiscent of waves caused by dropping pebbles on a pond. It runs over all Periods from the later to the earlier in order to make sure no overflow remains. If at the first Period of a Diffuse Activity there is still overflow caused by that Diffuse Activity, that overflow gets destroyed, even if there were free Periods preceding the start of that Activity.

- **Tsunami Distribution:** This algorithm is based on the Student Syndrome [18] and starts by gathering all the overflow within the time interval of the Diffuse Activity and puts it on its ending Period’s Occupation. Then it applies the Wave Distribution algorithm. We called it Tsunami because instead of generating small and disperse “waves” for each fully occupied Period that occurs within a Diffuse Activity’s time span, it causes a single “wave” with all its concentrated overflow.

The algorithm to be used and the slope in Equation 3 can be chosen by configuring Kronos’ Feature Toggler. By default, Tsunami is used with a slope of 1.
At the end of the update process, the estimated occupations are then inserted into the KB.

Part of the update process was initially implemented using the JBoss Drools Business Rules Management System (BRMS) [14]. Using the Drools Rule Language (DRL) to specify how to traverse the relationships between the many domain objects and how occupations should be applied according to the type of activity, Drools was able to successfully infer the schedules for each person. However, since the update logic would only branch on whether the activity was diffuse or simple, the modifiability benefits granted by the DRL could not outweigh the overhead of using Drool's stateful knowledge session. This led to the Drools implementation being eventually abandoned, but it heavily inspired the interface of the Knowledge Manager.

The KB may be cached or not. If it is cached, the update process is only performed if the requested Occupation is not yet in the cache, sparing the time it would take to perform the update process. When there are changes to a user’s schedule, that user’s Occupations are removed from the cache. If the cache mode is set to Disabled, the occupations are discarded after each query and each requested Person’s schedule will need to have its Occupations recalculated every time. If it is set to Enabled, the occupations are calculated the first time they are requested and will be kept in memory and reused for every subsequent query. If it is set to Initialised, all Occupations for all Persons are calculated and stored in the cache, during KB initialisation.

The query’s parameters are an array containing the unique identifiers of all mandatory Persons, an array containing the unique identifiers of all optional Persons, the minimum number of available Persons required in order for a Period to be considered valid for scheduling, the desired workload in Periods for the intended Activity and the set of periods where the search should occur. Queries are not performed directly on the KB, but through an object inspired by the scrollable cursors used in databases [19]. Query data is retrieved on a Period by Period basis and thus the cursor loads only the data related to the Periods of a single day at a time, avoiding allocating more memory than necessary. This cursor object interacts with both the Domain Manager and the Knowledge Manager to obtain each requested Person’s occupation rate for each requested Period. In order to take full advantage of the scrollable cursor for the users’ queries, whenever a user initiates the query, the cursor is wrapped into a presentation object, representing a calendar, and cached in the user’s session. The cursor remains cached in that user’s session until a new query is performed or the user starts a new session.

Apart from caching each user’s queries, user sessions also implement a basic form of authentication for each users’ requests. When logging into the system, the user receives an identifier, used to identify the user in all requests, and a token, which is the result of hashing a combination of multiple user and system values, generated once at the start of each session. For every subsequent request, the client application must send both the user identifier and the token, which will be matched against that user’s session token, rejecting the request if the tokens do not match.

Client applications access system’s services through a REST API, using JSON to serialize the data. The REST controllers are implemented using the Spring Model View Controller (MVC) Framework and, using the Jackson library, the controllers read the requests’ JSON data and build the JSON responses for the client applications. The HyperText Transfer Protocol (HTTP) method for all endpoints is POST, so all parameters have to be sent in the request body. All endpoints throw exceptions if any of the parameters sent in the request are incorrect or invalid. In cases where no response data is sent, the endpoints return a boolean specifying whether the operation was successful or not.

The Browser Client Application is a single page application developed in JavaScript, HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) and uses AngularJS [12] and Bootstrap [15], ensuring responsiveness and compatibility across different devices and browsers. When the user first accesses the application’s web site, the entire application’s code and files are downloaded into the user’s device. From then onward, the client application runs on the user’s browser and communicates with the server application exclusively through the REST API. The client application has a service module, a main module and an interface module for each of the application’s screens.

The service module handles all requests to the REST API and processes all received response data into structures suitable for the user interface modules. In the service module all requests are asynchronous, but in some cases the interface modules will wait for the data before showing a new screen to the user. The service module also holds data shared between the other modules. The main module of the application selects which interface module is generating the screen visible to the user and also dynamically specifies which module is loaded when the user presses one of the navigation buttons, like the Back and Confirm buttons on the application’s screen. Having the screen sequences being set dynamically instead of being hard coded allows easier reusing of the interface modules for similar screens. Each interface module corresponds to a screen and has a AngularJS controller that will generate the screen from its corresponding block in the HTML file, filled with the application data.

After the user successfully logs into the application, a menu is shown, allowing to initiate the scheduling of a new activity, to view the user’s own schedule or to view the activities that involve the user in some way. When scheduling a new activity, the user must first set the desired settings for the activity and select the intended participants to invite. On the same screen, the user may press a button to open a pop-up window, allowing to navigate through the organizational tree, going from departments, to degrees, to courses and to classes, and select the members of each such organization that are to be invited for the activity. For ease of use, a drop-up button allows selecting, deselecting or hiding from the list multiple persons at once, based on their role in
that organization. After closing the pop-up window, all the selected persons will be displayed in a box below the options and the user will still be able to review and individually remove or set them as essential for the activity. After confirming the activity’s settings and intended participants, the user is taken to a calendar screen displaying each day of the month in a color from a hue gradient ranging from green, for highest availability, to red, for lowest availability. This availability is each day’s average availability of all intended participants’ combined schedules. When the user selects a day, the screen will change from a monthly view to a weekly view, which details the estimated availabilities of each of that week’s days’ periods. By opening the options pop-up, the user may choose to view these availabilities in percentage or in total number of available participants, which accounts for the participants that have an availability above 0%. In the options pop-up, the user may also configure the intensity of the impact of the diffuse activities on the availabilities or even disable it completely. If a person who was set as essential for the activity is unavailable at a given period, it will be shown in red and be invalid for selection even if all remaining participants would be fully available at that period. The same applies for periods that do not meet the required number of available participants. In simple activities, a period is only valid for selection if all other periods that fall within the activity’s duration are also valid. For diffuse activities, there are no such restrictions, as the user simply needs to specify the starting and ending dates and times. Having chosen the activity’s schedule, the user will be given the opportunity to name the activity, review its details and confirm it, which will invite all selected participants to the activity and the user will then be able to keep track of the invitations on the activity’s details in the “My Activities” screen. For viewing an activity the user is responsible for, as well as checking how many participants have accepted their invitations, the user may select the activity in the “Created Activities” tab in the “My Activities” screen, which shows the activities hosted by the user. Besides checking the details and the invitations’ statuses on this tab, the user may also cancel the activity, in which case it will be removed from all the confirmed participants’ schedules, as well as cancel all pending invitations. In the “Initations” tab of the “My Activities” screen, the user may check any pending invitations from other users’ activities and accept or decline them. If the user accepts the invitation, the user will become a confirmed participant for that activity and it will become registered in the user’s schedule. The “Participations” tab contains a list with every activity that is registered in the user’s schedule. The user may select an activity to view its details or may cancel a previously confirmed participation.

4. Evaluation
Most of the functional requirements were fulfilled by the endpoints of the REST API. As for the remainder, in the client application it is possible to increase, decrease or disable the impact of diffuse activities when viewing availabilities and the ability to select the time span for computing the availabilities became unnecessary thanks to the scrollable cursor, as the user can simply navigate through the calendar to search for the desired dates. Regarding the possibility to edit activities created by the user and automatic conflict detection, these were not implemented due to time constraints.

When it comes to confidentiality requirements, it is possible to conclude that: any access to data requires authentication, regardless of whether it is public or not; it is not possible for a user to request another user’s data, apart from their availabilities; it is not possible for a user to infer, from another user’s availability data, which activities are responsible for the occupation in a period. These three factors ensure that Kronos does not break any of the confidentiality requirements.

Regarding the modifiability requirements, they are fulfilled as follows: The Connector is the module responsible for collecting source data from other systems, and Connector implementations are easily interchangeable; The Controllers layer is the service layer responsible for exposing the system’s functionality to all client applications and external systems; Both the workload distribution algorithm and the estimation formula can be configured via Kronos’ Feature Toggler.

As for usability requirements, they are met by the REST API, since all endpoints that do not return any operation data will return a boolean stating whether the operation was concluded successfully and a client application can be developed for any device, as long as it is able to run an application capable of using a REST API over the web.

The aim of this project is to provide a solution to the scheduling problem by providing the requested information to the user within a reasonable time, which may vary according to the number of schedules being calculated. The solution should also have an acceptable deployment cost, which in this case can be measured in Central Processing Unit (CPU) and Random Access Memory (RAM) usage. In order to make sure our solution would be adequate for a realistic scenario, we define the following goals to evaluate our solution, in terms of performance:

- Identify the minimal required capacity in terms of CPU cores and RAM capacity;
- Identify the performance of the system, given a defined processor and memory, in terms of latency and throughput;
- Analyze different architectural solutions in terms of the use of cache and its management.

The system was deployed in a Virtual Box virtual machine with 9GB of RAM and using 4 virtual processors from a host with an Intel i7 CPU with a frequency of 3.6GHz and 4 cores with hyperthreading and hardware virtualization technology. The Java Virtual Machine (JVM) was configured to run with a fixed heap of 8GB and a concurrent mark and sweep Garbage Collector (GC) with 2 threads. In order to measure CPU and RAM usage, the system was profiled using Apache JMeter on the host, connected to a PerfMon Server Agent running on the guest machine. The JMeter test plan was designed to be able to fully simulate the requests as they
would be performed by a client application. In order to simplify tasks not relevant for measuring as well as to allow JMeter to configure Kronos’ settings for each test, a test controller was implemented, extending the REST API with test-specific services. We developed a simple embedded profiler for sampling memory usage and getting accurate execution times, which was used in some tests. All tests consisted on running queries in sequence for an increasing number of selected participants, ranging from 1 up to 10000. Combinations of the following settings were covered by the tests:

- Number of participants: 1, 2, 3, 5, 10, 20, 30, 50, 100, 200, 300, 500, 1000, 2000, 3000, 5000, 10000;
- Cache: initialised, on demand or disabled;
- Time span: A year or a week;
- Number of concurrent clients: 1 or 2.

In the test plan’s setup phase, JMeter starts by requesting the unique identifiers of 10000 persons and the calendar that will used be used in all tests. In order to make sure the queries do not contain trivial cases, Kronos only includes in the list the identifiers of persons that participate in activities. After collecting all necessary data for the test run, JMeter sets Kronos’ cache mode. After the setup phase, JMeter logs into Kronos with each user thread and starts the test run, which consists of initiating a query for each predefined number of participants and requesting availability data for every week within the intended time span. For each query in a test, the set of identifiers is selected incrementally, meaning that, for example, the set of participants for the queries for 20 participants will contain the participants from the queries for 10 participants. JMeter also starts and stops the embedded profiler before and after each query, respectively, in tests where the profiler is used.

Results for the tests are shown in Tables 1 to 3. These tables feature the following information:

- **Cache**: The cache mode;
- **Persons**: The number of schedules requested in each query;
- **Total Time**: The sum of the time taken by all iterations of a set;
- **Average Time**: The average time taken by each iteration of a set;
- **Average RAM Used**: The average RAM in use during each iteration of a set;
- **Average RAM Variation**: The average difference between the RAM in use at the end and the beginning of each iteration of a set;

Our first batch of tests targeted the effects of each cache mode on the first query for each set of Persons. Each query requests the schedule of each person in the set for the time span of a year. As we concluded, the *initialised* cache mode has the best performance for the first query of each set. With *enabled* cache mode, the performance is nearly identical to the *disabled* mode for smaller queries, but the benefits of the cache can be seen in the larger queries, as each query has the schedules of the previous set already cached, which for most queries is about half of the schedules.

We confirmed that memory usage for the *initialised* cache mode remains relatively constant throughout all the queries. When comparing to the other two modes, it is clear that memory usage in *initialised* mode is higher for most sets, as expected, but becomes closer to the other modes at the 3000 set and lower at the 5000 and 10000 sets. We can also see much lower GC activity in the *initialised* mode than in the other two modes. The relatively large RAM variation in these last sets allows to conclude that a very high number of objects are being created at these queries, and the high GC activity indicates this is likely due to these objects being thrown away at the end of each query.

The goal of the next batch of tests was to analyse the impact of each cache mode for multiple repeated queries, as well as the evolution of performance and memory consumption over time. Each query is repeated 50 times for each set, and each set is run sequentially, as in the previous batch. As before, each query requests the schedule of each person in the set for the time span of a year. We could conclude that, as expected, there are no significant differences in performance between the *initialised* and *enabled* cache modes, as only the first query of each set benefits from the initialised cache. Comparing the two cached modes to the *disabled* mode, it is clear that for sets of 20 and higher, the cache leads to a very significant increase in performance, as the time each query takes with cache is proportional to the number of persons, whereas the time taken without cache grows exponentially. Regarding memory usage, the differences to the previous batch of tests only appear significant for the larger sets, which also have a high RAM variation, indicating once more that this is very likely caused by a large amount of temporary objects. It is otherwise expected that memory usage remains the same for each set regardless of the number of repetitions, as each query in a set collects the same data.

The following batch of tests was performed with the intent of determining the amount of time it takes to ob-

---

### Table 1: Results for queries spanning one year with 1 iteration

<table>
<thead>
<tr>
<th>Cache</th>
<th>Persons</th>
<th>Total Time (s)</th>
<th>Average RAM Used (MB)</th>
<th>Average RAM Variation (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialised</strong></td>
<td>1</td>
<td>0.704</td>
<td>3511.600</td>
<td>37.245</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.497</td>
<td>3689.191</td>
<td>55.795</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.989</td>
<td>3763.392</td>
<td>260.221</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>6.457</td>
<td>3696.618</td>
<td>253.595</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>127.081</td>
<td>3762.974</td>
<td>243.028</td>
</tr>
<tr>
<td><strong>Enabled</strong></td>
<td>1</td>
<td>1.177</td>
<td>2721.250</td>
<td>33.989</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.178</td>
<td>2732.005</td>
<td>37.919</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>6.700</td>
<td>2764.319</td>
<td>46.229</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>20.471</td>
<td>2890.329</td>
<td>74.643</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>251.300</td>
<td>6977.249</td>
<td>3371.100</td>
</tr>
<tr>
<td><strong>Disabled</strong></td>
<td>1</td>
<td>2.010</td>
<td>2733.255</td>
<td>33.397</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.020</td>
<td>2743.535</td>
<td>36.069</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>7.455</td>
<td>2797.594</td>
<td>55.367</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>30.447</td>
<td>3018.763</td>
<td>340.850</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>386.582</td>
<td>6977.249</td>
<td>3247.123</td>
</tr>
</tbody>
</table>
tain the availabilities for a single week, which is the time span of the client prototype’s availability requests when fetching the results of a query. Since the client shows the results as soon as it receives them, this effectively represents the time it takes for the user to receive the first results after submitting a query, disregarding network latency and the client application’s performance. In this batch, each query is repeated 50 times for each set and each set is run sequentially, for the time span of a week. From the results of the test, we could conclude that collecting the availabilities of a single week takes a very short amount of time, when the cache is enabled. The difference between RAM usage at the end and the beginning of each query is also very low, due to the relatively small number of objects necessary for a single week. With cache disabled, however, the query takes almost as much time as it takes for an entire year. This is due to the fact that the workload distribution algorithms must take into account all periods, or else the results could differ when querying for a smaller time span, due to overlapping activities outside the selected time span not being accounted for in the calculations. Since the cache makes it possible to run the algorithms only once, the cached queries run much faster and with lower resource usage.

The last batch of tests was performed with the intent of identifying the impact of concurrent queries on CPU and RAM usage, as well as determining the system’s throughput. Comparing the results, it is possible to conclude that CPU usage is somewhat higher when 2 simultaneous queries are being performed. As for memory usage, there is a noticeable increase, which is explained by the higher amount of presentation objects being used. Regarding query times, the concurrent queries appear to have taken slightly more time than in the single-user case, which could be due to the much higher GC activity. When comparing the results of these tests with the similar single-user ones, it appears that the concurrent requests are processed slightly slower and consume memory more quickly. Once more, GC activity appears to be significantly higher when 2 users are performing concurrent requests.

With these results it is possible to conclude that our solution is able to fulfill the performance requirements. Knowing that the server responds right after storing the query’s cursor into the user’s session, we can conclude from the average time it took to retrieve the availabilities of one week for 1 person, that the server is able provide this feedback immediately.

When using cache, the time taken to collect the results is generally proportional to the number of scheduled requests, if the GC activity is not too intense. The initialised cache mode gives the benefits of the cache for the first time each schedule is requested, but has the downside of taking several minutes to calculate all availabilities during system initialisation. Analysing the memory usage from the results, it is evident the system consumes 2.7GB of RAM from start, before even conducting the first query. Most queries can be performed with an additional 1GB of RAM, which would set the system’s minimum RAM requirements at about 4GB. Larger sets, however, seem to require at least 8GB of RAM to be reserved for the JVM Heap. Regarding CPU usage, each concurrent request runs on its own thread, and thus uses only one processor, plus the CPU usage of the GC. This means the minimum number of CPU cores necessary for an acceptable performance depends on the expected load for the system.

5. Conclusions and Future Work

In this work, we contribute with a platform for scheduling activities that makes use of the existing information about the schedules of different members of the academic community at IST, in order to display the availability of the intended participants in the process of creating a new activity. This platform allows to:

- Create activities of two types: simple (with a well-defined schedule) and diffuse (with no schedule specified);
- Select the participants individually or as groups, indicating those who are essential to the activity;
- Compute and display the availability of the selected participants, by inferring their workload from the

<table>
<thead>
<tr>
<th>Cache</th>
<th>Persons</th>
<th>Average Time (s)</th>
<th>Average RAM Used (MB)</th>
<th>Average RAM Variation (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialised</td>
<td>1</td>
<td>0.387</td>
<td>3457.016</td>
<td>36.805</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.362</td>
<td>3458.372</td>
<td>37.333</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.942</td>
<td>3473.772</td>
<td>36.171</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>6.804</td>
<td>3860.401</td>
<td>37.075</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>152.332</td>
<td>5841.983</td>
<td>37.194</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.416</td>
<td>2684.058</td>
<td>36.766</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.371</td>
<td>2690.535</td>
<td>37.302</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.968</td>
<td>2722.733</td>
<td>36.212</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>6.223</td>
<td>3232.565</td>
<td>37.955</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>141.556</td>
<td>5856.663</td>
<td>2904.245</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.452</td>
<td>2702.758</td>
<td>35.966</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.302</td>
<td>2737.610</td>
<td>37.194</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.547</td>
<td>2880.836</td>
<td>56.390</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>26.444</td>
<td>3424.291</td>
<td>187.270</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>423.377</td>
<td>6195.114</td>
<td>2435.996</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Persons</th>
<th>Average Time (s)</th>
<th>Average RAM Used (MB)</th>
<th>Average RAM Variation (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>1</td>
<td>0.043</td>
<td>2657.064</td>
<td>8.454</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.037</td>
<td>2641.263</td>
<td>8.855</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.149</td>
<td>2680.636</td>
<td>14.461</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.431</td>
<td>2836.078</td>
<td>14.493</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>5.029</td>
<td>5782.548</td>
<td>341.885</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.452</td>
<td>2702.758</td>
<td>35.966</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.302</td>
<td>2737.610</td>
<td>37.194</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.547</td>
<td>2880.836</td>
<td>56.390</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>26.444</td>
<td>3424.291</td>
<td>187.270</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>423.377</td>
<td>6195.114</td>
<td>2435.996</td>
</tr>
</tbody>
</table>
data collected from Fénix;

- Send invitations to the participants, which, upon acceptance, are considered in future computations.

From the results of our performance tests, we could conclude that the requirements for this project were met. Even for a large number of schedules the system is able to provide a response within a very short time, and the total time necessary to collect the entire results for the time span of a year is within acceptable values.

While our solution successfully fulfills all the requirements, some concerns were left out of the scope of this project and need to be addressed in the future. Besides some improvements that may still be performed on Kronos, this solution may serve as a starting point for other projects that extend the existing functionality, in order to solve other related problems.

Security and availability were not the focus of the development of our solution and still need to be addressed. The system could also still benefit from some optimization regarding CPU and RAM usage. The following improvements should be pursued:

- Use multi-threading for optimal CPU usage;
- Calculate occupations only for necessary activities;
- Reuse data structures for optimized memory consumption;
- Split Business and Presentation into two separate processes.

Besides the problem tackled by our solution, there are other features and functionality that could enhance the usefulness of Kronos or solve other related problems, such as:

- Loading external calendars;
- Automatic schedule generation;
- Schedule prediction based on history analysis.

References


