Kronos: Calendar Management System

Ricardo André Vicente Costa Laranjeiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

October 2016

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.

Resumo

Os membros da comunidade do Instituto Superior Técnico encontram frequentemente dificuldades para agendar actividades, como reuniões, workshops, eventos, avaliações e aulas, devido à falta de meios que permitam obter facilmente a informação necessária à escolha das melhores datas. No entanto, a maioria dessa informação já existe ou pode ser inferida a partir de dados disponíveis no sistema Fénix.

Por este motivo, apresentamos o Kronos: uma plataforma que permite aos utilizadores registados no Fénix estimar e visualizar a disponibilidade conjunta de um grupo de pessoas com as quais pretendam agendar uma actividade. Os utilizadores podem indicar as suas actividades pessoais, além das que são importadas a partir do Fénix, para serem contabilizadas no cálculo da sua disponibilidade. As actividades contempladas no Kronos podem ter um horário fixo (ex: testes) ou não (ex: projectos).

A partir dos testes que conduzimos, concluímos que o Kronos é capaz de apresentar toda a informação necessária num curto espaço de tempo, proporcional ao intervalo de tempo e número de horários cruzados.

Palavras-chave: Agendamento de actividades, estimação de disponibilidades, ocupação difusa, distribuição de carga de trabalho, cruzamento de horários.

Acknowledgments

I wish to thank Prof. António Rito da Silva for believing in this project from the start and giving me the chance to make it a reality and, above all, for his guidance during its development. I would also like to thank my parents, Ana and Eduardo Laranjeiro, and my brother, Tiago Laranjeiro, for their patience and support. I thank DSI for providing me a data sample to work with. My thanks also go to my friends who supported me during this project. In particular, I want to thank Soraia Meneses Alarcão for the precious advice she gave me, David Duarte and Simão Martins for hearing me out and providing ideas on my many development dilemmas, and also Gonçalo Sousa for getting me started with Fénix. Finally, I wish to leave a special thanks to my colleague and great friend Vânia Mendonça, for the many hours she spent proof-reading this document, and for the countless times she provided me invaluable help.

I dedicate this work to my parents, to my brother, to Matilde and José Félix, and to Vânia Mendonça.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

BRMS Business Rules Management System

CPU Central Processing Unit

CSS Cascading Style Sheets

DML Domain Modelling Language

DRL Drools Rule Language

DTO Data Transfer Object

GC Garbage Collector

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol (HTTP) Secure

IST Instituto Superior Técnico

JSON JavaScript Object Notation

JVM Java Virtual Machine

KB Knowledge Base

MVC Model View Controller

ORM Object-Relational Mapping

RAM Random Access Memory

REST Representational State Transfer

WASS Web Appointment Scheduling System

1

Introduction

The present document describes Kronos, a software platform for scheduling activities considering the availability of their target audience. In this chapter, we provide the motivation for our work, we state the problem we want to solve and the goals we are pursuing, we describe our solution and the main contributions of this work, and finally we present the document outline.

1.1 Motivation

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.

In the case of student organizations, it is essential for them to schedule their activities to dates and times when their target students are available to attend, as well as to ensure the organization's collaborators are available to staff those activities. Teachers and student delegates also need to figure out the best schedules for lessons and evaluations according to the courses the students are enrolled at, in order to avoid overlaps and unbalanced workload during the semester. Teachers may also need to to schedule extra classes during the semester, which can be a difficult task due to the fact that students often end up having different schedules. 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, having to wait until each participant has provided their availability, in order to find the most favourable time for the meeting.

This problem is most critical for degrees with a strong practical component where, at some periods during the semester, students start skipping classes, and sometimes even proper sleep, in order to complete all their projects. While the causes may vary from bad time management to excessive workload

over some parts of the semester, they all have time as a common factor. This indicates that the real problem may come from the lack of enough information for making better decisions, regarding project and class scheduling, as well as personal time management. 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' availabilities or the impact of evaluations on the days preceding them, due to students' being more likely to study harder on those days.

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. Considering the many combinations of courses possible, manually crossing the schedules of the hundreds of students of a degree is a daunting task. The schedules also cannot, on their own, provide enough information regarding activities with an undefined schedule, such as projects.

From this problem 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.

1.2 Goals

Our goal is to address the problem described above 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;
- Take into consideration not only the time spent in activities with a well-defined schedule (e.g, written evaluations, classes or meetings), but also the time spent (and not explicitly specified) in long duration activities, such as course projects or study sessions;
- Include additional periods of unavailability due to other activities the user is involved in (not necessarily academic activities).

1.3 Solution

In order to reach the goals listed above, we developed a platform for scheduling activities with one or multiple potential participants. This platform collects information from Fénix, allowing to:

- Select the intended target audience for the activity, either by selecting individuals or groups (e.g, teachers of a certain department or students of a certain degree), also specifying those who are essential to the activity;
- Collect the workload data of the selected target audience during the time interval specified, compute their availability and display it in a calendar view;
- Schedule activities of two types: activities with a well-defined schedule (e.g, classes, events) and activities with no schedule specified (e.g, projects);
- Invite the selected target audience to the scheduled activity and, upon acceptance of the invitations, consider the activity in future computations of the participants' availabilities.

1.4 Document outline

The remainder of this document is structured as follows: in Chapter 2, we review existing scheduling systems, explain which components of the Fénix system are used in the solution and give an overview of the underlying technology used; in Chapter 3, we describe our solution: Kronos, a system for scheduling activities considering the availability their target audience; in Chapter 4, we present the results of the evaluation performed to the Kronos system; finally, in Chapter 5, we provide some final remarks and future work perspectives.

2

Related Work

In this chapter, we provide a review of existing systems for scheduling activities, focusing on those closer to providing a solution for our problem. We also describe the Fénix system and which components are used in our solution. Lastly, we provide an overview of the technologies used for implementing our solution, as well as our reasons for choosing them.

2.1 Existing Scheduling Systems

We started by searching existing scheduling solutions that could potentially be used to solve our problem. In this section we describe a set of existing applications for scheduling of activities and discuss why they are not able to solve our problem.

2.1.1 Web Appointment Scheduling System

The Web Appointment Scheduling System (WASS) [3] is a system for scheduling meetings and similar activities over a web interface. This system allows to create and view activities, and supports e-mail notifications and iCal format support. 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.

This approach is particularly advantageous over the most usual calendar and scheduling systems, both in terms of privacy and in ease of use, since users do not need to specify each and every activity they have in order to expose their availability to others. However, this approach also brings some limitations, since it is mostly oriented towards single person schedules and even in a group calendar, the group's availability is the union of each group member's availability, not their intersection, which means it will show when at least one member is available, but makes it difficult to find out at which periods all of the group's members are available.

2.1.2 Doodle

A well-known scheduling system, Doodle[1] 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 most favorable for them. 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.

The main limitations with Doodle's approach lie in the fact that the user must wait for the invitees to select their available periods and that it is also oriented solely to activities that fully occupy a contiguous time period, thus focusing only the most voted periods, which makes it hard to find, for example, the days with most periods at which there is at least one person available.

Doodle 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, but it is aimed at a single person's activities and is not viable for grouping multiple users' schedules.

2.1.3 BookFresh and BookingBug

Systems such as BookFresh[2] and BookingBug[3] work on a publisher/subscriber model, allowing not only the management of schedules for individual or groups of publishers, such as teachers who intend to schedule their classes, but also to deal with groups of subscribers, such as students who wish to attend those classes.

Since these systems were designed to deal with many different kinds of activities, a user may specify an activity's details such as the location, periodicity, subscription limits and even subscription fees. It is also possible to monitor the activities a user has subscribed to, making it possible, for example, to verify if there are overlapping activities in the user's schedule. In both systems, the interface is a calendar in which it is possible to view each day's scheduled activities.

Despite allowing subscribers to easily find out which activities will fit into their schedule, publishers may only view the subscribers' schedules restricted to each publishers own activities.

2.1.4 Google Calendar

Google Calendar[4] allows users to register not only their personal activities but also 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. It has a feature that allows a user to find a date and time at which all invited participants are available, as long as they have given the necessary permissions to access their personal schedules. The interface is a calendar supporting multiple different views in which the user's activities may be seen, such as daily, weekly, or monthly schedule.

[1] `https://doodle.com` [last accessed: Sep. 10, 2016].
[2] `https://www.bookfresh.com/` [last accessed: Sep. 10, 2016]
[3] `https://www.bookingbug.com/` [last accessed: Sep. 10, 2016]
[4] `https://www.google.com/calendar` [last accessed: Sep. 10, 2016]

Users of both Google Calendar and Google+ may be notified by email when they receive an invitation for an activity. If a user accepts to participate in an activity in Google+, it will be registered in Google Calendar as well.

2.1.5 Discussion

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-need-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 Instituto Superior Técnico (IST), as well as in other universities. It holds information about almost everything in IST, from staff management to student enrollments. 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)[5], a public web interface specially designed by IST's Fénix development team to allow the creation of third-party applications that may use data from Fénix. These applications must be registered on Fénix and, if they require any personal information from a user, they must have the user's permission to access the data. Starting from the */person* endpoint, it is possible to traverse the domain relationships, according to the role of each user, to obtain the necessary data. For example, in the case of a student, the */person* endpoint exposes the student's degree. Through the */degrees/{id}/courses/* endpoint, it is possible to obtain the list of all courses in the degree identified by {*id*} and so on. It is also possible to obtain each person's calendar with classes and evaluations directly from the */person/calendar/classes* and */person/calendar/evaluations* endpoints, respectively.

Alternatively, the necessary data from Fénix can be obtained by a direct access to the system's domain. However, due to the security risks involved in giving access to Fénix's highly sensitive data to a

[5] `https://fenixedu.org/dev/api/` [last accessed: Sep. 19, 2016]

prototype in development, this 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

In this section, we will give an overview of the technologies involved in the development of each part of the solution prototype.

Fénix Framework

Fénix Framework[6] is an open-source Object-Relational Mapping (ORM) framework developed at IST, that provides a transactional and persistent domain model for Java-based applications. The domain model is specified with the Fénix Framework's Domain Modelling Language (DML) and all domain objects and relationships are then generated by the Framework. Previous experience with the framework allied with the fact it is the framework used in the Fénix system itself, led to it being the chosen ORM framework for developing our prototype.

Spring Framework

The Spring Framework[7] provides a large set of tools for handling the infrastructure of a Java application. For our prototype, we just used some of the Spring Core and Spring Web modules, as we just needed it for deploying the server applications and implementing the web REST API.

AngularJS

Due to the complexity of the client application, we chose AngularJS[8] to develop it. AngularJS allows using HTML to template the web interface, to organize JavaScript code into modules, and provides multiple services that allow, for example, asynchronous connections to a REST API.

JMeter

Apache JMeter[9] is an open-source testing application, focused on testing web applications, that allows building and running complete test plans. We chose JMeter as it allows testing the server's performance and efficiency by simulating the web client application, with several simultaneous instances for simulating concurrent users.

[6] `https://fenix-framework.github.io/` [last accessed: Sep. 19, 2016]
[7] `https://spring.io/` [last accessed: Sep. 19, 2016]
[8] `https://angularjs.org/` [last accessed: Sep. 10, 2016]
[9] `http://jmeter.apache.org/` [last accessed: Sep. 19, 2016]

3

Kronos: Scheduling Activities Considering its Participants' Availability

In this chapter we describe our solution for the scheduling problem we introduced in Chapter 1. We start by listing the requirements and use cases for the solution. Then, we describe our domain model and the architecture of the developed solution. Finally, we will cover our implementation of the solution in detail.

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, not only related to functionality but also to data confidentiality. There are also some modifiability requirements, aiming to keep the system adaptable to needs that are foreseen as possible to occur in the future.

3.1.1 Requirement Elicitation

To facilitate the requirements' elicitation, students, teachers and faculty staff were asked to fill a questionnaire where they indicated how important they consider the following features:

1. Configuration of the types of activity to be considered in the computation of the availability;
2. Import a personal calendar into the system;
3. View the availability of a group of persons in order to schedule an activity in a specific period;
4. View the availability of a group of persons in order to schedule an activity with a specified duration, but where it is not necessary to have a face to face meeting;
5. Configure the types of organization to consider when viewing availabilities;
6. Detection of schedule conflicts and computing the number of affected persons;

7. Define relevance levels for the participation of certain persons or organizations;
8. View the impact of the workload or occupation of long duration activities;
9. Opt to receive notifications regarding the proximity of or change of schedule of an event;
10. Add activities with hypothetical workloads in order to simulate known situations;
11. Prioritize certain activity types when computing availability;
12. Select who may view the user's activities;
13. Select who may view the user's schedule;
14. Keep the user's schedule anonymous when computing availability.

The questionnaire was answered by 443 students, 9 teachers and 2 faculty staff members. As we can see in Figure 3.1, although students ranked all features with an importance above 6, it is clear that some were considered more important than the rest, namely conflict detection (8.35), notifications (7.84), viewing the availability of a group of persons (7.57) and viewing the impact of the workload of long duration activities (7.38). As for privacy features, students gave a higher importance to the ability to choose who can view their schedules (7.33) and the activities (7.17) they participate in. The least important feature for students was the possibility of configuring the types of activities to be accounted for in the availability estimates (6.28).

Similarly to students, teachers ranked all features with an importance above 5, but considered the privacy feature of choosing who can view their activities as the most important (7.88). They also considered highly important the possibility of viewing the availability of a group of persons (7.44), conflict detection (7.22) and notifications (7.0). The least important feature for teachers was the possibility to import a calendar from another system (5.11).

Comparing the results between students and teachers, it becomes clear 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.

3.1.2 Functional Requirements

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;

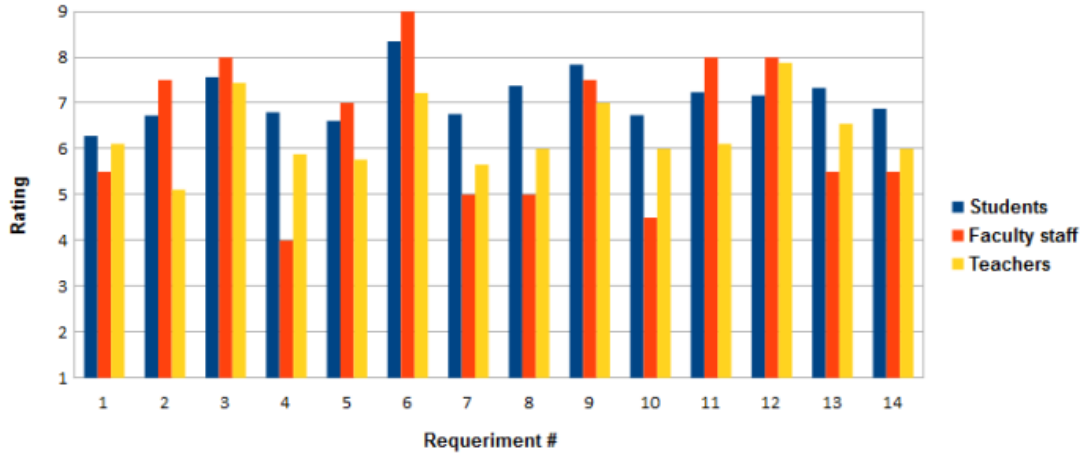

Figure 3.1: Questionnaire Results by Role

- Send invitations for the selected 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.

Not all features referred in the questionnaire were considered as requirements in the scope of this project, however. Importing a personal calendar into the system was not considered a requirement as it is merely an automation of the client application, which is not the focus of this project. Notifications were also left out of the requirements as they are not part of the core functionality of the project.

3.1.3 Confidentiality Requirements

On a system of this nature, there is the need to ensure user data confidentiality. The data may be categorized, according to its origin, as follows:

- Data collected from Fénix;
- Data inferred from data collected from Fénix;
- Data inserted by the users.

Data from Fénix may be public or private. For public data, no special access permissions are required. For private data, all security requirements they had originally in the Fénix system must be preserved. As for inferred data, if it can be used to guess the source data, then it must also preserve the same security requirements as the source data. Access policies to user inserted data should also be explicit. 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.

3.1.4 Modifiability Requirements

In order to maximise its usefulness, the system should be able to read data from and have its data read by other systems. Since the system may be set up in different environments and the other systems it interacts with may change, modules that read or expose data should be highly adaptable and easy to replace. 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.

3.1.5 Usability Requirements

The user interface of the client application is not the main focus of this project, so the user interface requirements are not covered. However, it is necessary to ensure the core of the system does not cripple the usability of the system as a whole. Therefore, 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.

3.1.6 Performance Requirements

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.

3.1.7 Use Cases

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;
- **Cancel participation**: Participants of an activity may cancel their participation at any moment, having the activity removed from their schedule;
- **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, as can be seen in Figure 3.2, was highly influenced by the Fénix domain model[1], 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 [1] 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.

[1] `https://fenix-ashes.ist.utl.pt/fdb/` [last accessed: Oct. 12, 2016].

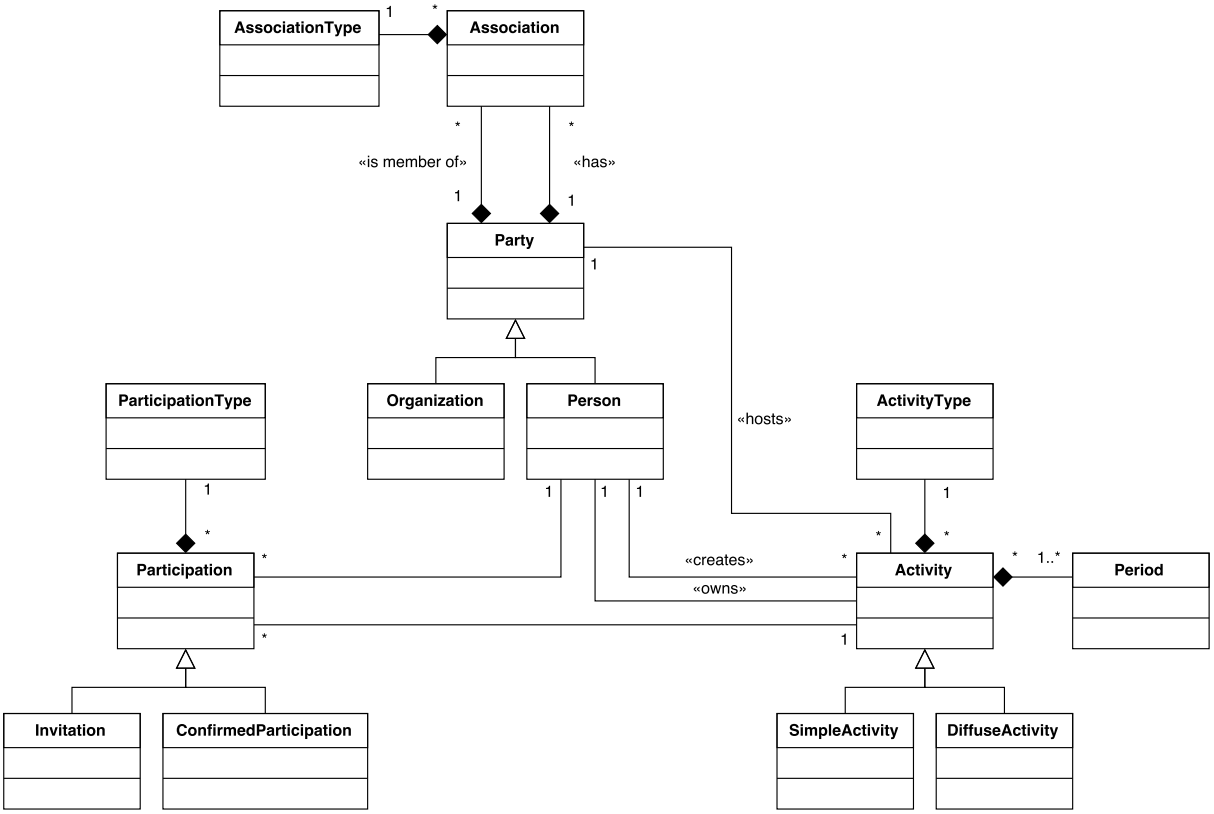

Figure 3.2: Domain Model

  - **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.
- **PartcipationType**: The role of a Participation's participating Person, such as "Teacher" or "Student".

Despite being based on Fénix's domain, the correlation between both domains is not direct and data from Fénix must be mapped to Kronos' domain model. Each Person in Kronos corresponds directly to a Person in the Fénix 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 Fénix 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, as shown in Figure 3.3. A description of each layer as well as their purpose is given in the following subsections.

3.3.1 Controllers

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.

3.3.2 Presentation

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 availabilities are to be requested and exposed according to days and periods of a calendar, then this layer structures that data to be accessed as if it were indeed stored according to a calendar. This is also the layer where user sessions and authentication are handled.

3.3.3 Processors

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.

3.3.4 Business

This 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, described in Section 3.2, 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 [2] 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.

3.3.5 Data Access

In this layer, the system has the ORM and Connector modules.

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 Fénix, and mapping it to Kronos' domain objects. Each Connector implementation acts as a Mapper [2] 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.

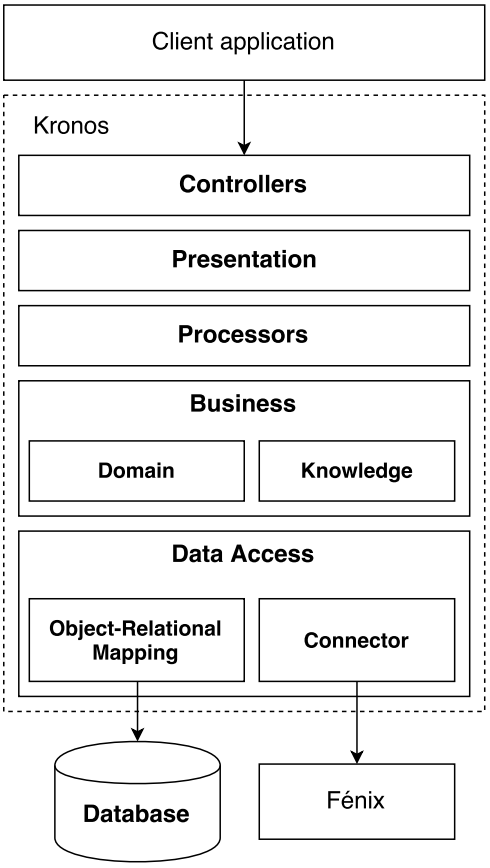

Figure 3.3: Layered Architecture Diagram

3.4 Implementation

In this section, we explain the implementation approaches of both the Kronos server prototype and its client application prototype.

3.4.1 System Core

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[2]. The Kronos Engine 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 writting the system's log messages to its configured output streams. It can differentiate between file streams and console streams, allowing it to print progress bars to the console while keeping the file logs clean.

The Feature Toggler is where the system's configuration is held and its settings can be changed at any time during execution. It configures features such as caching mode, algorithm settings and allows switching between different module implementations.

3.4.2 External Data Loading

As explained in Section 3.3.5, external data is loaded and mapped to the Kronos domain by modular Connectors. Initially, we implemented a Connector specifically designed to fetch data from the FénixEDU REST API. However, at the time of development, the FénixEDU API could not provide all the data required by Kronos, mainly due to the fact it required every potential user to first register to the Kronos application. Thus, a new Connector was implemented to read a JSON file exported directly from the Fénix system, which contained all the necessary data.

On our prototype implementation, the Fénix data is loaded only once at the first initialisation of the system, due to the fact the data does not change. On a production-ready implementation however, when using a Connector interfacing directly with Fénix, the Connector should check for possible data updates and trigger the update of the domain model.

It is also during this initialisation phase that the Periods are generated. As mentioned in Section 3.2, a Period represents a block of time. This block of time can represent any amount of time and in our implementation a span of 30 minutes was used, as it is the basic block of time used in Fénix's schedules. The number of Periods in a day is also adjustable, making it possible for a day in the system to be, for example, the set of Periods from 8:00 to 22:00. These settings can no longer be adjusted once the Domain is initialised, as the Connector requires them to properly map source schedules to Kronos' Activities and Periods.

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, and will be further explained in the next subsection.

[2] `http://martinfowler.com/articles/feature-toggles.html` [last accessed: Oct. 14, 2016].

3.4.3 Domain Management and Persistence

Domain objects and relationships are persisted by the FénixFramework and are specified in its DML, of which a small excerpt is shown in Listing 3.1.

Listing 3.1: Domain Modelling Language

```
class Party {
    String fenixId;
    String name;
}

class Person extends Party {
    String istId;
}

class Activity {
    String name;
    boolean locked;
}

class Participation {
    boolean locked;
}

relation Participates {
    Person playsRole participant {multiplicity 1..1;}
    Participation playsRole participation {multiplicity 0..*;}
}

relation ActivityParticipation {
    Participation playsRole participation {multiplicity 0..*;}
    Activity playsRole activity {multiplicity 1..1;}
}
```

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 Fénix Framework transactions, allowing all transactions to be opened in the faster *read* mode by default and only upgrading them to *write* mode precisely when necessary. When returning data to other modules, the Domain Manager always boxes domain data into DTOs, allowing transactions to be closed as soon as all domain operations are concluded.

As previously mentioned, creation of new instances of domain elements is handled by a Domain Factory class, which belongs to the Domain Manager. This Domain Factory implements all domain creation logic, such as registering new activities, which involves creating not only the activity but all its relationships with other domain entities, as is shown in Listing 3.2.

Listing 3.2: Excerpt of the Domain Factory

```
@Atomic(mode = TxMode.WRITE)
public String registerPerson(String fid, String name, String istid) {
    return new Person(fid, name, istid).getExternalId();
}

@Atomic(mode = TxMode.WRITE)
public String registerOrganization(String fid, String name, String acronym) {
    return new Organization(fid, name, acronym).getExternalId();
}
```

```
@Atomic(mode = TxMode.WRITE)
public String registerActivity(String name, String creator, String host, Set<String>
    periods, String activityType, boolean locked) {
    ActivityType activtype;
    if(!this.activtypes.contains(activityType)) {
        activtype = new ActivityType(activityType);
        this.activtypes.put(activityType, activtype.getExternalId());
    } else {
        activtype = FenixFramework.getDomainObject(this.activtypes.get(activityType));
    }
    Set<Period> p = new HashSet<>();
    for(String id : periods) p.add(FenixFramework.getDomainObject(id));
    SimpleActivity a = new SimpleActivity(name, FenixFramework.getDomainObject(creator), p,
         activtype, locked);
    a.addHost(FenixFramework.getDomainObject(host));
    return a.getExternalId();
}
```

When it comes to modifying or destroying existing domain data, this is handled by the Domain Manager and the logic is implemented directly into the domain classes. For example, when an Invitation is accepted, it instances a new ConfirmedParticipation containing the Invitation's data and then deletes itself, as shown in Listing 3.3.

This self-deletion functionality was implemented in order to avoid burdening the Domain Manager with domain logic that was too specific for each object. It consists in first destroying all the object's relationships and then, if applicable, destroying domain objects that would be orphaned relatively to the domain model, such as the Participations of a deleted Activity, as presented in Listing 3.4. Finally, using the Fénix Framework's own domain object deletion functionality, the object deletes itself.

Not all elements of the domain model may be modified, however. All objects that are loaded from Fénix have a flag that marks them as *locked*. This means no destructive methods will have any effect upon them, either by throwing an exception or simply doing nothing.

Listing 3.3: Accept Invitation

```
@Atomic
public String accept() {
    String newId = new ConfirmedParticipation(this).getExternalId();
    delete();
    return newId;
}
```

Listing 3.4: Activity self-deletion

```
@Atomic
public void delete() {
    if(!getLocked()) {
        for (Participation p : getParticipationSet()) p.delete();
        getCreator().removeCreated(this);
        for (Party p : getHostSet()) p.removeHosted(this);
        getOwner().removeOwned(this);
        for (Period p : getPeriodSet()) p.removeActivity(this);
        getType().removeActivity(this);
        deleteDomainObject();
    }
```

}

3.4.4 Knowledge Inference

All inferred schedule data is produced and maintained by a Knowledge Manager. Besides holding the Knowledge Base (KB), the Knowledge Manager uses a Knowledge Updater to infer all schedule occupation data from the domain data. This inferred data is stored in small Occupation objects, that contain the occupation rate and a flag to signal whether that occupation is caused by a Simple Activity or by workload from Diffuse Activities. In the Knowledge Manager, each pair of Person and Period has an assigned Occupation object, which must be updated whenever any Activity involving that Person and Period pair suffers any changes in the domain. The Knowledge Manager implementation can be selected by the Feature Toggler, but changing it requires restarting the Kronos Engine.

While it does not actively react to domain changes, the Knowledge Manager's interface provides a method that tells it to update the Occupations related to a specific Person, which is called not only when another module makes changes to a Person's Participations, but also when initialising the KB. For each Period in which a Person participates in at least one Simple Activity, an Occupation with a rate of 100% will be assigned, regardless of how many Simple Activities in which that Person has a Participation also occur over that Period.

Unlike Simple Activities, Diffuse Activities add a percentage of their workload to the occupation rate of each Period in its time span, based on their average Period workload (W_{avg}) and each Period's relative position within the Diffuse Activity's time span ($R(p)$). The average workload of a Diffuse Activity is equal to the quotient of the total workload in Periods (W_t) by the Activity's time span in number of Periods (D_p), as shown in Equation 3.1.

$$W_{avg} = \frac{W_t}{D_p} \tag{3.1}$$

The relative position within the time span ($R(p)$) of a Period (p) is given by the quotient of the number of days from that Period's day to the day of the Activity's ending Period ($E(p)$) and the total days in which the activity takes place in (D_d), as shown in Equation 3.2.

$$R(p) = \frac{E(p)}{D_d} \tag{3.2}$$

The workload attributed to each Period ($W(p)$) is given by the Equation 3.3. The slope m can be used to determine whether the workload is heavier towards the Activity's end, its beginning or even keep it even along the entire time span.

$$W(p) = W_{avg} \times (1 + m \times (0.5 - R(p))) \tag{3.3}$$

The value of the slope can be set at start up in Kronos' settings and must be a real number in the range $[-2, 2]$ in order to ensure that the total distributed workload along the set of Periods in an Activity's time span (P) is approximately equal to the Activity's total workload, as shown in Equation 3.4.

$$\sum_{p \in P} W(p) \approx W_t \tag{3.4}$$

The reason for the total distributed workload not being exactly equal to the Activity's total workload is that each Period's relative position in the time span is calculated in days, which has a lower granularity and will result in some loss of precision when the Activity starts in a Period that is not the first Period of a day or ends in a Period that is not the last Period of a day. This was done intentionally, in order to make the workload even within each day.

The workload is distributed regardless of the occupation rate the Person already has for each Period, which may cause some Occupations to get an occupation rate above 100%. Thus, the next stage of the schedule updating process is to redistribute the workload from Occupations where it overflowed past 100% occupation. For this purpose, the following algorithms were designed:

- **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 [4] 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 3.3 can be chosen by configuring Kronos' Feature Toggler, described in Section 3.4.1. 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, as shown in Listing 3.5.

Listing 3.5: Knowledge Base occupation insertion

```
@Override
public void insertPersonPeriodOccupation(String personId, String periodId, Occupation
    occupation) {
```

```
    Hashtable<String, Occupation> occupations = periodOccupations.get(periodId);
    if(occupation.getValue() == 0f) {
        if(occupations != null) {
            if(occupations.containsKey(personId)) {
                occupations.remove(personId);
                if(occupations.size() == 0) periodOccupations.remove(periodId);
            }
        }
    } else {
        if(occupations == null) {
            occupations = new Hashtable<>();
            periodOccupations.put(periodId, occupations);
        }
        occupations.put(personId, occupation);
    }
}
```

Part of the update process was initially implemented using the JBoss Drools Business Rules Management System (BRMS)[3]. 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, the modifiability benefits granted by the DRL could not outweigh the overhead of using Drool's stateful knowledge session, when compared to the efficiency of a solution implemented directly in Java. Since the update logic would only branch on whether the activity was diffuse or simple, Drool's rule engine had little room to outperform a direct Java implementation and its rule language granted little benefit over standard Java. This led to the Drools implementation being eventually abandoned, but it heavily inspired the interface of the Knowledge Manager, since Drools continued to be used interchangeably with the other Knowledge Manager implementations, until it was no longer worth maintaining.

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. The cache has the following working modes:

- **Disabled**: The Occupations are discarded after each query and each requested Person's schedule will need to have its Occupations recalculated every time;
- **Enabled**: Occupations are calculated the first time they are requested and will be kept in memory and reused for every subsequent query;
- **Initialised**: During KB initialisation, all Occupations for all Persons are calculated and stored in the cache.

3.4.5 Queries and User Sessions

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[4], an excerpt of which is shown in Listing 3.6. Query data is retrieved on a Period

[3] `http://www.drools.org/` [last accessed: Sep. 10, 2016].

[4] `http://www.mathworks.com/help/database/ug/importing-data-using-a-scrollable-cursor.html` [last accessed: Sep. 25, 2016]

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 the requested data, which comes in the form of a PeriodAvailabilities object for each Period, and each of these objects in turn containing a set of Availability objects, each containing a requested Person's occupation rate for that Period. These objects are essentially DTOs for subsets of the KB structures storing the Occupations.

Listing 3.6: Knowledge Base cursor behaviour

```
private void loadAvailabilitiesOfRange(long start, long end) {
    Set<PeriodDTO> periodsOfDay = dm.getPeriodsInRange(start, end);
    for(PeriodDTO period : periodsOfDay) {
        Set<Availability> results = km.getOccupationsInPeriod(persons, period.getExternalId
            ());
        PeriodAvailabilities pa = new PeriodAvailabilities(period, results, mandatory,
            required);
        data.put(period.getTime().getMillis(), pa);
    }
}

private void loadDayOfInstant(long instant) {
    data.clear();
    DateTime selected = new DateTime(instant);
    DateTime startTime = selected.minusMillis(selected.getMillisOfDay());
    long dayStart = startTime.getMillis();
    long dayEnd = startTime.plusDays(1).getMillis() - 1;
    loadAvailabilitiesOfRange(dayStart, dayEnd);
    checkEachPeriodValidity();
}

private PeriodAvailabilities getPeriodAvailabilities(long instant) {
    if(!data.containsKey(instant)) loadDayOfInstant(instant);
    return data.get(instant);
}
```

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, as shown in Listing 3.7. The cursor remains cached in that user's session until a new query is performed or the user starts a new session.

Listing 3.7: Cursor wrapped and stored in session

```
AuthenticatedData adata = authenticateAndExtractData(json, mapper);
...
AvailabilityQueryResult data = KronosService.checkAvailability(mandatoryPersons,
    optionalPersons, required, duration, instants);
ResultsCalendar calendar = CalendarFactory.newResultCalendar(data);
adata.getSession().setQueryResult(calendar);
```

As shown in Listing 3.8, this makes it possible for client applications to request small blocks of data as needed, instead of having to request and load all of a query's data even when only a part of it would be used, as well as allowing the client applications to be more responsive.

Listing 3.8: Fetching results from the cursor

```
AuthenticatedData adata = authenticateAndExtractData(json, mapper);
JsonNode request = adata.getData();
if(!request.hasNonNull("days")) throw new InvalidRequestException();
ResultsCalendar calendar = adata.getSession().getQueryResult();
ObjectNode response = mapper.createObjectNode();
ArrayNode rdays = response.putArray("days");
Iterator<JsonNode> days = request.get("days").elements();
while(days.hasNext()) {
    JsonNode day = days.next();
    if(!day.hasNonNull("y")) throw new InvalidRequestException();
    if(!day.hasNonNull("m")) throw new InvalidRequestException();
    if(!day.hasNonNull("d")) throw new InvalidRequestException();
    int y = day.get("y").asInt();
    int m = day.get("m").asInt();
    int d = day.get("d").asInt();
    ObjectNode rday = rdays.addObject();
    rday.put("y", y);
    rday.put("m", m);
    rday.put("d", d);
    ArrayNode periods = rday.putArray("periods");
    synchronized (calendar) {
        for (int p = 0; p < calendar.getNumRows(); p++) {
            ObjectNode period = periods.addObject();
            period.put("eid", calendar.getPeriodId(y, m, d, p));
            period.put("available", calendar.getAvailableCount(y, m, d, p));
            period.put("availability", calendar.getAverageAvailability(y, m, d, p));
            period.put("schedulable", calendar.isSchedulable(y, m, d, p));
            period.put("valid", calendar.isValid(y, m, d, p));
        }
    }
}
```

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 and a token. The identifier is used to identify the user in all requests while the token 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. If the tokens do not match, the requests are rejected.

3.4.6 Representational State Transfer API

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, except for the *login* endpoint, receive the user's id and authentication token, and are as follows:

- ***/activity/cancel/***: Receives the unique identifier of an Activity and cancels that Activity;
- ***/activity/details/***: Receives the unique identifier of an Activity and returns the details of all Participations of that Activity;
- ***/activity/owned/***: Returns the details of every Activity owned by the user;
- ***/activity/propose/***: Receives a name, a type, a set of Period identifiers, a set of Person identifiers and creates a new Activity with the sent data and an Invitation for each selected Person;

- ***/availabilities/details/***: Receives a set of days and returns detailed data for each Person's availability for each Period within those days, from the results of the last query in the user's session;
- ***/availabilities/query/***: Receives a set of days, a duration, the number of minimum available participants required, a set of identifiers for mandatory Persons, a set of identifiers for optional Persons and initiates a query, storing its results cursor in the user's session. In order to prevent a user from checking another user's schedule, if a query only has one participant selected, that participant must be the user;
- ***/availabilities/result/***: Receives a set of days and returns the average availabilities for each Period within those days, from the results of the last query in the user's session;
- ***/calendar/***: Returns the base calendar;
- ***/login/***: Receives a username and a password and returns the user's Person unique identifier and a session token;
- ***/organization/***: May receive the unique identifier of an Organization and returns the set of identifiers of the Persons who have an Association to that Organization, the AssociationType for each Person in the set and the set of Organizations that have an Association as child to that Organization. If no identifier is received, the set of all Organizations that do not have an Association as child to any other Organization is returned;
- ***/participation/accept/***: Receives the unique identifier of an Invitation and accepts it;
- ***/participation/cancel/***: Receives the unique identifier of a ConfirmedParticipation and cancels it;
- ***/participation/confirmed/***: Returns the details of the user's ConfirmedParticipations, including the Activity's name and the user's role in the Activity;
- ***/participation/invitations/***: Returns the details of the user's Invitations, including the Activities' names and the user's role in each Activity;
- ***/participation/reject/***: Receives the unique identifier of an Invitation and rejects it.

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.

3.4.7 Browser Client Application

The Browser Client Application is a single page application developed in JavaScript, HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). It also uses AngularJS[5] and Bootstrap[6], 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.

[5] `https://angularjs.org/` [last accessed: Sep. 10, 2016]
[6] `http://getbootstrap.com/` [last accessed: Sep. 10, 2016]

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, as seen in Figure 3.4, 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.

Figure 3.4: Main Menu

When scheduling a new activity, the user must first set the desired settings for the activity and select the intended participants to invite. The options, as shown in Figure 3.5, are described as follows:

- **Activity Type**: Set whether it is an activity with a defined schedule (presential) or an activity without a defined schedule (project);
- **Personal**: Set whether the user is the only intended participant for the activity. This should be used for inserting personal activities in the schedule that are not related to any other user in any way;
- **Activity Duration/Workload**: Sets the duration for simple activities or the workload in periods for diffuse activities;
- **Available Participants Required**: Specifies the minimum amount of available participants on a period to make that period valid for selection.

On the same screen, the user may press a button to open a pop-up window, as shown in Figure 3.6, 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

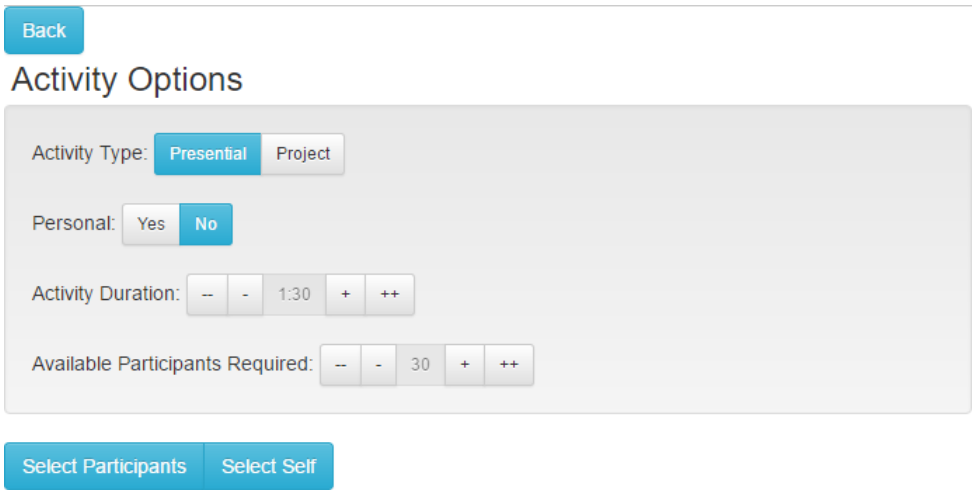

Figure 3.5: Activity Options

remove or set them as essential for the activity.

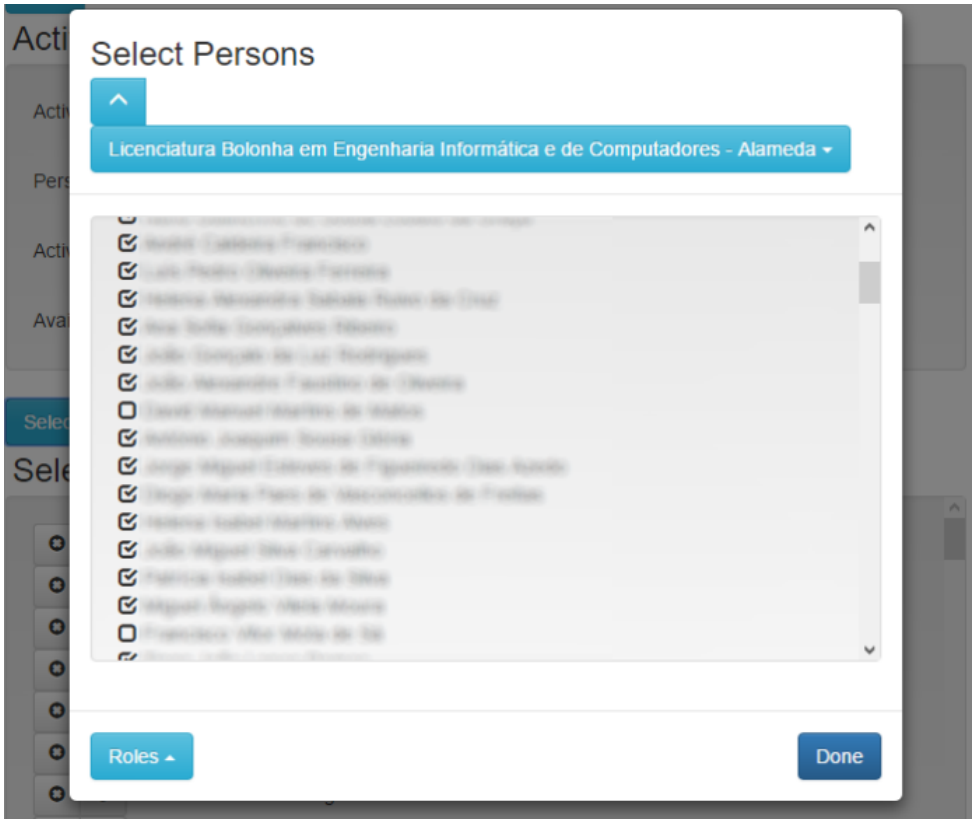

Figure 3.6: Person Selection pop-up

After confirming the activity's settings and intended participants, the user is taken to a calendar screen, seen in Figure 3.7, 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, shown in Figure 3.8, which details the estimated availabilities of each of that week's days' periods. By opening

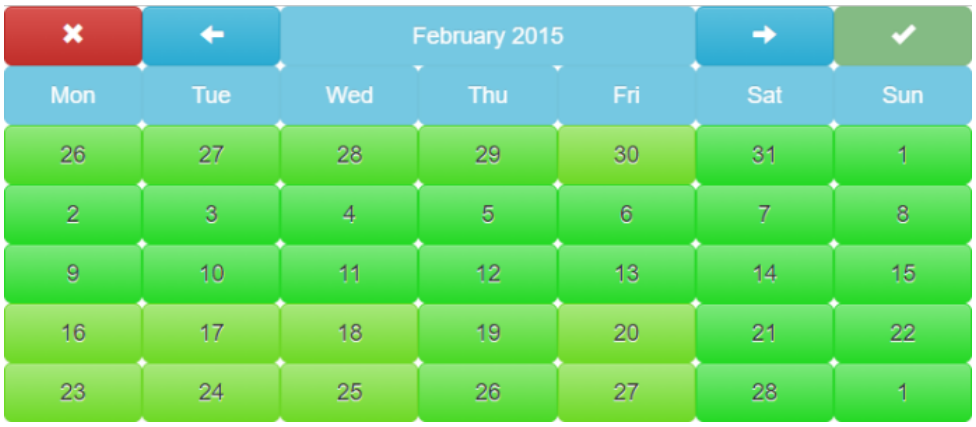

Figure 3.7: Monthly View of the Calendar

the options pop-up, which is shown in Figure 3.9, the user may choose to view these availabilities in percentage, having a similar meaning to the gradient colors, or in total number of available participants, which accounts for the participants that have an availability above 0%. The reason to consider any participant with an availability above 0% as available, even if that availability is as low as 1%, is so that the period may still be valid for the user to select, despite being colored red if most participants have a very low availability. 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 specified minimum 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, as shown in Figure 3.10.

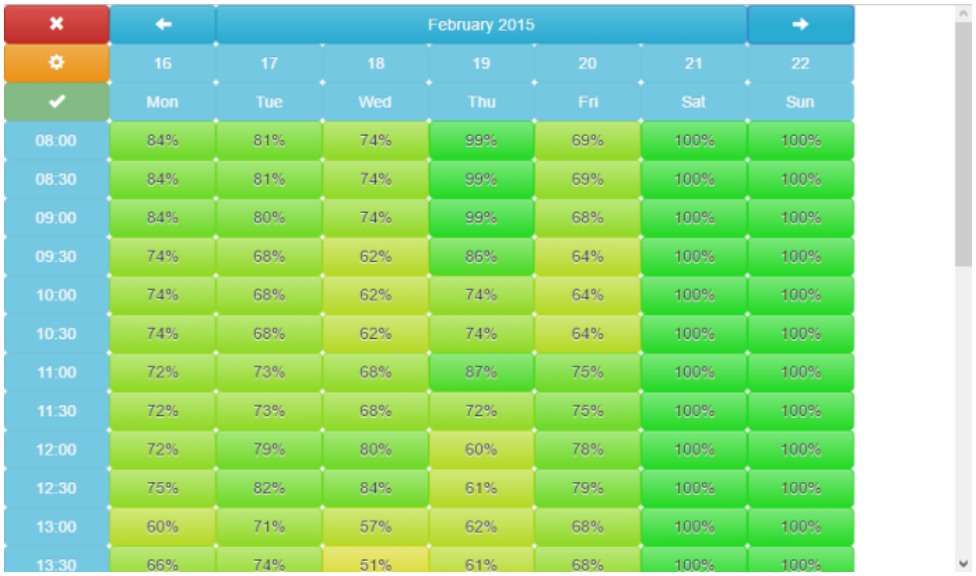

Figure 3.8: Weekly View of the Calendar

Finally, having chosen the activity's schedule, the user will be given the opportunity to name the activity, review its details and confirm it, as shown in Figure 3.11. Upon confirmation, all selected participants

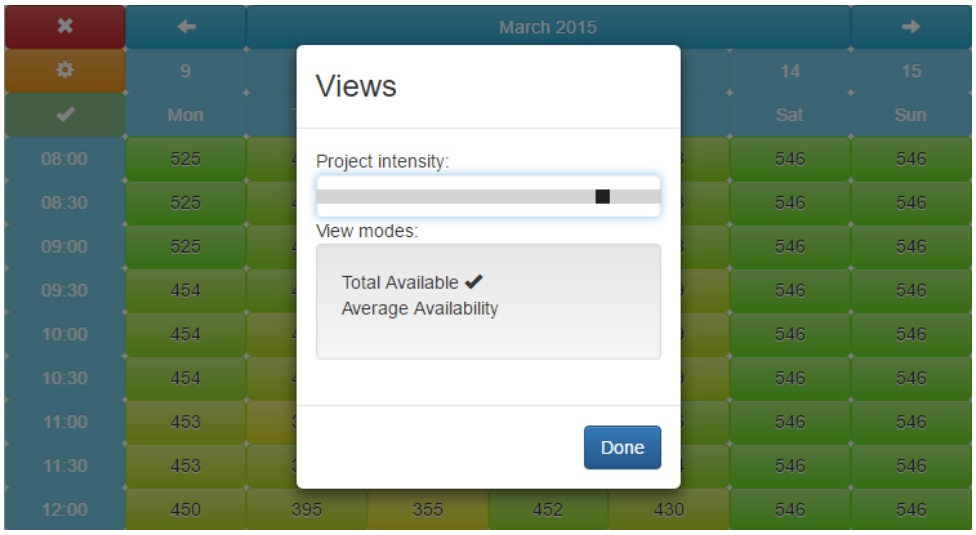

Figure 3.9: Calendar View Options

Figure 3.10: Monthly View of the Calendar for Diffuse Activities

will be invited to the activity and the user will 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, as it can be seen in Figure 3.12, 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 "Invitations" tab of the "My Activities" screen, shown in Figure 3.13, 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.

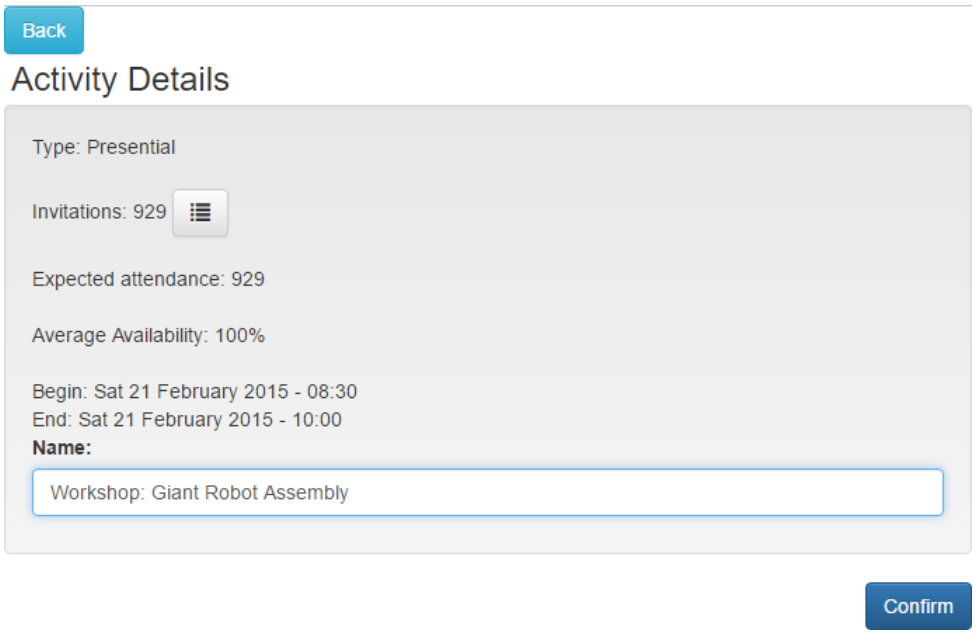

Figure 3.11: Activity Confirmation

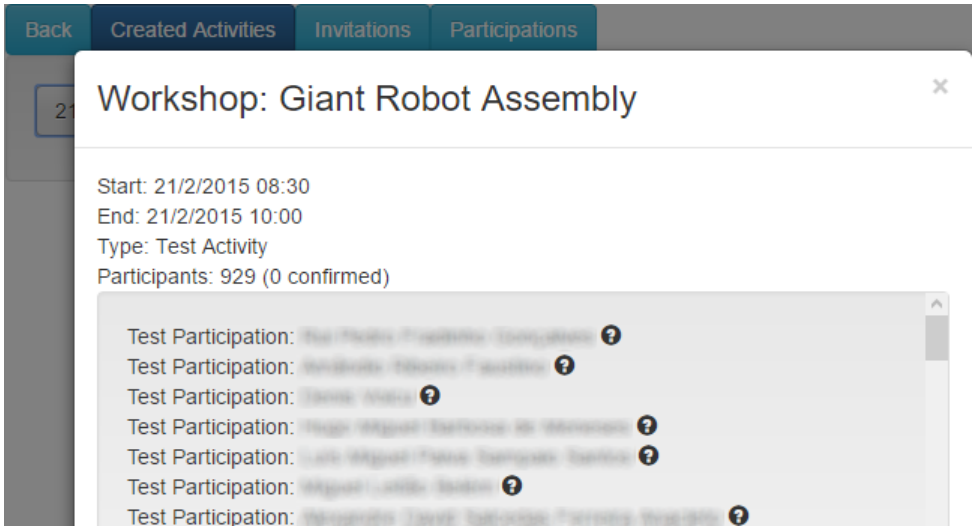

Figure 3.12: Viewing an Owned Activity

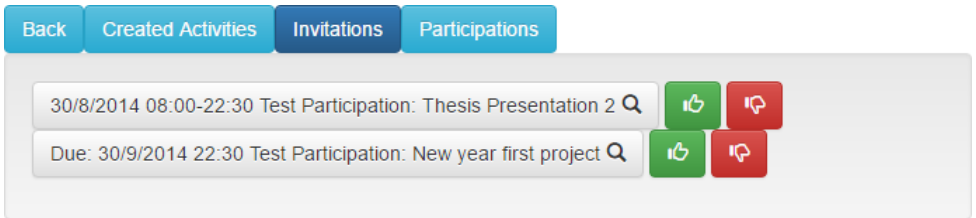

Figure 3.13: Viewing Invitations

The user may select an activity to view its details, as shown in Figure 3.14, or may cancel a previously confirmed participation.

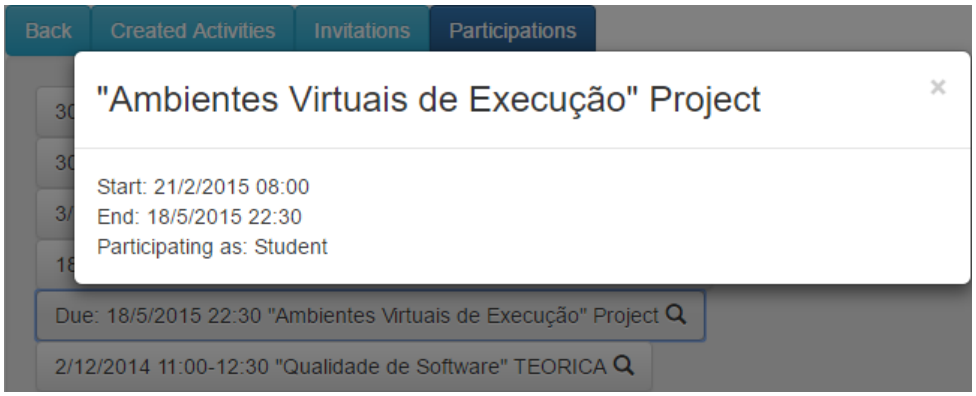

Figure 3.14: Viewing a Participation in an Activity

3.5 Summary

In this chapter, we presented the steps of developing the system proposed in Chapter 1, from the requirements elicitation to the design and implementation of a platform that would meet those requirements.

First, by sending a questionnaire to the target user groups and analysing the answers, we concluded that the most wanted functional features were: viewing the availability of a group of persons, conflict detection, and notifications. Confidentiality was also considered highly important by the respondents. From these results, we derived most of the functional requirements. The confidentiality requirements for our solution were mainly imposed by the source and nature of the data, as IST's and national policies regarding its protection must be enforced within our system. Our modifiability requirements aim mostly at improving the longevity of the system, by ensuring it remains adaptable to changes in the source of the data, as well as making it an adequate tool for solving similar problems in other contexts.

Concerning the design of our solution, the domain model is inspired by Fénix's own domain model, as it is modelled after IST's organizational structure, but remains flexible enough to adapt to other organizational structures. Kronos collects data from external systems through modular Connectors, that are also responsible for mapping that data to Kronos' domain objects, which implement most of the domain logic and are persisted by the Fénix Framework. This domain data is then used to infer the schedules of each user, calculating each user's rate of occupation for each period using algorithms that estimate the impact of each activity in a user's schedule, and are capable of dealing with workload overflow. Queries to Kronos are performed using cursor objects, which are stored in user sessions and make it possible for client applications to request query data in small chunks, which improves responsiveness and lowers memory usage.

We implemented a REST API, as a means for client applications to communicate with Kronos and perform all available operations. We also developed a client prototype that runs as a single page application in the user's browser and is designed to be compatible with multiple kinds of devices, from handheld to desktop.

4

Evaluation

In this chapter we validate our solution's fulfillment of the requirements described in Section 3.1. We also present our procedures for testing the system's performance and the results we obtained, which we discuss in the end of the chapter.

4.1 Functionality

From Section 3.4.6 and Section 3.4.7 is is possible to conclude that most of the functional requirements were fulfilled. The endpoints of the REST API allow to:

- Create simple and diffuse activities, personal or with other participants, who will receive invitations;
- Select participants individually or in groups and choose which of those are essential;
- Accept another user's invitation for an activity, in which case the activity starts being considered in availability computations;
- Delete activities created by the user.

In the client application it is possible to increase, decrease or disable the impact of diffuse activities when viewing availabilities, as is shown in Figure 3.9.

Being able to select the time span for computing the availabilities became unnecessary thanks to the scrollable cursor described in Section 3.4.5, as the user can simply navigate through the calendar to search for the desired dates, without the hassle of having to specify the time interval in which the query would be performed.

Some requirements, however, were not fully implemented, due to time constraints:

- It is not possible to edit activities created by the user;

- It is possible for the user to manually check for activities in which he participates that have conflicting schedules, but the system does not automatically detect and warn about these cases.

4.2 Confidentiality

When it comes to confidentiality requirements, it is possible to conclude, from the description of the REST API endpoints in Section 3.4.6, 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. However, they are not enough to enforce them, since security requirements, such as using IST's authentication system and requiring all communication with the REST API to be performed over HTTP Secure (HTTPS), were not covered in the scope of this project. While Kronos attempts to prevent a user from querying the schedule of another user in particular, it is still possible to overcome that barrier by checking the detailed availabilities in the week view, shown in Figure 3.8, or by performing multiple queries containing the target user and singling out the similarities. However, the availability data by itself is not enough to infer which activities are behind the occupations, unless the user already has knowledge of a schedule with the same set of activities and no other activities could have identical schedules, making it very unlikely that a malicious user could use these means to obtain information that he could not get otherwise.

4.3 Modifiability

Kronos fulfills all of the modifiability requirements presented in Section 3.1 as follows:

- The Connector is the module responsible for collecting source data from other systems, and from Section 3.3.5 and Section 3.4.2 it is confirmed that Connector implementations are easily interchangeable;
- The Controllers layer, described in Section 3.3.1 is the service layer responsible for exposing the system's functionality to all client applications and external systems;
- From Section 3.4.4, it is stated that both the workload distribution algorithm and the estimation formula can be configured via Kronos' Feature Toggler.

4.4 Usability

All usability requirements are met by the REST API:

- As stated in Section 3.4.6, all endpoints that do not return any operation data will return a boolean stating whether the operation was concluded successfully;
- 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.

4.5 Performance

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. What can be considered a reasonable time may vary according to the number of schedules being calculated. Although waiting 1 minute for obtaining the availabilities of 300 persons for a time span of a year can be considered acceptable, it is not acceptable to take the same amount of time for retrieving the availabilities of 5 persons in the same time span. 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.

4.5.1 Goals

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.

4.5.2 Procedure

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, shown in Figure 4.1, 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.

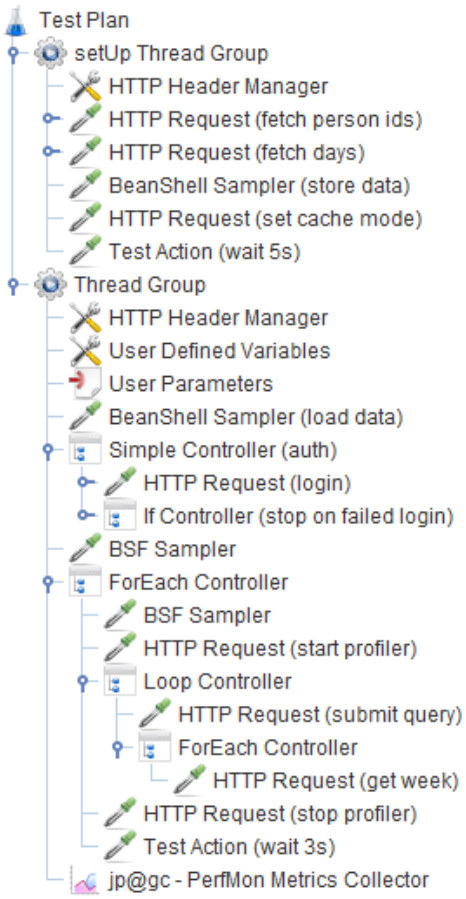

Figure 4.1: JMeter Test Plan

4.5.3 Results

Results for the tests are shown in Tables 4.1 to 4.3 and Figures 4.2 to 4.11. All graphs were generated by JMeter while profiling Kronos. Higher resolution versions can be consulted in Appendix A. CPU usage is displayed by the blue line and the JVM's RAM usage is displayed by the red line. This RAM usage corresponds to JVM's Heap, not to memory being held by Kronos, thus it does not appear to decrease when the GC is triggered. The sharp CPU usage spikes in the graphs correspond to the concurrent GC. The graphs include the test plan's setup time, which can be told apart from the queries by the first time the CPU usage drops to 0 in each chart, corresponding to a pause of 5 seconds. The queries for each set of persons were done sequentially, with a pause of 3 seconds between each one, which can be noticed in the graphs by the CPU usage dropping to 0.

The tables were generated by parsing the data written in Kronos' logs by our embedded profiler,

which was only used for single-user tests. These tables feature the following information:

- **Cache**: The cache mode, as explained in Section 3.4.4;
- **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;

Cache Modes and Sequential Queries

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 shown in Table 4.1, 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.

In Figure 4.2, it is shown that memory usage for the *initialised* cache mode remains relatively constant throughout all the queries. When comparing to the other two modes, displayed in Figure 4.3 and Figure 4.4, 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. This continues to be seen in the remaining test results in this Section, but identifying the exact cause would require a much more exhaustive profiling.

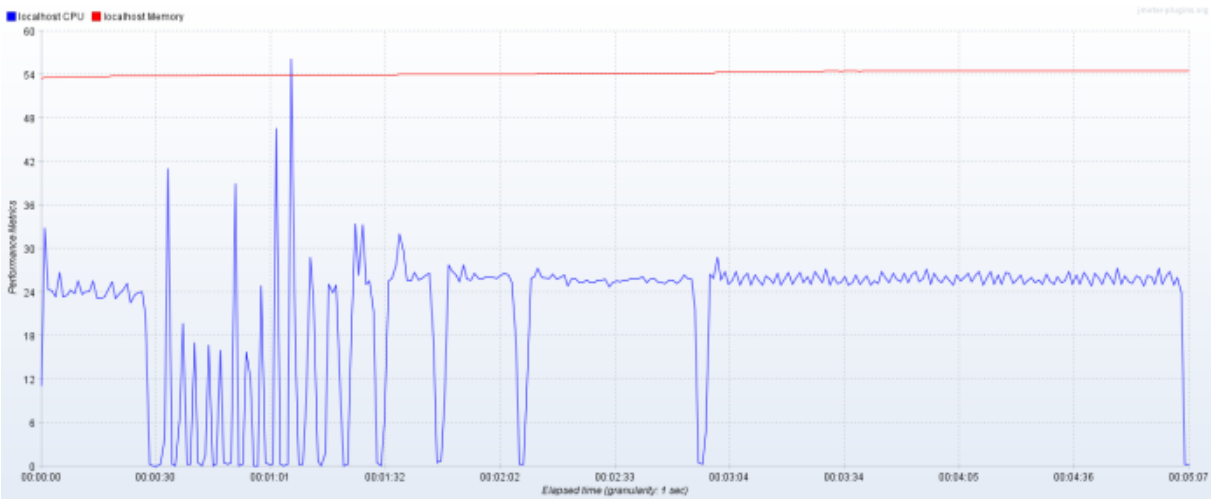

Figure 4.2: JMeter Profile: Cache initialised, 1 user, 1 iteration, 1 year

Table 4.1: Results for queries spanning one year with 1 iteration

Cache	Persons	Total Time (s)	Average RAM Used (MB)	Average RAM Variation (MB)
Initialised	1	0.704	3511.600	37.245
	2	0.558	3549.939	38.100
	3	0.436	3589.709	39.570
	5	0.467	3633.165	43.454
	10	0.497	3689.191	55.795
	20	0.564	3705.553	265.945
	30	0.562	3596.122	97.242
	50	0.689	3702.775	263.669
	100	0.989	3703.392	260.221
	200	1.616	3700.390	259.980
	300	2.199	3704.290	258.403
	500	3.457	3703.765	256.609
	1000	6.457	3696.618	253.595
	2000	12.851	3709.001	257.799
	3000	18.916	3717.332	253.061
	5000	44.220	3726.274	239.419
	10000	127.081	3762.974	243.028
Enabled	1	1.177	2721.250	33.989
	2	0.945	2726.244	35.136
	3	0.591	2728.715	39.568
	5	0.903	2728.296	36.799
	10	1.178	2732.005	37.919
	20	2.204	2737.650	39.211
	30	1.966	2741.897	39.366
	50	3.905	2753.330	40.668
	100	6.700	2764.319	46.229
	200	10.723	2793.715	38.916
	300	7.856	2807.556	44.163
	500	11.839	2834.625	59.996
	1000	20.471	2890.329	74.643
	2000	35.819	3151.934	297.002
	3000	43.441	3575.560	719.294
	5000	103.884	5060.743	1617.220
	10000	251.300	6970.552	3371.100
Disabled	1	2.010	2733.255	33.397
	2	0.988	2737.815	36.970
	3	0.721	2738.298	37.989
	5	0.975	2739.421	36.784
	10	1.202	2743.535	36.069
	20	2.417	2753.222	44.454
	30	2.357	2761.296	39.715
	50	4.461	2774.450	42.657
	100	7.455	2797.594	55.367
	200	12.510	2835.025	54.855
	300	11.632	2869.761	47.060
	500	17.792	2925.906	92.629
	1000	30.447	3018.763	340.850
	2000	71.718	3044.635	362.763
	3000	107.271	3979.649	933.729
	5000	176.686	5166.118	2160.781
	10000	386.582	6977.249	3247.123

The goal of next batch of tests is 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.

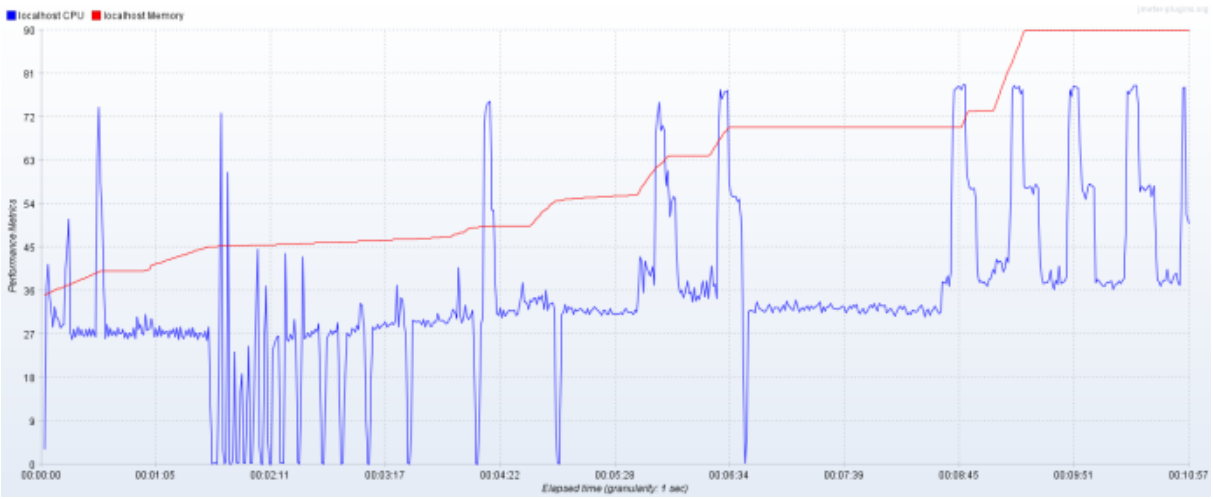

Figure 4.3: JMeter Profile: Cache enabled, 1 user, 1 iteration, 1 year

Figure 4.4: JMeter Profile: Cache disabled, 1 user, 1 iteration, 1 year

From the results in Table 4.2, we can 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. The values of the sets 5000 and 10000 appear as an exception to this but, as shown in Figure 4.5 and Figure 4.6, they experienced heavy GC activity, similarly to the previous batch, which is the most likely reason for the discrepancies.

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 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.

Table 4.2: Results for queries spanning one year with 50 iterations

Cache	Persons	Total Time (s)	Average Time (s)	Average RAM Used (MB)	Average RAM Variation (MB)
Initialised	1	19.366	0.387	3457.016	36.805
	2	16.829	0.337	3457.382	36.700
	3	16.057	0.321	3457.791	37.051
	5	17.157	0.343	3458.224	37.400
	10	18.105	0.362	3458.372	37.333
	20	21.134	0.423	3458.601	37.321
	30	23.889	0.478	3458.801	37.182
	50	30.367	0.607	3459.254	37.217
	100	47.085	0.942	3473.772	36.171
	200	82.356	1.647	3519.314	37.304
	300	113.381	2.268	3584.275	36.686
	500	181.367	3.627	3676.590	36.886
	1000	340.223	6.804	3860.401	37.075
	2000	717.532	14.351	4585.123	338.783
	3000	1150.150	23.003	5118.858	1070.943
	5000	2695.980	53.920	5555.234	2711.360
	10000	6616.587	132.332	5841.983	3741.126
Enabled	1	20.776	0.416	2684.038	36.766
	2	17.049	0.341	2685.895	36.819
	3	16.149	0.323	2687.006	37.064
	5	16.530	0.331	2688.613	37.433
	10	18.547	0.371	2690.535	37.302
	20	21.876	0.438	2697.173	37.337
	30	23.879	0.478	2702.453	37.258
	50	31.129	0.623	2712.065	37.208
	100	48.397	0.968	2742.733	36.212
	200	80.861	1.617	2809.977	37.326
	300	104.926	2.099	2890.180	36.558
	500	169.477	3.390	3008.635	37.463
	1000	331.129	6.623	3232.565	37.955
	2000	680.497	13.610	4171.596	345.524
	3000	1066.472	21.329	4547.130	1006.650
	5000	2500.770	50.015	5024.984	1807.822
	10000	7077.803	141.556	5856.663	2904.245
Disabled	1	22.611	0.452	2702.758	35.966
	2	18.363	0.367	2703.564	36.772
	3	18.301	0.366	2708.825	37.336
	5	20.131	0.403	2722.670	37.613
	10	25.076	0.502	2757.610	37.194
	20	37.115	0.742	2870.767	39.073
	30	46.826	0.937	3057.510	39.279
	50	68.323	1.366	3308.681	39.055
	100	127.365	2.547	2880.836	56.390
	200	288.822	5.776	3440.992	71.490
	300	424.475	8.490	3293.553	76.648
	500	657.035	13.141	3285.911	94.745
	1000	1322.194	26.444	3424.291	187.270
	2000	2742.310	54.846	3870.260	615.717
	3000	4197.830	83.957	3488.924	1511.238
	5000	8184.917	163.698	4079.103	2146.035
	10000	21168.830	423.377	6195.114	2435.996

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

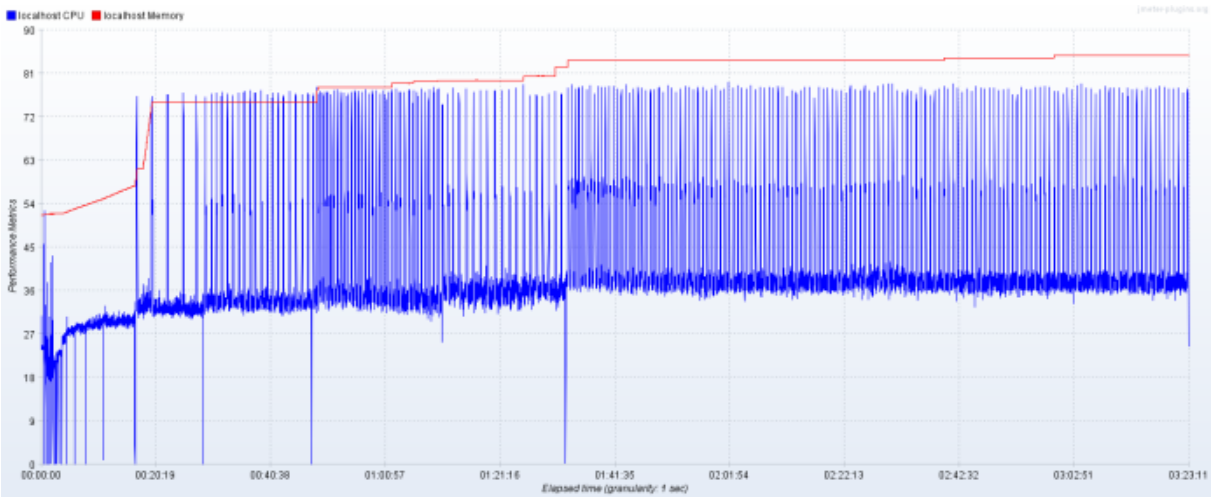

Figure 4.5: JMeter Profile: Cache initialised, 1 user, 50 iterations, 1 year

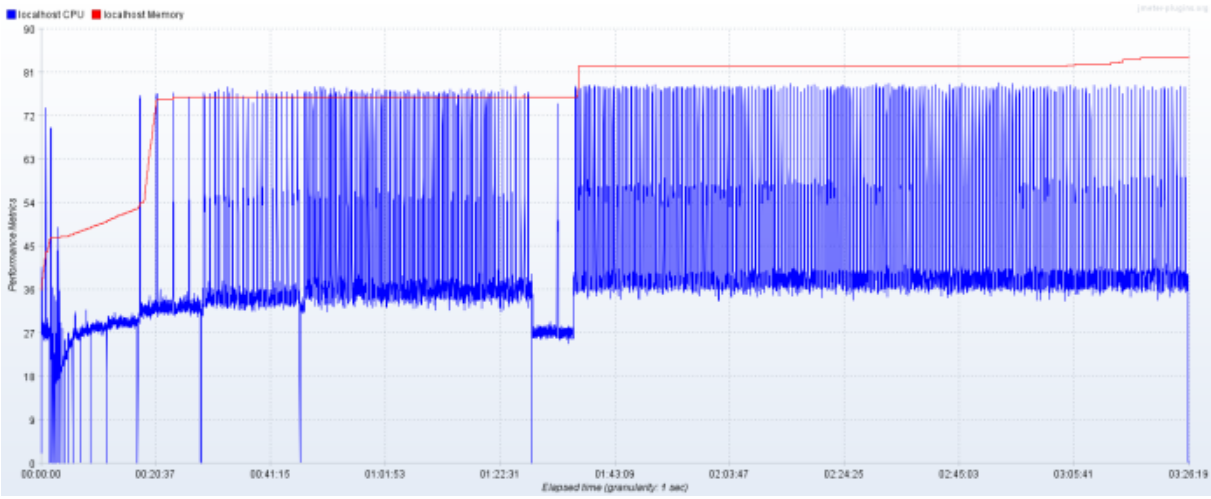

Figure 4.6: JMeter Profile: Cache enabled, 1 user, 50 iterations, 1 year

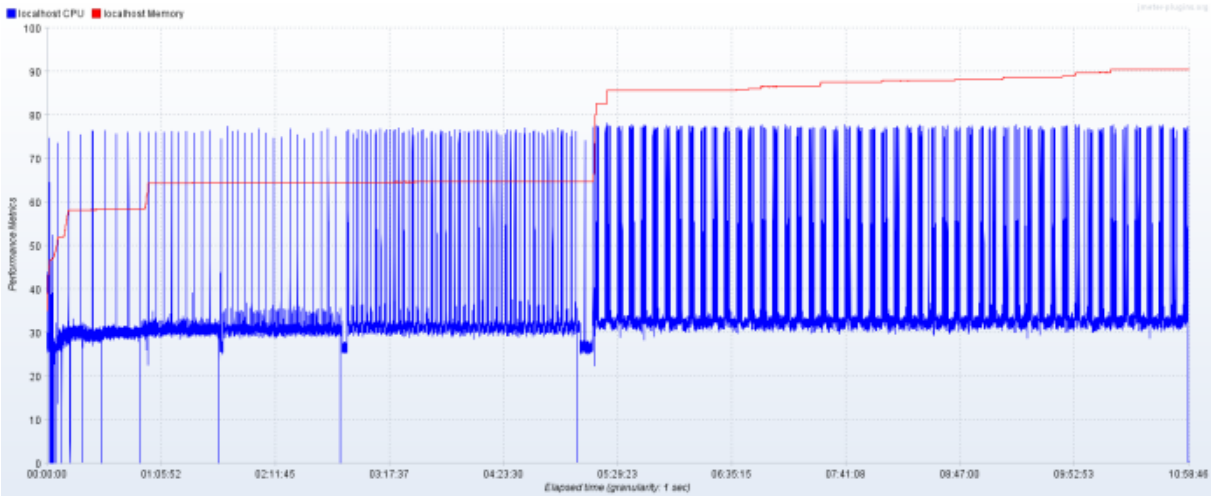

Figure 4.7: JMeter Profile: Cache disabled, 1 user, 50 iterations, 1 year

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 in Table 4.3, it is shown that collecting the availabilities of a single week takes a very short amount of time, when the cache is *enabled*. RAM variation 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.

Table 4.3: Results for queries spanning one week with 50 iterations

Cache	Persons	Total Time (s)	Average Time (s)	Average RAM Used (MB)	Average RAM Variation (MB)
Enabled	1	2.145	0.043	2637.064	8.454
	2	1.725	0.034	2638.348	7.978
	3	1.404	0.028	2637.649	8.361
	5	1.654	0.033	2639.402	8.221
	10	1.842	0.037	2641.263	8.855
	20	2.961	0.059	2647.946	9.942
	30	2.792	0.056	2653.176	9.641
	50	4.612	0.092	2663.949	10.828
	100	7.435	0.149	2680.636	14.461
	200	11.526	0.231	2706.138	19.266
	300	8.670	0.173	2729.544	21.832
	500	12.791	0.256	2757.330	20.085
	1000	21.546	0.431	2836.078	14.493
	2000	37.364	0.747	3263.079	24.316
	3000	46.550	0.931	3680.787	45.339
	5000	98.175	1.963	4421.276	104.331
	10000	251.446	5.029	5782.548	341.885
Disabled	1	4.441	0.089	2677.808	19.989
	2	3.239	0.065	2677.301	19.486
	3	3.417	0.068	2680.074	22.855
	5	4.829	0.097	2678.963	20.890
	10	7.955	0.159	2681.614	13.952
	20	17.480	0.350	2749.435	23.198
	30	24.539	0.491	2905.778	18.732
	50	41.606	0.832	2815.899	16.482
	100	85.150	1.703	3080.180	24.990
	200	194.625	3.893	3369.529	51.991
	300	298.961	5.979	3307.335	60.235
	500	492.867	9.857	3329.938	73.074
	1000	990.642	19.813	3427.881	185.707
	2000	2061.357	41.227	3500.046	314.506
	3000	3064.699	61.294	3170.127	491.078
	5000	5293.336	105.867	3299.812	812.896
	10000	11089.454	221.789	4266.157	1070.967

Concurrent users

The tests shown by Figures 4.10 and 4.11 were 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 Figure 4.10 to Figure 4.3, 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,

Figure 4.8: JMeter Profile: Cache enabled, 1 user, 50 iterations, 1 week

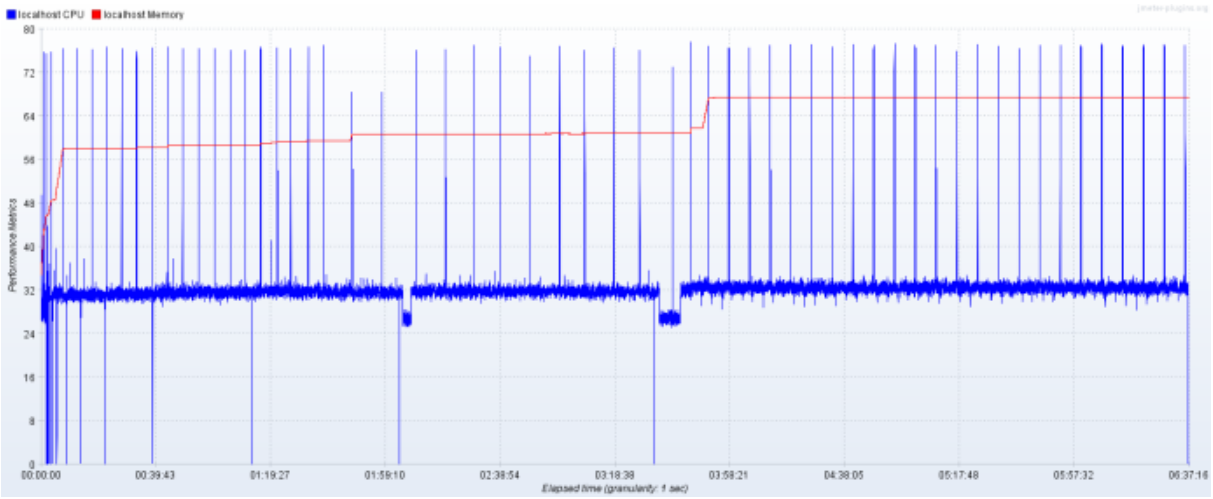

Figure 4.9: JMeter Profile: Cache disabled, 1 user, 50 iterations, 1 week

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.

Regarding Figure 4.11, when compared to Figure 4.8, it appears to show the concurrent requests to be processed slightly slower and to consume memory more quickly. Once more, GC activity appears to be significantly higher when 2 users are performing concurrent requests.

Conclusions

With these results it is possible to conclude that our solution is able to fulfil the performance requirements described in Section 3.1. From Section 3.4.5, we know that the server responds right after storing the query's cursor into the user's session. From the average time in the first row of Table 4.3, we can conclude that the server is able provide this feedback immediately.

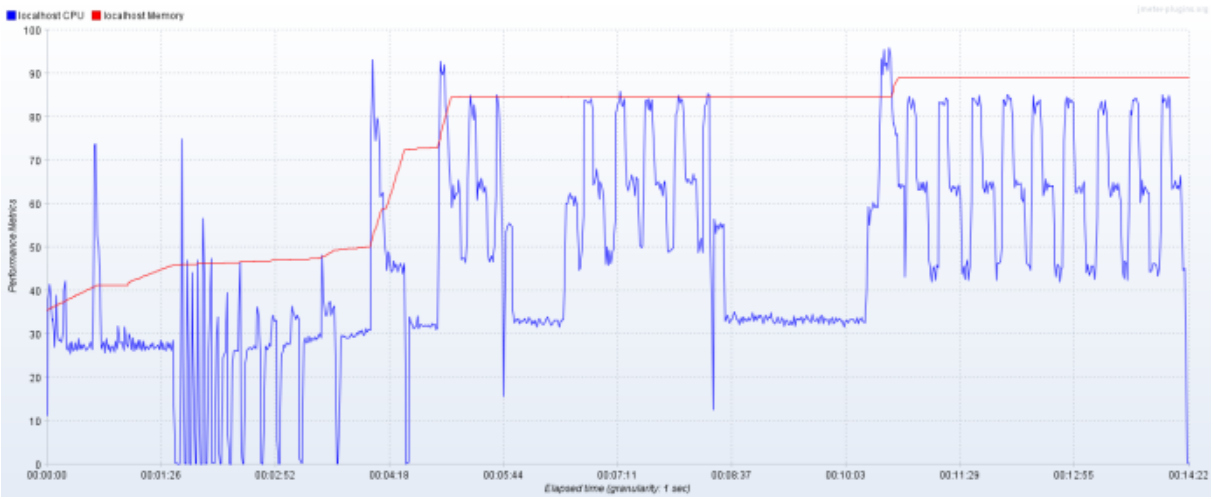

Figure 4.10: JMeter Profile: Cache enabled, 2 users, 1 iteration, 1 year

Figure 4.11: JMeter Profile: Cache enabled, 2 users, 50 iterations, 1 week

When using cache, the time taken to collect the results is generally proportional to the number of schedules requested, 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 chapter, we conclude this document with a short summary of our work, followed by a description of our contributions and lastly we leave our suggestions for future work.

5.1 Summary of the Dissertation

The work developed in this dissertation was motivated by the difficulties in scheduling activities relevant to the academic community at IST, which lead to problems such as overlapped or unfinished activities and unattended classes or events. Most information concerning classes and evaluations schedules is available in the Fénix system, but there are currently no mechanisms to cross such information in order to provide the necessary information for the scheduling of new activities. Existing calendar/scheduling systems do not provide an effective solution for this problem, specially considering that none of them supports diffuse activities, such as projects or studying for exams.

In this context, we defined as our goal the creation of a platform that would take advantage of the information stored in Fénix in order to allow the scheduling of activities considering the availabilities of the potential participants. To this end, we elicited the requirements and designed the platform with those requirements in mind. The developed platform, Kronos, allows to visualize the availability of the intended participants when creating an activity, supporting both activities with a well-defined schedule and diffuse activities. Kronos' domain model is inspired in Fénix's, while remaining adaptable to other organizational structures. The platform has a layered architecture, which includes a dedicated Knowledge module to infer the availabilities of a participant from his schedule and a module that allows Kronos to connect to external systems, such as Fénix, to collect scheduling data. Client applications communicate with Kronos through a REST API, as it is the case of the web application we developed for the purpose of testing and demonstrating our solution.

In order to assess if our solution would be appropriate in a realistic scenario, we measured the capacity that would be necessary in terms of CPU and RAM usage, and the performance of the system

in terms of latency and throughput. For this, we profiled Kronos by running multiple batches of tests with different settings, namely the number of participants and time span selected, the caching strategies used and number of concurrent users.

5.2 Contributions

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 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 requisites 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.

5.3 Future Work

While our solution successfully fulfils 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.

5.3.1 Improvements

Security and availability were not the focus of the development of our solution and still need to be addressed. It 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**: The current scrollable cursor implementation only allows to collect the availabilities of a day at a time for each user session, forcing each user's query requests to be handled sequentially even on separate threads. By adding a task queue to the cursor, where each task consists in collecting the availabilities of a day, the cursor could take full advantage of the multiple CPU cores;
- **Calculate occupations only for necessary activities**: When calculating the occupations, all activities of the entire year are considered. The algorithms could be changed to first consider only activities in the selected time span, and then all activities that overlap any activity being calculated;

- **Reuse data structures for optimized memory consumption**: As discussed in Section 4.5.3, a significant part of the memory consumption is due to a very large number of objects being created and discarded at each query. By identifying the single-use data structures responsible and replacing them with reusable ones, the memory efficiency would increase and the GC would not use as much CPU;

- **Split Business and Presentation into two separate processes**: In order to achieve better security and availability, by protecting the system from request flooding caused by usage peaks or deliberate attacks, the system should be split into two separate processes. The architecture, described in Section 3.3, is already designed to allow this by adding a service layer between the Processors and Presentation layers, with the purpose of transmitting the serialized DTOs between those two layers.

5.3.2 Extended Functionality

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**: Loading schedule data from external sources, such as iCal files, is a feature that was considered important by the questionnaire respondents and would add to the usefulness of the system;

- **Automatic schedule generation**: With the aid of Kronos' capabilities, a system could be developed to generate evaluations schedules with a balanced workload, taking into consideration the number of lessons required for each evaluation and project deliveries. Such a system could also generate optimal schedules for classes, by taking into consideration factors such as room capacity and location;

- **Schedule prediction based on history analysis**: Due to the fact that class and evaluation schedules must be set before the students are enrolled, it becomes necessary to predict student enrollments in order to minimize overlaps of classes, evaluations and projects. By profiling each student's academic history, such as their grades and completed courses and comparing to previous students, a system could identify trends and generate predictions for their next enrollments, which would allow a more effective use of Kronos for planning each semester's schedules.

Bibliography

[1] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley series in object-oriented software engineering. Addison-Wesley, 1997.

[2] M. Fowler. *Patterns of Enterprise Application Architecture*. A Martin Fowler signature book. Addison-Wesley, 2003.

[3] Serge Goldstein. WASS: Web Appointment Scheduling System. Technical report, Princeton University, 2013.

[4] D. C. Smith. The Effects of Student Syndrome, Stress, and Slack on Information Systems Development Projects. *Issues in Informing Science and Information Technology*, 7:489–494, 2010.

A

JMeter Graphs

A.1 Cache Modes and Sequential Queries

A.1.1 1 Iteration, 1 Year

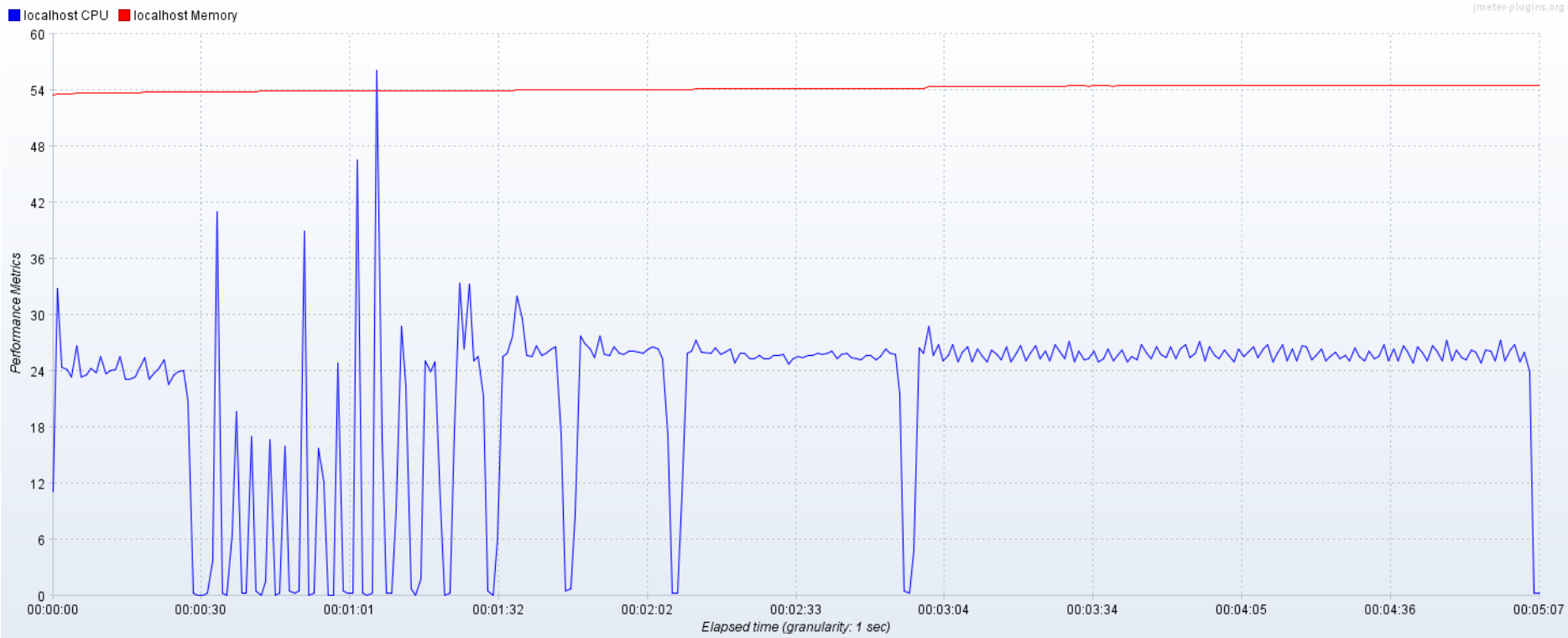

Figure A.1: JMeter Profile: Cache initialised, 1 user, 1 iteration, 1 year

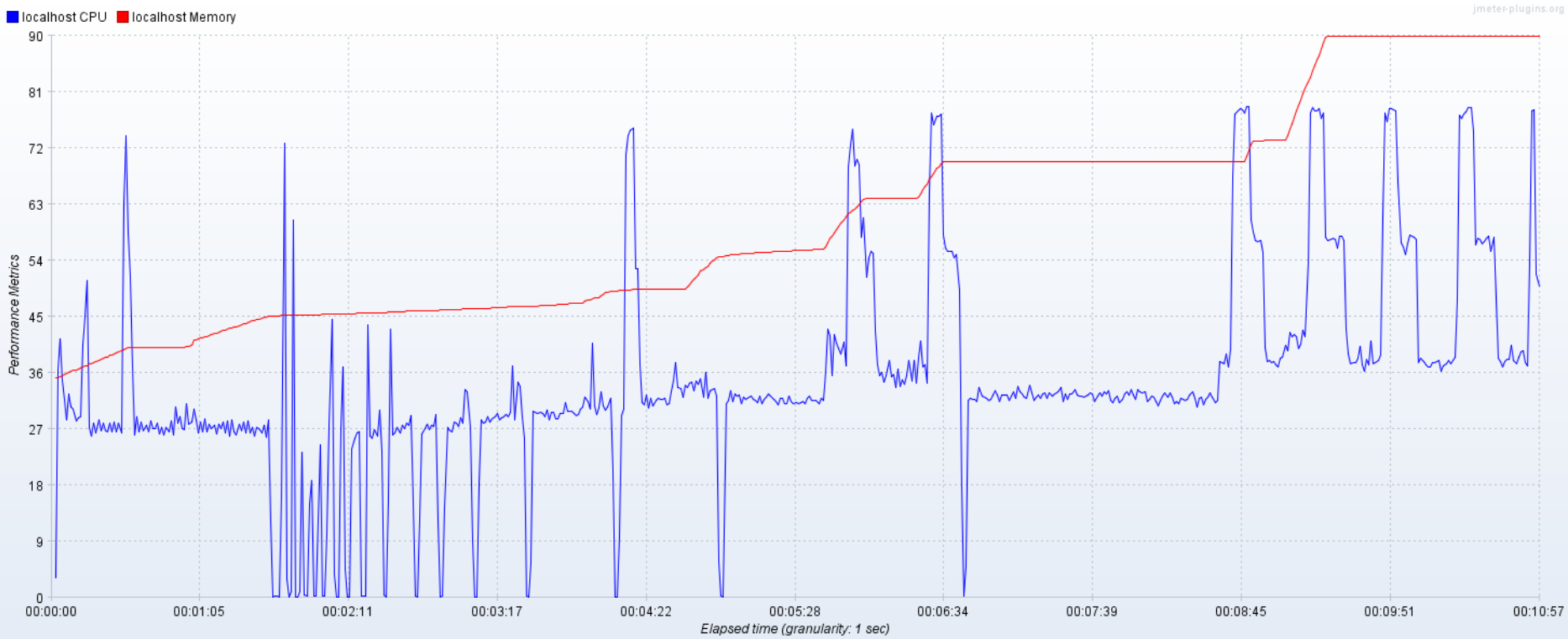

Figure A.2: JMeter Profile: Cache enabled, 1 user, 1 iteration, 1 year

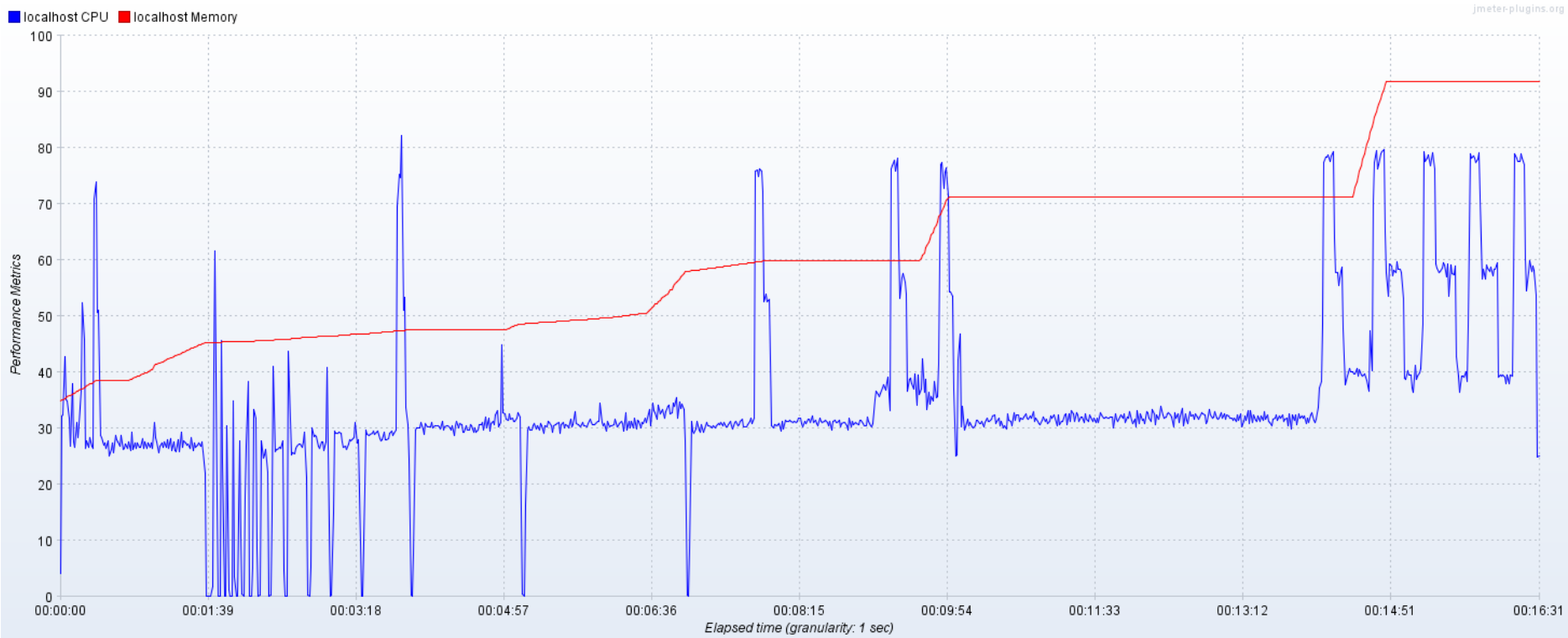

Figure A.3: JMeter Profile: Cache disabled, 1 user, 1 iteration, 1 year

A.1.2 50 Iterations, 1 Year

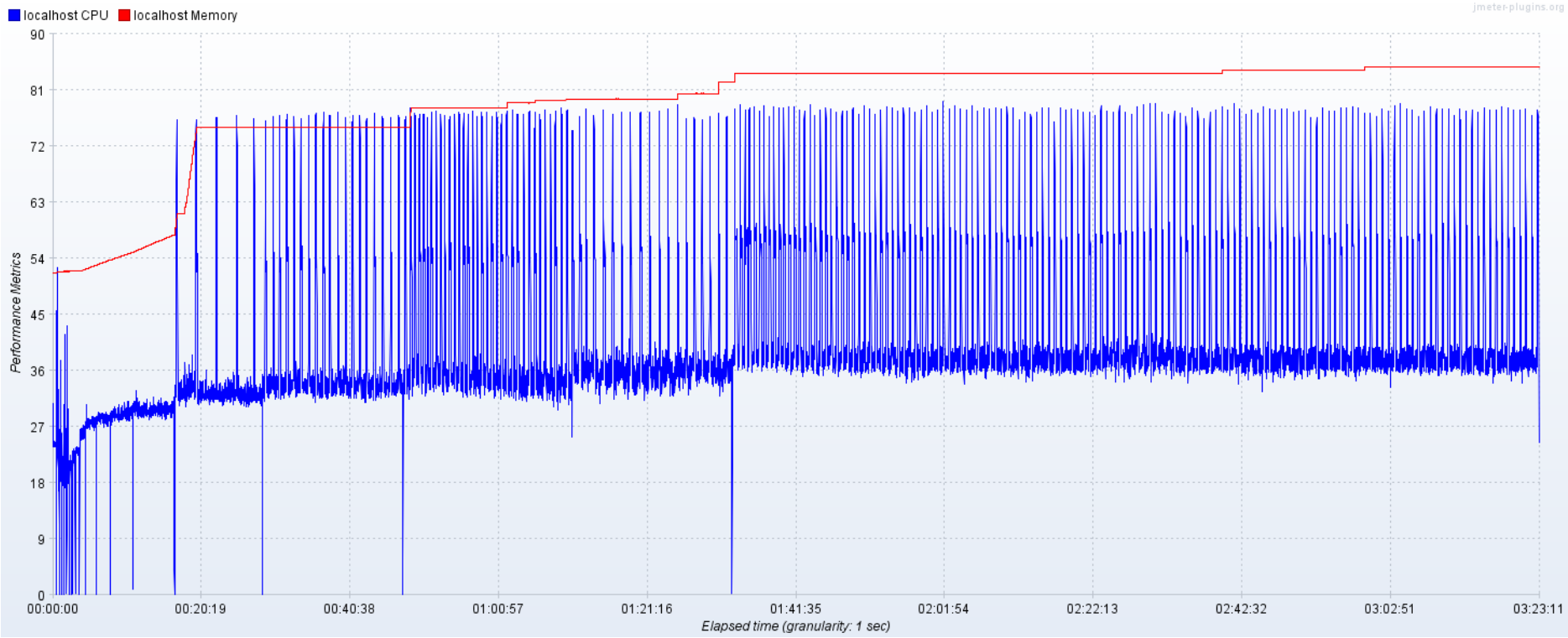

Figure A.4: JMeter Profile: Cache initialised, 1 user, 50 iterations, 1 year

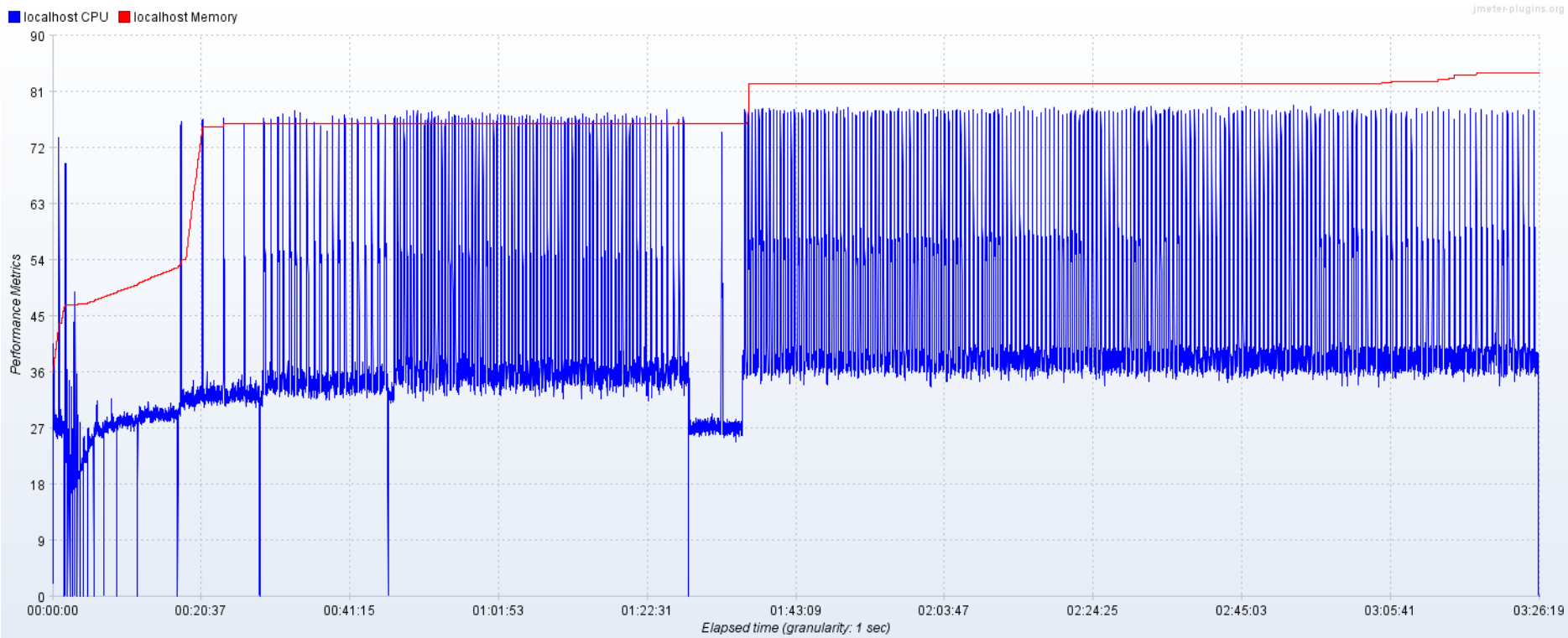

Figure A.5: JMeter Profile: Cache enabled, 1 user, 50 iterations, 1 year

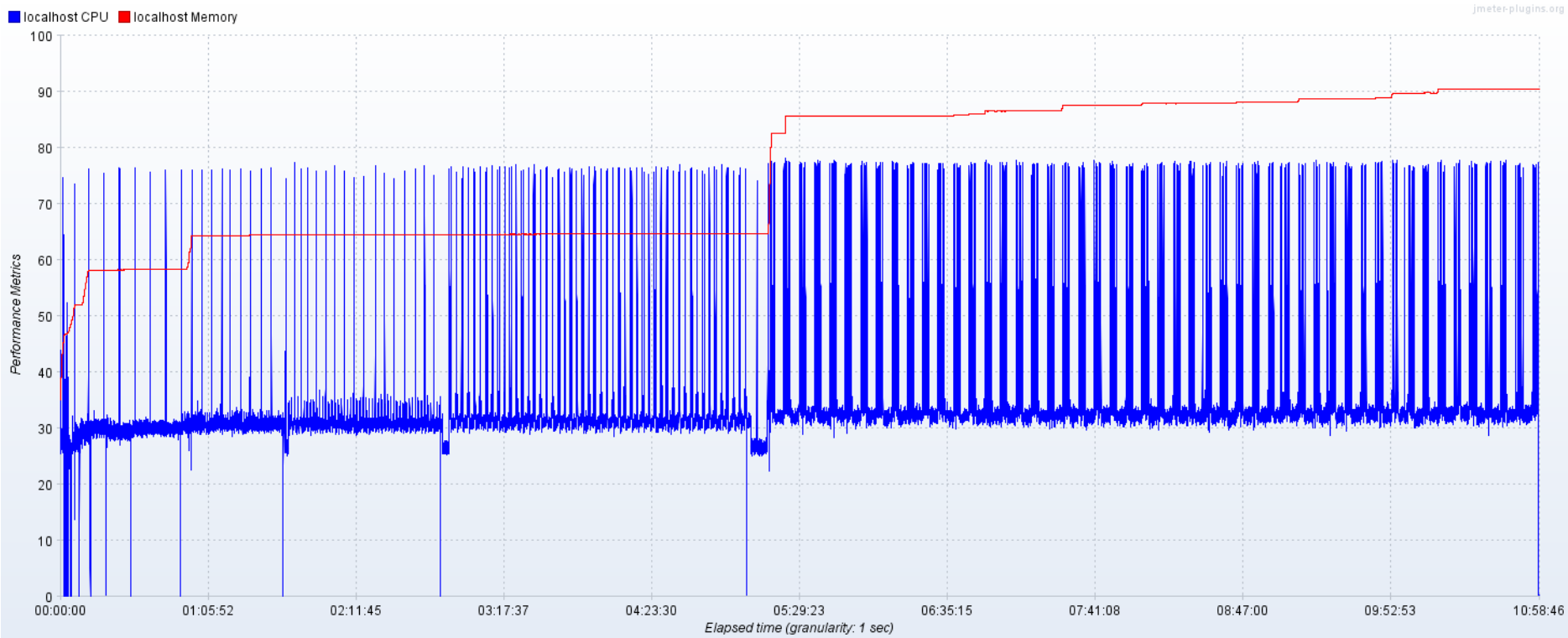

Figure A.6: JMeter Profile: Cache disabled, 1 user, 50 iterations, 1 year

A.1.3 50 Iterations, 1 Week

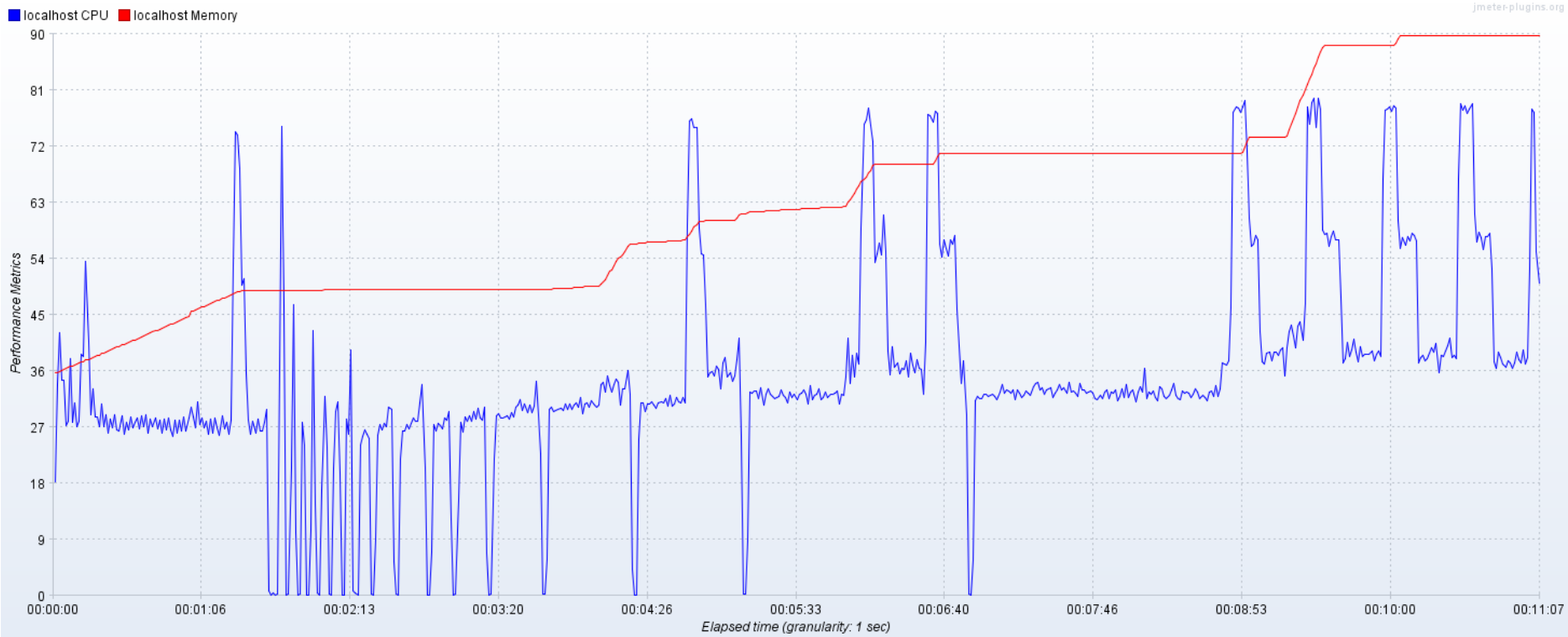

Figure A.7: JMeter Profile: Cache enabled, 1 user, 50 iterations, 1 week

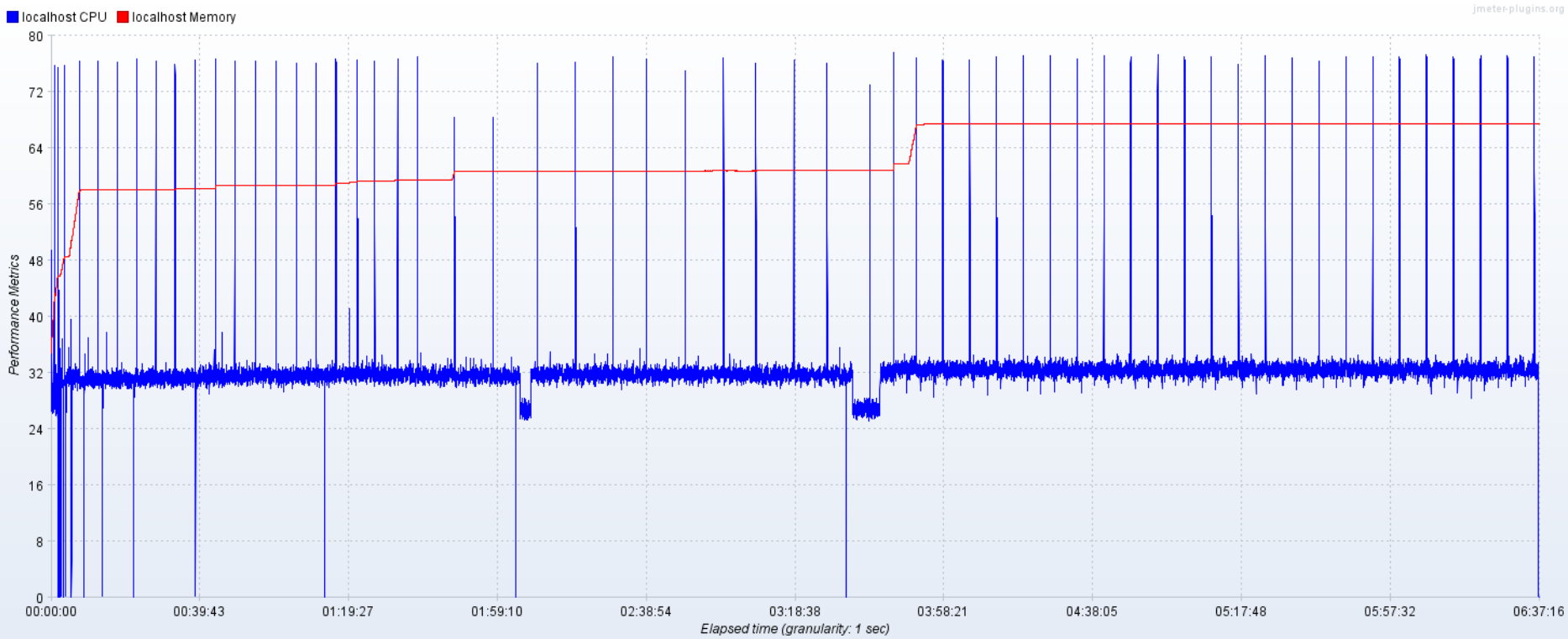

Figure A.8: JMeter Profile: Cache disabled, 1 user, 50 iterations, 1 week

A.2 Concurrent users

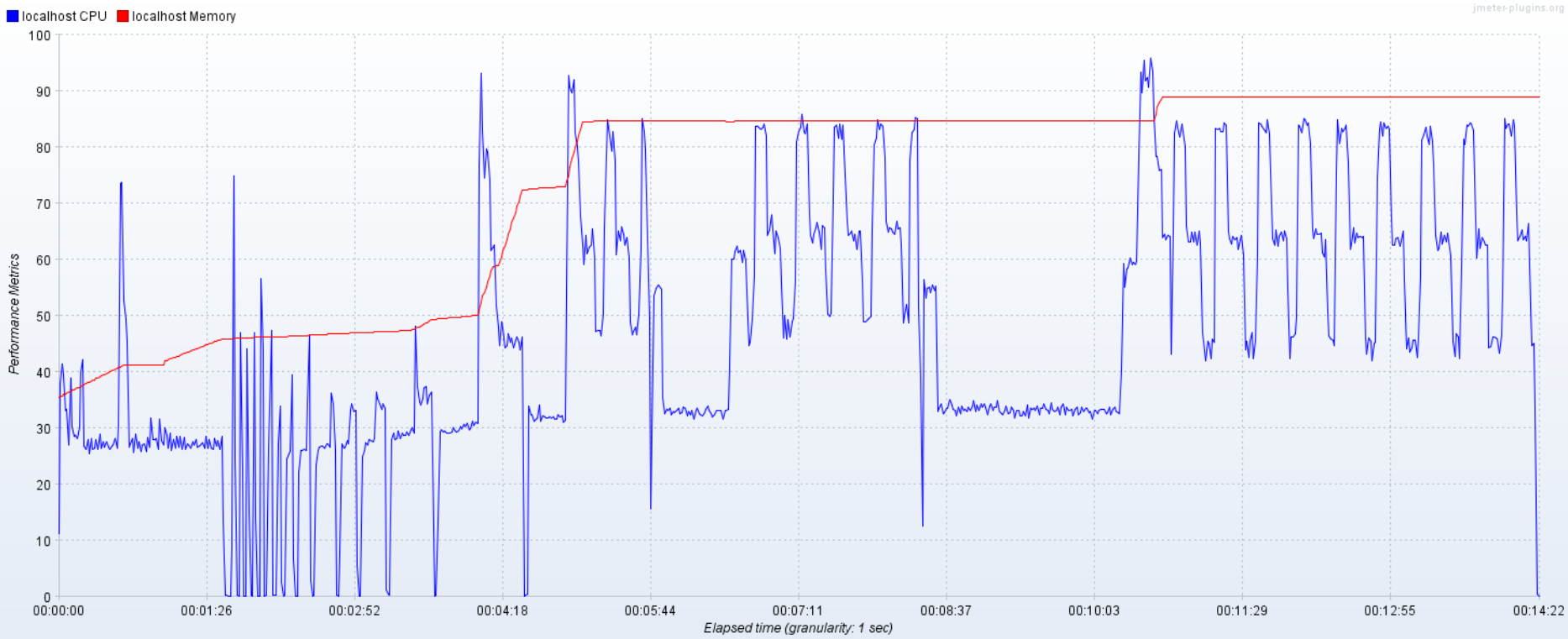

Figure A.9: JMeter Profile: Cache enabled, 2 users, 1 iteration, 1 year

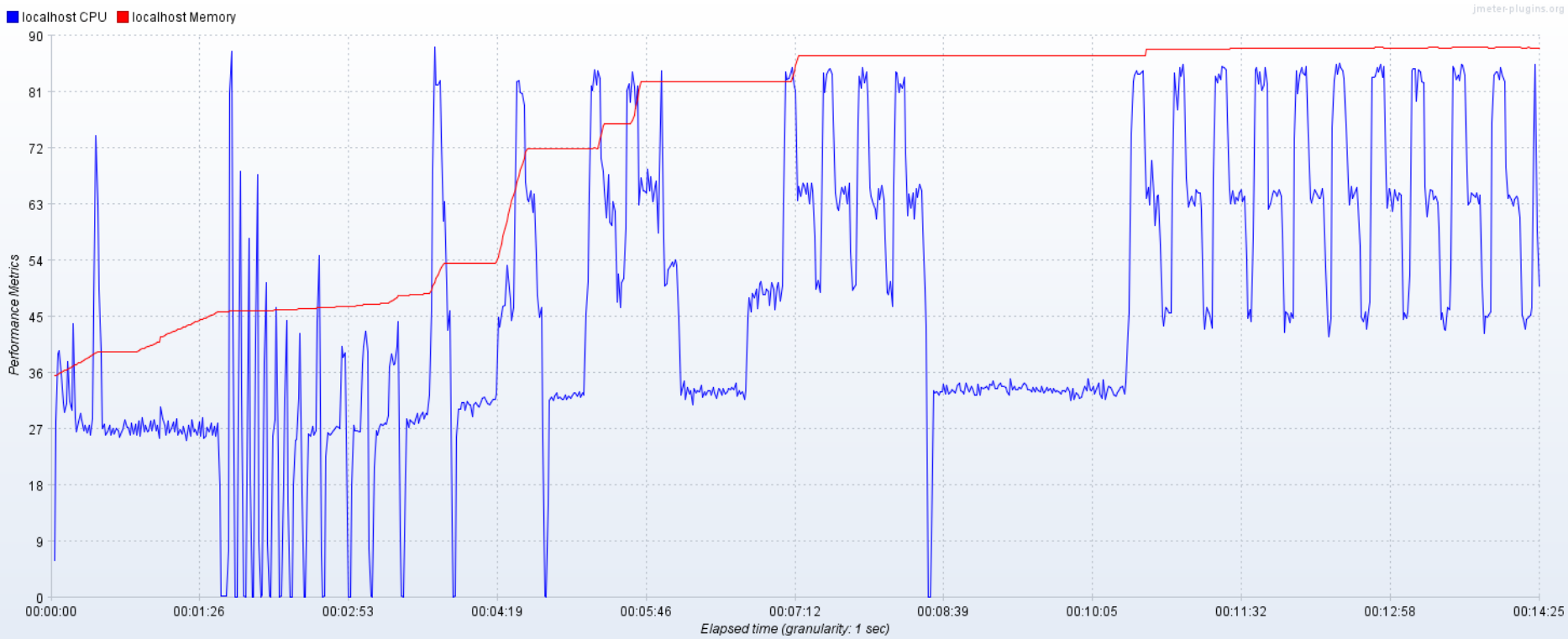

Figure A.10: JMeter Profile: Cache enabled, 2 users, 50 iterations, 1 week