SIGA: Integrated Queue Management System

Gonçalo António Rendeiro da Silva, *Student 67492, MEEC IST*

***Abstract*—Academic services at Instituto Superior Técnico are currently managed with manual ticket dispensers. After obtaining a ticket, customers wait their turn in one queue, devoid of waiting time estimates. Staff is unaware of the growth of this queue. Service operation activity is not recorded.**

With the goal of improving and modernize these services, a queue management product, the SIGA System, is designed and implemented in this work. It provides more information to both customers and staff while keeping record of all service related activities. It is adaptable to other contexts and can integrate with other existing systems (e.g. authentication, CRM).

***Index Terms*—Queue Management, Ticket Dispenser, Backoffice Interfaces, Web Application, Mobile Integration**

I. Introduction

Numbered tickets, served by ticket dispensers, are probably the simplest existing technology for managing waiting lines. A staff member operating a queued service can simply call the next ticket aloud and register the last called number. Alternatively, this same operator can press a button that makes a speaker signal the call and a LED display to show the ticket number being called. This latter example is representative of the current queue management systems deployed in the academic services at Instituto Superior Técnico, where only one queue is formed by the customers, independent of the issues they might want to solve.

With this current technology the waiting customers have no way to know when their ticket is about to be called. They can only look at the current number in the LCD display, and make an educated guess by watching its progress, not being automatically of the current average time estimate. This forces them to wait near the service, possible for long, or otherwise they risk losing their turn.

Likewise, there is no means of providing automatic feedback to the staff about current queue growth or about the effectiveness of their queue operation, based on tickets dispensed and customers served.

No record of the overall activity of the services is taken. Recording service activity data in the long run is useful to detect patterns in the service operation, like periods of higher affluence of customers, periods of lower service efficiency or any others patterns that might be found. By having this bulk data, its analysis could pinpoint the weak points of the system and where to act in order to improve the service.

Although the current technological state of our school's service management tools is not the most advanced, this does not reflect the state of currently existing queue management solutions. Several entities, like hospitals or public services, already have queue management solutions that are more sophisticated, dividing their customers in several queues related to their issues, and providing them with average wait time estimates. Several queue management solutions can be found across the web, such as Sedco solutions[1], Qminder[2] and Lonsto solutions[3], presenting varying functionality. From these three, Qminder is the solution that best fits our problem, and the only one to disclose its pricing, which is placed at 250 dollars per month and per branch. For the three academic services at IST, this would ammount to 750 dollars per month.

An internally developed system has several advantages. First, being a tool developed in IST, it can be used by our community with pedagogical purpose: through continuous iterations and improvements, future interested students can contribute to this product and learn with it. Second, the control of costs and functionality shifts to our side, and by developing it we can better control its cost-effectiveness, a part of our system requirements, along with functionality needed for systems integration (e.g. authentication, user databases, CRM). Last but not least, our academic services and current infrastructure also permits deployment of this system to be tested and improved with real service operation data. So, after proven in our environment, the ability to sell the developed system as a product to interested entities is also a plus.

As such, with the initial goal of optimizing our school services and provide a much better experience both for customers and staff, a server-based integrated queue management system is designed and implemented to be deployed in three academic services of IST.

This comprises interfaces for the backoffice operation in the form of a web application, to be used by the staff, providing information of queue growth upon ticket dispensing, and the ability to call tickets from several queues. Staff is responsible for configuring (creating, deleting or editing) the possible queues.

These backoffices work along with a ticket dispenser kiosk which enables users to get numbered tickets for different types of queues that reflect the issues they want to solve, and provides them with average time estimates for each queue. They can also consult the current queue status on a display near the service, and remotely if needed.

Also, upon integration with an existing user database, authenticated users are allowed to obtain virtually dispensed tickets and get notifications about queue status on a mobile application.

Software is selected such that integration with existing user databases or existing Costumer Relationship Management (CRM) software is feasible.

Being a server based solution, the operations that occur in this system are properly stored in a database, and thus service activity operation is recorded and can be queried any time.

[1] http://www.sedco-online.com/en/content/queuing-and-routing

[2] https://www.qminderapp.com/

[3] http://www.lonsto.co.uk/pc/6/queue-management/ticket-controlled-queuing-systems.html

In overview, in this project we developed a web infrastructure providing backoffice and client interfaces and physical and virtual ticket dispensing that together support a queue management system offering:

1) Support for different services and different queues in each service;
2) Interface for service operators;
3) Interface for users to get tickets;
4) Service activity logging to enable performance assessment and other types of reports;
5) Basic visual statistics;
6) Enable the future implementation of additional services that may require authenticated users (e.g. CRM).

Although the solution developed during this work stems from the specific need of our school, its design and implementation kept the broader vision of achieving a general-purpose product of potential interest for any service with waiting lines.

II. Problem Statement

At Instituto Superior Técnico, the academic services store no information on how they manage service customers: one queue is formed, numbered tickets are dispensed to customers who in turn are called by their arriving order. We now exemplify the current lack of information for the two parties concerned (customers and staff), thus bringing to light how its existence could improve their experience and the efficiency of operation.

Weighting in decreased efficiency, we have the lack of information on the operation side. Staff elements have no way to see which issues are on higher demand as the queue grows, thus cannot prioritize issues over others. At the end of the day, or month, or year, there is no way to account for statistics on service operation. This data, particularly if obtained for a long period, can be used to improve planning and thus bring efficiency to the operation of the service. Also, to increase efficiency and service quality, more information could be given to customers. Because they have no means to know their estimated waiting time, waiting near the service office becomes necessary.

Thus, the problem we propose to address in this work is that of the uninformed queue management. As pointed out in [1], having detailed information on how queues are operating (e.g. user arrival time distribution, staff element productivity, etc) can lead to better modelling of the service operation, thus enabling better decisions towards its (multi-objective) optimization (e.g. customer waiting time, staff idleness, service utilization).

As an example, in a service with a first-come first-serve policy, queueing theory studies point the multiple-cashier single-queue style as the most efficient [1]. By dynamically prioritizing one queue over others and support multiple queues calling from different staff members, the multiple-cashier would just happen naturally in our school scenario, because only one physical queue would exist in practice. However, the study in [2], considering the social aspect of the problem and focusing on minimizing waiting times, concludes that parallel physical queues are the best solution. Other studies, also highlighting the social component that the operating staff brings to this question, debate on how visual feedback might be important for increasing staff efficiency [3] and what trade-off can be expected from changing the intensity of service [4][5].

With this in mind, and as before mentioned, we wish to develop a queue management system that could provide both the school and its students and staff with better information. Our work sets out from a practical standpoint, without restricting our system to a particular queueing theory: we wish to develop a configurable system, where any of the above mentioned theories and can be tested and fine tuned to the specific needs of the service where it is to be deployed. After deployment, it could be even used to test and find new theories for queue management improvement, based on the acquired data and configurable parameters.

We now proceed to describe the base features that this system should offer to its users.

The number of services and respective queues this system serves should be configurable by the technical administrator that installs the system in the service provider entity (e.g. in IST).

For the operations, it should provide the service with the possibility of calling tickets of different issues based on either a system's suggestion (from a list of possible heuristics or a default one) or by operator's choice of a certain queue. The term *queues* will refer to those different issues henceforth. An example: Queue A - Payments; Queue B - Enrolment; Queue C - Certificates. Staff members should be able to see real-time queue data to help them make an informed decision on which queue to call next user from.

In parallel, customers should be able to get a ticket for a given queue. Also, we wish to provide the customers with updated waiting time statistics, and even notifications to a mobile application that is linked with the service, which can dispense virtual tickets upon authentication and inform them when they are about to be called.

This system should record all the operation data for further analysis with the objective of pointing out possible improvements and better informing the staff on how to prioritize different matters (that is, manage several queues) in both real-time and specific periods of higher demand. It should also provide the staff with an easy to use back-office interface, and the customers with an easy to understand queue progress display.

Because this is a broad problem, present not only in our school, but in all kinds of services with users waiting to be called, we wish to develop a product that is customizable, making it also possible to integrate with other services and existing CRM systems and user databases.

III. Functional Requirements

Functional requirements capture the intended behaviour of a system, and thus, a way of providing a structured functional blueprint, useful for both developers and users. To capture the fuctional requirements, we capture the customers and clients main goals for better guiding the interfaces development from a user perspective.

A. System Actors and Their Goals

System actors and what they should be able to do with the system (their "goals") are now listed.

1) Staff:

- Super Admin
  - Login and logout to and from the super admin interface
  - Create/Modify/Delete Services for a generic service provider entity that installs this system (e.g. for our University: Post-Graduate Academic Service, Undergraduate Academic Service, International Mobility Service)
  - Define service open-hours (service management)
  - Define maximum number of queues for certain service (service management)
  - Define a logo for the tickets to be printed (service management)
  - Create/Edit/Delete staff members of any type
  - Associate a Kiosk/Ticket Dispenser to an existing service

  The Super Admin doubles as the technical administrator and maintainer of the whole system. This means he will have access to all the created data and used technologies. As an example, he may create a new user authentication system for SIGA or integrate it with an existing one.
- Service Admin
  - Login and logout to and from the service admin interface.
  - Give and remove service admin privileges to and from operators
  - View and edit Service Settings, use Operation Mode and visualize Statistics
  - Settings: Create/Edit/Delete Queues (e.g. Enrolment, Certificate Requirement, Grade Improvement, Others)
  - Settings: Select Service Session duration period (between normal office hours or manual)
  - Settings: Select heuristic for "next ticket to call" suggestion from:
    1) First-come,first-serve
    2) Minimize average wait-time for the service
  - Operation Mode: Can choose a desk and call tickets, performing the role of the Operator.
- Operator
  - Login and logout to and from the system. Upon login, operator is prompted to select a desk number
  - Call a customer (by system suggestion or from a queue) to his desk
  - Open or Close the service, that is, stops tickets creation

2) Customers:

- Kiosk Customer
  - Get a ticket for a queue that categorizes this customer's issue
  - Identified by the ticket, gets called by the service to solve that issue
- Authenticated Customer
  - Log in the appropriate service application, mobile or web, and get a virtual ticket for a queue categorizing this customer's issue.
  - Get notifications about that queue's progress, until called by the service to solve the issue, given that the customer used the given info to approach the service in time.

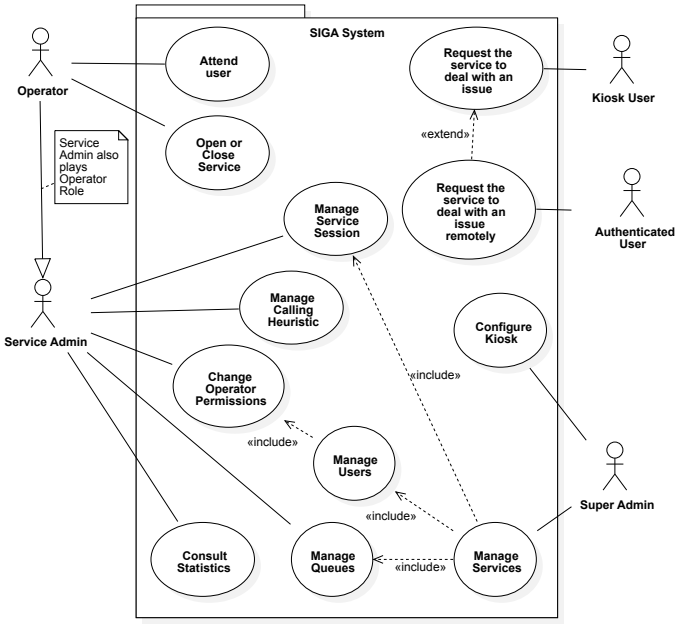

Fig. 1. Simple use case diagram for the SIGA system

B. Additional Functional Requirements

1) Records: On top of the functional requirements specified which were based on the use cases of the system, the system shall never really delete any information that was collected. As an example: all tickets created for the queues, even after queues are deleted through the interface, stay stored and accessible, for purposes of data retrieval for future analysis and automatic report generation .

2) Security Requirements: Also, the kiosk interface shall not have a direct internet connection, to prevent tampering from the customers side. It should never be able to permit clients to use it for other purposes besides getting tickets, or staff configuration.

The kiosk must also prevent clients that may request many tickets for the ill-purpose of wasting resources (e.g. paper), by having an acceptable (0.5-1s) time-wait cooldown between prints, besides blocking ticket printing request while printing the ticket.

The printing of tickets may only be authorized to the physical ticket dispenser or to a user that is authenticated.

3) Integration Requirements:

1) Authorization backend customization: staff should be able to login into back-office operation with already existing login back-end system.
2) App customers should be able to request remote tickets, with the app and respective notification service also using the previously integrated authentication backend.

C. Non-functional Requirements

This system should be easy to work with for both customers and staff (user friendly interfaces), customizable and deployable for different service provider entities (e.g. other universities, hospitals, etc.) and achieve cost-effectiveness.

It should be server-based, easy to configure and scalable, ideally enabling the remote deployment of client units that will self-configure upon server connection, making it easier to deploy in large organizations and extensible to provide interfaces to devices external to this system.

IV. Proposed Approach

A. System Architecture

Our solution will be server based, providing backoffice interfaces for the staff and client interfaces for the customers. An overview of this system's architecture is depicted in Figure 2.

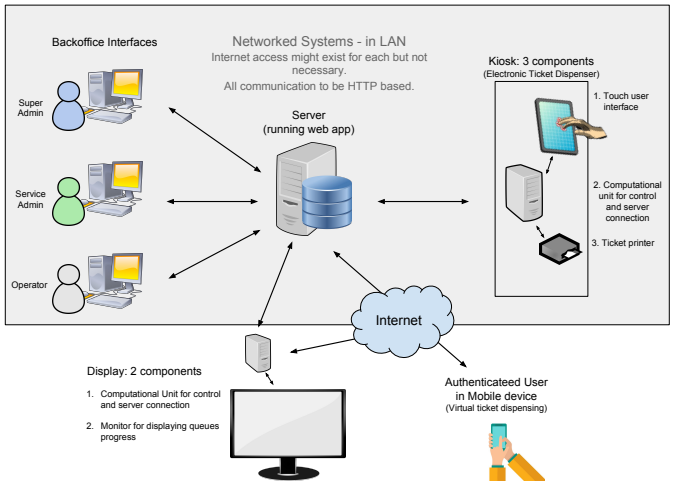

Fig. 2. System Architecture proposed approach

B. Server

The server shall be the main component of our system. It will be used to log all interaction with the system on a database, as well as enable the needed interfaces. In order for this to work, each component must have a corresponding computational unit to be able to establish a connection with the server. It will serve the multiple back-office and customer interfaces, for which we now present mockups and architecture.

C. Backoffice and Display

1) Backoffice: Our back-office interfaces will have three variants, one for each of the staff roles, for which we have made mockups: the super admin in Figure 3, the service admin inn Figure 4 and the operator in Figure 5

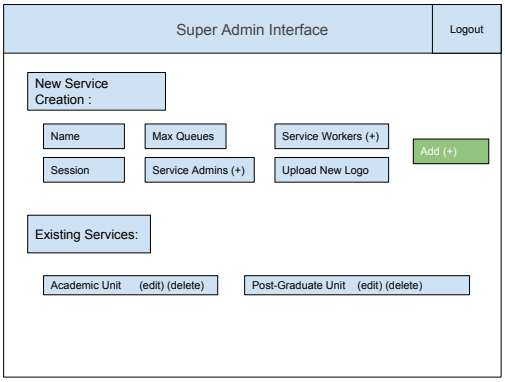

Fig. 3. Mockup of the super admin interface.

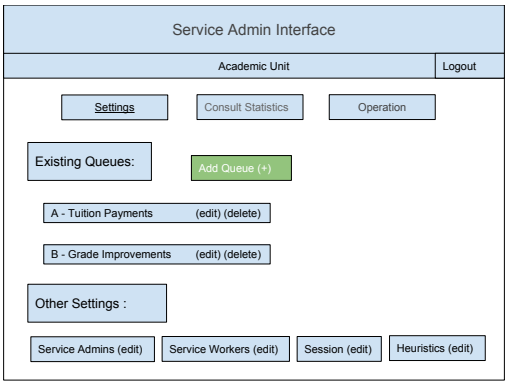

Fig. 4. Mockup of the service admin interface.

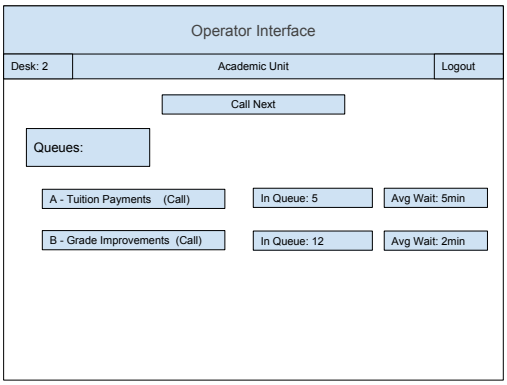

Fig. 5. Mockup of the operator interface.

2) Display: Customers who take a physical ticket will have a Display near the service that informs them on the current queue status. This display is actually a monitor or TV, connected to a computational unit that is fetching a specific web page for queues progress for this service from our server. A depiction of an the intended information for the Display interface is shown in Figure 6.

D. Ticket Dispenser

There will be two ways for customers to get tickets:the kiosk, where they select the intended queue from a touch-screen display interface; through a mobile application, from

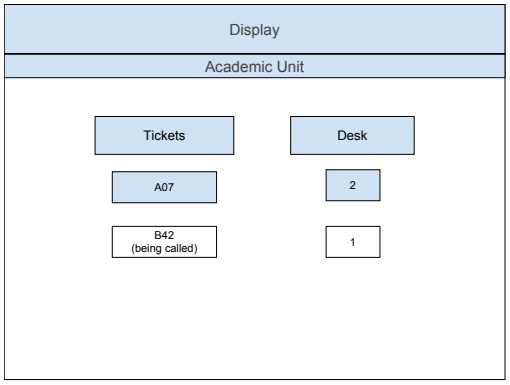

Fig. 6. Mockup of the Display interface.

where they can get a ticket for a queue, and receive notifications updating the status of that queue.

As one can see in Figure 2, the Kiosk will have three main components: a touch interface, a computational unit, and a printer. This computational unit will be responsible for interpreting the touch-screen interfaces and communicate them to the server (e.g. create ticket for queue A). It will also be responsible to interpret the server response and give order for the printer to print a ticket. The interface to be presented in the touch-screen is depicted in Figure 7, along with a printer and a ticket.

In 8, we see the configuration mode for the first time the Kiosk attempts to connect to a service: it presents the existing services, as portrayed by the ninth use case. A similar interface shall be displayed to the authenticated user before proceeding to the queues, and the first time a display is configured.

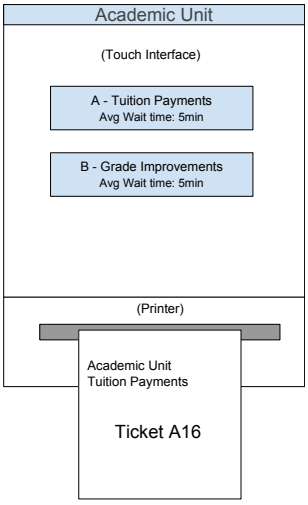

Fig. 7. Ticket dispenser representation, with touch screen for queue selecting and ticket being printed by an attatched printer.

The virtual ticket can be obtained through an application, as depicted in Figure 9. The several tickets obtained are also depicted. Note that, as previously explained in Figure 2, this user needs an internet connection in order to communicate with the server.

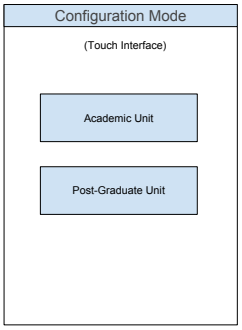

Fig. 8. Mockup interface for associating dispenser with one of the existing services.

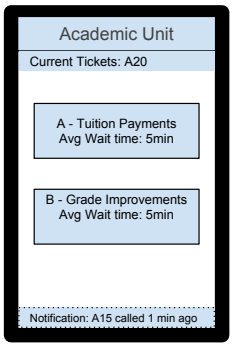

Fig. 9. Representation of the mobile application integration, allowing to request tickets and receive notifications. User has taken ticket A20 and received a notification on the last ticket called.

V. Implementation

A. Hardware Components

Before this project started, some hardware was already chosen and acquired. The acquired materials included:

Kiosk metal frames As in Figures 10 and 12). This frame is a national product, completely manufactured in Portugal, by Partteam[4].

Android Tablet This model, the SM-T550[5], comes with Android version 5.0.1 (Lollipop). Can be seen in Figure 10.

TMII-20 Epson Thermal Printers The Kiosk hols a TMII-20[6] into its printer compartment, as shown in Figure 12.

RaspberryPi models 2B and 3B[7], as depicted in Figure 11.

Having these items to build the kiosk, its architecture almost outlines itself.. A diagram outlining the intended interaction is presented in Figure 13.

B. Server Software

1) Used Web-Framework: The main framework used to develop this project is Django[8], version 1.9. This is a free high-level web-framework that aids the developer into building web applications faster. It uses Python[9], a high-level general

[4]http://www.partteams.com
[5]http://www.samsung.com/us/support/owners/product/SM-T550NZWAXAR
[6]https://www.epson.pt/products/sd/pos-printer/epson-tm-t20ii-series
[7]https://www.raspberrypi.org/products/
[8]https://www.djangoproject.com/
[9]https://www.python.org/

Fig. 10. Photo of the kiosk with fixed tablet running an application with blue background

Fig. 11. Picture of RaspberryPis used in the implementation.

Fig. 12. Photo of the kiosk from the back, with opened doors, where we see the tablet attached to the frame, the printer compartment.

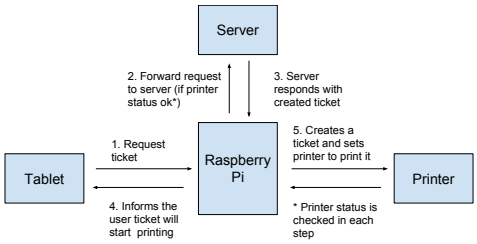

Fig. 13. Picture detailing intended kiosk components interaction

purpose language, which is interpreted and dynamic. One of the best advantages of Python, also present in Django, is the ease of install and usage of modules developed by the community. Another highlight, it is Django's good and extensive documentation [6], that is also backed by Python's own [7].

2) RESTful Web services in Django: To take advantage of the Android operative system pin mode, we opted to make an application. However, this is not as easy as making a web page as interface, directly accessed through the browser. Therefore, in order to have communication between server and tablet, we used Django Rest Framework[10](also known as DRF)

This framework lets us define a Web API (application programming interface), that is, URI endpoints that can transmit machine to machine information in JSON format. Endpoints were created to obtain details of services and their respective queues, along with an endpoint enabling ticket creation.

These endpoints will also be used to deliver notifications, as they will be the web services connecting the mobile user to the system.

3) Security considerations: To make sure the ticket creation endpoint is not tampered with (e.g. a user using the endpoit to create several tickets in the server, pretending to be a kiosk), a specific hash that needs to be set in each request is defined, and a certificate for enabling HTTPS is recommended. Also, the number of tickets creation per authenticated user must be limited.

4) Front-End: Back-office and Display Interfaces: Figures 14 and 15 depict the initial status of these interfaces, made with the help of TwitterBootstrap[11]. They are fully functional but the final design is not closed: full-fledged interface designs are still under development by the design team of our school. A first, almost closed design, was made fully functional with CSS3 Flexbox[12]. This design is presented in Figure 16). The display is also a web application, running on a Raspberry Pi that is connected to the internet.The final result is depicted in Figure 17.

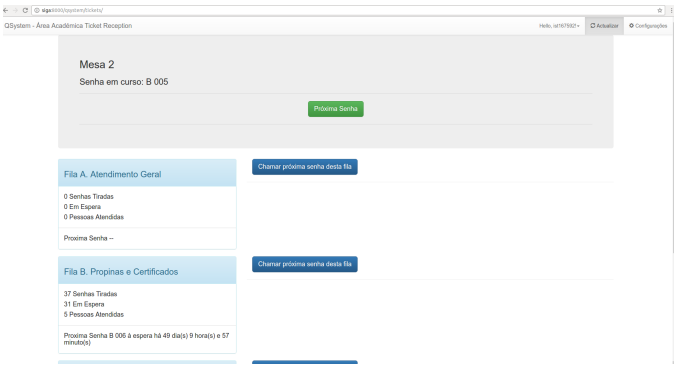

Fig. 14. Functional draft interface for tickets operation.

This was made using HTML/CSS with Flexbox, and AJAX calls to poll the server for updates on tickets status.

[10]http://www.django-rest-framework.org/
[11]http://getbootstrap.com/2.3.2/
[12]https://www.w3.org/TR/css-flexbox-1/

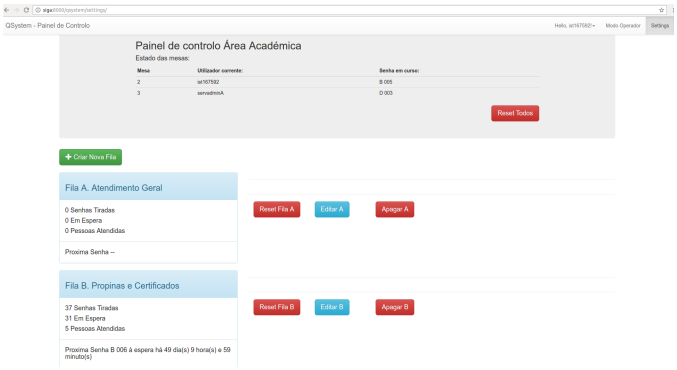

Fig. 15. Functional draft interface for settings operation.

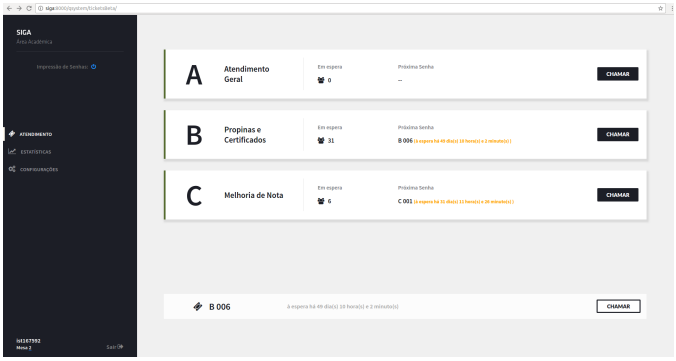

Fig. 16. Operation interface with a more advanced design, although still in a preliminary version.

Also, the right-pane is available to display information coming from an RSS feed. We used an open-source tool, Feednami[13] to integrate it.

C. Ticket Dispenser

1) Architecture: The architectural description is in figure 18

2) RaspberryPi to Tablet connection: The initial idea was that the tablet communicated with the RaspberryPi through inverse-tethering over USB. In short, the USB connection

[13]https://github.com/sekando/feednami-client

Fig. 17. Display. Next ticket for B queue has been recently called: shows in alternate color.

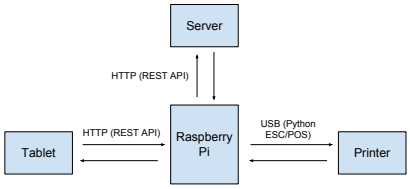

Fig. 18. Architecture overview of the Kiosk subsystem.

would mimic an Ethernet one, offering a private TCP/IP connection where HTTP would be supported.

With the need to occupy the USB port only for charging, we found other solution to provide the physical layer that enables HTTP between the tablet and the Raspberry: a private Wi-Fi Access Point network, generated and controlled by the Raspberry Pi, which can be programatically configured in the tablet, providing an out-of-the-box solution with these elements.

For security concerns, one can define a password for this network with WPA2 encryption[14]. This is so that one cannot connect to the created Wi-Fi AP pretending to be a tablet.

As one might note in fig.19, that depicts all the needed kiosk connections, there is no physical connection between the RaspberryPi and tablet. The orange cable is the USB cable that goes directly to the plugged transformer.

Fig. 19. Picture of the kiosk from the bac with connections made/

3) Printing Tickets: When handling ticket creation requests, our Python application invokes methods to communicate with the EPSON Printer through USB. Fortunately, an open-source Python library was found. This library is named Python ESC/POS[15]. Its most important feature used was sending an image to the printer. This library makes use of appropriate imaging libraries that rasterize images very quickly, taking advantage of GPU computations if possible. Our ticket, for purposes of increased customization (e.g. not being tied with the available printer text fonts), is an image. The ticket design was iteratively developed by the design team of our school.

When the user requests a ticket in the tablet, and the request is forwarded by the Raspberry Pi to the server, if everything goes well server-side we get a ticket response. From this ticket response we extract the following needed information to

[14]http://standards.ieee.org/getieee802/download/802.11i-2004.pdf
[15]https://github.com/python-escpos/python-escpos

Fig. 20. Pictures of digitally generated tickets with Imagick

construct an image: 1) Logo to use; 2) Service name; 3) Queue name; 4) Short Queue Name; 5) Date; 6) Hour; 7) Tolerance; This image is created with a bash script using Imagick[16], that receives these arguments and is called within the RaspberryPi request handler upon server ticket creation response. The final result of our image construction, can be seen in fig. 20. Its analogue counterpart produced by the printer is depicted in fig. 21.

Fig. 21. Tickets in their analogue version.

4) Printer Status: One important feature that during the development and at the time of writting was not present in Python ESC/POS was that of assessing printer status.

So, if printer status is abnormal, it will return a response to the tablet and the server listing the present errors, which may be the following: 1) Offline Mode (printer turned off); 2) Cover is Open; 3) Paper End; 4) Autocutter error; 5) Unrecoverable error; 6) Automatically Recoverable Error;

Two other types of errors, not pertaining only to the printer, are also checked for and sent: printer USB connection errors and server connection errors.

5) Tablet Software: the SIGA App: Our app was developed with the free Android Studio IDE, that easily allows the use of Android's Java API Framework [8] for Android application development. The end result for the main interface, both for Portuguese and English settings (as defined in a server running our Django application), is shown in Figure 22.

Some third party libraries for Android were used for the development of REST API communication on the tablet side, one using the others: Square's Retrofit using RxJava and GSON.

The main interface is the one clients will use: the list of queues for the selected service, as in Figure 22. This one updates to changes that occur in the back-office.

6) Security Considerations - Pinning Mode: This feature, present in Lollipop (Android 5.0) and above as task pinning, is of utmost importance for the correct operation of our system, as previously discussed. One can manually pin one application to the screen, which means that it will not leave the current screen view unless one presses two specific navigation buttons simultaneous, for a while. In our metal frame, such buttons are not accessible, making our app never leaving the screen. We add two protections to this one. First, the app has a boot listener, to self launch after the tablet boots. Second, because there is a slight delay between the OS boot and the app launch, a password must be added to the tablet user, only known to the super admin, so that the screen is locked unless for password input.

7) Ticket Dispenser: Final result : The result of the integration of the several elements for the electronic ticket dispenser is shown in Figure 22.

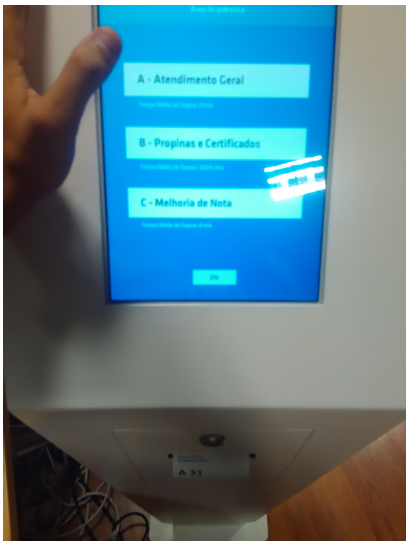

Fig. 22. Ticket dispenser after printing.

8) Virtual Tickets: Push notifications are a way to receive mobile notifications (in our smartphones) from a certain application, if we so allow.

The mockup uses a fabricated back-end (not IST's), constantly alerting users who have had requested a ticket from a certain queue, of the updates to that queue. This was done with the aid of FCM, and a Django Package for FCM (django-fcm[17]) that simplifies the sending configurations. Much parts of this mockup app are re-proposed from the SIGA App.

VI. Test and Validation

A. Tickets Printing

We tested the speed with which we can print the tickets.

In a first test, we clicked continuosly on the same queue, even when not appearing in the interface to show a printing dialog. We created 10 tickets in one minute (and thirty microseconds), which totals an average of 6.0 seconds per ticket.

[16]http://www.imagemagick.org/script/index.php

[17]https://github.com/Chitrank-Dixit/django-fcm

This measure includes the ticket printing, and the waiting for the screen to re-establish the queues after a printing dialog.

In a second test, an analysis was made take by take, from a total of ten measures and without continuous clicks (only after the establishing of the screen) we measured both the printing time after click, and the screen repositioning.

The average speed for the physical ticket creation was 1.9 seconds after button click.

The screen with the queues reappeared, in average, 1.6 seconds after the ticket printing.

Although this amounts to less than the first test, totalling 3.5 seconds for each ticket creation, the missing 2.5 seconds can be accounted for button unresponsiveness after printing, which was not directly measured in this second test.

Thus, without counting with virtually dispensed tickets, 4800 would be the theoretical maximum number of customers a service session could serve, limit set by the printing ticket speed. Virtually dispensed tickets let us overcome this maximum.

B. RESTful API Endpoints

Using the Apache Benchmark tool[18], we load tested our several REST API endpoints.

The requests were launched from an external server to the development server, running a development webserver, WSGIServer/0.2, a Lenovo X220 laptop running Ubuntu 15.04, with a Intel(R) Core(TM) i5-2540M CPU running at 2.60GHz (dual core), 8GB of RAM and 128GB SSD.

We tested the several endpoins for a configuration of 3000 requests, accounting for a population of roughly 2000 students making several requests at different times, with 10 concurrent requests (that is, 10 requests being sent at the same time).

The GET requests all got similar results, the medium response ranging from 100ms to 250ms. The requests never failed. The percentage of requests served in a certain time for a get request is represented by the all services endpoint and is represented by the plot in Figure 23.

From the 3000 sent requests, *none failed.* with only the ticket creation being different.

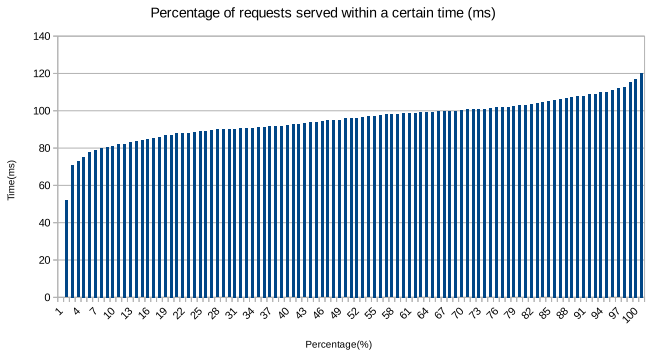

Fig. 23. Percentage bins for request duration for the Services endpoint.

The percentage of requests served in a given time is presented in the plot of Figure 24. From the 3000 sent requests, only 638 created and returned new tickets, while *2352 failed*. There is a big variance between time waits for these requests: while 60% are below a request time of 500ms, 30% present a minimum of 1500ms and a maximum of 3000ms, with the remainder 10% between 500ms and 1500ms.

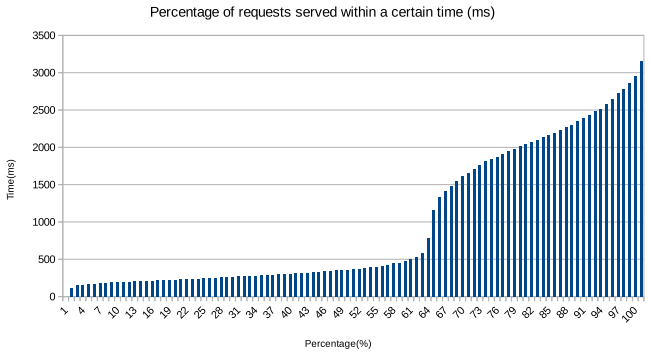

Fig. 24. Percentage bins for request duration for a specific queue ticket creation endpoint.

The rate of failure for requests is quite large for ticket creation. This is because upon creating a ticket, a transaction is initialized in the database, and locks are made to the data, as to ensure that no ticket can get the same number. Failed requests reflect failed concurrent requests that tried to obtain the last ticket given for a certain queue during database transaction lock.

VII. Conclusions and Future Work

A. Conclusions

At the end of this project, a solution to the uninformed queue management problem was documented and prototyped. Having in mind the needs of our school, several functional and non-functional requirements were assessed, and use-cases defined, to better lead the product design and implementation. The implementation was carried out, with modern software tools and school provided hardware, and the developed API was tested for load and concurrency in a development environment. We feel that the base requirements were fulfilled, both functional and non-functional, highlighting the simplicity of the interfaces and the cost-effectiveness of this project.

The development of this project covered great amount of different tools and technologies, which were studied and learned, making this project an extremely enriching experience.

B. Future Work

The back-office designs can be customized, and for such, with the help of the design team to provide improved designs, these must be integrated in our system.

Also, the designs themselves, back-office and kiosk, can be object of further usability testing.

On the technical side, it would be interesting to implement our Django Application with websockets, a tecnhology that would allow real-time information for the back-office without

[18]http://httpd.apache.org/docs/2.4/programs/ab.html

needing to poll the server with AJAX calls or constant page reload.

The application can be further battle tested, in an environment simulating production.

An RSS feed for displaying information on the TV Display can be provided to integrate into our Display interface.

Also, the developed Django application can always be extended to provide more statics and heuristics for the next ticket to select.

In overview, future work includes

1) Add improved back-office designs
2) Back-office usability testing
3) Possible inclusion of websockets
4) Extend next-ticket heuristics (based on acquired data)
5) Assess useful statistics options (based on back-office needs)
6) Integrate with IST mobile application
7) Test in a production or simulated production environment
8) Deploy to production

A good product is achieve upon continuous iterations. With this project we have made the base foundations for the SIGA system, aiming that one day, after its continuous improvement in the context of our schools, it becomes a full fledged product, battle tested, and ready to be deployed in schools or entities looking for a better management of their services.

REFERENCES

[1] M. Halperin, "Waiting lines," *RQ*, vol. 16, no. 4, pp. 297–299, 1977. [Online]. Available: http://www.jstor.org/stable/41354440

[2] H. Do, M. Shunko, M. T. Lucas, and D. A. Novak, "On the pooling of queues: How server behavior affects performance," *SSRN Electronic Journal*, 2015. [Online]. Available: http://dx.doi.org/10.2139/ssrn.2606071

[3] K. L. Schultz, D. C. Juran, J. W. Boudreau, J. O. McClain, and L. J. Thomas, "Modeling and worker motivation in JIT production systems," *Management Science*, vol. 44, no. 12-part-1, pp. 1595–1607, 1998. [Online]. Available: http://pubsonline.informs.org/doi/abs/10.1287/mnsc.44.12.1595

[4] K. S. Anand, M. F. Paç, and S. Veeraraghavan, "Quality–speed conundrum: Trade-offs in customer-intensive services," *Management Science*, vol. 57, no. 1, pp. 40–56, 2011. [Online]. Available: http://dx.doi.org/10.1287/mnsc.1100.1250

[5] M. Delasay, A. Ingolfsson, B. Kolfal, and K. L. Schultz, "Load effect on service times," *Available at SSRN 2647201*, 2015.

[6] Django Software Foundation, "Django documentation," https://docs.djangoproject.com/en/1.9/, [Online; accessed 22-April-2016].

[7] Python Software Foundation, "Python documentation," https://docs.python.org/3.4/, [Online; accessed 20-Jun-2016].

[8] Google Inc. and Open Handset Alliance, "Android API guide," https://developer.android.com/guide/index.html, [Online; accessed 26-May-2016].