PaperLess: A Scalable Electronic Receipt Processing System

Sean M. Arietta and William P. Burns

Abstract

We describe PaperLess, an electronic receipt processing system. Our approach is to provide a seemless integration to existing point of sale (POS) systems. Our system is designed to withstand spikes in demand and faults from hardware and connectivity. We describe our system and the ways in which we accomplished each of our goals. The final product is an integrated system that allows consumers and sellers to effortlessly access their receipts without needing a physical copy.

1 Introduction

As the world becomes more environmentally conscious, it becomes increasingly necessary to revise the financial transaction infrastructure to accommodate the trend. Although paper receipts create very little waste on a per transaction basis, the volume of receipts being printed daily has a negative effect on the environment. The Paperless system aims to reduce this wastefullness by providing customers with the option of receiving a digital receipt in place of its paper counterpart. Customers can then access their digital receipts through the Paperless system via an online web interface, providing an environmentally friendly alternative to paper receipts with the added benefeit of a scalable distributed backend capable of compiling statistical data on customer purchasing activity.

In the initial phase of this project we enumerated a small set of general goals for the system. We wanted the system to be invisible to both the consumer and the seller to minimize both training and confussion during the transition to the Paperless system. In addition, due to the large number of transactions that occur daily, we knew from the start that the system had to be scalable and would likely need to be distributed. Additionally, we imagined that companies utilizing the system would want to aggregate statistical data on customer activity. This would require permanent storage of the receipts in our digital warehouse.

During the next phase of the project we examined the potential risks associated with our undertaking and determined what could reasonably be accomplished in a semester. Since we did not have direct access to a functional point of sale (POS) system nor the means to procure one we decided to focus primarily on the processing and warehousing aspects of the system.

In this paper, we discuss the project as it was proposed and the progress that we have made thus far. In Section 2 of the paper we present an overview of the important requirements for the system and discuss their significance. Section 3 provides an overview of the system from a structural perspective. This section is not intended to detail the inner-workings of the system but rather the components that comprise it. In Section 4, we discuss our implementation methodology as it pertains to Software Engineering. While our goal was to create a functional product, we were primarily concerned that whatever progress we made over the course of the semester was accomplished through the use of good Software Engineering practices. Section 5 discusses how we tested the system once it was functional and what we learned from our initial testing that will be addressed in future work. Section 6 addresses our future work on the system and we conclude the paper in Section 7 with our reflections on the overall experience.

2 Requirements

In accordance with good Software Engineering practices, we compiled a set of high-level system requirements during the planning phase of the project to guide development over the course of the semester. As the project progressed, the set of requirements was updated to reflect unforseen changes that occured during the development process. In this section, we discuss the requirements that we consider to be the most significant for the Paperless system. We divided our requirements in five distinct groups: User requirements, Reporting requirements, System and Integration requirements, Security Requirements, and User Interface requirements.

2.1 User requirements

From and end-user perspective, the most crucial aspects of the system is that it's easy to use and always available. To accomodate these requirements, we elected to use a simple web interface to provide users access to their digital receipts. We consider it low-risk to assume that the average person is familiar enough with the internet to be able to navigate a simple website with little effort. Additionally, we require that the system be distributed with no single points of failure and that data be stored redundantly. To ensure no single point of failure for receipt processing, we developed a distributed receipt processor that load balances over available processing nodes. These processing nodes can be started and shutdown on the fly and ensures that receipts can be processed as long as one node is running. To ensure no single point of failure for the receipt warehouse, we elected to use a mySQL Cluster to store the receipts. As with the receipt processor, storage nodes can be added and removed from the cluster on the fly and the database will continue to function. Additionally, we can adjust the level of redundancy over the cluster as necessary. Redundancy is managed by the mySQL Cluster natively. If a storage node is removed from the cluster, the data that was lost is automatically copied to a different running node to restore redundancy.

2.2 Reporting requirements

From the onset of this project, we wanted the system to have the ability to aggregate statistical data on customer activity. This data is not only important for us to analyze system behavior, but is of great benefeit to retailers that wish to better accomodate their customers or develop targeted advertising campaigns. This requirement was essentially fulfilled through the use of mySQL Cluster as the backend of our storage warehouse. Using the native operations provided by mySQL Cluster, simple scripts can easily extract data from our storage warehouse that can then be processed offline. Each receipt is broken down into its atomic elements prior to database insertion which eliminates the need to pre-process data before it can be analyzed. Additionally, receipts are timestamped at the time of the transaction as well as when they are inserted into the database. This allows us to analyze system performance and locate bottlenecks in the processing and storage system.

2.3 System and Integration requirements

The system and integration requirements for the system are primarily to ensure the integrity of the Paperless system. The biggest risk while the system is running is the loss of data during transmission. This can occur at various stages of the storage process. The receipt must first be sent from the POS system to our processing servers.

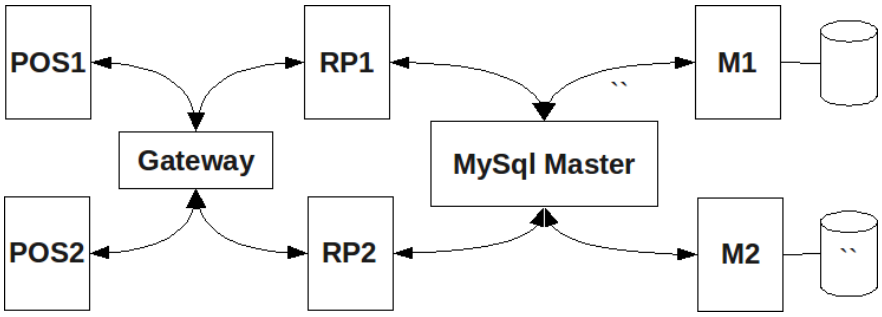

Figure 1: **System Overview** Our system consists of three main parts: a gateway, a processing group, and a data warehousing group. The gateway accepts receipt requests from a point of sale (POS) system and passes them on to the processing group (RP's). Once a receipt has been processed it is passed to the data warehousing group to be stored on one of the data nodes (M's) for future use.

Then it must be sent from the processing servers to the storage servers. If the receipt is lost during either of these stages, the integrity of the system is compromised. To mitigate this risk, we ensure that all transmissions are confirmed on the receiving end before they are considered successful. If a problem arises during transmission, the transaction is ignored and placed back in the queue to be re-transmitted.

2.4 Security requirements

From a security perspective, the secrecy of our users personal information is of utmost importance. All passwords are stored as MD5 hashes in the database, unnecessary ports are blocked by a firewall, and while not implemented at this stage, all data will be encrypted prior to transmission. Good coding practices will be used to develop the web interface to prevent mySQL injection attacks, buffer overflow attacks, DOS attacks, and other known vulnerabilities.

2.5 User Interface requirements

One potential risk for the web interface is slow response time. We envision a massive ammount of data being stored in our database which makes targeted receipt retrieval an expensive task. For this reason, we will require that the mySQL Cluster be sufficiently distributed to balance requests and mitigate the bottleneck from disk access. Additionally, the management nodes in the mySQL cluster cannot be used as storage nodes. They should only be concerned with re-directing requests to the appropriate storage nodes.

3 System Overview

In this section we present the system-level implementation of our approach. We describe the various components to our system and describe the interactions between them. This is not meant to be a formal specification, but rather a more informal description of our specific implementation. For a more formal description of our system please see the supplemental material.

We now describe our system in two parts. The first is the receipt processing group, which is responsible for receiving, processing, and saving receipts from customers. The second part of this section describes the backend data warehousing group. We use this group to store the receipt data from the processing group for future access and analysis.

3.1 Receipt Processing

A crucial property of our entire system is that it scale well with the demands of the customers. As such, our design of the receipt processing pool reflects this goal. The vision of our system is that it will able to efficiently and reliably serve many millions of requests with little or no degradation of service. We set out to accomplish this by attempting to minimize possible downtime and/or degradation of service. Our approach is twofold. First, we have designed our system to be redundant. If any particular subsystem is faulty for any reason, then our entire system is able to adapt and use other resources to "fill in the gaps." Unfortunately redundancy is only half the battle. Even a highly redundant system can fail to satisfy the needs of a growing number of requests. Therefore, as our second goal, we wish to create a system that is also highly scalable.

Figure 1 is an overview of our system. The entry point to the entire processing portion is the gateway. All receipt requests arrive at the gateway. From there, our ProcessDistributor attempts to schedule the receipt to be processed by a particular ReceiptProcessingServer. Each receipt is assigned its own thread that will be responsible for communicating with a particular processing server. The threads themselves are designed to be lightweight and completely independent. This allows the gateway to handle a large number of receipts at once. Although this puts a critical confidence in the gateway servers, we are able to provide redundancy and scalability by using a simple scheme that uses DNS to choose an alternative gateway if one goes down or to simply add a new one to the pool and add a DNS entry if demand is higher.

One important issue for the gateway is its ability to load-balance. This is accomplished by keeping a list of recently scheduled jobs for each server. If a server does not respond to a request, it is put on a blacklist and another server is used instead. Periodically the distributor will check its blacklist to see if any of those servers have come back online. New servers can be easily added and removed by simply updating these lists allowing us to easily scale our entire system as needed.

The servers themselves can be run on effectively any hardware with an Internet connection. The server process simply polls its listening socket for any connections from the gateway, and upon receiving a request, processes that request and returns the result. Specifically, when a processing server receives a request to process a receipt, it first loads the receipt from a redundant distributed file system, performs optical character recognition (OCR), parses the result into a set of lines, and saves the information to our data warehouse.

3.2 Database Overview

As previously stated a primary goal of our system is that it is scallable. To accomodate this requirement on the backend, we elected to run our data warehouse on a mySQL Cluster. The two most appealing aspects of mySQL Cluster for our project are that it

allows for multiple management nodes and that storage nodes can be added and removed on the fly. It is essential that in the absence of catastrophic failure our system remains functional. The system must be able to recover gracefully from errors such as power failures, disk failures, connection failures, etc. mySQL accomodates this necessity by automatically replicating data over storage nodes when a storage node fails to satisfy our desired redundancy.

In accordance with our requirement that there be no single point of failure in our system we decided that a distributed architecture was essential. We have designed the system such that the data warehouse consists of multiple geographically separated locations which together represent the entire database.

Another crucial aspect of the mySQL cluster is that it load balances automatically. This was a major benefeit for our project because while we wanted to develop a system that was customized to our needs, we didn't want to re-invent the wheel. We believe that any issues regarding load-balancing in mySQL Cluster are low-risk. If load balancing becomes an issue, it can be resolved easily by adding more management and storage nodes. It may not be an elegant solution, but commodity hardware is cheap and the benefeit of developing a custom load balancer to suit our needs does not outway the effort and time required to do so.

Figure 2: **Rapid Collaborative Refinement** By closing the short-term iteration loop our team was able to efficiently develop critical pieces of the system. The rapid switching between coding in seclusion and collaborating to uncover and resolve problems allowed us to very quickly refine aspects of the system.

4 Implementation

In the spirit of good software engineering practices we made every effort to abide by a rigorous approach during development. The outcome of our efforts was a similar, but slightly tailored version, of the agile method. We dub our method rapid collaborative refinement. In essence, due to our small development team (2), close proximity, and unqiue abilities we were able to very quickly refine a particular piece of code or refine our approach by alternating between personal interaction and separated investigation. The lifecycle for a particular system-level decision was therefore reduced to a small overhead leaving us more time to focus on developing useful software and well-constructed software. This approach is depicted graphically in Figure 2. Notice that we are able to more tightly close the loop on developing a particular aspect of the system due to our methodology.

Our development environment consisted of several testing machines running Linux and Mac OS X, the Eclipse IDE, git source code management through github.com, mySQL Cluster Server, and MyEclipse for making logical diagrams. All code was written in JAVA so that we would have more flexibility in deplyoment options.

We know describe in detail the actual software behind our system. In Section 3 we discussed the high-level overview of our system. Now we will explain exactly what we did, and what we didn't do. We will start by describing our implementation of the processing group and then move on to the data warehousing group.

4.1 Processing Group

From a high-level perspective, the processing group is responsible for processing a set of receipts in an efficient and scalable manner. For this reason we chose to separate the role of process assignment and process fulfillment. Our ProcessDistributor acts as the assigner. It polls its listening socket and waits for a request to process a receipt. When it receives a request, it simply puts the request into a FIFO queue. A separate thread continuously checks the queue for elements and upon finding one creates a new handling thread that will process the item.

Each handling thread executes a remote procedure call (RPC) to a ReceiptProcessingServer. The particular server chosen is determined by the ProcessDistributor and is currently just a round-robin selection. The RPC itself was written from scratch so that we would have the flexibility to create a lightweight object tailored to our very simple needs. RPC's are implemented as serializable objects that can be transferred via a socket to a ReceiptProcessingServer. Upon receiving the RPC, the processing server extracts the necessary information from the parameter field, saves the relevant data to the mySQL backend (see Section 4.2), and then returns a response.

As discussed in Section 3.1 our goal was to be able to process imags of receipts. These images would be stored on a distributed file system and the RPC parameters would just consist of a file handle to the particular receipt image to be processed. Due to complications, we were unable to accurately perform OCR on images. Section 5 describese in more detail exactly what these complications were. Nevertheless, we still wanted to implement a working system if not complete. In our current implementation, the parameter to the RPC is a string representation of a receipt. Our system then parses the string representation (just as it would parse the output of OCR) for the item name, price, quantity, seller, user information, etc. This information is saved to our mySQL cluster database through the JAVA mySQL connector.

If at any time there is an error on the processing server, a message is sent bck to the distributor notifying it of the error. The distributor is then able, depending on the error, to re-queue the receipt for processing again. Some situations where this would be appropriate would be if the processing server happens to lose connectivity with the mySQL database.

The beauty of the system is that independent functional parts are separated by logical and information boundaries. The distributor knows only a list of possible processing servers. It holds the queue and iteratively attempts to empty the queue by sending RPC's through its handling threads to a processing server. The processing servers, in turn, know nothing about the distributor. They simply listen for data on their sockets, process requests that come to them, and return the results. Similarly the database has no knowledge of the specifics of neither the processing servers nor the distributor. This setup allows us to very easily scale the entire operation as needed. We need only notify the distributor that another poten-

tial processing server is available. All other parts are completely agnostic to its pressence or lack thereof.

4.2 Data Warehousing Group

The data warehousing group is comprised of a distributed mySQL Cluster. The Cluster consists of multiple management nodes which oversee a variable number of storage nodes running on commodity systems. The level of redundancy is maintained by the management nodes and can be adjusted as necessary. Data is copied to multiple storage nodes to reflect the desired level of redundancy to ensure data integrity.

As a proof of concept, we configured a small three system cluster with one system acting as both a management node and storage node. This allowed us to run experiments using different levels of redundancy and simulated failures. In the early stages of the project we simulated a distributed cluster using a single machine. While this provided evidence that the system would function as expected when deployed, we were determined to get a truly distributed configuration running. Although our cluster is small, we found it sufficient to run large workloads for experimental purposes. It was far more impressive to see in action than the simulated version.

To improve access time and remove redundancy from the database we spent a good portion of our planning phase devising an appropriate schema for the database. Our schema is organized as tree structure containing six distinct tables: Line, Receipt, User, Location, Store, City. We will now discuss these in further detail.

The Line object contains all information pertaining to a single item from a receipt. This includes a the item name, the price per unit, and the quantity purchased. The Line object also contains a reference to a Receipt object. The Receipt object consiste the date of the purchase, and references to both a User object and a Location object. The User object simply contains the registration information for a particular user. It is used to determine which receipts belong to which users. The Location object contain refrences to both a Store object and a City object. These references are used to accomodate retail chains that may have more than one location in a single city. The City object contains fields for city, state, and zip. This reduces redundancy when the system contains multiple clients in the same city. The Store object contains the name and address of the retailer. This is also to reduce redundancy in the database for retail chains. Additionally, if a company were to change its name, this would only require updating a single record.

5 Results

Our goal was to create a system that could process receipts in a scalable and reliable way. We have defined scalability as the ability to quickly and seemlessly add or remove processing entities. Reliability refers to the persistance of data after it has been received by our system. We have described our system and its specific implementation to date in Section 3 and Section 4. Now we set out to evaluate our work.

First we will consider scalability. In order to test the scalability of our system we required a large test set. As mentioned in Section 4.1, we were unable to get adequate results from an optical character recognition package we tried. We tried Tesseract and SimpleOCR on a small set of test receipts. Although according to 1995 UNLV Accuracy Test, Tesseract is one of the best OCR packages available. However, our initial tests deemed it unable to extract text correctly from receipts most likely due to the poor resolution available and the non-standard layouts present in receipts from many different companies. Therefore, we chose to consider receipts as strings with fields corresponding to what would have been extracted from an OCR package. A typical receipt line then takes the form:

{receipt:RECEIPT ID} {item: ITEM NAME} {price: ITEM PRICE} {quantity: QUANTITY} {store: STORE ID} {date: PURCHASE DATE}

For our tests, we generated 30,000 lines of this format. We then wrote a simple test harness that looped over the list and sent each line to the process distributor gateway. We ran this test with three processing servers available. The bottleneck in the system is the distributor. It receives all of the requests nearly simultaneously and must federate them off to individual threads while maintaining a queue of the non-processed receipts. In a setting where there are hundreds of thousands of receipts coming in at once, this can become a burden. However, even with a modest hardware configuration consisting of a MacBook Pro with 2GB of RAM and a 2.16GHz Core2 Duo we were able to handle 30,000 requests arriving nearly simultaneously.

We also ran tests to assess the ability for our system to adapt to new or failed servers. We began by running the same test as before but in the middle of the test we killed off a node. The distributor was able to distribute the receipts effectively across the remaining servers in the pool. Likewise, if we added a new server, the disitrbutor was able to adapt and send receipts to the newly added one too. Our tests show that we have indeed created an effective receipt processing system that can be scaled to meet demand.

Since one of our goals was to develop our system in line with standard Software Engineering practice we also provide results for our logical, infrastructure, and use case diagrams. These are included in Appendix A.

6 Future Work

As stated previously, our tests revealed a few short-comings of our system. Our small scale configuration could not handle the large test sets that we were runing on it, we elected to focus on the server side of our system while ignoring the client side, and the OCR packages were not able to decipher actual receipts with sufficient reliability.

With a primary goal of scallability, we really wanted to see large test sets run through our system with ease. Unfortunately, we were somewhat limited by the ammount of hardware we could procure over the duration of this project. Configuring and managing the MySql Cluster nodes requires priledges that were not available to us on department hardware so we were limited to what few computers we could procure. The results from our tests revealed the bottle-necks to be hardware related, we didn't have enough of it. Accordingly, we were not satisfied that we suffiently tested the limits of the actual system. Fortunately we designed our system to scall easily so that if hardware did become available we could incorporate it into the system with ease. This would likely be the next step in the project so that we can evaluate the limitations of the system on a large scale.

Another goal of our project was to integrate seamlessly with existing POS systems. Without access to an existing POS configuration nor the means to procure one during the short timeframe for this project, we used an oracle to create simulated receipts. This issue would have to be resolved before a real-world installation could occur. We would like to develop a resident client that can be installed on a POS server and interface with the POS system which actually performs the financial transactions. This would eliminate the need for OCR since the POS system would already have the receipt in

a digital form. It is not clear to us at this point how difficult this portion of the system would be to develop.

While a resident client on the POS server would eliminate the need for OCR for future transactions, it does not allow for customers to store old receipts on our system. It would be any easy solution to create a web form where end-users could type the information in by hand, but this would be a tedious task. As a more elegant solution, we would like to develop our own OCR program customized for receipts. It is unrealistic to expect a general purpose OCR package to be able to decipher all printed text. In our tests, it was not accurate enough to satisfy our requirements. However, we believe that a custom OCR package that is developed specifically for receipts would be more successful. Most receipts have a similar structure which would be an advantage for a custom OCR program. Additionally, there are not a lot of POS systems in use so there are not a lot of different character sets to accomodate. A reliable OCR program would be a great benefeit for our system.

7 Conclusions

We set out to develop a distributed and scallable system for providing storage and access to digital receipts. We feel that we have accomplished our goals for this phase of the project. The use of digital receipts has many benefeits. It reduces paper waste, reduces paper cost, receipts are stored permanently, and statistical data can aggregated. The system that we developed provides these benefeits.

While many of our goals for this project were focused on the actual system, it was essential to the success of this project that our development was guided by good Software Engineering practices. Using rapid collaborative refinement, we were able to quickly develop requirements for the system and iteritively develop new working components while improving existing ones. This ensured that at any point in our development we always had a working prototye. We preferred this method over throw-away prototyping because we didn't want to waste what little time we had on non-system code. Additionally, our method eliminated development bottle-necks because the coders were always working on code separated by a well defined interface.

To conclude this paper we would like to discuss what we learned over the course of this project. Identifying risks early in the development process helps to determine what is realistically possible in a given time frame. If we had attempted to develop the client-side and server-side systems in parallel we would have likely ened up with an elaborate communications system that did minimal receipt processing if any. We also confirmed that rapid collaborative reinement is a viable method for Software Development as long as communication is emphasized. At all stages of the development process there was a clear understanding of what each developer was working on. This eliminated the frustration of attempting to merge changes in the git repository and reduced the chance of breaking the build.

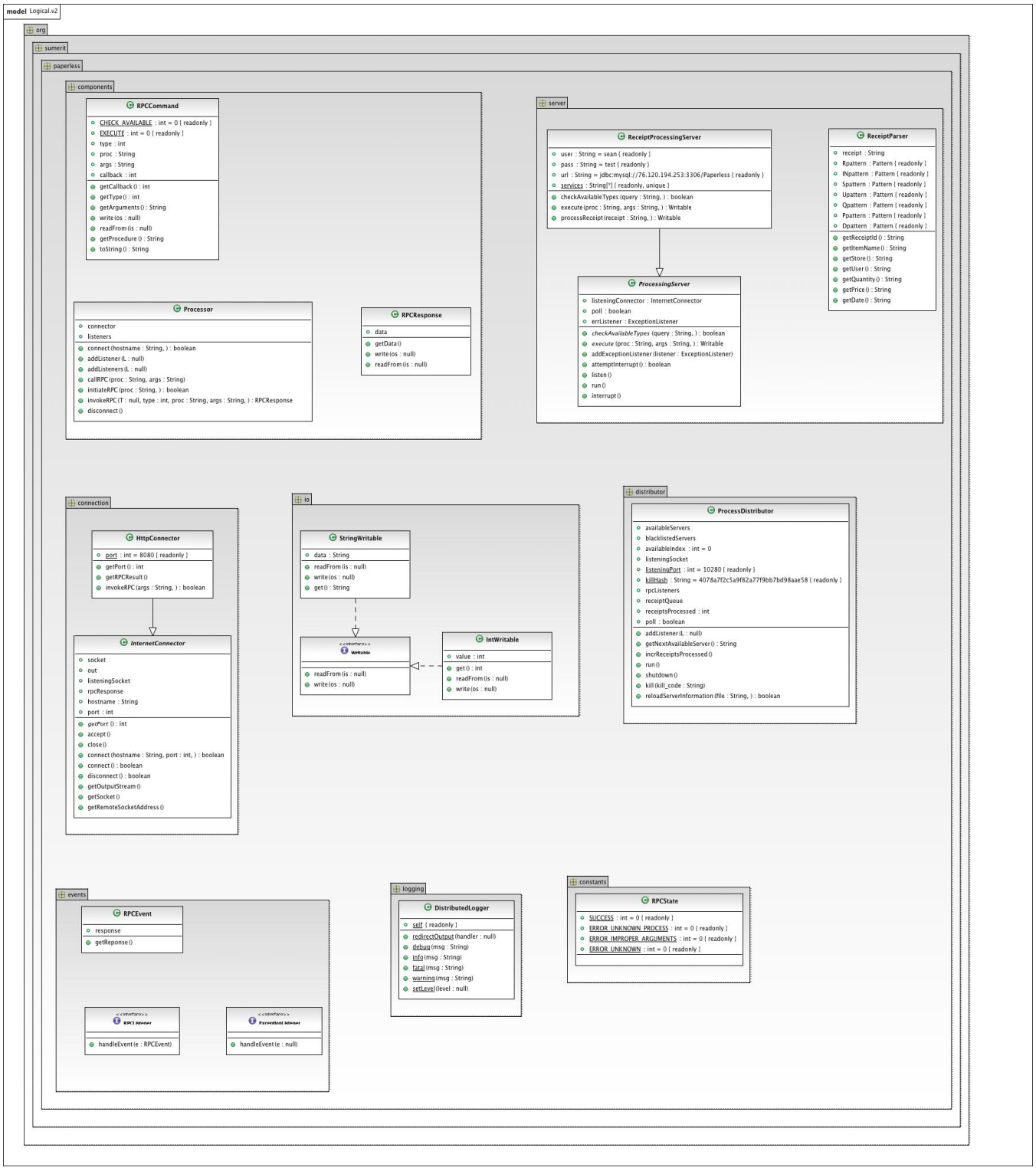

Figure 3: **Logical View** Shows the UML diagram of our processing software package.

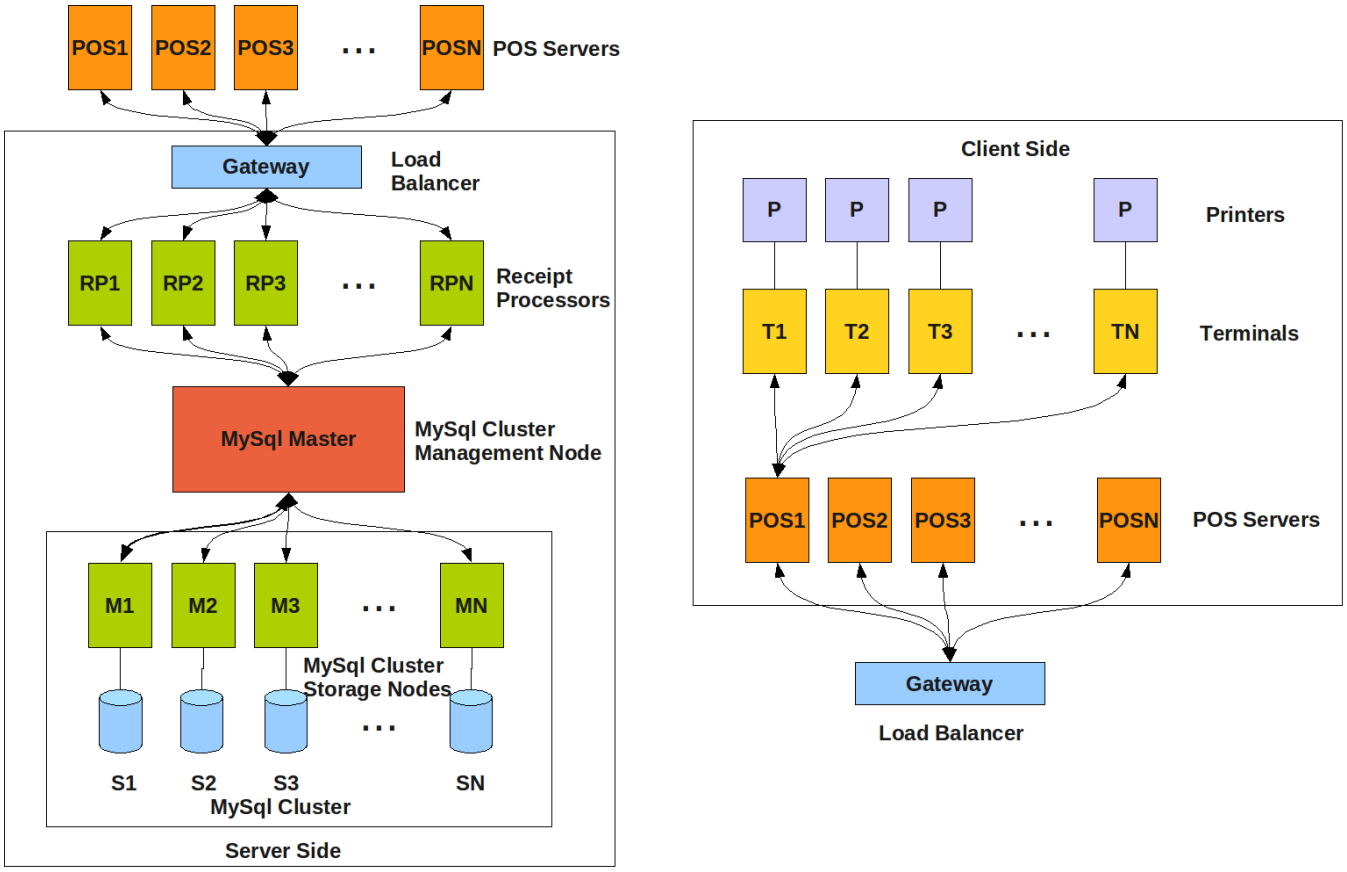

Figure 4: **Infrastructure View** Shows the components and the connections between components of our system

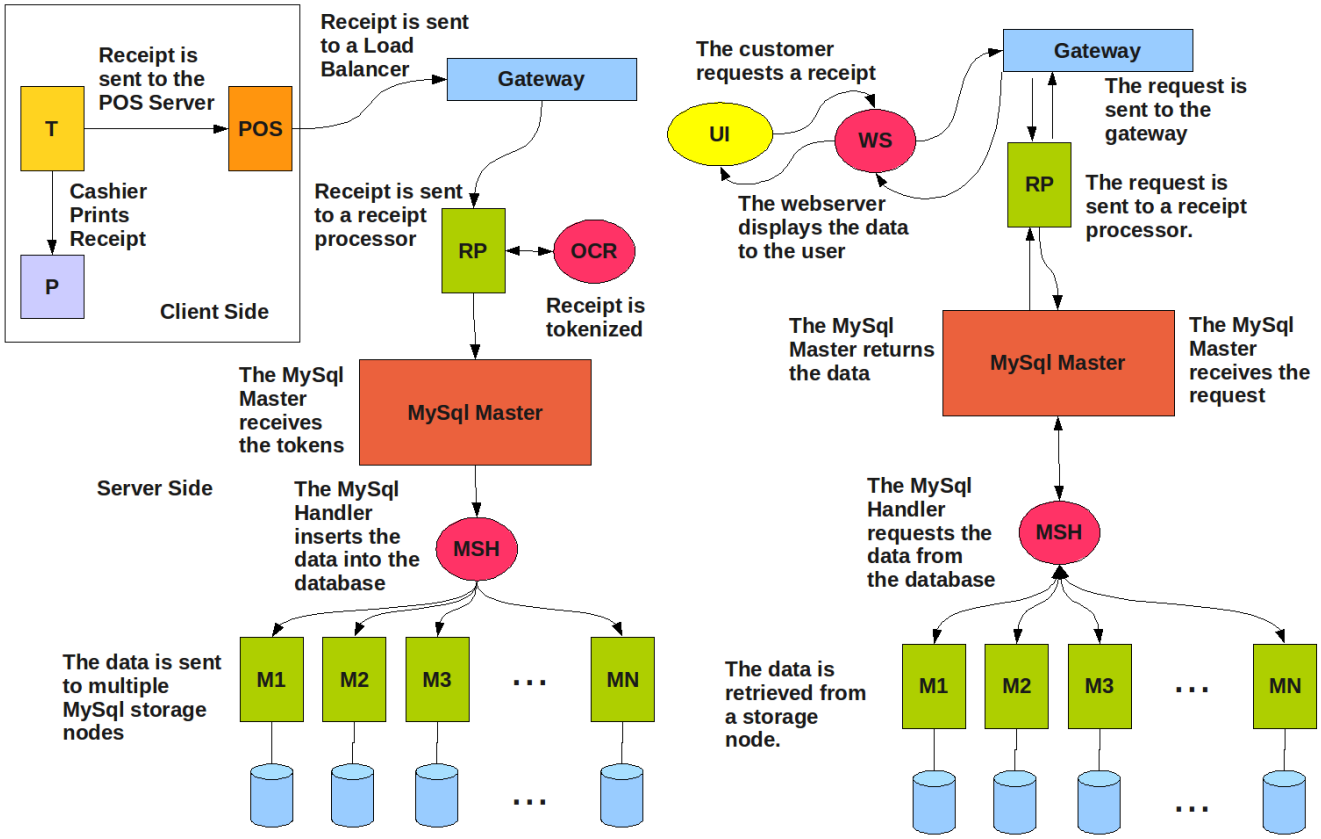

Figure 5: **Use Cases** Some typical use cases from both the seller and consumer points of view