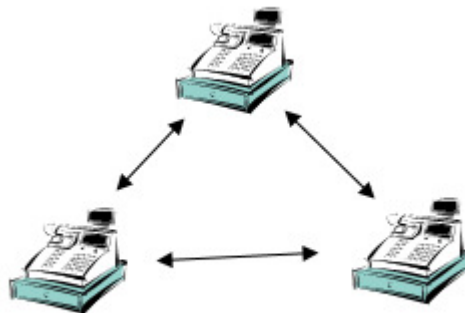


CHALMERS



A Peer-to-Peer Point of Sale System

A design of a distributed system with a peer-to-peer architecture to replace a solution based on a client-server model

Master of Science Thesis in Computer Science and Engineering

JOEN PETER

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, October 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

A Peer-to-Peer Point of Sale System

A design of a distributed system with a peer-to-peer architecture to replace a solution based on a client-server model

JOEN PETER

© JOEN PETER, oktober 2015.

Examiner: TOMAS OLOVSSON

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden oktober 2015

Abstract

In this thesis we investigate an enterprise IT system based on a simple 3-layer client-server model, with clients connecting to local servers which are then in turn connecting to a central enterprise wide server. The system that is analyzed is a retail system, consisting of Point of Sale (POS) clients and one server in every shop to handle the clients. The local servers in turn communicate with a central server that keeps track of all retail transactions from all the shops, and all the reference data (articles to sell, prices, campaigns, etc) being sent to the shops.

Because of cost and maintenance issues there is a demand for being able to deliver the service without the overhead for the local servers. Finding a way to eliminate the local server from the system is the purpose of this thesis. This is achieved by closely examining the data flow and functionality of the local server, thereby being able to suggest two different approaches to solve the problem using a peer-to-peer approach. The two different designs are compared and evaluated, showing the difference in the approaches. This result can also be used in a more generalized manner to look at removing layers of servers in similar systems.

Preface

This report is the result of a master thesis project done for Chalmers University of Technology in Göteborg, Sweden and for Extenda AB. It has been written as part of the Master's Program Networks and Distributed Systems.

The project was performed by Joen Peter during 2014/2015, with the help of examiner and supervisor Tomas Olovsson at Chalmers and the advisor Peter Norrhall, chief architect at Extenda. Also helpful in this project with extensive knowledge of the system investigated was Peter Mathsson, POS architect at Extenda.

Table of contents

Abstract	i
Preface	ii
Table of contents	iii
1 Introduction	6
1.1 Retail system.....	6
1.1.1 System use.....	6
1.1.2 System design.....	7
1.1.3 Customer expectations	8
1.2 The problem.....	8
1.3 Objectives	9
1.4 Scope	9
2 Background	10
2.1 Detailed system overview.....	10
2.1.1 Relevant functions on POS/POS Server.....	10
2.1.2 Communication	12
2.1.3 Data synchronization.....	12
2.1.4 Code structure and principles.....	13
2.1.5 Code implementation	13
2.2 Inversion	14
2.2.1 Dependency Inversion Principle	14
2.2.2 Inversion of Control	15
3 Theory	16
3.1 Peer-to-peer networks.....	16
3.1.1 Pure peer-to-peer networks	16
3.1.2 Hybrid peer-to-peer networks	16
3.1.3 Overlay networks	16
3.2 Distributed systems.....	17
3.2.1 Causality.....	17
3.2.2 Atomicity.....	19
3.2.3 Coordinator.....	20
3.2.4 Consensus.....	21
3.2.5 Eventual consistency	22
3.3 Theory applications	22

4	Methodology	24
4.1	Investigation phase	24
4.2	Design phase.....	24
4.3	Proof of concept phase	24
5	Design.....	26
5.1	General approach.....	26
5.2	Controlled actions.....	26
5.3	Causality	26
5.4	Peer to peer network	27
5.4.1	Network structure	27
5.5	Coordinator design alternative.....	27
5.5.1	Leader election	27
5.5.2	Synchronization.....	28
5.5.3	Reference data flow	28
5.5.4	Transaction data flow	29
5.5.5	Points of integration	29
5.6	Logical clocks design alternative	29
5.6.1	Synchronization.....	30
5.6.2	Reference data flow.....	30
5.6.3	Transaction data flow	31
5.6.4	Points of integration	31
6	Proof of concept	32
6.1	Purpose and scope	32
6.2	System overview.....	32
6.3	Different approaches.....	33
6.3.1	Client-server	33
6.3.2	Coordinator.....	33
6.3.3	Logical (Lamport) clocks	34
6.4	Test environment/Proof of concept	35
6.4.1	Till interface	35
7	Results	38
7.1	Test cases	38
7.1.1	Start up all nodes	38
7.1.2	Log in and create a receipt on all tills	38

7.1.3	Put till A offline.....	38
7.1.4	Shut down till B.....	39
7.1.5	Put till A online	39
7.1.6	Put till C offline.....	39
7.1.7	Put till C online	40
7.1.8	Start up till B	40
7.1.9	Test login of the same user on all tills at the same time.....	40
7.2	Network traffic in general.....	40
7.3	Death of a node.....	41
7.4	Birth of a node	41
7.5	Differences between the approaches	41
7.5.1	Workload distribution	41
7.5.2	Local performance.....	42
7.6	Design issues	42
7.6.1	Data ownership and conflicting updates	42
7.6.2	Data storage.....	42
8	Discussion	43
8.1	Failures and unknown states.....	43
8.2	Design choices and designs not explored.....	43
8.2.1	The consensus approach.....	43
8.2.2	The atomic operation approach.....	43
8.2.3	Distributed database	44
8.2.4	Mixed approach.....	44
8.2.5	Issues not looked at	44
8.3	Usability.....	44
8.4	Future work.....	45
8.5	Other applications of the work	45
9	Conclusions	46
10	References	47

1 Introduction

As the Internet grows larger, bandwidth is becoming cheaper and workstation computer grows more powerful, the traditional client-server model is growing old. As large servers are not only expensive, but also frequently becoming bottlenecks in large systems, an alternative to this model is an important field of research and development. The solution to these problems could be a distributed architecture [1].

One way of catering for the increased demand from users is to create large-scale distributed systems that serve cheap “thin” clients with distributed services. This is generally referred to as cloud computing.

Another approach is using the excess resources readily available in most modern workstation computers. By sharing resources, the requirements for expensive servers are reduced. Instead, clients utilize resources on other clients as needed to create a peer-to-peer system. This thesis will investigate using such a peer-to-peer architecture to create a distributed system.

1.1 Retail system

In this thesis we are looking at a retail system built for an enterprise environment. Primarily, it is a point of sale (POS) system, with centralized administration and integrations with several other systems. The software is developed by a company called Extenda, and it is widely used both in Sweden and internationally.

Due to the business critical nature of a POS system (the shop generally cannot legally sell merchandise without a functional cash registrar) there are very high requirements on availability at the POS.

1.1.1 System use

The system is used on several different levels and by different personnel for different purposes.

Cashier

The cashier is the end user at the point of sale, or cash registrar. This computer is generally referred to as the POS client. The cashier interacts with the system using a combination of a graphical interface, touch screen, keyboard, scanner, and other peripherals. In some cases, for example at a self-checkout terminal, the customer is actually their own cashier.

The cashier primarily serves customers, but could also perform some simple administrative tasks at the POS client, for example cashing up or changing their password.

Shop manager

Personnel that performs economic and administrative tasks for the entire shop. This could include regular tasks like banking cash or performing end of day for the shop, a process in which the current business day is closed and can be filed to accounting.

Company administrator

This is typically the IT department or some other centralized part of the company that handles economic transactions or shop administration. This could include tasks like adding new users

(for example cashiers), administrating prices, articles, receipt texts, etc (some of these tasks could also be delegated to the shop manager). An administrator can control all aspects of the POS environment.

1.1.2 System design

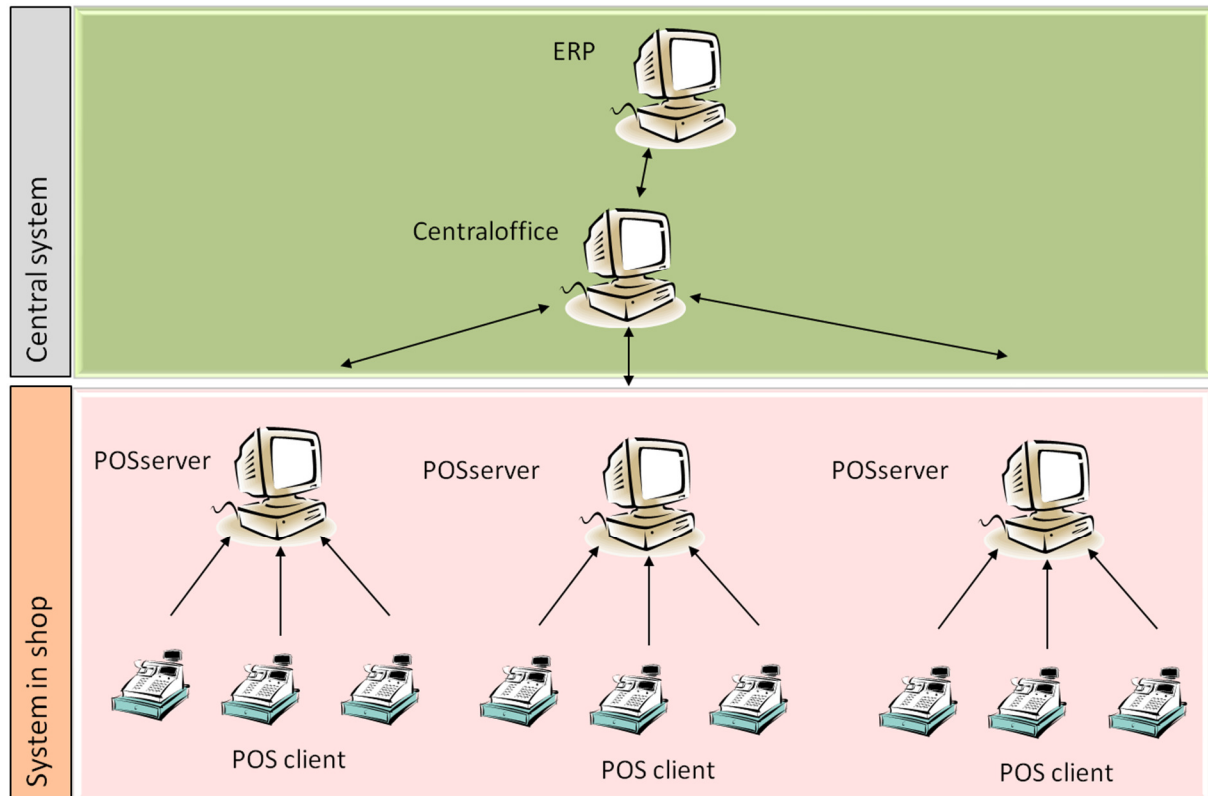


Figure 1 - Extenda system design, based on a client-server model.

The system is interconnected in a way that makes it possible for a large organization to specify all data that is needed in a shop (articles, users, customers, etc, also referred to as “reference data”) on an enterprise-wide scale. It also ensures that all sale transactions (receipts) are collected on an enterprise-wide level. In most cases, most data originates from and all receipts are forwarded to an external Enterprise resource planning (ERP) system which maintains control of the overall economy and resources.

POS client

The POS client is the actual till where sales are performed. The POS client has all the reference data it needs stored locally. This data is the same for all clients operating in the same shop, and is acquired from a POS Server that operates in the shop. When a receipt is created, it is first stored locally and then sent to the POS Server.

There are two main reasons why the POS client is not operated as a thin client. Firstly, the availability requirements demand that the shop can still operate the POS client, even if the network connection is not working properly. Secondly, the client integrate with a lot of complex peripherals, like card terminals, automated cash changers, special keyboards and cash drawers. This puts some requirement on the computer hardware at the client.

POS Server

In every shop there is a local server, referred to as the POS Server, which has two basic tasks. The first task is to handle the synchronization of reference data to all POS clients in the shop, and the second task is to handle the receipt flow from the POS clients. On top of this it acts as a local control tool to synchronize certain requests from the clients.

The reference data is pushed to the POS Server from a central system called Centraloffice (CO, see below). The POS Server performs a version control and increments the database version for each update from CO. The clients regularly poll the server to see if there is a new version available – if there is, it will fetch the new version from the server.

The POS Server delivers updates to the client of two different types. A minor update is delivered as a change of some specific reference data, for example a new price for a specified article. A major update is delivered as a complete database, with all the reference data. Usually, major updates are only generated when there is a specific instruction from CO to perform this.

All receipts are sent to the POS Server from the clients, and are then processed at the server. The server performs some conversions of the receipts to match the format that CO expects. The receipts are also saved locally on the POS Server.

Centraloffice

The Centraloffice (CO) application stores all available information regarding all shops for the customer, both reference data and receipts. CO is usually populated by data from an external ERP system, and all economic transactions are forwarded to the ERP system from CO as they arrive from the shops.

All accounting activities in the system are handled in CO. This is where the amount of cash in the different tills are calculated, as well as where cash corrections and end of day can be performed. CO has a web-based user interface where the shop manager or a company administrator can perform different tasks and look at information from shops.

1.1.3 Customer expectations

From the shop's point of view the reliability of the POS client needs to be very high, since the company will likely lose both money and customers if they can't process sales in a shop. It is also very important that sales transactions (receipts) always are delivered to the accounting and ERP systems, even in case of failure at some step in the process. This is because the company may otherwise get in trouble if they are audited. The customer will also expect the reference data in every POS client to be correct at any point in time.

While not a strict requirement in the same sense, speed at the POS client is generally an important factor. In some supermarkets (or hypermarkets) a few seconds extra delay for every receipt is enough to have a negative impact on sales over a longer period.

1.2 The problem

The POS Server has no interaction with the end user, and is mostly noticed by the shop as the extra computer they need to have in every shop causing higher maintenance costs. For this reason, some shops have requested an alternative to having a POS Server. Primarily these are

smaller shops (1-6 clients), but the problem has been defined to find a general solution to be able to remove the POS Server from the system in the environments where it is feasible.

1.3 Objectives

The goals of this thesis are:

1. Find a method to replace the functionality of the POS Server and put it in the POS clients or CO servers.
2. Design a way to implement this, fulfilling the following objectives:
 - a. It should be possible to perform sales at the client even if the network connection is disrupted.
 - b. Receipts should be saved in strictly more than one place in the shop.
 - c. There should only be one active connection to CO from any given shop at any time.
3. Build a proof of concept to show that the design works as expected.

1.4 Scope

This master thesis project consists of four large parts, as presented below.

Investigation

During the investigation, the specific requirements for the end product were structured, and available methods for similar implementations were explored. Different approaches and algorithms were looked at and documented. The investigation was limited to looking at different approaches to distributed systems.

Design

During the design stage, the information gained in the investigation were used to construct a rough design for a proof of concept. At the early stages of the design phase, two specific approaches were chosen, and all the work into the design were based on these approaches. The design was revisited at times during the next phase, as new information was discovered.

Proof of concept

A proof of concept was also implemented, using the design specifications. This phase was mostly limited by time, and had a strict deadline for last delivery. The main goal of the proof of concept was to validate the design and to be able to see the differences between the two approaches. It was therefore important that enough development was done to get an idea whether the designs were functional or not.

Report

At the end of the testing, a report was written consisting of a complete documentation of the work and results of the project.

2 Background

2.1 Detailed system overview

This part describes the current system in detail, specifically the areas that are directly affected by the proposed changes.

2.1.1 Relevant functions on POS/POS Server

These are the functions that are identified in the current environment that will likely be affected by this project, with a detailed description of how they work today.

Reference data update (“minor update”)

A minor update is an update to the reference data in a specific shop. This data is always sent as a change, including only the specific data that is being changed.

The POS Server is a single point of contact in the shop from CO, which is designed in a way that it always communicates with exactly one IP address for every shop (the POS Server in the shop). Whenever there is new data, CO will push the new data to the POS Server with a reference data message. It stores this message in the *pos_messaging* database on the POS Server, and validates and processes the message.

As the POS Server processes the message, it updates its own *serverpos* database containing a complete current reference data database, using the update information in the reference data message. At the same time it adds the change to the *serverupdates* database that contains all the changes in data. The POS Server then increments the database version.

The POS clients are informed of any database update on the POS Server. The update will be fetched by the POS client from the server, and the client will add the updates to its local *clientupdates* database.

The client has two databases for reference data containing the complete reference data set, *posa* and *posb*, which both should mirror the *serverpos* database. The client only uses one database at a time, and always applies updates to the inactive database to avoid database changes in the middle of a receipt. If the inactive database has a more recent version than the active database, it will switch active databases on next opportunity. This is done between two receipts, or when no user is logged on to the till.

After the information from the *clientupdates* database has been applied to both *posa* and *posb*, the record is removed from *clientupdates*.

Complete reference data update (“major update”)

When there is a large change that affects a large portion of the database, or when manually triggered, CO will send out a “major update”. It will collect all reference data for the specified shop, package it, and send it all to the POS Server using a web service request.

The POS Server will drop its *serverpos* database, and completely rebuild it using the update from CO. When it is completed, it will empty the *serverupdates* database, create a database dump, increase the reference data database version, and signal the clients that there is a major update available by putting a special record into the *serverupdates* database.

The POS clients will fetch the database dump, drop their inactive database and recreate it with the dump. When possible, it will switch active databases and do the same thing with the other database.

Transactions

A transaction can be either a receipt or a control transaction (login, logout, open cashdrawer, etc). A transaction is always created at a POS client. It is quickly saved in the *clienttransactions* database as a Binary Large Object (BLOB), and then forwarded to the POS Server in binary format. When the POS Server receives the transaction, it is first saved in the *servertransactions* database. It is then converted to a standard POSlog format, and sent on to CO.

Receipt parking

Parking a receipt is technically the same operation as finishing a receipt, except the receipt status is set to *SUSPENDED* instead of *COMMITTED*.

Fetching a parked receipt however, requires some new functionality. The POS client can send a request to the POS Server to get a list of parked receipts. When a specific receipt is selected in the POS client to be restored, the POS client sends a request for the receipt information to the POS Server.

Post-void and returns

Shops can in specific circumstances perform a post-void on a receipt – that is, they can cancel the receipt after it has already been printed and sent to CO. It is also possible to perform a return on an old stored receipt, even if it is from another shop. To handle these special cases, there is both a service to search for old transactions on the POS Server, and a service to fetch (and update) transactions already stored at CO.

Shop-specific status and counters

On the POS Server in the *serverstates* database there are accumulators that are updated when transactions are received from the clients. There are several different accumulators storing different types of data, as well as some historical data.

Single point of contact

In some situations it is impractical to have each POS client make their own connection to a service, both inside of the shop and on an enterprise-wide scale. In the later case, it could in fact be a problem for the service if all the POS clients in all the shops try to connect. This has to be considered in the design.

There is one core implementation that relies on the server being a single point of contact. This is the Global Blue Refund service for tax free shopping that requires the integration to be limited to one point per shop, due to limitations at the service provider.

In-shop integration

The POS Server acts as an integration point between the POS system and other services in the shop. For example, this applies to the bottle deposit system that maintains a record of returned bottles to a Tombra bottle recycle machine in order for a customer to get their deposit back.

2.1.2 Communication

CO to POS Server

The communication of data from CO to the POS Server is done using web service requests. In most instances, CO will be the one to initiate the communication.

Messages from CO to POS are encoded in an Extenda specific XML format called Generic External Interface (GEI). Messages arriving from CO at the POS Server are handled by the messaging component, which validates the message and stores it locally before letting the POS Server process it. The messaging component will also reply to CO when the message has been validated and successfully stored.

POS Server to CO

Communication from the POS Server to CO is done by using web service requests, and is initiated by the POS Server. This communication consists of transactions from the POS clients, encoded in ARTS XML POSLog format [2].

The outgoing POSLog message is first saved locally database before being sent to CO.

POS client-server communication

Communication between the server and clients in the shop are performed directly via TCP/IP sockets. The data sent can be binary data, SQL statements or a database dump.

Transactions are sent as binary data, and reference data updates are sent as SQL statements – unless it is a “major update”, in which case the entire database is sent as a database dump.

The client will initiate the communication just after boot. It will peer up with the server and register all the “services” (more on this later) that the server provides. It will also perform a connectivity check (“heartbeat”) at regular intervals to ensure connectivity. The address of the server is a configurable variable set in one of the client configuration files.

The POS Server will maintain a list of currently connected clients.

2.1.3 Data synchronization

Data synchronization in the current system is fairly straightforward as all transactions are handled by the POS Server.

When a retail transaction is committed on a till, it is saved to the local *clienttransactions* database. It is also sent to the server, where it is saved in its *servertransactions* database. In that way, the transaction is saved on both the client and the server.

Maintaining track of the reference database is done by a version control where the POS Server increments the version by 1 for every new update to the database. The SQL query for the corresponding database update is stored in *serverupdates*, and sent to and saved in the POS clients’ corresponding *clientupdates* database. A POS client will receive the “next” update compared to their own database version and apply it, until they have the same database version as the server.

2.1.4 Code structure and principles

The entire retail system is built on Java. The architecture is built upon the dependency inversion principle [3], and utilizes an in-house core framework to facilitate this. The POS client and the POS server share the same source code, but use different “node configurations”, which tells the frameworks which classes to load for the specific application. In this way, several completely different types of applications can be started from the same source, with just a few changes in the configuration files.

From the POS code it is possible to run the POS client, the POS Server, but also a large variety of different client configurations (touch client, self-checkout client, test client with fake peripherals, etc) and even other types of services depending on the same code – for example an implementation called POS-services, which forms the base for a number of potential mobile POS implementations, like wireless selfscan systems.

2.1.5 Code implementation

Most of the source code consists of different modules that can perform simple operations, while being completely independent of each other. This should ensure that changes in part of the code do not have unpredictable effects in other parts of the code. This is achieved by applying the dependency inversion principle and using a service locator. The service locator creates an appropriate object (as per configuration) for an instance during runtime.

Services, requests and responses

The system is event driven and loosely coupled. When an event occurs, for example caused by a user is pressing a button, a `ServiceRequest` object is created with all the information on what action needs to be taken.

The request is then sent to a `ServiceDispatcher` that looks at the request to determine which type of `ServiceHandler` should handle the request. The handler found can either be representing a local or a remote service, which in the remote case is then located at the POS Server.

If it is a remote service, the resulting handler will forward the request to a node where the service is located and then receive the `ServiceResponse` back, which is then returned down the chain back to the original invoker. If the `ServiceDispatcher` cannot find a matching handler for the requested service (for example because the client is not connected to the network), then it will throw an exception.

Because of the loose couplings between the objects, the `ServiceDispatcher` does not need to know anything about how or where the request is handled – it simply has a list of handlers with one or more associated services.

Data synchronization

Data synchronization is done using the same event-driven architecture. When a change is made to one of the transaction databases (for example *serverupdates*), the `RecordReporter` object is notified and ensures that the update is properly delivered to any receivers (be it the POS server or the clients), and their corresponding database.

This is done via the ServiceDispatcher – the RecordReporter on the server sends the new record to the ServiceDispatcher, and it is forwarded to the client where a RecordCollector object handles it and saves it to the database.

2.2 Inversion

As the software explored is built around the dependency inversion principle and inversion of control, a short introduction to the concepts around dependency inversion is introduced in this chapter.

The Dependency Inversion Principle (DIP) “is about the view from one part of your system to another; [you should] strive to have dependencies move towards higher-level (closer to your domain) abstractions” [4]. Schuchert proceeds to point out that DIP “is about the level of the abstraction in the messages sent from your code to the thing it is calling”, in contrast to Inversion of Control (IoC), which is a way of moving the control of the program flow to an external component or Dependency Injection (DI), which is a way to wire uncoupled modules together during runtime execution.

Schuchert defines the difference between the three related expressions with the following: “DI is about wiring, IoC is about direction, and DIP is about shape.” The three are often used together. In this study, only DIP and IoC is explored as DI falls outside of the scope of this project (it is not used in the product).

2.2.1 Dependency Inversion Principle

The concept of the Dependency Inversion Principle (DIP) was first introduced by Robert C. Martin in 1995 in an article called “OO Design Quality Metrics” [5], and later named and further explored in a 1996 article called “The Dependency Inversion Principle” [3]. In the 1996 article, Martin defines the principle as:

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

Definition: The Dependency Inversion Principle

The Dependency Inversion Principle is a way to decouple modules in a computer system. By defining clear modules and interfaces between them, it is possible to build a computer system where modules do not need to have any knowledge of how other parts of the system works.

If the DIP is applied, these interfaces are defined in their own module and defined from the point of view of the higher-lever modules [6]. It is also important that any module in the system only handle issues on its own level. For example, a higher-lever module would never handle database interaction (or indeed even have knowledge that there is a database involved anywhere), and a module handling database interaction would not have any business logic.

If this is done properly, the modules should be completely independent of one another. This has several advantages, which includes better reusability (you can simply move functionality

to a new project or other part of the system) and easier maintenance (a fix in one module could never cause problems in another).

2.2.2 Inversion of Control

Inversion of Control is a way of wiring modules together during runtime and controlling the program flow [7]. In its most used form, it is used in user interfaces to for example wire a method to be activated when a user presses a button. However, in the investigated system it is used more widely – the entire program flow is controlled by an external framework.

When using the Dependency Inversion Principle in designing a computer system, there has to be a way of specifying what implementations to use for the lower-level modules. However, specifying which implementation to use in the higher-level modules would cause rigidity in the system that is unwanted. It would not be possible to change implementation of the lower-level modules without changing the code of the higher-level modules, which goes against the principle.

Inversion of Control is a method of wiring the implementations together without having to limit the options in the source code. Instead, an external framework is used that will populate the different interfaces with implementations during runtime. This can either be done using Dependency Injection or Dependency Lookup [8].

The framework will maintain control of the program flow, and will call into the application code when appropriate. IoC is also referred to as the Hollywood Principle – “Don’t call us, we’ll call you” [9]. Typically, the framework will wire the system according to user-defined configuration files.

3 Theory

3.1 Peer-to-peer networks

The definition mostly used for a peer-to-peer network (as opposed to a network based on the client/server model) is that each peer (or “node”) is sharing part of their resources with the other connected peers [10]. The idea is that all participating nodes are participating on the same conditions, or as “equals”, as described by Kini & Shetty [1].

Kini & Shetty proceeds to suggest that peer-to-peer based architectures are a better way to utilize the resources on a large scale system such as the internet, compared to the traditional client-server model where the server can become a bottleneck.

Schollmeier [10] defines two types of peer-to-peer networks. The pure peer-to-peer networks where all participants are equal, and the hybrid peer-to-peer network where there is some central authority required for some parts of the functionality.

3.1.1 Pure peer-to-peer networks

A pure peer-to-peer network [1] [10] per definition does not utilize any sort of centralized mechanism. As such, peer discovery – the process of finding other nodes to connect to – becomes a challenge.

The process can be executed either by using some type of broadcast or multicast to discover nearby nodes, or by maintaining a list of addresses to possible nodes. The first option is mostly useful in a smaller intranet or an environment that is well-defined by the developer, and the second option can be very limiting for the application’s reach – it might not find many nodes.

On the other hand, not being dependant on any single server for providing the service can be an advantage as a server can be a single point of failure.

3.1.2 Hybrid peer-to-peer networks

There are several types of hybrid peer-to-peer networks. The most simple hybrid peer-to-peer network is based around a log server that provides a list of all connected peers. Any new node wishing to connect can simply ask the server where to find a peer to connect to. There are more complex variations of hybrid peer-to-peer networks, but they are outside the scope of this thesis.

3.1.3 Overlay networks

To facilitate communication between the peers in the peer-to-peer network, an overlay network is created with the connections between the peers. An overlay network is a form of “virtual” network that works on the application level and is based on top of some other network (for example TCP/IP) [11].

The manner in which this overlay network is designed can be essential for the performance of the peer-to-peer network, and usually depends on the type of service the network is intended to provide to its peers.

3.2 Distributed systems

A distributed system is a loosely coupled system of several processes working in parallel, where processing times and delays may be arbitrary [12].

Leslie Lamport describes a distributed system as: “A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable” [13].

As shown below, the main issues in a distributed system can typically be traced back to the asynchronous communication and parallel execution which in most cases make it impossible for any single node to have knowledge of a current global state. This can cause difficulties when trying to synchronize and maintaining a coherent dataset throughout the system (data integrity).

Below, we are introducing some of the fundamental concepts used to work around those problems. These aspects are imperative when designing a distributed system as in this project.

3.2.1 Causality

Causality is about the ordering of events within a distributed system. Knowledge of the order of events is useful for many applications, for example for determining the most recent value of a shared variable [14] or the order in which database updates should take place [15].

While the ordering of events in a single-threaded process is trivial, Lamport shows in his article “Time, Clocks, and the Ordering of Events in a Distributed System” [16] that the ordering in a parallel distributed system is not always possible to determine. He also defines the causality concept of “happened-before”, with which we can define a partial ordering of events in a distributed system.

Lamport defined the happened-before relation (\Rightarrow) as:

Definition A1. If events a and b occur on the same process, $a \Rightarrow b$ if the occurrence of event a preceded the occurrence of event b .

Definition A2. If event a is the sending of a message and event b is the reception of the message sent in event a , $a \Rightarrow b$.

Lamport timestamps

Lamport suggested [16] that the partial ordering can be decided by using “logical clocks”, which are simple integers. If every process starts at 0 and then increment by 1 every time there is a local event, every process would keep its own local time. By adding the sender's current time to all messages sent, the logical clock is updated at the receiver to match *Definition A2*.

In this way, Lamport timestamps satisfies the happened-before relation in an implementation of a distributed system. However, sometimes it will report events as being ordered when they are in fact concurrent.

Lamport timestamps exhibit the weak clock condition, which states that, for events a and b [14]:

Definition A3. $\mathbf{a} \rightarrow \mathbf{b} \Rightarrow L(\mathbf{a}) < L(\mathbf{b})$, where $L(x)$ is the logical time of event x .

Vector clocks

Colin J.Fidge [17] describes an improvement to the Lamport timestamps that can provide all possible ordering of events in a distributed system, thereby being able to identify all causal relationships in the system and specifying which events are in fact concurrent. This is done by each process maintaining a vector of logical clocks keeping track of the last known logical time of each of the other processes. This entire vector is then included in every message.

This way, the receiver do not only know at what logical time the message was sent, but also what information the sender had when sending the message. Using this information some conflict resolver in any of the processes can resolve conflicting concurrent events.

The rules for comparing the vectors of a system based on vector clocks are defined as [14]:

Definition A4.

$$\begin{aligned}
 \mathbf{V} = \mathbf{W} &\Leftrightarrow 0 \leq j \leq N - 1 : \mathbf{V}[j] = \mathbf{W}[j] \\
 \mathbf{V} \leq \mathbf{W} &\Leftrightarrow 0 \leq j \leq N - 1 : \mathbf{V}[j] \leq \mathbf{W}[j] \\
 \mathbf{V} < \mathbf{W} &\Leftrightarrow \mathbf{V} \leq \mathbf{W} \text{ and } \exists j \text{ such that } \mathbf{V}[j] < \mathbf{W}[j] \\
 \mathbf{V} \parallel \mathbf{W} &\Leftrightarrow \exists k \text{ such that } \mathbf{V}[k] < \mathbf{W}[k] \text{ and } \exists j \text{ such that } \\
 &\mathbf{V}[j] > \mathbf{W}[j]. \quad \square
 \end{aligned}$$

Note that we can now define parallel (\parallel) clock vectors. This means that for event a and b :

Definition A5.

$$\begin{aligned}
 a = b &\Leftrightarrow \mathbf{V}(a) = \mathbf{V}(b) \\
 a \rightarrow b &\Leftrightarrow \mathbf{V}(a) < \mathbf{V}(b) \\
 a \parallel b &\Leftrightarrow \mathbf{V}(a) \parallel \mathbf{V}(b)
 \end{aligned}$$

This is called the strong clock condition. Torres-Rojas & Ahmad suggests that this type of clock characterizes causality, while a clock that satisfies only the weak clock condition is consistent with causality.

The main drawback with Vector Clocks is that the data structures for maintaining the clocks will grow linear with the number of participants in the distributed system. The consequence of this is that for very large systems, this can be very expensive both concerning bandwidth when sending messages and concerning processing power when comparing large clock vectors [15]. This can be counteracted by carefully designing garbage collectors and routines for maintaining reasonably sizes vectors [18].

Other logical clock algorithms

There are several other, later, algorithms based on logical clocks as well. Francisco Torres-Rojas and Mustaque Ahamad [14] provides a cheaper variant of the vector clocks algorithm, that will however sometimes order concurrent events. They call it plausible clocks, and in tests they have performed the algorithm ordered 93% of the event pairs in the same way that vector clocks would have in a system with 76 sites and 96 million event pairs using only 7 clocks.

Almeida, et al., [19] suggests a vector clock algorithm for dynamic distributed systems where the number of nodes vary over time. They call this Interval Tree Clocks. The algorithm is based around three basic functions; fork, event and join, through which the causality of all the classic distributed operations can be described.

Causality in practice

In practice, causality can have several different applications, including being able to collect a global state of a distributed system [16] which is useful for among other things debugging.

Causality is also useful when it is important that certain events are applied in the same order on a number of loosely coupled nodes, for example a distributed database or file system. In this case, the order of a read and write could make a big difference for an application. Implementing a state machine with a logical clock that can provide some total ordering of events is a very useful way of maintaining consistency throughout the system.

For example, using logical clocks, a read operation on a specific node could be delayed until such a point that the node can be sure that no write operation happened anywhere else causally before the read operation [15].

3.2.2 Atomicity

Atomicity is about only letting one process at a time enter some specific critical section, thereby avoiding conflicting updates to some shared environment. As early as 1965 Dijkstra [20] realized that this was an important field of study, and today this is one of the methods of ensuring data integrity in a distributed system. That is, only allowing one process at a time to perform changes in the shared data set.

Non-blocking

An algorithm for atomic operations that is non-blocking will progress in some bounded amount of time.

A non-blocking algorithm is considered lock-free if, at any time, some thread is always allowed to make progress (ie, the system as a whole will never halt), although a specific thread may run into starvation [21].

A non-blocking algorithm is considered wait-free if every thread will continue to make progress in a bounded time, even facing delays or failures of other threads [21].

ACID

Härder and Reuter [22] defined four properties for database transactions that are often associated with persistent data in distributed systems. These properties are atomicity, consistency, isolation and durability (ACID). The idea is that a user should never be able to see any “half-finished” transactions – it’s a principle of complete success, or complete failure of a transaction. If it is successful, then it will be changed as an atomic operation regardless of the size of the operation, and it will remain successful.

These transaction properties allow several operations to be performed at the same time, but if they try to change the same data then one of the operations will fail, which means the failed operation will look like it has not been performed at all. To fulfill ACID requirements, the

database manager has to be able to isolate different processes from each other's database operations.

Semaphores

Mutual exclusion is one of the first ways to provide atomicity in a loosely coupled system. The concept of semaphores in this area was first introduced by Dijkstra, who introduced an algorithm for mutual exclusion [20] and a definition of a semaphore [23].

Dijkstra introduced semaphores as a new type of primitive to simplify the solution of the mutual exclusion problem. They are a type of non-negative integers that allows for an atomic operation of either increasing its value by 1 (the "V-operation"), or decreasing the value by 1 if the result is non-negative (the "P-operation").

The idea is that there are X resources available, and to use one of them a process has to use the P-operation. If it fails, there are no resources currently available. When a process is finished, the resource is released using the V-operation.

This makes the mutual exclusion problem simpler, as the P-operation would otherwise (on a normal integer for example) be two different operations (first reduce the value by one, then check the result to see if it is negative), where another process could potentially change the value of the semaphore between these operations.

Lamport's bakery algorithm

Leslie Lamport [24] suggested an improved mutex algorithm to Dijkstra's original problem, which he called the bakery algorithm. Lamport describes it as based on a system commonly used in bakeries, where the customers will receive a number upon entering the shop. The customers are then served in the numbered order.

This is a true solution for the mutual exclusion problem, in that it does not require any lower-level mutual exclusion such as semaphores [25, p. 12].

Compare-and-swap

Compare-and-swap (CAS) is an optimistic (it is assumed to be correct until the CAS operation fails) way of performing some atomic calculation based on a value and then updating the value with the result. It relies on an atomic CAS primitive that will take a variable, a value, and a new value. It will then compare the value with the current value of the variable and if they match it will replace it with the new value.

This can be of use in any implementation where an algorithm reads a value, performs some calculations and then needs to replace the old value with the new value. In order to ensure that no other thread has changed the value in the mean time, CAS can be used. If CAS fails, the algorithm will have to read the new value and perform all calculations again.

3.2.3 Coordinator

An alternative to implementing some distributed algorithm to perform atomic operations is to use a coordinator, or "leader". In this case, any critical operation has to be approved by the leader before it can be performed. In this way, atomicity can be guaranteed in much the same way as a simple client-server model.

There are two problems with using a coordinator in a distributed system. Firstly, it means that one node, the leader, will need to devote some processing power to coordinating, which can be disruptive for the normal work of that node. The second problem is the leader election problem, where the nodes have to agree on a leader in order for the system to make progress.

Leader election

A leader election in a distributed system is a process in which a number of nodes decides among themselves which node should be the coordinator. After the leader election, only one node should call itself coordinator, and all other nodes know the identity of the coordinator [26].

Garcia-Molina [26] suggests two algorithms for leader election. The first is the Bully Algorithm, in which all nodes will check if any nodes with higher priority (or node ID) are available. If not, it will decide that it is the new leader, and inform all other nodes of that decision. The other nodes will then keep checking that the leader is still available.

This algorithm will not work properly with arbitrary link failures, which may cause a situation where different nodes have different opinions about which node is actually the coordinator.

The second algorithm is called the Invitation Algorithm, in which all nodes first become a leader in their own “group” (of one node), and then merge with other groups that it has full connectivity with.

Sigh & Kurose [27] suggested that in practice, finding a leader may not be enough – the elected leader should in fact be the best suited leader. They suggest different methods of letting the nodes vote on who should become the coordinator.

3.2.4 Consensus

In the consensus problem, the system must reach a decision when provided with (potentially different) suggestions from the nodes in the system [28]. A consensus algorithm can, in much the same way as Lamport clocks [16], be used to implement a state machine to maintain consistency throughout a distributed system.

However, instead of just tracking the order of events, the system has to agree on which event should be next before it is performed. The safety requirements for consensus are [29]:

- Only a value that has been proposed may be chosen,
- Only a single value is chosen, and
- A process never learns that a value has been chosen unless it actually has been.

Consensus can be perceived as difficult to implement in a distributed system, as much of the research is very theoretical [28]. It includes difficult and sometimes vague concepts such as eventual guarantees where it is not possible to be certain that a property will hold within a bounded amount of time.

Even worse, Fischer, Lynch and Paterson [30] showed that in a completely asynchronous system there are always cases in which consensus cannot be reached if a single process fails. While this is a theoretical result (in practice, it is very unlikely [28]), it shows how difficult this subject is.

Paxos algorithm

The Paxos algorithm was created by Leslie Lamport [31], and is a consensus algorithm most often implemented to drive a state machine [29].

The algorithm defines three types of nodes in a distributed system. The proposer nodes that may suggest a new value to be used next at the beginning of the round. The acceptor nodes that decides which value will be the new value for a given round. Lastly, the learner nodes, that learns and remembers which value was chosen for a particular round. A node in the system can take on several of these roles at the same time.

Paxos, when used to implement a state machine, works by having a proposer propose a new value for the system in the first round. If a majority of the acceptors agree, the learners are informed and the command can be executed. If it fails in the first round, a second round with an increased message number will be attempted. When the algorithm agrees on a value, the instance of Paxos is terminated and a new instance is started.

3.2.5 Eventual consistency

While most users typically expect their system to demonstrate ACID [22] properties, in some larger systems this may affect availability negatively [32]. Furthermore, Gilbert & Lynch could prove that it is in fact impossible to guarantee consistency, availability and partition-tolerance in the same distributed system. Only two out of the three can be achieved at the same time.

Eventual consistency is about being able to maintain availability at the cost of consistency. This means that while the system will always respond, a read operation may in some cases not reflect the latest write operation [33]. Not being able to read the most recent information can result in uncanny effects, especially in business critical systems [34].

Eventual consistency is a weaker property then what can be achieved using atomic operations or a state machine powered by consensus or Lamport clocks. However, it allows for progress in situations where atomic operations, consensus or a logical clock approach would otherwise fail.

3.3 Theory applications

A lot of the important work being done in the area of distributed computing was published during the 1970s and 80s, which is also where most of the related work to this thesis can be found. While this does not necessarily mean nothing has happened since then most of the basic concepts in the field were published during this time period. Most of the concepts have evolved over time to perform better or be able to do more. However, since one of the objectives was to find a simple and easily understandable solution to the problem, an effort has been made to locate the most basic variants of the different approaches.

Synchronization approaches

As the more complex parts of this work is related to the synchronization between the different nodes in the network, this is where most of the research into previous performed work was concentrated.

Most notably the works of Leslie Lamport [16] [24] [25] [29] [31] and to some extent Edsger Dijkstra [20] [23] was utilized in the design process when looking at different approaches to synchronize the system. While Dijkstra's works lay a solid ground to basic synchronization in a distributed system, his approach (semaphores) proved too rigid for this application. Instead, Lamport's more adaptable "logical clocks" approach (or "Lamport clocks") [16] was implemented almost straight out of the article early in the process.

Also used in the design process was some of Garcia-Molina's work on leader election algorithms [26], where one of the algorithms suggested, the "Bully Algorithm", was implemented in a final design.

Useful concepts

Other related work that was not directly used in the designs still added valuable theory around the issues in the different design approaches.

The concept of ACID transactions [22] have been very useful to describe expected behaviors in the system, and define whether a design is working as intended or not. Eventual consistency was not intentionally implemented at first, but ended up being a natural side effect in some specific situations. Gilbert & Lynch's article [32] on the subject was helpful in determining how to look at this issue.

4 Methodology

4.1 Investigation phase

During this phase both the Extenda system and general theory in the area of distributed systems were explored.

Theory and related work

Research was done mainly in the field of distributed systems and peer-to-peer networks. The focus was on algorithms and principles for coordinating an asynchronous distributed system. This was done mainly using library sources, but also some online sources.

The system

The Extenda system, the system where the work was to be applied, was researched in three ways. During the first time, the documentation of the system was of great use.

However, when more specific aspects of the system needed to be looked into two other sources proved more useful. The first was help from the system architects, and the second was reading the source code. Using those sources, a more complete picture of the system could be constructed which was useful for beginning the next phase.

4.2 Design phase

During this phase the knowledge of the system was applied to the different principles researched in the previous phase, in order to decide on the best way forward. At this point, the advisor was consulted in determining what might be the best way to proceed, based on the pros and cons with the different alternatives.

After consulting the advisor, a decision was made to explore two different approaches in order to see the options in action.

4.3 Proof of concept phase

A proof of concept was created for both options, and compared to the current solution. In order to be able to easily see the result, a simulator was used to test the different approaches. A large part of the phase was dedicated to building a solution where a simple version of the system could be used with the simulator.

As the first step was to build a working demonstrating copy of the actual system, a lot of that time went into testing the current client-server solution in the simulating environment. When it performed as expected, the other two options were developed.

While developing the two concepts in the simulator, a lot of problems arose with the implementation where the behavior was not as expected when based on how the current system performed. These problems were solved, and can be seen as indications as to what type of problems will arise when trying to implement these concepts in the actual product.

As problems were found and resolved, parts of the design was revisited and updated accordingly.

The results were analyzed using different test cases, where aspects of the implementation were tested in the simulator. Using this, the key aspects of each used algorithm could be devised. As the results were analyzed a final description of the two concepts and their pros and cons was created.

5 Design

This section includes all design considerations and decisions that were made in this project, and is a specification of what a final design should look like.

5.1 General approach

The general idea with the project is to minimize the impact on the production code when implemented. Because the system in production is very clearly divided into layers, the goal is to only change the code responsible for delivering messages between the nodes. In this way, none of the actual business logic in the tills would need to be changed.

The idea with the proof of concept was to try and determine if this is at all possible, by using the same business logic code in all POC implementations. An effect of this was that it became clear which parts of the code would need to be changed in order to implement this in reality.

Common for all distributed implementations is that all transaction data (receipts) have to be distributed to and stored on all the tills. This is because if some nodes crash, the transactions should still be recoverable from any other node in the system. Because problems with missing transactions from cashed nodes sometimes need to be resolved manually by system administrators it is important that all transactions can always be expected to exist on all nodes.

5.2 Controlled actions

The main focus of this design is to define how a number of special actions, now referred to as “controlled actions”, are to be handled in the system. A controlled action is an action that needs to be handled in a controlled manner – for example a login or resuming a parked receipt.

Generally, a user expects this type of action to demonstrate ACID properties (see 3.2.2). It should for example not be possible for two different tills to resume the same parked receipt. This makes controlled actions more complicated to handle.

Not all actions on a till are controlled actions – committing receipts, for example, have lower requirements. It does not need to demonstrate atomicity or isolation properties, two receipts can thus be ordered in different ways when stored on different nodes – or even be missing from some node. This does not affect the overall functionality of the system.

5.3 Causality

Causality is not strictly a requirement in any of these implementations, as for example receipts do not have to arrive to Centraloffice in the correct order.

5.4 Peer to peer network

The peer-to-peer network is of a pure p2p design (see 3.1.1), which utilizes the fact that all clients are in the same IP network. It uses broadcast to find the other peers in the network and then establishes connections to them.

5.4.1 Network structure

The overlay network is a full mesh network (see Figure 2), which ensures that the time for a message to reach all the connected nodes is relatively low.

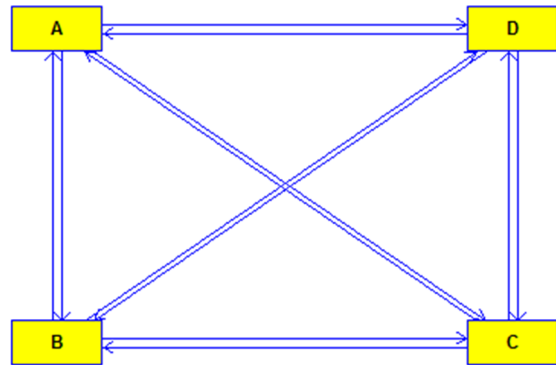


Figure 2: The full mesh overlay network used for testing all the designs.

5.5 Coordinator design alternative

The most straightforward way to implement a peer-to-peer variant of the system is a coordinator approach. In this design, a coordinator (or “leader”) is selected. The leader then performs all of the same duties as the server would. In this way, all controlled actions are ordered and handled by the leader, ensuring consistency.

5.5.1 Leader election

To facilitate this, some type of leader election has to be considered. This is implemented using a simple bully algorithm (see 3.2.3), which is modified to be more rigid in that it will not change leader unless the old leader is unavailable or some other specific criteria is fulfilled.

A new leader will be elected when:

- a) The old leader is no longer available, or
- b) the current leader received a significant priority change and there has not been a leader election for some time (to avoid flapping).

Typically, b) may happen when a user logs into the till on the leader node.

The algorithm for setting priority between the nodes for leader election should take several different considerations into account. The node with the highest priority will be picked as leader in an election.

Current use of the node

If the till in question is currently logged in, it should have a lower priority. This is because some of the coordinator work can be CPU intensive, potentially affecting sales on the till negatively due to delays.

Historical use of the node

Based on the number of transactions created on the till over the past week a lower priority can be set on nodes that historically generated more transactions. This should be an indication of whether the node in question is a till that is being used a lot, thereby being able to avoid electing a node with a very popular till.

A configurable priority

In some cases, a shop may want to manually increase or decrease the likelihood of some node being picked as the leader.

The name of the node

Lastly, the name of the node (till number) can be used to determine which till should be elected leader.

5.5.2 Synchronization

In the client/server configuration all synchronization was done via the POS server and the POS Server kept all records. However, in the coordinator design the leader could change at any time, which means that the new leader needs to know everything the current leader knows.

As it is difficult to predict which leader would take over, all the nodes need to have the same information. This means that instead of sending transactions to just one other node (the POS server), all transactions need to be sent to and stored at all other nodes in the system.

Controlled actions

In this design, all controlled actions are handled by the current leader. In this regard, the leader fills the same role that the POS server has today.

Besides just handling all the controlled actions, the leader also has to inform all other nodes in the system of every controlled action performed, so that each node knows the complete state of the system (in case it becomes leader).

Conflicting updates

In some cases, when the system is segmented due to for example network failure, several leaders can be active and perform controlled actions at the same time. When they reconnect to each other there may be conflicting data on the nodes from the different segments.

Because there can only be one leader, the first step is to select which leader will remain leader, and that leader can then decide on how to proceed with the synchronization.

In this design, the conflicting updates themselves are handled by the higher logic and are not considered here. This is to conform to the current level of abstraction in the product, to avoid invasive changes in the higher logic

5.5.3 Reference data flow

Reference data is pushed to the leader from Centraloffice. To make this possible, a new functionality in Centraloffice has to be implemented to allow for the leader to inform Centraloffice of where to send updates. The leader will then handle the reference data in the same way that a POS server would have.

5.5.4 Transaction data flow

When a retail transaction is created, it is first saved at the local node as a BLOB (Binary Large Object) and then distributed to all other nodes in the system, where it is also saved as a BLOB. This is important so that the transaction is still available if the creating node crashes.

After committing the transaction, the creating node transforms the transaction into a POSlog XML file and sends it to Centraloffice.

It would also be possible to configure it so that instead the current leader transforms the transaction into a POSlog and sends it to Centraloffice. In this case, information that the POSlog has in fact been created (and sent) has to be distributed to the other nodes when it has been sent to Centraloffice. This is to avoid that some transactions are not being sent to Centraloffice at all if the leader changes at the same time as a transaction is committed, as it may not be clear which node is responsible for sending the transaction to Centraloffice at that point. This more complex information exchange is not needed if the node creating the transaction is always responsible for transforming it and sending it to Centraloffice, as it is always clear which node is responsible for sending which transaction.

5.5.5 Points of integration

Integration in the shop that require access to point of sale logic using the POS-services service will most likely require a separate server in the shop. For other, customer-specific integrations, one of the tills could act as a point of integration. Examples of customer-specific integrations could be services such as automated sales solutions like Vensafe [35] vending machines.

Tax refund integration

The integration to Global Blue tax refund requires the communication to Global Blue central servers to be from one specific IP address. One of the nodes should be pre-configured to run the GB service. This decision will not affect any other functionality.

Bottle deposit integration

The Tombra bottle deposit integration will be handled by the leader, and works basically the same way as on a standalone POS server. However, the bottle deposit data fetched by the leader has to be forwarded to all other nodes in the system before being deleted from the bottle deposit server. This is necessary in case the leader crashes.

Another important requirement is that the nodes keep old bottle deposit receipts in storage even after they have been cashed in, for future reference in case of synchronization issues. This is in case some receipts are fetched again from the bottle deposit server after a crash, but have already been cashed in. In that case, it should not be possible to cash them in again.

5.6 Logical clocks design alternative

This design is based around a simple logical clock algorithm for causality-based synchronization (see 3.2.1). Controlled actions are handled by ensuring a total ordering of all events in the system, using a logical clock as baseline. If two controlled actions have the same time stamp, total ordering is achieved by ordering on node name (ie, till number).

5.6.1 Synchronization

Synchronization is done using a logical clock algorithm, which is an algorithm where each node has its own logical clock that is incremented for each action taken by that node. It also has a list of the last reported time from all other nodes in the system. Using this information, each node can independently provide a total ordering between all events in the system. As long as all the nodes apply the same events in the same order, they will all have the same information.

The requirement here is that before applying an event with timestamp N , all other nodes have to have reported a local time $\geq N$. This means that for the system to make progress, all nodes need to be responsive. If some node stops responding, the system will halt until it either starts responding again or the connection to the node times out (at which point the system will proceed without the node).

Controlled actions

As all events in the system are causally ordered, and the ordering is total, no two events can ever be said to be concurrent. This means that all actions do in fact demonstrate atomic properties.

Lazy commit

Constantly keeping up to date with the current time of all other nodes is very resource intensive, as it requires all the nodes to be in contact with all other nodes for any single action performed on any single node. This is not very practical.

The implemented logical clock algorithm is instead lazy in nature – that is, it will not strive to commit a message until it is required in order to process an action originating from itself. Because of this, it is important that the message queue is persistent as it is likely to contain important data that is yet to be committed.

Typically, all nodes will be fully synchronized within two heartbeats if there are no actions taken during this time. The exact timing depends on the heartbeat frequency.

Conflicting updates

In some cases, when the system is segmented due to for example network failure, conflicting updates could be performed in tills that are not currently connected to each other. These have to be handled properly when the tills reconnect.

In this design, conflicting updates are handled by the higher logic and are not considered here. This is to conform to the current level of abstraction in the product, to avoid invasive changes in the higher logic.

5.6.2 Reference data flow

For reference data to be handled correctly, one of the nodes need to be receiving the reference data pushed from Centraloffice. This can be determined by some simple type of priority, for example based on the node name. The selected node have to inform Centraloffice where to send the reference data, which requires new functionality in Centraloffice.

As reference data messages arrive, they are applied as actions, and are thus applied on all nodes in the system in the same order.

5.6.3 Transaction data flow

When a retail transaction is committed, it is handled as an action and is thus committed on all nodes. It is saved on all nodes as a BLOB, and the creating node then transforms the BLOB into a POSlog XML file which is sent to Centraloffice.

5.6.4 Points of integration

Integration in the shop that require access to point of sale logic using the POS-services service will most likely require a separate server in the shop. For other, customer-specific integrations, one of the tills could act as a point of integration. Examples of customer-specific integrations could be services such as automated sales solutions (like Vensafe [35] vending machines).

Tax refund integration

The integration to Global Blue tax refund requires the communication to Global Blue central servers to be from one specific IP address. One of the nodes should be pre-configured to run the GB service. This decision will not affect any other functionality.

Bottle deposit integration

The Tombra bottle deposit integration is implemented by fetching data from an FTP server. To facilitate this, one node has to be selected to fetch this information. This can be done through configuring a specific node for it, allowing the reference data handler perform the integration, or perform some selection (based on current local time on all nodes) singling out a specific node every time the FTP needs to be checked.

Either way, an important requirement here is that the nodes keep old bottle deposit receipts in storage even after they have been cashed in, for future reference in case of synchronization issues. This is in case some receipts are fetched again from the bottle deposit server after a crash, but have already been cashed in. In that case, it should not be possible to cash them in again.

6 Proof of concept

6.1 Purpose and scope

The purpose of the proof of concept is to demonstrate the difference between different approaches to solving the problem with running the system without a dedicated server.

In order to do this, a simulator is used to visually show how the nodes in the system communicate. Each till is represented by a very basic user interface, with the ability to log in/out, create a receipt, shut down/start up the node and disconnect the node from the network. There is also some key information from the till showing in the user interface.

The idea is to be able to test some basic functionality of the tills in different situations, and most of the functionality can be broken down to two categories – things that synchronize with the server when possible, and things that require a reply from the server (controlled actions). In this case, logging in/out is used to simulate things that require a reply from the server while create receipts are things that will sync when possible.

Being able to shut down/start up the till and disconnect/reconnect the till from the network are features that are useable when trying different test cases.

6.2 System overview

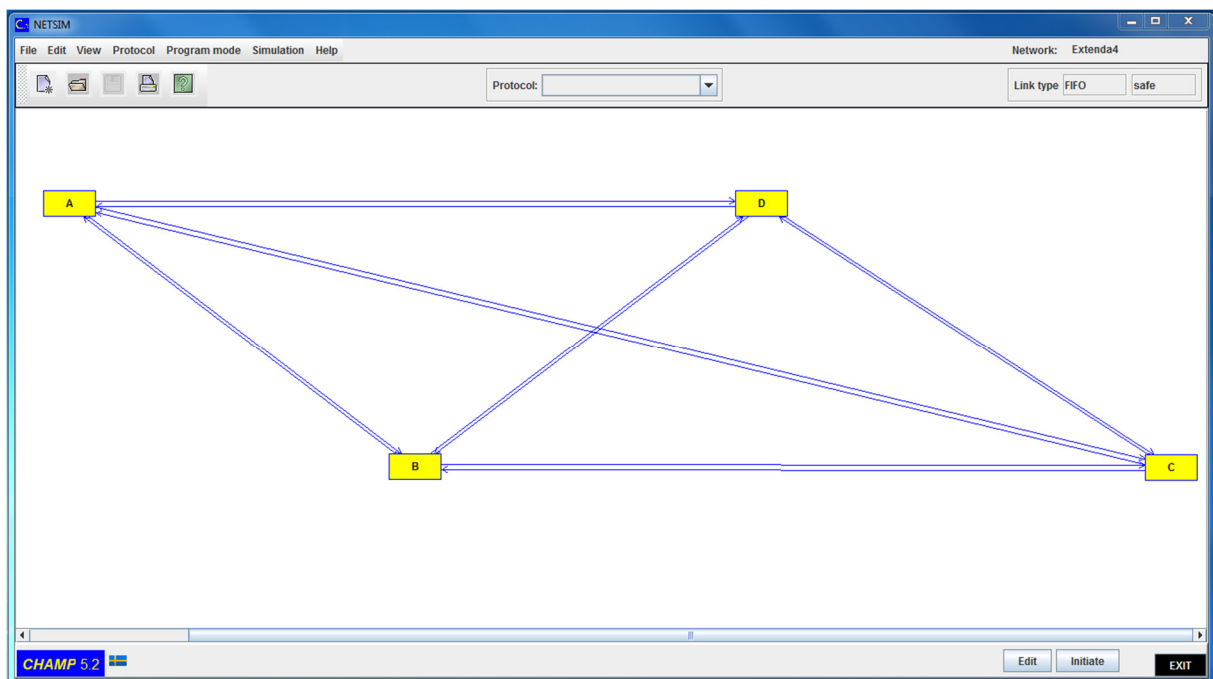


Figure 3: The simulator used for the proof of concept

The proof of concept is based on a networking simulator called Netsim, in which you can program a node in Java to fit your purpose.

In this case, a basic till construct was made with some basic functionality. To simulate the actual system, a service dispatcher was created to operate in much the same way as the

production system. In order to make the tests as realistic as possible the behavior of all components were designed to match the production system as closely as possible.

To test the functionality, a client-server communication implementation was created (Figure 4). As all functionality was working with the server/client system, the alternatives were developed and tested.

6.3 Different approaches

6.3.1 Client-server

For reference, a server/client implementation was built based on the current product (Figure 4).

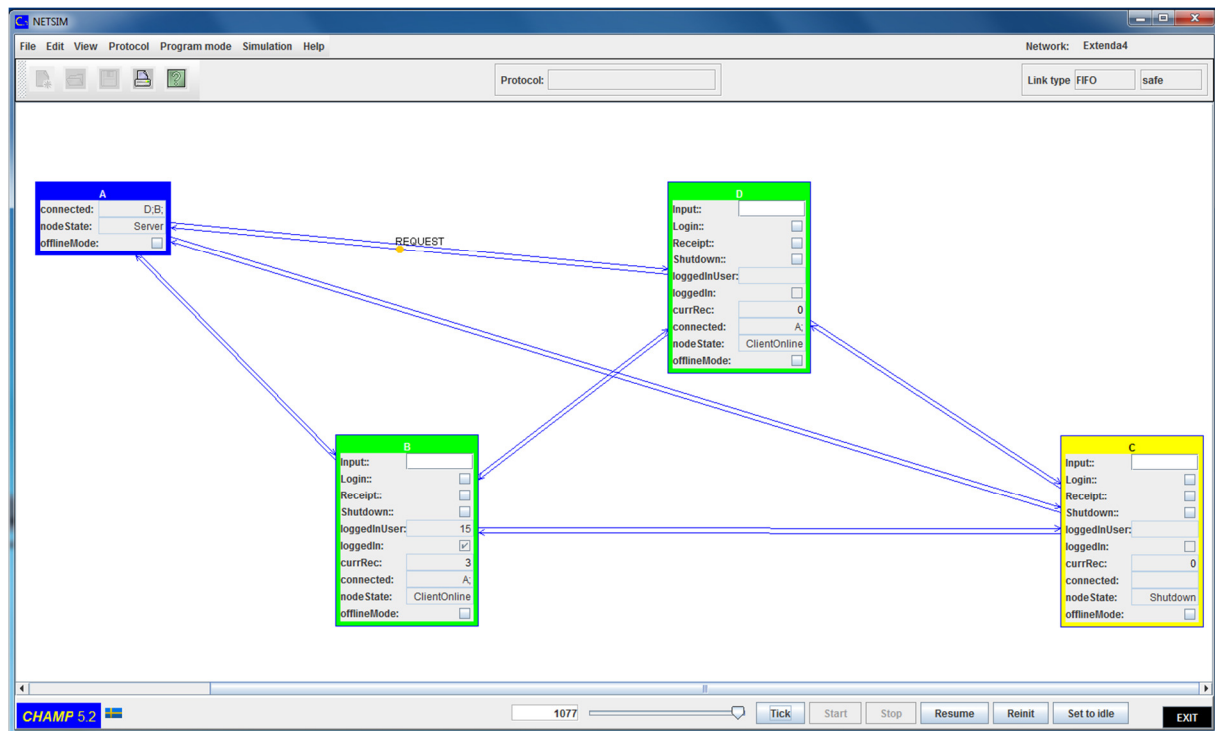


Figure 4: Client-server implementation, with a server (blue), two connected clients (green) and a turned off client (yellow) with a message being sent.

6.3.2 Coordinator

One of the options explored was to elect a coordinator or “leader” based on some priority and have that node act as server. This is a simple solution because it does not require much special consideration from developers during development or when troubleshooting an incident.

The most complex aspect of this solution is how to handle leader transition, when for some reason a new leader has to be elected.

This implementation is still largely built on the same principle as the standard server/client one, which means it has most of the same advantages and disadvantages as such a solution. Only minor changes were done to adapt the POC tills to this approach (see Figure 5).

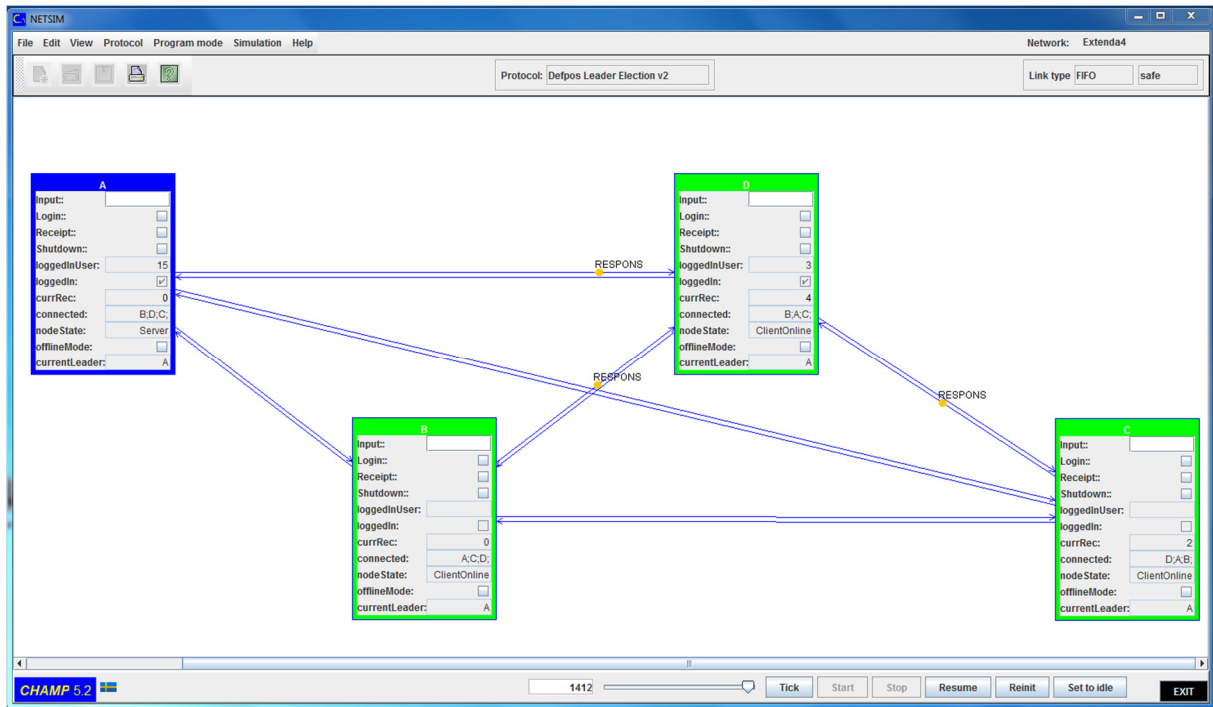


Figure 5: The coordinator implementation, with a coordinator (blue) and 3 other connected nodes (green).

6.3.3 Logical (Lamport) clocks

The other explored option was an implementation based on logical clocks (see Figure 6). This is a more complex solution as it requires an understanding of Lamport timestamps in order to follow what is happening in the system.

This implementation does not have a single entity that can make decisions, which means it is theoretically quicker at recuperating after a node crash.

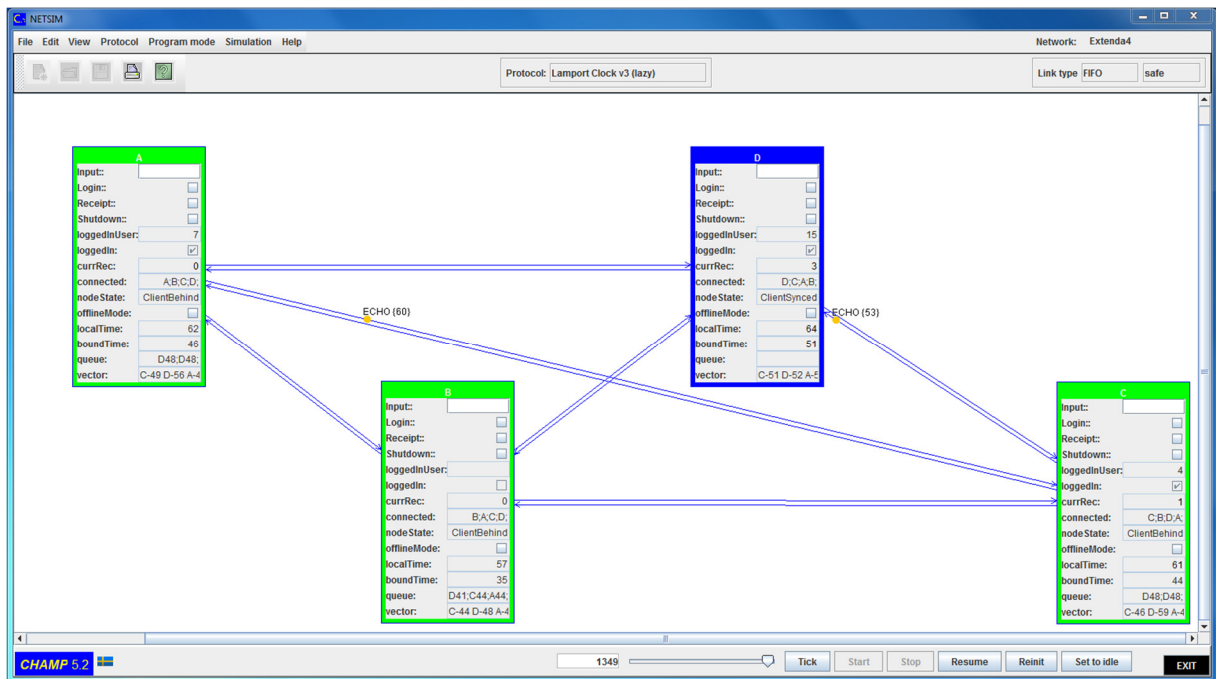


Figure 6: The logical clocks implementation, with 4 connected client. Client D (blue) is completely synchronized, while the other clients (green) are not yet fully synchronized due to the lazy commit.

6.4 Test environment/Proof of concept

The main practical result of this project is a testing environment, where different types of synchronization can be tested on a simplified point of sale node. The core functionality of each node is a service dispatcher, designed to function as closely as possible to the service dispatcher in the actual point of sale system.

6.4.1 Till interface

Each node acting as a till has a basic interface that looks like the one below, giving some information to the user and a few options with which to test different functions (see Figure 7, Figure 8 and Figure 9).

Figure 7: The basic till interface, which is the same for all implementations.

Input – argument to be inputted

Login – Log in or out a user (specify the user name in the input field)

Receipt – Create a receipt on the till

Shutdown – Shut down the till

loggedInUser – shows the currently logged in username

loggedIn – is ticked if a user is currently logged in

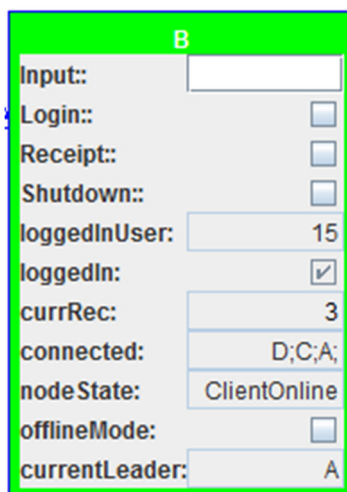
currRec – shows the current (next) receipt number

connected – displays all other nodes that this node is connected to

nodeState – shows the current state of the node (these varies depending in implementation)

offlineMode – simulate disconnecting the network cable from this node

Additional informational fields are also provided on the specific implementations.



The screenshot shows a window titled 'B' with a light gray background. It contains several fields and controls:

- Input:** A text input field.
- Login:** A checkbox, currently unchecked.
- Receipt:** A checkbox, currently unchecked.
- Shutdown:** A checkbox, currently unchecked.
- loggedInUser:** A text field containing the value '15'.
- loggedIn:** A checkbox, currently checked.
- currRec:** A text field containing the value '3'.
- connected:** A text field containing the value 'D;C;A;'. Below it is a list box containing 'ClientOnline'.
- offlineMode:** A checkbox, currently unchecked.
- currentLeader:** A text field containing the value 'A'.

Figure 8: The coordinator implementation till interface.

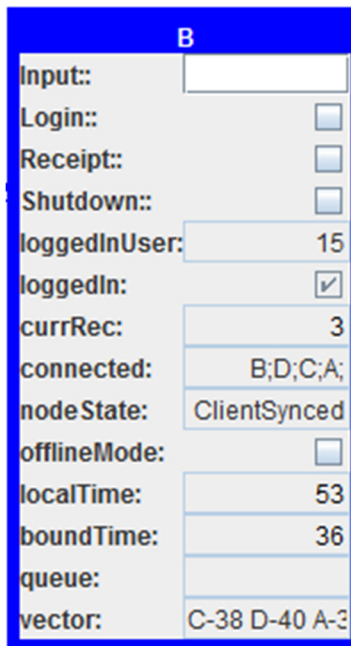


Figure 9: The logical clocks implementation till interface

currentLeader – the node currently winning or the node that has won the leader election

localTime – the current logical time on this node

boundTime – the lowest logical time reported by another node, the latest synchronized time

queue – requests waiting to be handled on this node

vector – the last reported time from each connected node

7 Results

In general, both the coordinator design and the logical clocks design performs almost as good as or better than the original client/server configuration when looking at the protocol's built-in delay times and message complexity in different situations.

7.1 Test cases

To check the performance of the implemented alternatives, a few test cases were devised and tried on the POC. During these tests four fully connected nodes were used, named from A alphabetically descending. Priority in leader election was based on name, in descending order.

In each of the test cases, login/logout operations as well as receipt operations are tried from on all tills. The test cases are performed in order. Both new implementations are tested in these conditions. For reference, the server/client implementation is also tested in each case. For timeout implementations, a heartbeat is sent every 2000 time units and there is a 500 time unit timeout on a connection. A connection has to time out twice to be disregarded. Message travel time between nodes is on average 100 time units.

7.1.1 Start up all nodes

Client/server

All tills online and ready within 200 time units.

Leader election

All tills online and ready within 600 time units.

Logical clocks

All tills online and ready within 1 time unit.

7.1.2 Log in and create a receipt on all tills

Client/server

All receipts committed on server after 300 time units.

Leader election

All receipts committed on all tills after 300 time units.

Logical clocks

Most receipt committed on all tills after 300 time units, a few took up to 400 time units.

7.1.3 Put till A offline

Node A was put offline at time unit 1000.

Client/server

All tills went offline at time unit 5000. No online services available during or after these 4000 units.

Leader election

All tills except till A went offline at time unit 5000. New leader B elected at 5001, at which point B could reach online services. All tills online to new leader B at time unit 5100. No online services available on till C or D during these 4100 time units or B during 4000 time units.

Logical clocks

Online services readily available on till A. Online services available on B, C and D, but with an up to 3000 time units long delay on first invocation.

7.1.4 Shut down till B

Client/Server

No change. Till B shut down. C and D still offline.

Leader election

Till B shut down. Till C took over leadership in 100 time units. Till D online again within 200 time units.

Logical clocks

In worst case results in a 100 time unit delay for online actions on till C or D.

7.1.5 Put till A online

Till A is put online at time 5900.

Client/server

At time 6500 clients attempted a timed reconnect, C and D online at 6700.

Leader election

Till A attempted reconnect at time 6000. After 300 time units till C and D have been notified that till A is taking over as leader. At this point till A and D are online while till C is taking a timeout. After another 1700 time units till C is online and connected with till A as leader. This is at time 7900.

Logical clocks

Till A attempted reconnect at time 6000. The connection does not affect the status of any of the connected node.

7.1.6 Put till C offline

Client/server

Till C loses all online functionality.

Leader election

No apparent effect, till C is elected as its own leader.

Logical clocks

D and A risk an up to 3000 time unit delay on attempted online actions. C will perform actions without problems.

7.1.7 Put till C online

Client/server

200 time units for C to connect with server again.

Leader election

200 time units for C to connect to the other nodes, then 1700 time units with timeout and reconfiguration to become an online client.

Logical clocks

No apparent effect.

7.1.8 Start up till B

Client/server

Till B online after 200 time units.

Leader election

Till B online after 200 time units.

Logical clocks

Till B online at bootup.

7.1.9 Test login of the same user on all tills at the same time

Client/server

First message to arrive to the server wins the login.

Leader election

First message to arrive to the leader wins the login, which will always be the current leader.

Logical clocks

Till with the currently lowest logical clock will win. If two are the same, they are prioritized on till name. Message order is not relevant.

7.2 Network traffic in general

In order for any these distributed protocols to work, it requires some extra network traffic. This is because all updates performed on a till (for example creating a receipt) need to be replicated to all other tills instead of just the server. The network traffic required for synchronizing with a new node is comparable to the amount of traffic generated in the client/server model in the same situation, as the first node that the new node connects to should be able to send all required data directly when connecting.

7.3 Death of a node

When a node shuts down or is otherwise unexpectedly lost from the network, the system usually needs to reconfigure. Although this is done somewhat differently in the coordinator approach and the logical clock approach, it turns out that in both cases the difficult part is actually detecting the death. Because this is done using a timeout there is typically a small time slot where the system is not functioning correctly because some node is not responding properly.

In the coordinator approach, this is most notable if the leader is the dead node. During the timeout, controlled actions are still sent to the leader node but are never handled – and thus never approved. This has the potential of resulting in lost information if not handled properly by the higher logic.

In the logical clock approach, the controlled actions are sent to all nodes but only approved when all nodes report ready. This means that during a timeout all controlled actions are put on hold. While this means no information is lost, it could cause a noticeable delay for the user performing the action.

7.4 Birth of a node

When a new node is introduced to the system, this does not normally affect the performance of the other nodes. However, a notable exception is that in the coordinator approach, if another leader connects so that the system has two leaders, the leader with the lower priority will need to revert to being a follower.

This may cause a small window where information could end up being sent to the wrong node and thus get lost.

7.5 Differences between the approaches

Despite the fundamental design differences between the two alternatives, the perceived functionality is very similar. There are however some key differences in behavior.

Most notable is the behavior when a node disconnects or connects. Generally, it turns out that the logical clock approach is better at handling new connections (they sync up faster), while the coordinator approach is better at handling nodes that disconnect.

7.5.1 Workload distribution

When performing controlled actions, the two different approaches perform the operations in slightly different ways. In the coordinator approach, the *leader informs all other nodes* of the changes being made. In the logical clock approach, the *performing node informs all other nodes* of the changes being made, and the time at which the change was made.

This is important when considering node workload. In the logical clock approach the workload, as well as the network load, will be much more balanced among the nodes when comparing to the coordinator approach where the leader will always have a considerably higher workload.

This also means that the logical clock approach will, at least theoretically, scale better.

7.5.2 Local performance

A client in the coordinator approach will have the same performance as a client in the current system, as there are no apparent changes in behavior depending. The current leader, on the other hand, will need to keep track of all the same information as a server normally would on top of running the till application for the user. This may result in degraded performance on the till in question.

In the logical clock approach, every node has to have an idea of what every other node is doing. To ensure this, every node has to keep track of the latest received timestamp from every other node as well as handle a message queue. The vector with timestamps will grow proportionally to the number of nodes in the network. The number of messages normally in the queue will be related to the number of nodes actively sending messages. All in all, this means that the performance of every single till may go down slowly as the network grows larger and more active.

Additionally, creating POSlog files at the individual nodes rather than a dedicated server is likely to cause a slight performance loss at the tills. It is not clear at this point whether this would be noticeable for the user.

7.6 Design issues

7.6.1 Data ownership and conflicting updates

Because there is no central control or management in the system, a node will maintain ownership of any data produced by it. This means that if a specific node receives reference data from Centraloffice that is then distributed to the other nodes, the other nodes need to not only store the reference data but also where the data was produced (which node owns the data).

So why is this important? It turns out that in some cases different nodes can produce conflicting data in parallel, for example while the network is segmented. To be able to merge this data later on, it is important to not only have the specific data but also some way of determining which data is “correct”. A straightforward approach is to let the data from the highest priority node (calculated in some consistent way) win, ensuring all nodes will have consistent data over time.

7.6.2 Data storage

Because every node owns its own data, every node also need to ensure that the data created is sufficiently secured in case the node fails. Typically, this is done by sending the data to all other nodes.

8 Discussion

While the original plan was to have a finished solution by the end of the thesis project, the scope was somewhat altered during the course of the project. This was due to the fact that it was more difficult than expected to decide on the best approach as several different approaches to the problem have similar properties.

Instead of implementing a solution, the thesis work was concentrated on comparing a few different solutions in order to be able to find the best suited solution and be able to suggest an implementation for future work. This means that instead of implementing a single finished implementation, several proof of concept implementations on a smaller scale were constructed and used for testing and simulation.

8.1 Failures and unknown states

If a node crashes while performing an action, it could be difficult (or impossible) to determine whether this action was actually finished or not. This is an important aspect to consider when handling controlled actions.

For example, if a leader crashes when handling a login request, it is possible that the login request is approved, but the result is only distributed to some of the other nodes. In that case, the different nodes do not agree on the current state of the system. The problem will be most noticeable if the node requesting the login and the new leader disagree on the end result.

Completely eliminating this problem would require a lot of overhead network traffic. Instead, regularly performing some kind of sanity check on the system to ensure consistency is probably a more efficient way of dealing with the problem. This would be a kind of eventual consistency, which could have some undesirable results (see 3.2.5). However, if it is only used for error handling it may be a better alternative than the overhead required to avoid the problems entirely.

8.2 Design choices and designs not explored

Despite having expanded the scope to include more than one design alternative, several alternatives had to be discarded in an early stage due to time considerations.

8.2.1 The consensus approach

A consensus approach was considered, based on the Paxos (see 3.2.4) algorithm. However, this was discarded due to the complexity of maintaining accurate information regarding the number of nodes in a single system. Because a consensus algorithm is based around majority decisions it needs to be clear how many nodes there actually are in the system, which is not a desirable requirement as shops should be able to add or remove tills as needed.

8.2.2 The atomic operation approach

Another possible solution to the problem would have been an atomic operation approach, with either underlying compare-and-swap or semaphore primitives, or using Lamport's bakery algorithm (see 3.2.2). This approach, however, was not considered further as it was not believed to add anything of value compared to the coordinator approach or the logical clock approach.

8.2.3 Distributed database

The option of using an underlying distributed database to distribute information in the system was considered, but discarded as it would require too heavy modifications to the current system architecture. It is also not clear that this approach would solve the problem – additional logic to handle controlled actions would still be needed and intimate knowledge of the inner workings of the database manager would likely be needed in order to ensure correct behavior.

8.2.4 Mixed approach

While looking at this problem it became increasingly clear that while each approach can solve the problem in its own way, a mixed approach may be a better solution. For example, a logical clock approach with some services being handled by a coordinator. This has however not been further explored.

8.2.5 Issues not looked at

Due to the change of scope of the thesis project, some issues were not taken into consideration when designing and comparing the alternative designs.

POSlog handling

In order to send transactions to Centraloffice a POSlog XML file has to be created from the transaction and then sent. In this thesis, we generally assume this operation is performed locally at the same node that created the transaction. However, no consideration has been taken regarding the performance implications of this behavior.

No functionality for tracking which records have been sent to Centraloffice and not have been considered either. This could be an important consideration if a design decision is made to not create POSlogs locally, but rather at some specific node in the system. The system as a whole would then need to keep track of which POSlogs have not yet been created in case of failure in some nodes.

Customer specific store-wide integration

The POS server is commonly used as an integration point in the shop between the Extenda system and other systems in the shop (such as for example vending machines). While this has not been specifically considered, these integrations can likely be built to be run on the leader in the leader election approach.

If it is not possible to have the integration point move around in the shop, the implementation will likely require the shop to run a POS server.

8.3 Usability

The main target of this project was small shops, where the POS server accounts for a large portion of the hardware at the retail chain. This is also where this solution is best suited to be implemented.

This solution is not well suited to be implemented in large shops, as the network traffic and workload on all tills will increase linear with the number of nodes in the system. This means

that the system will not perform well in a large environment. In these cases, maintaining the POS server is a more efficient way of handling the workload.

8.4 Future work

This thesis project has outlined a possible direction for further development of the product. The next logical step would be to determine how to technically implement this into the current product. Because of the architecture of the product, implementing this change should not be a particularly invasive process.

8.5 Other applications of the work

Much of the work done here should prove useful for any generic project where a server/client system is to be transformed into a peer-to-peer based system. Much of the same problems should appear in any generic system with similar traits, and the concept of controlled actions versus actions that do not require the same control in regard to ordering or consensus should be useful when considering other systems.

9 Conclusions

The purpose of this thesis was to find a solution to the problem of removing the in-shop local server in a retail (point of sale, POS) system, and instead letting the clients (POS client or till) communicate directly with each other and the enterprise wide Centraloffice (CO) management server.

The objectives for the thesis were as follows:

1. *Find a method to replace the functionality of the POS Server and put it in the POS clients or CO servers.*
2. *Design a way to implement this, fulfilling the following objectives:*
 - a. *It should be possible to perform sales at the client even if the network connection is disrupted.*
 - b. *Receipts should be saved in strictly more than one place in the shop.*
 - c. *There should only be one active connection to CO from any given shop at any time.*
3. *Build a proof of concept to show that the design works as expected.*

The three objectives were achieved by first looking at the current system and deriving what the current functionality of the local server (POSserver) was in the shop. Using other work in the field and articles as a base, two approaches based on a peer-to-peer structure were designed and described. One design was based on one of the nodes acting as a leader, and the other was based around a logical clock approach first suggested by Leslie Lamport (see 3.2.1).

A simulator to test the base functionality of the system was designed, and both designs as well as the current client/server system were tested and compared in the simulator. In the end, most of the objectives were fulfilled:

1. Two different methods of replacing the POS Server functionality and moving it to the POS clients have been identified.
2. Two designs have been built, although they have not been designed detailed enough to be able to confirm whether requirement c can be fulfilled or not.
3. Proof of concepts for both variants have been implemented.

The result showed small differences in performance in some situation, but the most important difference was simplicity and stability in the leader approach versus adaptability in the logical clock approach. Both implementations, however, require more network traffic and would create a higher workload on the clients.

10 References

- [1] U. A. Kini and S. Shetty, "Peer-to-Peer networking," *Resonance*, vol. 6, no. 12, pp. 69-79, 2001.
- [2] National Retail Federation (NRF), "XML Schemas | NRF," 2014. [Online]. Available: <https://nrf.com/resources/retail-technology-standards/xml-schemas>. [Accessed 10 July 2014].
- [3] R. C. Martin, "The Dependency Inversion Principle," *C++ Report*, May 1996.
- [4] B. L. Schuchert, "DIP in the Wild," 21 May 2013. [Online]. Available: <http://martinfowler.com/articles/dipInTheWild.html>. [Accessed 30 July 2014].
- [5] R. C. Martin, "OO Design Quality Metrics - An Analysis of Dependencies," *C++ Report*, September/October 1995.
- [6] D. Greer, "Examining the Dependency Inversion Principle," 2008. [Online]. Available: <http://aspiringcraftsman.com/2008/12/28/examining-dependency-inversion/>. [Accessed 30 July 2014].
- [7] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 23 January 2004. [Online]. Available: <http://www.martinfowler.com/articles/injection.html>. [Accessed 15 July 2014].
- [8] P. Hammant, A. Hellesoy, J. Tirsén, R. Johnson, M. Royle, S. Curl, M. Tarruela and M. Fowler, "IoC Types," 2003. [Online]. Available: <http://docs.codehaus.org/display/PICO/IoC+Types>.
- [9] M. Fowler, "InversionOfControl," 2005. [Online]. Available: <http://martinfowler.com/bliki/InversionOfControl.html>. [Accessed 30 July 2014].
- [10] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Peer-to-Peer Computing: Proceedings of the International Conference 2001: Linköping, Sweden*, Linköping, 2001.
- [11] B. Porter, F. Taiani and G. Coulson, "Generalised Repair for Overlay Networks," in *SRDS '06. 25th IEEE Symposium on Reliable Distributed Systems*, Leeds, 2006.

- [12] F. Mattern and R. Schwarz, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, no. 3, pp. 149-174, 1994.
- [13] L. Lamport, "Subject: distribution," 28 May 1987. [Online]. Available: <http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt>. [Accessed 6 August 2014].
- [14] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: constant size logical clocks for distributed systems," *Distributed Computing*, vol. 12, no. 4, pp. 179-195, 1999.
- [15] P. Bailis, "Causality is expensive (and what to do about it)," 5 February 2014. [Online]. Available: <http://www.bailis.org/blog/causality-is-expensive-and-what-to-do-about-it/>. [Accessed 6 August 2014].
- [16] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.
- [17] C. J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," in *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, 1988.
- [18] J. Creasy, "Causality in a Real-World Distributed Consensus Implementation," 5 February 2014. [Online]. Available: <http://blogs.wandisco.com/2014/02/05/causality-real-world-distributed-consensus-implementation/>. [Accessed 6 August 2014].
- [19] P. S. Almeida, C. Baquero and V. Fonte, "Interval Tree Clocks," in *Principles of Distributed Systems*, Luxor, Egypt, 2008.
- [20] E. Dijkstra, "Solutions of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.
- [21] M. Herlihy, V. Luchangco and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *23rd International Conference on Distributed Computing Systems*, Providence, RI, USA, 2003.
- [22] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, no. 4, pp. 287-317, 1983.
- [23] E. W. Dijkstra, *Cooperating Sequential Processes*, F. Genuys ed., Academic Press, 1968.

- [24] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453-455, 1974.
- [25] L. Lamport, "The writings of Leslie Lamport," 10 July 2014. [Online]. Available: <http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html>. [Accessed 7 August 2014].
- [26] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Transactions on Computers*, Vols. C-31, no. 1, pp. 48-59, 1982.
- [27] S. Sigh and J. F. Kurose, "Electing "Good" Leaders," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 184-201, 1994.
- [28] M. K. Aguilera, "Stumbling over Consensus Research: Misunderstandings and Issues," in *Replication - Theory and Practice*, Berlin, Springer Berlin Heidelberg, 2010, pp. 59-72.
- [29] L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51-58, 2001.
- [30] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374-382, 1985.
- [31] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133-169, 1998.
- [32] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51-59, 2002.
- [33] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40-44, 2009.
- [34] J. Creasy, "Why Cassandra Lies to You," 20 June 2013. [Online]. Available: <http://blogs.wandisco.com/2013/06/20/why-cassandra-lies-to-you/>. [Accessed 13 August 2014].
- [35] Vensafe AS, "Vensafe," 2015. [Online]. Available: <http://www.vensafe.com/>. [Accessed 23 02 2015].