Implementation of a transaction-intense multi-user system

Students:
Jon Ekdahl 760328-1614
Thomas Pettersson 760624-0518

Examinator:
Seif Haridi
Advisor:
Thomas Sjöland

Mint advisors:
Wouter van der Wijngaart
Gunnar Karlberg

March 1, 2001

Abstract

This thesis describes the specification and implementation of a payment system where the consumer uses his or her GSM phone for identification, instead of a magnetic card. By using this service the consumer will not have to carry around a wallet with cash, credit cards and bonus cards. The consumer makes a purchase simply by calling a number (and perhaps punching a PIN code). The shop owner has to have a Point Of Sales unit (POS) to communicate with Mint. Both consumers and shop owners can do follow-up of the transactions on their personal Mint web page.

The problem was to build a system that could handle high load (i.e. many concurrent transactions) and that was fault tolerant. Our main focus has been on specifications and implementation of the Mint server, to which all calls are directed, both from the consumer's GSM phone and the shop owners POS unit.

The result of our work is a system built on a transaction based system, the Teligent P90/E system from Teligent AB. The tests show that our system can handle 300 transactions per minute, which is well within the boundaries of the calculated load.

Contents

1 Introduction

1.1 Possible Mint scenarios

Today or in the future when Mint is popular all over the world the following scenario is very possible and it shows the usefulness of Mint as a payment system.

1.1.1 The impulsive person

You are out on a walk and feel that a calm VHS-night is a good idea for yourself and someone you care about. You pass by the local VHS-distributor on your way home and enter. After all the trouble with finding the right movies you reach for your wallet and going for the cash register, but the wallet is not there. You begin to curse to yourself and are just about to return the movies to their right places when you remember something. You don't have any cash nor the creditcard but you do have your GSM-phone, which you always carry around. The night before, you activated your Mint service on your phone and this particular VHS-distributor is a registered Mint retailer. So, happy again, you take the movies back just before another consumer tries to fetch them and return to the cash register. You tell the cashier that you want to pay with your mobile. He or she tells you to call a certain 020-number (free of charge) from your mobile phone and a few seconds later you are out of the shop with the movies and an SMS-receipt plus an additional paper receipt if you wish. The VHS-distributor you just left may have a campaign which lets you rent the 10th movie for free and without bothering Mint has taken care of all the administration for you and the movies you have rented are registered in the bonus system/campaign of the VHS-distributor.

1.2 History of Mint

Mint was registered as a company in February 2000. It was founded by three persons, Wouter van der Wijngaart, Fredric Ankarcrona and Patrik Mossberg. Mint's business idea is to provide a solution for mobile payments, integrated with loyalty programs. Its vision is to move everything in peoples wallets on to a server, including cash, plastic cards (such as credit cards and bonus cards) and eventually even IDs, membership cards and drivers licenses. All you should need to have to access this wallet is your mobile phone, together with a PIN code. You should also be able to log in to your personal Mint web page to check your latest transactions and other Mint specific information.

In February 2000, Mint had developed a test system, including a Point-Of-Sales unit (unit standing in store, from here on called a POS) and a test server coded in LabView with enough database functionality to perform a purchase. The communication between the two consisted of modem communication over a GSM link. In April 2000 the server had been upgraded to a Java program working towards a PostgreSQL database and the POS was now a little fancier with a color touchscreen and some added functionality. This solution was used in a test run in May 2000 in a coffee shop in central Stockholm. The test included 1 POS and about 100 consumers. The feedback from the tests were very positive and Mint decided to go out and apply for financing. The goal was to have a ready production platform by September 2000.

1.3 Goals for thesis work

In order for the Mint system to be able to handle thousands of POS units, it of course needed to be implemented in a very scalable way. The goal of our thesis work was to implement a production version of the Mint server (excluding the database), that could support thousands of POS units and had functionality for activation of consumers and changes that they make to their profile information. The activation is part of the Mint security. Since Mint handles peoples' money, the system has to be reliable and consistent, meaning that all transactions in the system must be traceable. It should also have a high percentage of up-time, otherwise people won't use it.

Our tasks were to suggest a way of implementing the system, including choice of programming language/development platform and middleware, be part of developing the requirement specifications for the system, and finally implement the server software according to the requirements.

In this report we first explain why we chose to work with the tools and methods we have used and how they work. This is explained in section 2 to section 7.

In section 8 and 9 we describe how we implemented the system and we also describe what problems we encountered and how we solved them.

Section 10 discusses the tests we have conducted on the system and some conslusions that we have made based on them.

In the final section (section 11) we suggest improvements that should and probably will be implemented in the future.

2 Hardware and software choice

It was included in our task to aid in the choice of hardware and software. The object was to find the optimal solution for the system as a whole, or at least the best combination of different components. Scalability and fault tolerance were desired system features. We started the work, having in mind a server, or server cluster, that could handle multiple incoming connections through modem pools or telephone switchboards.

It turned out that there were not that many options. Pretty soon we had three candidates for the hardware implementation. The first one was IBM, in which case we would use their hardware, middleware and transaction server. The other alternative was a swedish platform called the P90/E, developed by the telecom company Teligent. The third and last contender was to build our own platfrom from scratch and use hardware from different distributors.

The choice of software was very related to the choice of hardware. For the IBM solution and the own platform solution, we proposed the use of Java to implement the server application. The reasons for this were:

- It was fast enough - the bottleneck would not be the server program, but rather the access of the database or the POS unit software, or the communication between the server and the POS unit. These conclusions were made from experiences from the test run with the prototype, which was implemented in Java.
- It was scalable enough - with an appropriate Virtual Machine (VM), it's possible to spawn thousands of threads [1]. The architecture also allowed for several VM:s to run on different machines.
- It could be implemented fast enough - the implementation of the prototype was done in Java, so we could probably reuse some of that code. We also preferred Java instead of any other languages, such as C++, because of our own experience of developing in Java.

For the P90/E solution there were no choices. The choice of Teligent hardware forced us to use a development tool called Application Builder (see Section 5).

To be able to decide in favor of the different solutions, a more thorough evaluation had to be made. Therefore, estimations were made on costs for the three alternatives, to develop a first version of the system.

2.1 Decision

IBM wanted a lot of money just to make an initial architecture as a basis for cost estimation. Therefore, the IBM alternative was dropped at a very early stage.

The Teligent solution is based on standard Intel PC:s and therefore not very expensive. The cost estimation for this alternative was 10.0 million SEK. Implementation time was approximated to 5 months, including tests.

The alternative where we build our own platform was based on Sun machines. The cost for the hardware and the licenses for OS:es and software was calculated to 12.7 million SEK. No specific time estimation was made but the guess was that this alternative would take a lot more time than using the Teligent solution, with many ready components and experience in the field of telecommunication.

With the estimations of cost and developing time in hand, Mint decided to go with the Teligent solution. The fact that we wanted to implement the solution in Java and build our own platform was overruled, due to the higher price and longer time-to-market.

3 Unified Modeling Language (UML)

Throughout our thesis work we've been working with specifications and requirements for the system using software from Rational. To visualize the solution we have used Rational Rose 2000, which is using UML notation, and to describe it in text we have used Microsoft Word.

In this section we are trying to describe how we made the specifications and to help you understand how the software we used works. To get a good introduction to UML we recommend you to read [2].

3.1 History of UML

Modelling languages have been introduced to the market during 1990s, all with different notations, values and weaknesses. Three of the most popular methods were OMT (Rumbaugh), Booch (Booch) and OOSE (Jacobson) [2]. All three inventors used their own standard, but started to adopt good things from each other. However, they still had their own unique notations. The use of different notations brought confusion to the users since one symbol could have different meanings to different people. They finally came up with an attempt to standardize the artifacts of analysis and design. In October 1995 the first draft (version 0.8) of UML was released. The Object Management Group (OMG) adopted UML as the standard modeling language in November 1997.

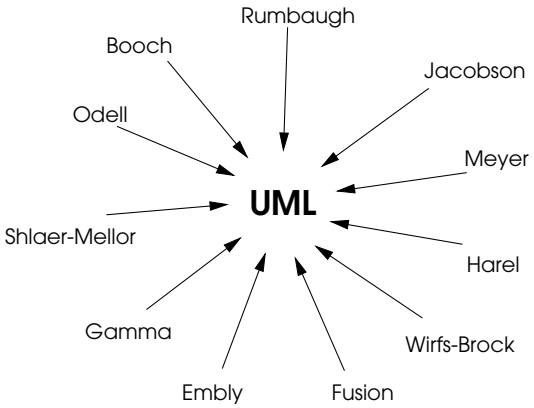

Figure 1: Booch, Jacobson and Rumbaugh had most influence when specifying UML but they were not alone [2].

3.2 Why UML?

UML is a modeling language that is used to visualize a solution with graphs and pictures instead of just ideas in someone's head. Models that are created with UML are useful for understanding problems and promote better understanding of requirements, cleaner designs and more maintainable systems. After working with the programs for a while we realized that they were pretty good and made the specifications more interesting because we at the same time got to know useful programs.

It is very important to build models of complex systems because it is not possible to comprehend such systems in their entirety. In high school, for example, you didn't need to draw models of each assignment you had to do because you could easily visualize them in your head, but the system we have built is far more complex. Models help us organize, visualize, understand and create complex things and also makes it easier for someone else to understand the structure and functionality of the system.

3.3 Use cases and actors

To make a good general model of the system we had to identify all situations (use cases) that the system must be able to handle. The use cases represent the functionality provided by the system. The formal definition for a use case is: "A use case is a sequence of transactions performed by a system that yields a measurable result of values for a particular actor." [2]. A use case in the Mint system is for example `Pay with mint`.

`Actors` are not part of the system - instead they represent someone or something that interacts with the system. An actor can both have input to the system and receive information from the system. An Actor is usually the initiator for a use case. Typical actors in the Mint system are the POS unit and the Consumer.

In figure 2 you can see a use case (Pay with Mint) and an actor (Consumer) and how they look in UML.

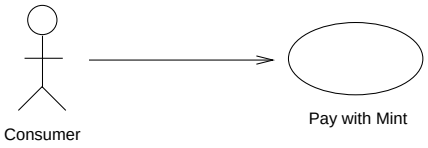

Figure 2: Notation of an actor and a use case.

3.4 Sequence diagrams

Use cases are very general and therefore you need to describe them in more detail in one or several sequence diagrams. For each use case, several sequence diagrams can be used to describe different flows within it. It is not possible to draw different paths in one sequence diagram so instead you will have to draw one sequence diagram for each path. Each sequence diagram contains classes, actors and their interactions with each other.

3.4.1 Classes and objects

To be able to make the sequence diagrams we had to identify the classes and objects we were about to use. A class is a description of a group of objects with common properties, common behaviour, common relationships to other objects, and common semantics. A typical class could be `Database`.

An object is a concept, abstraction or thing with well-defined boundaries and meaning for a system [2]. Each object is an instance of one class. Two typical objects in the Mint system are `Oracle database` and `MDS database`.

A simplified sequence of a successful payment call is visualized in figure 3.

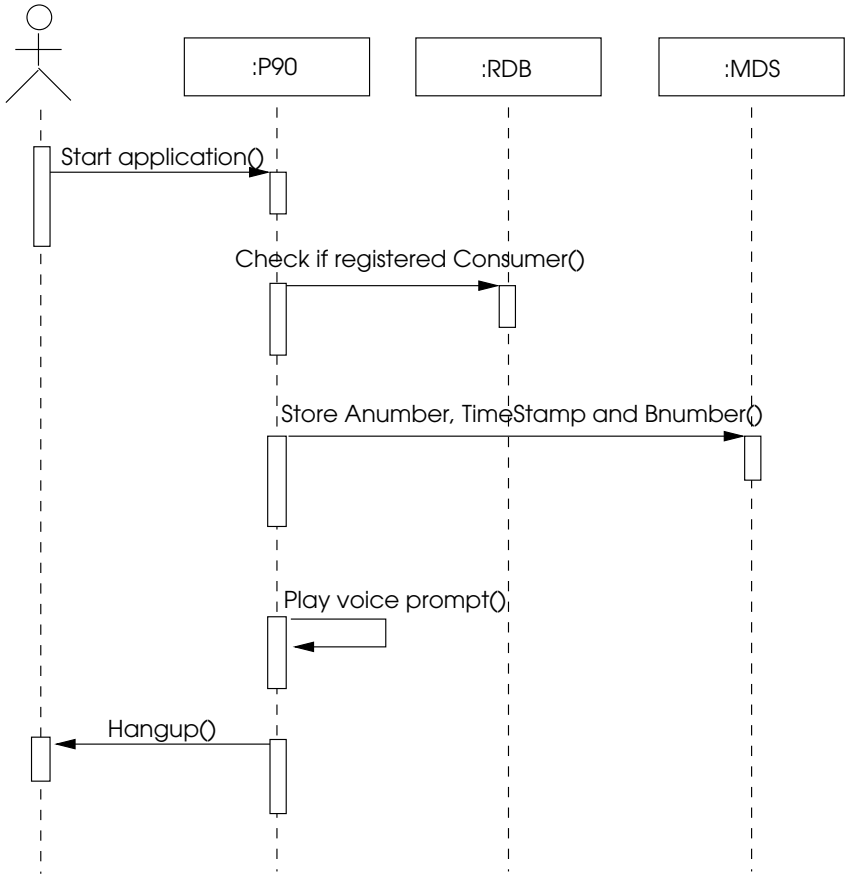

Figure 3: Example of a simple sequence.

3.5 Textual documentation

Each use case must be described in more detail and therefore we have created a small Word document for each one of them and described them in textual form. We have described what/who initiates them, how the main flow and possible alternate flows work, and what must be fulfilled before the use case starts and which consequences it has.

3.6 Use case diagrams

If we put all use cases, actors and their relations to each other together we have a use case diagram of the whole system. This diagram together with the textual documentation makes the system easy to visualize. We see which actors interact with which use cases and the documentation describes this interaction on a very high and understandable level.

After the model was finished we could set requirements for the system and begin to see how things could be done in practice.

3.7 Activity diagrams

An activity diagram could be drawn to visualize the dynamics of one or several use cases. These kinds of diagrams look very similar to the flows we had to draw/implement in Application Builder so we did not draw them in detail using UML. We have done one overall diagram for a part of the system though, see figure 4.

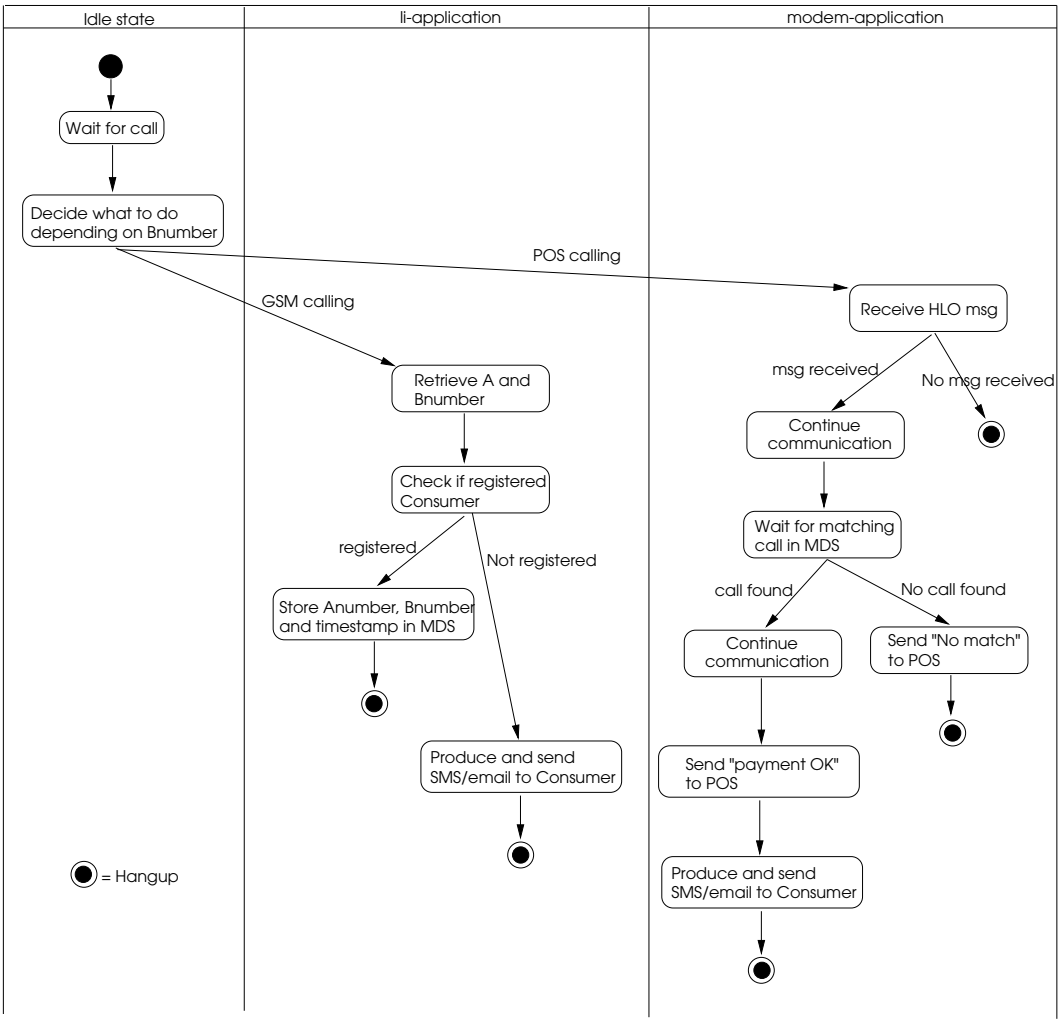

Figure 4: Activity diagram describing a part of the Mint system.

4 The P90 architecture

The P90 architecture [3] is transaction based communication system, implemented on top of TCP/IP. It is comprised of components that send transactions between each other. The most important component is the Transaction Manager (TM), through which all normal transactions are sent. The platform is implemented on top of different Unix variants such as Free BSD and SCO Unix, depending on which type of hardware the machine has.

A machine running the P90 software is called a node, and several nodes working together under one TM is called a segment. It is also possible to have different segments working together to obtain fault tolerance, fail-over functionality and load balancing.

Below, we give a description of how the transaction system is constructed. We also give a short description of the different components we've used in our solution.

4.1 P90 transactions

The P90 uses a message format that is called the Teligent P90 Transaction Format (TPTF) [3] and the TPTF messages are sent using TCP/IP. Each transaction message contains a time stamp, sequence number, the name of the service and the return destination. In addition to this you can send parameters, or FICS:es (Function Independent Containers), depending on which function you're calling. Licences for the TPTF package can be bought if you want to develop your own software that needs to communicate with the P90.

4.2 P90 components

We haven't used all components available for the P90 platform but the ones we have been using are explained in this section. Some of the components didn't fulfill our requirements in their original versions, so they have been changed to fit our purposes.

4.2.1 Transaction Manager - TM

The Transaction Manager (TM) is the center of the P90 architecture. All normal transactions are delivered through the TM. This makes it possible to keep logic for detecting faulty components and timeouts in one place, namely the TM. The TM is responsible for starting the different components in the system at startup and for keeping them alive by restarting them if necessary. The TM communicates with the components through a unique and dedicated socket (one socket per component).

The TM can be used to obtain control, prioritization, load balancing and surveillance. Several variables such as load, peak load and transaction counters are constantly being updated and can be used to monitor and increase performance in the P90 system.

However, since most transactions pass through the TM, it is a single point of failure (there are ways to send transactions directly to another component, without using the TM, but we don't address that in our report).

4.2.2 Transaction Logger - TLG

The TLG component logs transactions that are sent through the system. Usually, only transactions that change the state of the system are logged. The reason for this is that the log can be used to recover the system if it goes down for some reason, which means that there is no interest in logging "read"-transactions. The recovery result would still be the same, no matter if you include them or not.

It is possible to log read transactions as well. This can be very useful when debugging applications.

4.2.3 Application Controller - AC

The AC is responsible for running applications. Each application is run by a Script Interpretor (SI), and one AC can handle 100 SI:s. Several AC:s can be run under one TM so this is not a limiting factor.

4.2.4 Programming Interface - PIF

The PIF is used for connecting to a P90 system from a workstation running the P90 development program Application Builder. It enables a user to send transactions to the system. This makes it possible to run simulations on a workstation, handling the logic in the Application Builder but sending real transactions through the P90 system. At the same time you can follow the flow on your screen which makes it easier to debug the application. This is called a Live On System simulation (LOS).

4.2.5 Relational Database - RDB/RDS

The RDB is used for connecting to an Oracle database. Once connected, you use SQL queries to insert and extract data from the database. It is not possible to call stored procedures (SP:s) from these components. Once you have a connection, via an RDB component, to the database no other applications can use this component.

The solution to this problem is the RDS component. This component is limited to select queries only but allows sharing of the connection to the database.

4.2.6 Main Data Storage - MDS

The MDS is a real-time database for simple but fast queries based on a file database.

4.2.7 Line Interface - LI

The P90 system was designed to be used for telephony applications for intelligent networks (IN services). The LI is the interface towards the telephony system and it supports several different signalling protocols. With the LI you can can route incoming calls as you like, forward them to other locations etc. You can also answer them and play voice prompts to the calling party. This makes it possible to implement services like redirection and mailboxes for voice messages, using voice menus and DTMF tones for navigation.

4.2.8 Modempool component - LIM

The LIM handles the connection to a modempool. The data delivered to the modempool is delivered to the LIM through a virtual serial port. The LIM then delivers the data to the application.

Due to the fact that the initial LIM didn't support binary data, and the Mint application sends encrypted strings using this component, this component was adapted to suit the needs of Mint.

4.2.9 Security component - SEC

The SEC component contains an implementation of the encryption mechanism DESX. DESX is an extension of the Data Encryption Standard (DES) with longer keys, thus providing greater security. The SEC component can also be used for our triple DESX encryption (see section 7.3). This is a component made especially for Mint's purposes.

4.2.10 Alarm Handler - ALH

The ALH is handling alarms in the P90 system. Alarms are created when something abnormal occurs in the system i.e. an application or component crashes, a transaction is not acknowledged in time or at all. The ALH can be configured to act upon alarms in several different ways; when a component crashes it may have to recover by releasing resources allocated by the component or when an application fails for some reason it can send an SMS to the system administrator.

4.2.11 Email Sender - EMS

The EMS is basically an implementation of an SMTP client, allowing you to send email simply by sending a transaction through the P90 system.

4.2.12 Cross System Router - XSR

The XSR or Cross System Router is a component that keeps track of other P90 segments, and which components they have. It also handles the communication with these segments.

4.2.13 Execute component - EXC

The EXC supports only one transaction, Execute Unix Command. This is a sometimes useful shortcut and we use it to run an SMS sending program (see section 8.5) and the sleep command.

5 Application Builder

Applications for the P90 platform are written in a script language called the Transaction Script Language (TSL). There is a toolbox including compiler, linker and libraries for building applications. TSL was developed because the developers found it tedious to write the code in C, which is also possible. The next step in making it easier to build P90 applications was to develop a graphical design program which could generate the TSL code. This graphical design tool was called the Application Builder and it is this program we have used to build our applications.

The basic idea behind the Application Builder is that you draw your program, like a flowchart. In theory, it could be used to generate scripts in any script language but currently, it only supports TSL. Unfortunately, the program is rather new and still has lots of bugs in it, some of which makes it very hard to work with it in a normal way.

5.1 Basic building blocks in the Application Builder

The basic building blocks are different ENTRY- and EXIT-blocks, flow control blocks IF and WHILE, assignment ASSIGN and no operation NOP. There are also the P90-specific transactions which each are one block, and with which you are able to send transactions to the other components in the system. Each specific P90-transaction belongs to a specific component (see Section 4). On top of this, there is the function block, which can contain all of the above. Figure 5 shows a basic example of an Application Builder function.

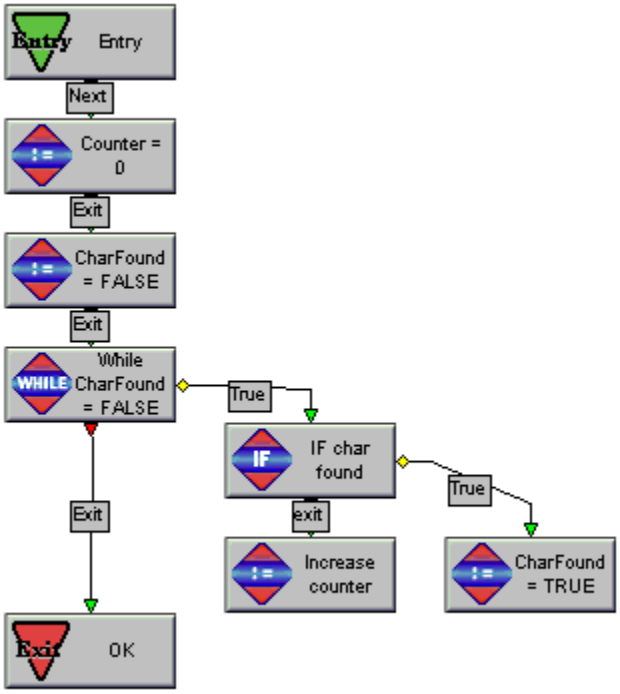

Figure 5: Example of Application Builder function

5.2 Working with the Application Builder

In the top left of Fig. 6 you can see the toolbox, containing defined variables, transactions, interfaces (ENTRY/EXIT), library functions and user defined functions. Below the toolbox you find the IF, WHILE, ASSIGN and NOP. The white area with the boxes to the right is called the workspace and this is where you build your flow. You do it by using drag-n-drop from the sections to the left, putting your building blocks on the workspace, and then connecting the boxes with arrows. On the arrows you can also specify input and output parameters to the functions.

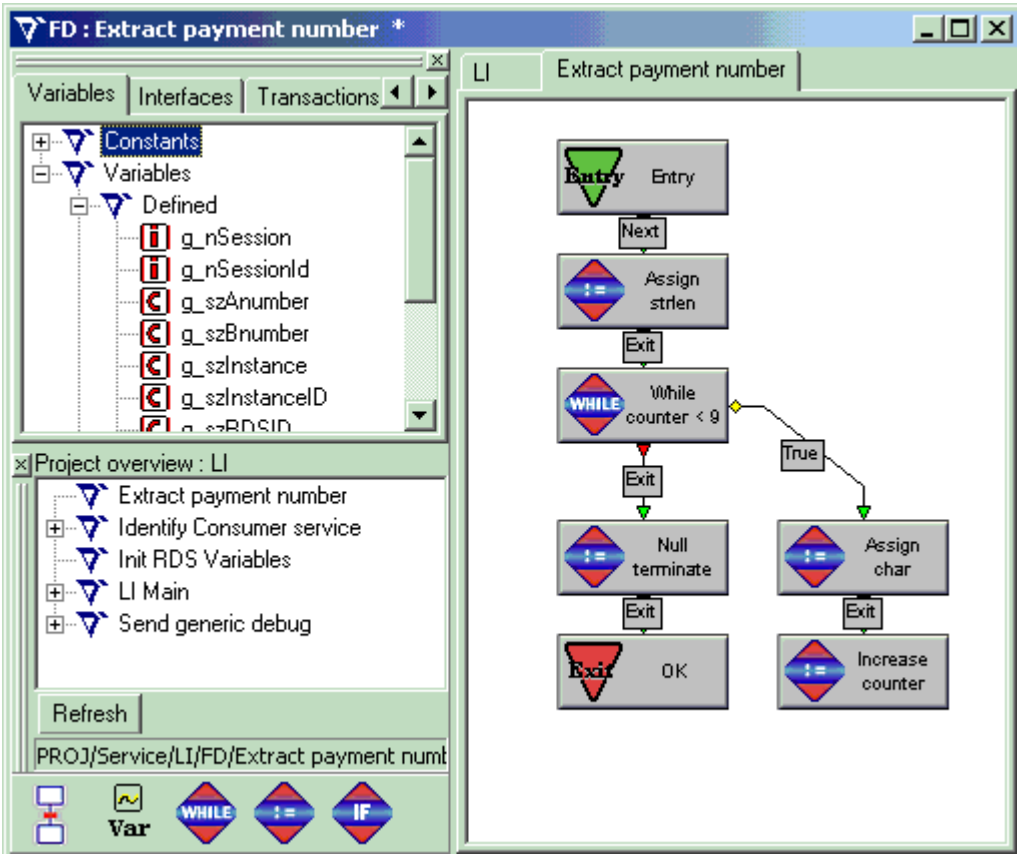

Figure 6: Application Builder

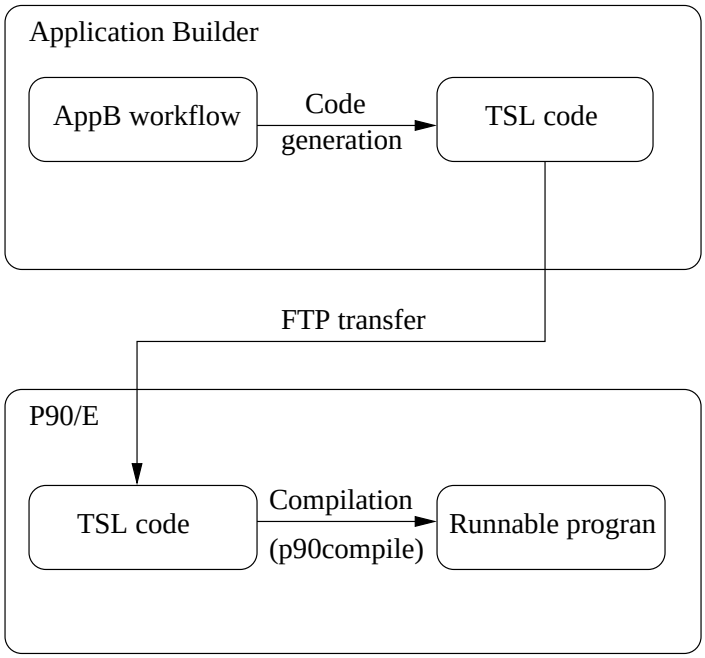

Figure 7: Flowchart of the compilation steps

Once you have built your flow you compile it in Application Builder producing a file with generated TSL code (see Fig. 7). This file is then uploaded to the P90 where it is compiled again. The output from this compilation is a program that you can load in the Application Controller (AC, see Section 4.2).

6 System description

The system we have built is one of the first mobile payment systems in the world. As a registered Mint consumer you can go to a registered Mint retailer and buya product or service by only using your GSM-phone. The registered Mint retailer will, at least in the beginning, have a Mint Point Of Sales(POS) unit that communicates with our platform. The POS unit software is developed by Mint and the plan for the future is that the software can be installed in the retailer's ordinary cash register and therefore it will not be necessary to have an extra POS unit.

6.1 Mint System

The Mint system consists of several, more or less complex, parts. Here we try to describe them all in a very general way for you to get a picture of the whole Mint system.

6.1.1 Payment System

The payment system is the main product from Mint. To make a payment we need three elements: a GSM-phone, a P90/E(P90) platform and a POS unit. The POS unit calls in to the P90 platform with our applications running on it. The P90 receives the call and establishes a connection for further communication. The application communicates according to the POS-protocol (see Section 7) with the POS unit. Simultaneously a GSM phonecall is received by another application (LI application, see Section 8.3) and the two calls are matched togheter in one of the applications and the payment is either accepted or declined depending on whether the calling consumer fulfill certain requirements (i.e. enough credit etc.) or not.

This system will in the future also handle loyalty services such as discount when you buy for a certain amount, every 10th pizza for free etc.

6.1.2 Web system

You can register as a new consumer on the web using an ordinary browser. When you have fulfilled the registration process, including activation of your Mint service using an activation code sent home, you can access your personal Mint site(MyMintSite). On your personal MyMintSite you can check the purchases you have made, change information, PIN-limits and if you have any problem you can get in contact with Mint support. When you register you can choose whether you want to be a postpaid or a prepaid consumer. If you choose the postpaid option (i.e. credit) a creditcheck will be performed online and you will immediately receive an accept or decline.

6.1.3 Economy system

The economy system we define as the system that takes care of the money flow. If you are a prepaid Mint consumer and deposit money on a specific bank account the economy system will register the amount of money to your Mint account. If you are a postpaid Mint consumer the economy system will, every month, send out an invoice on the amount you have made Mint purchases for. This system also handles the money flow to and from the Retailers.

6.1.4 Database

The database is the spider in the web in the Mint system. It is an Oracle8 database and it is running under Linux. All the other systems communicate with the database. The database is a single point of failure and if it should break, the payments, the dynamic webpages and the economy system will stop working. To make the database more failproof the computer it is running on is equipped with RAID discs and backups are done continously. If the database computer crashes Mint's maintenance group will get an alarm and be able to restart the database within a couple of minutes.

7 The POS protocol

In order to communicate between the Mint server and the POS unit, some kind of protocol had to be developed. It had to be flexible as well as fault tolerant. Flexibility was desired in case you ever want to change something. It's quite easy to change the server software but very tediuos to collect and reprogram all POS units, once they are put in store. But if the protocol is flexible enough, it should be possible to change only the server software and the POS units could be left untouched. Fault tolerance is of course a property that is absolutely necessary, there is no room for failure when dealing with peoples' money.

To obtain the flexibility that was needed, the protocol was based on the idea that the POS unit acts only on demand from the server, using messages like "Send payment amount", "Show message 22" and so on. The POS unit was not allowed to take any initiative.

7.1 Messages and message syntax

The POS protocol we have set up consists of several different messages where a few of them can have different meanings depending on submessages within the main message. We will not go into detail on the syntax and structure of the messages as this information is considered security sensitive.

7.2 Security and fault tolerance

In a money transaction system, like the Mint system, error handling is a crucial point. Even if errors occur, the system must handle them and never create incorrect transactions. Since the communication to the Mint server can be performed over a GSM connection, there is a possibility that the communication can be interrupted at any time. When designing the protocol, these issues had to be taken into consideration.

In order for the POS unit and the Mint server to agree on whether a transaction was actually made, we use a kind of Two Phase Commit (2PC). The idea is that the Mint server, with access to the database, checks the consumer's account. If the balance is bigger than the payment amount of the transaction, the transaction is approved. At this point the Mint server adds the transaction to the database, as a pending transaction, this is the first phase of the 2PC. The Mint server then sends the information to the POS unit, telling it to approve the transaction. The POS unit shows "Payment Accepted" on the screen and saves that the transaction was successful. Then it sends an acknowledgement back to the Mint server that the last message was received properly. This means that the Mint system can move the transaction from the pending status to committed.

If the communication would be interrupted at some point when the transaction is pending, the transaction-ID of the pending transaction is saved in the database. When a POS unit contacts the Mint system again, the Mint server checks the database to see if the unit has a pending transaction. If so, a recovery mechanism in the protocol is used, that allows the Mint server to ask for the status of the last transaction that a POS unit made. If the POS unit replies that the last transaction was OK, the pending transaction is committed. Otherwise, it is simply deleted.

7.3 Encryption

There is also the problem of someone attacking the system, either by "wire tapping" or actively trying to contact the system, acting as a POS unit. To protect the system against attacks like this, we use encryption. More specifically, we use an encryption mechanism

called DESX. The encryption algorithm is based on the well known Data Encryption Standard (DES), which is a symmetric algorithm i.e. the same key is used for both encryption and decryption. For more detailed information about the encryption we have been using see [4].

One of the differences between DES and DESX is the key length, DESX uses 128 bit keys while DES uses only 56 bits. This makes the time for an exhaustive key search (trying every key until you find the right one) longer. However, with enough time and computer resources, it is possible to break the encryption. There is also a difference in implementation. The three steps in DESX encryption are:

- XOR the cleartext and the first part of the DESX key.
- The second step of DESX is ordinary DES encryption. The DES key used is the second part of the DESX key.
- Finally, another XOR operation is performed.

7.4 Communication fault tolerance

To handle errors that occur during transmission of protocol messages, every message has a checksum attached to it. It's a regular Cyclic Redundancy Check (CRC) of 4 bytes. When receiving a message, the checksum is recalculated and compared to the checksum received together with the message. If they are equal, the message is considered to have been correctly transmitted.

Every message that is sent begins with an STX character (0x02) and ends with an ETX character (0x03). This is done to be able to delimit the messages from each other. However, since we're sending binary data, these characters can occur inside the messages as well. This means that they have to be escaped, in other words preceded with an escape character. The escape character we use is DLE (0x10). Hence, this character must also be escaped when it occurs inside the messages. An illustration of a typical protocol message is shown in Fig. 8.

STX	ENCTYPE	'#'	MESSAGE	'#'	CHKSUM	ETX

Figure 8: Example of a typical message as sent over the communication link.

8 Implementation

In this section we will explain which applications we have implemented and explain how and why we have chosen the functionality we have. We will go on by discussing some of the problems we had on the way and what we did to solve them. We have implemented all our applications using the program Application Builder (see section 5) except for the SMS sender program, which we implemented in C.

Due to the fact that when a payment is made, we have communication both with a POS unit and a GSM phone, we chose to split the program into two different applications on the platform. The applications that together handle a payment are called `li` and `modem`.

8.1 Scheduling

To be able to update the POS units when they are placed in a store, without going to the store, we had to insert some kind of scheduling for the POS units. It could be done in two ways, either the POS unit could initialize the update by calling the Mint server or the Mint server could call the POS unit. We decided to go for the first alternative because of the similarities with a payment, where the POS unit is calling the Mint server.

We use the scheduling for updates of parameters in the POS units and security updates.

The POS units have the next update time stored in memory and polls that field continuously to check whether the current time is after any of that time. If that is the case the POS unit automatically calls the Mint server, which has the same time stored in the database. If the Mint server and the POS unit have the same time stored for update they agree on the service type and the scheduled event is performed. If somehow the times are different and the POS unit is calling too early, the Mint server thinks it is a payment and sends out a `STP#PAY` message toghether with the next update time in the database (see section 7) and after the failed payment (because the application can't find a matching call) the times are equal again.

8.2 Activation

In the Mint system we have an activation procedure. When a consumer registers for the first time an activation code is generated and sent, by ordinary mail, to the Consumer's home address. To be able to pay with your mobile you first have to activate the service by dialling a certain telephone number and punch your activation code. This procedure has to be done from the registered mobile phone number the consumer entered during the registration process.

This activation procedure prevents us from having someone registering someone else's mobile phone number and immediately be able to pay with it.

When sensitive personal information (i.e. information such as mobile phone number, address, pin code etc.) is changed at the personal MyMintSite the activation code procedure is triggered. The sensitive changes are stored in a separate table in the database where they are pending until the consumer calls in and activates them.

8.3 LI application

This application handles the incoming GSM phone call. When someone calls a Mint payment or activation number the application first checks in the Oracle database whether or not the calling number (A-number) is a number that belongs to a registered Mint consumer. If this is not the case, a voice prompt will be played and an SMS will be sent to the specific

A-number. The voice prompt and SMS contains the same information: Explaining that you first must register as a Mint consumer before you can make any payments with your mobile.

If the calling person is recognized as a Mint consumer and the consumer is not blocked, the application checks whether the phone number the consumer called (B-number) is the activation phone number or a payment phone number and chooses one of the two branches described below (blocking can occur if the Mint service is misused, or if the consumer decides to temporarily block the service).

8.3.1 Payment branch

So, the calling A-number is registered in Mint's database and the called B-number is a payment number. Everything we have to do now is to store GSM phones A-number together with a timestamp and have the B-number as primary key. We store the information in the MDS, to get as fast result as possible. Since each B-number is connected to one and only one specific POS unit we can identify which POS unit the consumer is trying to make a payment to. Everything we have to do now is to play a voice prompt telling the consumer that the purchase has been initialized and that he or she will receive an acknowledgement from the POS unit. As you can see above, all important execution is done when the voice prompt begins to play so the system does not fail if the consumer hangs up before the voice prompt has reached the end.

When the A-number and timestamp is stored and the voice prompt is played everything is done and we can hang up the call.

The whole procedure takes about seven seconds because of the length of the voice prompt we are playing. However, as soon as the voice prompt starts playing, the modem application can pick the payment call from the MDS and continue the protocol. This means that it is possible to get an OK from the POS unit just three to four seconds after the GSM call is received.

8.3.2 Activation branch

When we have reached this far we know that the calling A-number is registered in Mint's database and the called B-number is the activation phone number. The reason for calling the activation phone number is to activate the Mint service for your phone when you have registered to be a user of the system. It is also necessary to activate security sensitive changes you have made in your Mint profile from your MyMintSite or through support. To activate these things the Consumer needs to have an activation code, which he or she gets sent home via ordinary mail.

In this part of the application we look in the database if the consumer who this A-number belongs to, has some changes that can be activated. If this is not the case we play a voice prompt telling him or her that he or she has nothing to activate and hang up.

If the consumer has something to activate he or she will be asked to punch in his or her 13-digit long activation code on the GSM phone. The application reads this punched code using DTMF (Dual Tone Multiple Frequency) and compares it with the activation code in the Oracle database. If the activation code is correct, the application commits the changes associated with the activation code and plays a voice prompt telling the caller that the service or changes now have been activated. The caller gets three tries to enter the right activation code before we hang up and the consumer must call in again or call support to get help doing it the right way.

8.4 Modem application

The modem application is the biggest and most complex application and it handles the communication between the POS units and the platform. All communication is based on the own designed POS protocol (see section 7) and sensitive information as payment amount, PIN code etc. are encrypted using the DESX algorithm (see section 7.3).

The platform has a modempool, which the POS units call in to when a payment is initialized. The POS units will call in to the system when the cashier presses a button and initializes a payment. The POS units can also be scheduled to call in to the system for an update. The update time is stored in the database and sent to the POS unit every time some communication is done. Thus, the POS unit will always have updated update times. Updates are scheduled for each POS when the specific retailer is closed. This means that retailers that are open around the clock cannot use the POS unit for a couple of minutes each night because payment is not possible during a scheduled event.

When a POS unit makes a call we check in the database to choose the right service. If the POS unit is calling in and we do not have any scheduled events (i.e. update) a payment will be performed. If we on the other hand have a scheduled event, this event will have a higher priority than a payment. This has implications on the way that the retailers should operate the POS unit. For the scheduled events to take place at night, the POS unit has to be turned on. If the retailer turns on the POS unit in the morning, and immediately tries to make a payment, the server will see that the unit should do an update. Instead of a payment the consumer and retailer have to wait until the POS unit has got the updates. We prioritize a scheduled event over a payment because a payment without valid information will not be succesful.

8.4.1 Update branch

If an update is scheduled, the application enters the update branch.

This branch starts off by sending message `STP#UPD`, stating that the servicetype is update. After getting the acknowledgement from the POS unit, the application starts gathering information from the database. Ideally, only the things that have changed in the database since the POS unit was last updated, should be transmitted. This is especially true if the POS units get somewhat more advanced, and need updates that are bigger than a couple of hundred bytes. For example, they might get the ability to display colour images. Then it would be important not to send the entire library of images if only one image has changed. In other words, the update should be as economical as possible. This is solved by having a timestamp on every piece of data concerning the POS unit. The timestamps are compared with the POS unit's last update time, and the data is included only if its timestamp is later the the POS unit's.

Once the application has collected all data it starts transmitting them. When this is accomplished, the next update time is changed in the POS unit to mark that no update needs to be performed and the last update time is changed as well.

8.4.2 Payment branch

The payment branch is taken whenever a POS unit without a scheduled event calls in. The first thing that happens in this branch is that the Mint server sends the `STP#PAY`-message, and this message is acknowledged by the POS unit with an `ACK#STP`-message.

At this point, the application checks the database to see if this particular POS unit has a pending transaction. If so, the message `PAY#LTS` is sent. The purpose of this message

is to find out if the POS unit did in fact approve the last transaction (see section 7.2). If it did, the status for the pending transaction is set to "performed", otherwise status is set to "deleted".

Then, the Mint system sends the PAY#PA-message. At this point, the POS unit prompts the retailer to enter the payment amount and the consumer has the option to enter a tip. The total payment amount is sent back to the Mint server using the SND#PA message. Knowing this information, together with the ID of the POS unit, the Mint server starts polling for incoming GSM calls in the MDS. If found, the profile and account information of the consumer that made the payment call are retrieved from the database. The balance and PIN limit are then compared with the payment amount. One of three things can happen:

- The payment amount is bigger than the balance - the payment is aborted.
- The payment amount is smaller than the balance, and smaller than the PIN limit - the payment is performed.
- The payment amount is smaller than the balance, but bigger than the PIN limit - the Mint server request a PIN code from the POS unit. If it matches the one connected to the consumer's account, the payment is performed. If not, the Mint server asks for a second try. If a consumer fails to enter his/her PIN code three times in a row, the account is blocked.

Before a "Payment OK" is sent to the POS unit, the transaction is saved in the database as a pending transaction. When the POS unit acknowledges the OK, the transaction is moved from status "pending" to status "performed".

If no matching GSM call is found within a limited time (typically 30 seconds) the Mint system sends a certain reason code to the POS unit, which then hangs up and displays a failure message on the screen.

8.5 SMS sender program

Since our application had to be able to send SMS:s this was an issue we had to look into. There were already components in the P90 that had this ability, but they only supported the X25 protocol. Our SMS deliverer, however, doesn't support X25, but instead uses HTTP or more specificly XML messages over a TCP/IP connection. The protocol works like this:

- First, you post a LOGIN document, supplying your username and password. The reply to the login document includes a couple of cookies that you need to supply in every following message that you send to the SMS server.
- Secondly, you post a BIND document, that specifies which service you want to use (usually "SEND SMS", but it's also possible to receive SMS:s or check that previously sent SMS:s has been delivered).
- Once this is done you can post SEND documents, as many as you like.
- Finally the LOGOUT document is sent which invalidates the cookies thus preventing further communication with the server.

The implementation of the SMS program was done in C, using the socket implementation.

8.6 Implementation problems

When implementing our applications, we ran into trouble from time to time. In this section, we'll describe the problems we had and what we did to solve them.

8.6.1 P90 related issues

A lot of problems originated in the P90 components that we used, and the fact that we used them in a way they apparently weren't designed for. Also, documentation was not always correct and there were bugs in the components that took us some time to figure out. There was no way for us to verify or correct these bugs, as we were not permitted to read the source code.

The component that is most critical to our application is the LIM. It handles data communication between the Mint server and the POS unit. The original LIM was designed to handle only ASCII data. This was of course a problem since our application uses encryption of messages and the encrypted data can very well contain binary zeros (null), and other control characters that might affect the functionality of the modem. There was also supposed to be a variant of the `Receive Data` transaction that took the additional string parameter `UNTIL`. The return value was the contents of the modem buffer until the value of the UNTIL parameter was found. This seemed perfect for our application, since we surround our messages by start and end tags (see Section 7.4). Unfortunately, it didn't work. We had to make a workaround solution, a function that reads character by character from the buffer, each time checking if the pattern is found.

To be able to use encrypted messages, we needed an encryption component. This component, the SEC (for SECurity), is an implementation of the DESX encryption technique and developed exclusively for Mint. Surprisingly, when it was implemented, it didn't support binary data either. This problem was fixed by Teligent.

Another important component is the RBD which handles communication with an Oracle database. Only one connection can be set up in an RDB. This is a very limiting factor because the server must be able to have several incoming calls simultaneously and every one of them needs access to the database. There is also no way of "sharing" a connection since the RDB works in a strict transaction manner, meaning that an SQL `INSERT` or `UPDATE` is not entered in the database until you explicitly use the "Commit"-transaction. If two applications would share one connection it would mean that there is a possibility of committing each others `INSERT:s` or `UPDATE:s`.

The solution we used was the new RDS component, in almost all cases a direct copy of the RDB. The exception is that the RDS doesn't allow any `INSERT:s` or `UPDATE:s`, only `SELECT` statements. With this functionality, it is safe to share the connection between applications.

While most of our SQL are `SELECT` statements, there are a couple of places in the code where we really need to use `UPDATE` and `INSERT`. In these places, we are forced to use the RDB. To make sure that the applications doesn't have to wait to get an RDB connection, there are four RDB:s in each segment (see section 9) and the applications occupy the connection as short time as possible. The conclusion is that we use the RDB component when we want to change something in the database and immediately disconnect when it is done and when we just want to retrieve information we use the RDS component.

We wanted to obtain a high grade of fault tolerance in our system. This means that the system should handle machines going down or telecommunication links breaking down. In the case where a machine goes down, all incoming calls must be routed to another machine. This is handled by the teleoperator and was verified in the tests (see section 10).

When both segments are up and running, the calls are randomly routed between them. This means that the POS call and the GSM call of a specific payment could end up in different segments. Since the modem application looks for a matching call in the MDS database, it will not find one if the GSM call is registered in the MDS of the other segment. This was solved by setting up the MDS:s to use the XSR to replicate each other between the segments.

8.6.2 Application Builder issues

The other big source of problems was the program Application Builder (AppB). Without complaining too much we try to describe the problems we had in AppB and what we did to solve them, if possible.

The purpose of AppB is to simplify the coding and instead of writing code in the Teligent script language (TSL) use boxes and lines to draw the program flow. This may be a good idea if you don't want to have control over what you are doing and have trust in the program you are working with. The problem with AppB is that we lost the trust for it after a couple of days. It doesn't work as it should and if you have done something wrong and want to correct it, it is sometimes not possible. The list below is a collection of the problems we encountered in AppB.

- String handling - This function is very poor in AppB. It is not possible to edit strings you have once written, but instead you have to create a new string and delete the old one.
- Missing variables - Variables may come and go as they prefer. Sometimes it is impossible to find variables in the variables list even when they are used in the program flow. This makes it impossible to use the same variable further on in the flow and you have to create another variable instead.
- Initializing of variables - In the beginning of each flow you have to initialize each variable to be sure that it doesn't contain any information you don't want.
- Memory capacity - When several flows (typically above 10) are opened in AppB it takes several minutes to switch between them. This get worse and worse the longer you have the program opened and you finally have to reboot the computer.
- Multiple starts - When you start AppB it sometimes (typically 40% of the times) doesn't start and you have to try again. Very time consuming.
- Parameter length - Parameters are a way to send data between functions and they are not allowed to have a name length bigger than 52 characters. This should be more then enough if it wouldn't have been for the naming convention when AppB translates the flows into TSL. AppB sets a prefix on every variable and parameter which can be up to 50 characters long and that gives the user (52-50=) 2 characters to find a good name for the variable.
- Missing of floats - There is no such thing as floats in AppB so when we want to deal with decimals (often used when we are working with currency) we have to multiply everything by 100.
- Modulo - When building with modulo boxes in AppB and the graphical flows are translated into TSL the modulo boxes are translated into plus. you manually have to edit the generated TSL code and change the plus signs (+) into modulo operators (mod).

The list is far from complete and the reason why we encountered so many problems with AppB is probably because we developed applications with functionality they hadn't tested before. AppB has been used to develop common telephony applications such as voice mailboxes, DTMF navigation applications and other small applications. When we used these features we didn't encounter so much problems but when trying to write more complex functions we surely did. The problems with AppB have been very time consuming and frustrating due to the fact that we couldn't fix the problem ourselves but had to rely on Teligent.

We've got 27 fixes for AppB to correct the problems we have reported but since the fixes sometimes made it impossible to even compile the flows without an additional fix we decided to go for the original version and live with the known problems. A fix for fixing a fix is never a good solution.

9 Setup of components and applications

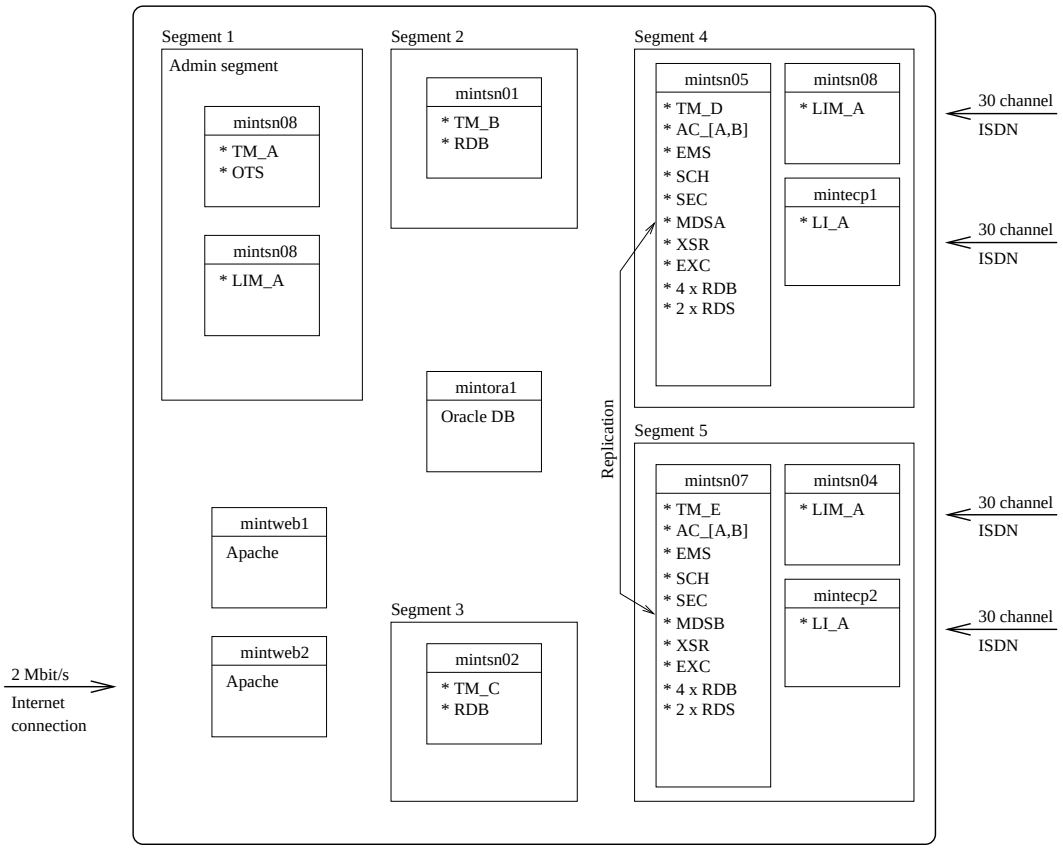

Figure 9: The segments of the system and their components

Figure 9 is an overview sketch of the Mint system. The outer boxes are P90 segments and the inner ones are nodes within the segments. Each segment has to have a TM to control the components that are running in the segment, and the communication between them. There are also computers that don't run any P90 software, thus not belonging to any P90 segment. These computers are mintweb1, mintweb2 and mintora1.

As you can see, Segment3 is a copy of Segment2 and Segment5 is a copy of Segment4. This setup is used to obtain fault tolerance in the system. The routing capabilities included in the TM:s can handle the case where one segment crashes and try to send the transaction to another segment. This ability makes the system very robust and fail-safe. Additionally, it allows you to take one segment down, for upgrades or bug fixes, still keeping the system running.

Another interesting feature is the replication of the MDS database. The setup is called "Mated pair", and states that the two databases must have the same view at all times, i.e. the same data exists in both databases. If this wasn't the case, it would mean that the GSM phone call and the POS call would have to be directed to the same segment, which basically erases all fault tolerance that we have disussed above.

10 Tests

We conducted both functionality tests and load tests on the system. In this section we will try to explain how these tests were made and the conclusions we have made based upon them.

10.1 Load tests

The goal for these tests were to make sure that the system could handle the load that were estimated for the near future. To make the load tests we had to hire an external company specialized at testing telephonysystems, namely Telia Validation. They can generate phonecalls in the net with a very high speed and verify if they get an answer or not.

Telia Validation programmed their computer that generates the GSM calls to generate an amount of calls per second and verified if they got any answer. When the call is answered all computation is done in the platform already, so an answer indicates that everything is working just fine. A channel is occupied for approximately seven seconds, due to the length of the voice prompts we are playing.

Call rate	Percentage answered	Time	Other
1/sec	100%	60sec	
1/sec	100%	60sec	Functionality test
1/sec	100%	60sec	Functionality test
5/sec	100%	60sec	
10/sec	75%	60sec	
7/sec	100%	60sec	

Table 1: Test results

Since we have 60 channels connected to the system and each call takes seven seconds the table above is pretty self explaining. If we have 10 calls per second we will run out of channels after six seconds and we will lose calls for one second (i.e. miss 10 calls), before the first second's channels will be free again.

10.2 Functionality tests

The functionality tests that were made during the tests were made to verify that the platform could handle breakdowns and broken telephony lines.

In the first functionality test one of the two telephony cables was disconnected from the system during the heavy traffic and all calls were automatically redirected to the other cable. The first cable was then connected again and the second one was disconnected. Every generated call got an answer so the system can handle failures in the telephony net.

In the second functionality test the two computers handling the incoming calls were shut down, one at a time, and recovered. While one of the computers was down the other computer handled all traffic and it was balanced again when both were up and running.

10.3 POS test

We had 28 POS units calling to one of the segments simultaneuosly as Telia validation generated GSM calls. In a perfect world the POS units should have found a matching GSM call for every one of the faked purchases, but because all POS unit were using GSM modems and were all calling from the same operator we overloaded the GSM cell in that area and far from all POS calls made it through.

The problem we got was to make enough POS calls at the same time so we could also verify that the POS channel could handle enough load. We have the same bandwith for both channels (LI and LIM), which is 60 simultaneously channels (i.e. phone calls), so from the telephone net's point of view both channels can handle the same amount of traffic. Due to the fact that we have different components answering the different channels and we were not able to load the POS channel with enough traffic due to the limitations in one GSM cell we are not sure if that channel can handle the same amount of traffic.

10.4 Conclusions

From the load and functionality tests we can make some conclusions. The system can handle at least 5 (five) simultaneous calls/second or 300 calls/minute, which is more than enough for the next two years. Due to the fact that we haven't been able to load test the LIM component we are not really sure how many purchases can be done simultaneously but from the telephony line's point of view it is 300 purchases per minute. During the tests we could only see that all POS calls that made it through also made successful payments.

If Mint however, in the future, would like to handle more purchases we have some ideas of how that could be done easily (see section 11).

11 Future work

Mint will in the future probably find many improvements that can be done to the system and get several ideas from their Consumers, which are very hard to foresee now but they already have some ideas of new services for the future. As we have written throughout this report we also have some ideas about how the system/service can be improved to handle more simultaneous calls, shorten purchase time etc. In this section we will try to summarize the ideas that we have in this area.

11.1 New services

In the beginning of this project we planned to implement a lot more services, but the lack of time forced us to reconsider and postpone the implementation to future versions. Some of the services that should be implemented in the future are listed below.

- Consumer2Consumer payments - In the future it should be possible for a Mint Consumer to transfer money from his or her Mint account to another Mint consumer's Mint account. This should be possible via different kind of media such as SMS and an ordinary phone call, navigating with DTMF.
- Loyalty services - For the next version of Mint payment system some loyalty services should be implemented. This means that you, as a registered Mint Consumer, don't have to bother about all bonuscards and coupons. Mint will take care of that for you.

11.2 Hardware improvements

Some improvements or upgrades that can be done to the hardware of the system are listed below.

- Connect more incoming channels - The system still has two unconnected lines (à 30 channels), for each segment. So the capacity can easily be doubled.
- Add an extra segment - More segments gives Mint more computers, which in turn gives Mint more incoming channels and a more fault tolerant system.
- Set up backup for web site - As it is today there is no backup machine.

11.3 Software improvements

In the software (our applications) there are some improvements that can be made.

- Shorten the calls - Use shorter voice prompts and by that shorten the calls by a few seconds. This will lead to shorter occupation of the channels and more calls can be answered within the same time period. It is also possible to add an extra check and check if the calling party have hung up or not, during the voice prompt. If the calling party has hung up the Mint server can also hang up and gain a couple of seconds.
- Read more then one byte at a time from the modempool - As it is today we read one byte at a time from the modem pool's input buffer and that generates a lot of transactions (one for each byte) in the LIM component. We haven't encountered any complications because of this but it however can be improved. Why we did like this in the first place was because of the poor string handling in Application Builder.

- Load balancing of components - Today we don't use all load balancing features in the P90 platform due to the amount of time it takes to change it, it was not done to version one. For example we only use one of the two RDS components at each segment and the RDBs are not used equally either.

References

[1] Neffenger, J. *Volano report.* <http://www.volano.com/report.html>. 2001-02-02.

[2] Quatrani, T. March 2000. *Visual Modeling with Rational Rose 2000 and UML.* Chapter 1-4.

[3] Teligent. 1999. *The P90 reference manual.*

[4] Scheier, B. 1996. *Applied Cryptography - Protocols, Algorithms and Source Code in C.* Chapter 11-12, 15.

[5] Killian, J. & Rogaway, P. 1997. *How to Protect DES Against Exchaustive Key Search.*