Computer Science

Stefan G M Sonesson

BISNet

Bachelor's Project

2000:24

BISNet

Stefan G M Sonesson

© 2000 The author(s) and Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Stefan G M Sonesson

Approved, Date of defense

Advisor: Dimitri M Ossipov

Examiner: Stefan Lindskog

Acknowledgements

I am indebt to my family for having taken up a lot of familytime. My wife Eva has been a great support at late hours in front of the catodetube.
Thank You Eva, Jennifer and Samuel.

My friend Antoine Haddad has meant much for me too. He is an excellent programmer and has object orientation written on the back of his palm. We have had quite a few discussions on the issue.
Thank You Tony.

Without the help of my advisor Dimitri M. Ossipov I would have had a hard time. His knowledge is an amazing resource which I greatly appreciate taking part of.
Thank You Dimitri.

Abstract

This document describes the BISNet system.

Chapter 1

Gives the reader a brief introduction to the project.

Chapter 2

Describes analysis of classes and their responsibilities. It also shows how the system works from the users point of view.

Chapter 3

Explains the design of classes, scenarios from systems point of view, structure of database tables, relationships aswell as queries.

Chapter 4

Reviews architecture, middleware and JDBC/ODBC-bridge.

Chapter 5

Brings up maintenance and system evolution.

Chapter 6

States a few assumptions.

Chapter 7

Conclusion, estimation of response time.

References

Books I have read and got material from.

Appendix

Code.

Contents

List of Figures

List of Tables

1 Introduction

This document describes my Bachelor's project BIS^{Net} that is a system for optimization of transports. The report is written in LaTeX.

Acke Kanderman at Bygginvest Scandinavia AB, had a proposition for a bachelor's project that aims at increasing the profit for trailer contractor T (supplier) and machine contractor M (customer).

Today return-transports are often empty due to the lack of information between T and M. Our intention is to solve the problem by using a database to manage orders and offers. The parties will probably access the database with mobile phones sending e-mail or SMS[1]. The parties T and M have expressed a need to get in contact with each other in a fast and efficient way. As of today the cooperation is said to occur through contacts or by phone.

The purpose of the service is to simplify the handling of the negotiation between M and two or more T. One can think of an auction as a metaphor. Our service would be the middleman.

The database service created will presumably lead to:

* an increase in T's reservations
* a decrease of M's transportation costs

Basically, the service will consist of an application that handles a database and the messaging between database and users.
C and S communicates with the service through use of e-mail.
C can place orders where he specifies that he wants a machine transported from one loca-

[1]Short Message Service, a way for mobile phones to send text messages

tion to another and that this can be done earliest from a specific date to another.
S can place offers where he states that he can transport machines between 2 specific dates.
Both C and S can respond to messages sent to them by the service. In chapter 4.1 the message protocol[2] is described.

[2]How the messages should look to be accepted by the system

2 Analysis

First we need to define what classes we should have and then scenarios from users point of view.

2.1 Classes, responsibilities

From the problem domain, we can see that there is a need of extracting data from messages, inserting to, and retrieving data from, a database aswell as sending messages to the users of the service. The users should register with the service through an application (paper form) to the DBA[3]. This is probably a good way to insure that there are no fake applications, since they would have to legitimate themselves on site when applying.

There should be classes responsible for:

- Finding new messages.
- Parsing named messages.
- Inserting and retrieving data from database.
- Take care of users interaction with the service.
- Controlling the negotiation.
- Sending messages.

[3]Database Administrator

2.2 Scenarios

Customer and supplier can make a deal if C has placed an order, S has placed an offer and a match between the two have occurred. Then the negotiation starts where the service first asks S if he can take the order that has matched his offer. Upon S agreeing to this, we ask C if he wants this offer, or several offers if we have any. If C agrees on one offer we send C and S a deal, which when they agree to becomes a closed deal.

After the deal is closed C and S will have to use some kind of direct communication with each other to cancel or change the deal. It could happen that they make this communication earlier in the process, leaving our negotiation to time out.
In the future, customer may view the status of his ongoing negotiation through some online service, where we could have, e.g. an applet that fetches data from our service and shows this to the user. C and S is not going to be able to delete their registration other than through the DBA because of the security problems it could inflict. For this service to work there exists several different scenarios how the system should work from a users point of view. These are described in brief in table 2.1.

Scenario	*Description*
S01	Customer places order.
S02	Supplier places offer.
S03	Supplier accepts negotiation offer.
S04	Customer accepts negotiation offer.
S05	Supplier accepts deal.
S06	Customer accepts deal.
S07	Supplier rejects deal.
S08	Customer rejects deal.
S09	Supplier removes previously placed offer.
S10	Customer removes previously placed order.
S11	User do not reply in time.

Table 2.1: Scenarios from users view

S01.

A user (from now on called C) wants to have his machine transported somewhere.

C sends a message, with information about his order, to the service and receives a message that his order has been placed.

S02

A user (from now on called S) wants to transport a machine.

S sends a message, with information about his offer, to the service and receives a message with acknowledge that his offer has been placed.

S03.

S receives message with a negotiation offer.

S sends a message back to the service telling it that he accepts this offer.

S04.

C receives a message with a negotiation offer.

C sends a message back to the service telling it that he accepts this offer.

S05.

S receives message about a deal .

S sends a message back to the service telling it that he accepts this deal.

S receives message that deal is made.

S06.

C receives a message about a deal.

C sends a message back to the service telling it that he accepts this deal.

C receives message that deal is made.

S07.

S receives message about a deal.

S sends a message back to the service telling it that he rejects this deal.

S receives message that deal has been rejected.

S08.

C receives a message about a deal.

C sends a message back to the service telling it that he rejects this deal.

C receives message that deal has been rejected.

S09.

S sends a message to the service saying that he removes earlier placed offer.

S receives message that the offer has been removed.

S10.

C sends a message to the service saying that he removes earlier placed order.

C receives message that the offer has been removed.

S11.

User fails/neglects to reply in time (specified by system).

C receives message that he has timed out.

3 Design

3.1 Classes

From Analysis we derived the classes, BISNet, MsgFinder, MsgParser, DBManager, ContractWatcher, DBHandler and MsgSender. BISNet is the main application and is responsible to start the service. MsgFinder checks a given directory to see if any new messages has arrived. MsgFinder keeps all new messages in a file. MsgParser does what it says, it parses the message/s that it takes as argument. DBManager takes care of the placement of orders and offers. It calls ContractWatcher on the negotiation scenarios 4.1. ContractWatcher maintains the ongoing negotiation. This is where most of the business logic is. DBHandler does the hard work for DBManager and ContractWatcher, i.e. inserting and retrieving, to and from the database. MsgSender is responsible for sending messages to C and S. I use Booch[1] notation for diagram 3.1.

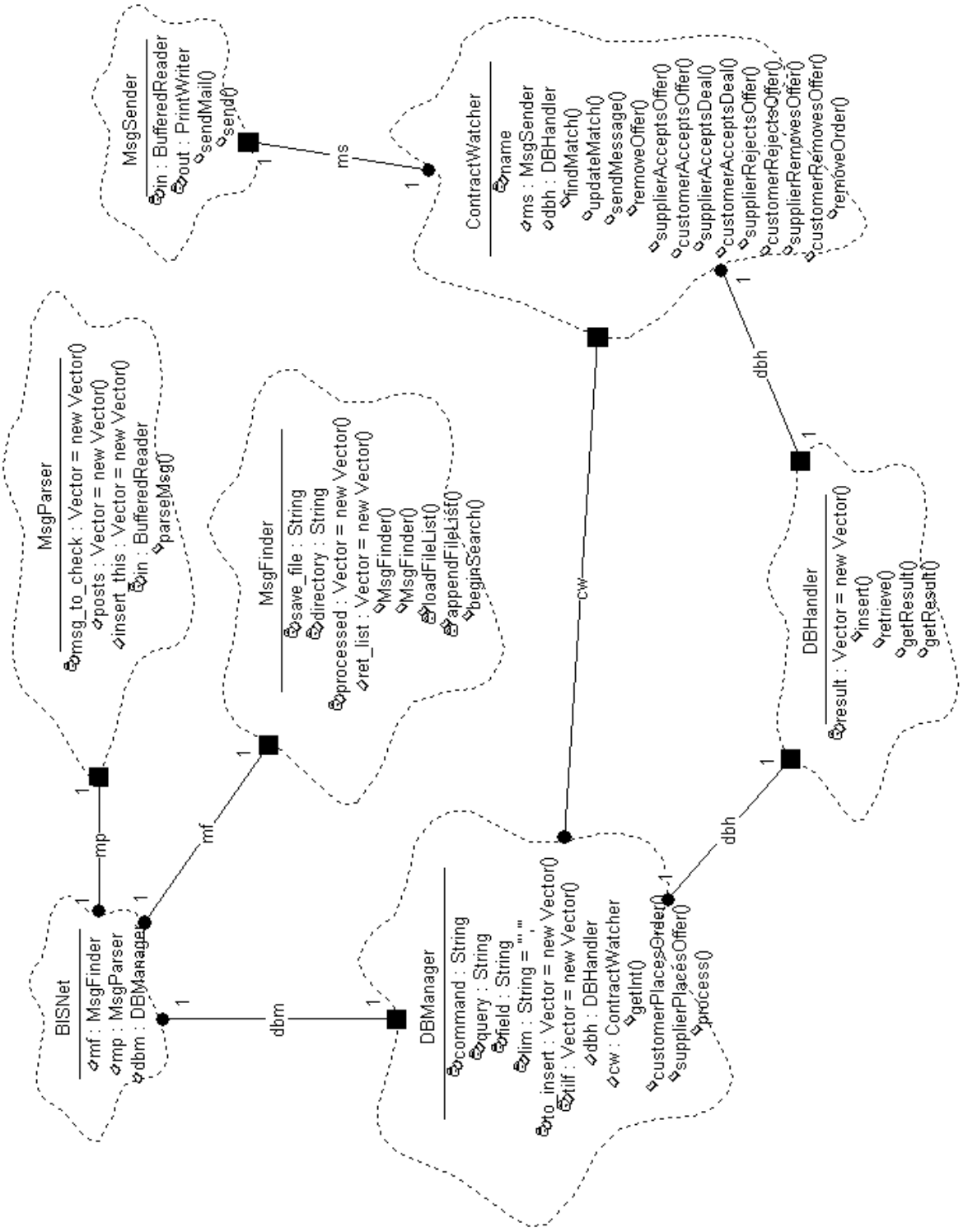

Figure 3.1: Class diagram from Rose model

From the diagram 3.1 we can see the relationships between classes. BISNet has three objects by value, these are mf, a MsgFinder object responsible to find messages. There are mp, a MsgParser object that parses data from a message and dbm that is a DBManager object responsible to place new orders and offers. DBManager has two objects by value, dbh that is used to insert newly placed orders/offers. There is also cw a ContractWatcher object that handles the negotiation process. ContractWatcher also have two objects by value, dbh used to insert and retrieve data from the database, and ms a MsgSender object that sends messages necessary to uphold the negotiation.

User sends messages to the service through any kind of media, but at current time it is done by email. E-mail server (see 4.5) retrieves and puts message as files in appropriate directories. BISNet (BIS) starts the whole service by calling MsgFinder (MF). MF searches directories for new messages that do not exist in the processed.dat file, i.e. they are new. Upon finding a message file it returns a list that contains new messages to BIS. BIS then calls MsgParser (MP) with a Vector containing the messages to parse. MP starts parsing the files for data, which it puts in String arrays. These are in turn put in a Vector, to be returned to BIS, which then calls the DBManager (DBM) that is responsible for inserting the data to the Database, depending on what type of message it was. If it was COR or SOF (see 4.1) DBM inserts the new order/offer and replies to the user. If it was any of the other message types (see 4.1) DBM calls ContractWatcher (CW) that takes the appropriate action depending on the message received. CW handles most of the systems business logic. CW queries DB to find matches, i.e. suppliers and customers who can make a deal. These matches are inserted to a Match-table where status about the ongoing negotiations are held. CW and

3.1.1 BISnet

This is the main class of the project. It is responsible to start the service and keep it running.

Constructor detail

No constructors.

Method detail:

`public static void main(java.lang.String[] args) throws java.lang.Exception`

BISNet creates five objects mf, mp, dbm, cw and ms. A while-loop runs forever to check for incoming messages and deal with them accordingly. Call is made to the `beginSearch()` method of MsgFinder class. If it returns a value greater than zero, it means that there has arrived new messages. In that case we assign `ret_list` of MsgFinder to the Vector tmp. Now we call the `parseMsg()` method of MsgParser class. If it returns a value greater than zero, the value is held in the parameter `posts` as the number of posts to process. The Vector tmp is cleared and `posts` of MsgParser is assigned to tmp.

Parameters: `args`, takes the mailbox directory as argument. In case we do not have the mailbox directory at a fix position, or simply want to place it elsewhere

Returns: void.

Throws: -

Continue with BISNet after finished with class...

3.1.2 MsgFinder

`public class MsgFinder extends java.lang.Object`

Searches given directory for new messages.

Field detail

`private java.lang.String directory`

Holds the name of the mailbox directory used by EMWAC4.5

`private java.util.Vector processed`

A Vector containing names of already processed messages.

`public java.util.Vector ret_list`

Holds the names, as Strings, of the new files that MsgFinder found.

`private java.lang.String save_file`

Holds the name of the file we are using to have persistance in case of a crash.

Constructor detail

There are two constructors.

`public MsgFinder(java.lang.String file_name)`

Constructs a MsgFinder object.

Assigns the `file_name` to the variable `save_file` and loads the file list.

Parameters: `file_name` - keeps the names of the processed messages.

`public MsgFinder(java.lang.String arg, java.lang.String file_name)`

Constructs a MsgFinder object.

Takes the `arg` as directory where to start looking for new files. That directory should not contain any files only folders. Assigns the `file_name` to the variable `save_file` and loads the file list.

Parameters: `arg` - The directory to be searched.

`file_name` - keeps the names of the processed messages

Method detail

`private void loadFileList()`

Has a while loop where lines are read from a `save_file` *processed.dat*.

These lines are filenames that we add to a Vector `processed`.

Returns: void.

`private void appendFileList()`

Cycles through a for loop, where it gets elements from `ret_list` and assigns them to `str`. `str` plus a newline, to get each file on separate lines, is appended to the `save_file` *processed.dat*.

Returns: void.

`public int beginSearch()`

This method searches directories for newly arrived messages. It starts of by checking that `directory` exists and that it can be read from. Then it checks if `directory` is a directory and not a file in the given directory, because we have said earlier that we do not want any files in the mailbox directory. Now we assign an array of strings naming the files and directories in the directory denoted by the abstract pathname to `dir_list`. We loop the length of `dir_list` to put the filenames with their paths in a `File` array `file_list`. This is done to retrieve the whole pathname of the messages encountered as new files. After that we loop length of `file_list` to check if the item at every position is a directory i.e. users mailbox. If it is so we list the contents of that directory in to a `String` array `dir_list2`. This secondary directory list is looped through to see if the Vector `processed` does **not** contain the item at this position, then it is a new

message which we add to the Vector `ret_list`. When this is done we check if `ret_list` is greater than 1, a new message has arrived, in that case we call the appendFileList() method.

Returns: Size of `ret_list`

3.1.3 MsgParser

This class parses messages for information.

Field detail

`public java.util.Vector posts`

Holds Vectors of tokens.

`public java.util.Vector insert_this`

Holds tokens that we want to insert to DB.

`private java.util.Vector msg_to_check`

Holds the filenames of the messages to parse.

`private java.io.BufferedReader in`

Read text from a character input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Constructor detail

MsgParser

`public MsgParser()`

Default constructor

Method detail

`public int parseMsg(java.util.Vector tmp)`

Assigns taken Vector `tmp` to `msg_to_check`. Loops through `msg_to_check`. Clears `temp` and `msg_to_check`. Takes every element (filename, message) from `msg_to_check` and assigns them as Strings to `msg`. Assigns a new BufferedReader with a new FileReader with the argument msg. Now we read lines from the given message and adds them to `temp`. After this we make a new `StringTokenizer` with the arguments the element at position 5 (meaning line 6 in the message) and our special delimiter character # and assigns this to `st`. Loops while st has more tokens, assigning each element to `value`. Each `value` is added as a new String element to `insert_this` and every `insert_this` is added to `posts` as a new Vector element. The size of `posts` is returned.

3.1.4 DBHandler

Field detail:

`public java.util.Vector result`

Holds the result from a query to the DB.

Constructor detail

`public DBHandler()`

Default constructor

Method detail:

`public java.lang.String getResult(int pos)`

Returns an element from a given position in `result`.

`public java.util.Vector getResult()`

Returns a new Vector with `result` as an argument.

`public int insert(java.util.Vector command)`

Inserts data from a Vector to the DB, this is done as a batchjob.

`public int retrieve(java.lang.String command)`

Queries the DB for information given as a SQL query in the form of a String.

Parameters: command - `String` with the SQL command to run.

Returns: The size of result.

3.1.5 ContractWatcher

Field detail:

`private java.lang.String cstate`

Holds the customer state from the match table.

`private java.lang.String sstate`

Holds the supplier state from the match table.

`private java.lang.String command`

Holds SQL commands we want to execute towards the DB.

`private java.lang.String query`

Holds the queries we want to ask the DB.

`public java.util.Vector result`

Holds the results given from DB when asking a query.

`static java.lang.String ACK`

A string with acknowledgement to the user.

`static java.lang.String NACK`

A string with negative acknowledgement to the user.

`static java.lang.String SYSMAIL`

A string with the email address to the system.

`static DBHandler dbh`

A DBHandler object.

`static MsgSender ms`

A MsgSender object.

Constructor detail

`public ContractWatcher()`

Default constructor.

Method detail:

`public void findMatch()`

Finds matches[4] and inserts them to the match table.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

`public void supplierAcceptsOffer(java.util.Vector tilf)`

Retrieves the e-mail address of the user sending the current message. If we fail getting email address, we send a message to ourselves with information of the command that we could not execute. Once we have the address we try to insert information to database, if ok, we send an acknowledgement to the supplier.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

`public void customerAcceptsOffer(java.util.Vector tilf)`

Retrieves e-mail addresses of the customer sending the current message and the supplier who corresponds to the same offer. If we fail getting email address, we send a message

[4]Orders, offers that share some predefined criteria.

to ourselves with information of the command that we could not execute. Once we have the addresses we update cstate to *cresp* in database, if ok, we send a message about a deal to customer and supplier.

Parameters: tilf - `Vector` with the information to process.
Returns: void.

`public void supplierAcceptsDeal(java.util.Vector tilf)`

Updates sstate to *SAD*, checks cstate for *CAD*, if cstate is CAD update historylog, remove negotiation from database, send message to supplier and customer, if not CAD send timeout message to ourselves.

Parameters: tilf - `Vector` with the information to process.
Returns: void.

`public boolean customerAcceptsDeal(java.util.Vector tilf)`

Updates cstate to *CAD*, checks sstate for *SAD*, if sstate is SAD update historylog, remove negotiation from database, send message about deal to customer and supplier, if not CAD send timeout message to ourselves.

Parameters: tilf - `Vector` with the information to process.
Returns: void.

`public boolean supplierRejectsOffer(java.util.Vector tilf)`

Updates log, removes record from match, get e-mail address to supplier, sends acknowledgement message to supplier.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

`public boolean customerRejectsOffer(java.util.Vector tilf)`

Updates log, removes record from match, get e-mail addresses to customer and supplier, send message that negotiation is aborted to supplier and customer.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

`public boolean supplierRemovesOffer(java.util.Vector tilf)`

Updates log, checks if ofid exists in match table, gets e-mail address to supplier and customer, removes record from Offers, send message to supplier and customer that this negotiation is off.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

`public boolean customerRemovesOrder(java.util.Vector tilf)`

Updates log, checks if oid exists in match table, gets e-mail address to customer and supplier, removes record from Orders, send message to customer and supplier that this negotiation is off.

Parameters: tilf - `Vector` with the information to process.

Returns: void.

3.1.6 MsgSender

The methods of this class is taken from [4] p165.

Field detail:

`private java.io.BufferedReader in`

Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

`private java.io.PrintWriter out`

Print formatted representations of objects to a text-output stream.

Constructor detail

`public MsgSender()`

Default constructor.

Method detail:

`public void sendMail(java.util.Vector message)`

Opens a socket to our mailserver smtpds on port 25. Calls `send` with every element of the `message` Vector.

Parameters: message - `Vector` with the information to send.

Returns: void.

`public void send(java.lang.String s)`

Parameters: s - `String` with the information to send to print stream.

Returns: void.

Throws: java.io.IOException

3.2 Scenarios

From the systems point of view, all actions performed by the user in the scenarios from Table 2.1, will act as events for us to do some action upon noticing. These can be extracted as a state-event matrix. Abbreviations in *Event description* column are explained in Table 4.1. Table 3.1 shows what type of event the scenarios from table 2.1 entail. Table 3.2 presents the actions and their descriptions.

Scenario	*Event*	*Event description*
S01	E01	COR received
S02	E02	SOF received
S03	E03	SAC received
S04	E04	CAC received
S05	E05	SAD received
S06	E06	CAD received
S07	E07	SRJ received
S08	E08	CRJ received
S09	E09	SRM received
S10	E10	CRM received
S11	E11	TUT received

Table 3.1: Events

Action	*Description*
A01	Insert order to DB, send ack to customer, find match, send negotiation offer to supplier.
A02	Insert offer to DB, send ack to supplier, find match, send negotiation offer to supplier.
A03	Update sstate in match table to sresp, send negotiation offer to customer, send message with timeout date to ourselves.
A04	Update cstate in match table to cresp, send deal to customer and supplier, send message with timeout date to ourselves.
A05	Update sstate in match table to SAD, check cstate for CAD, send timeout date to ourselves.
A06	Update cstate in match table to CAD, check sstate for SAD, send timeout date to ourselves.
A07	Update historylog table, remove negotiation from DB, send message to customer and supplier.
A08	Update historylog table, remove match in DB, send message to supplier.
A09	Update historylog table, remove match in DB, send message to customer and supplier that negotiation is off.
A10	Update historylog table, remove match in DB, send message to customer and supplier.

Table 3.2: Actions

Figure 3.2 presents state-transition diagram to show negotiation that is performed by system service. Negotiation starts after event E01 see table 3.1 has occurred and A1 see table 3.2 is performed, or after E02 followed by A02. This takes us to state SRESP see 3.2, where we are waiting for E03 that triggers A03 and takes us to state CRESP. We can also get E07 or E11 both followed by A08, in that case we are taken to state END and the negotiation is over. From state CRESP, E08 or E11, can lead to A09 being performed and take us to state END which is the end of negotiation. E04 gives that we should perform A04 and move to state WDEAL. WDEAL is the state where we wait for customer and Supplier to accept a proposed deal. In WDEAL state, E05 followed by A05 takes us back to same state. E06 makes A06 happen and also takes us back to same state as we were

in. If we have E11 trigger A10 we go to state END. By finding that both customer and supplier have accepted deal, SAD&CAD, A07 leads to state DEAL. In state DEAL we send customer and supplier confirmation telling that the deal is made and after that we will update historylog see table 3.8 and remove record from match aswell as appropriate records in, order table, see 3.5, offer table, see 3.6, and the negotiation is over.

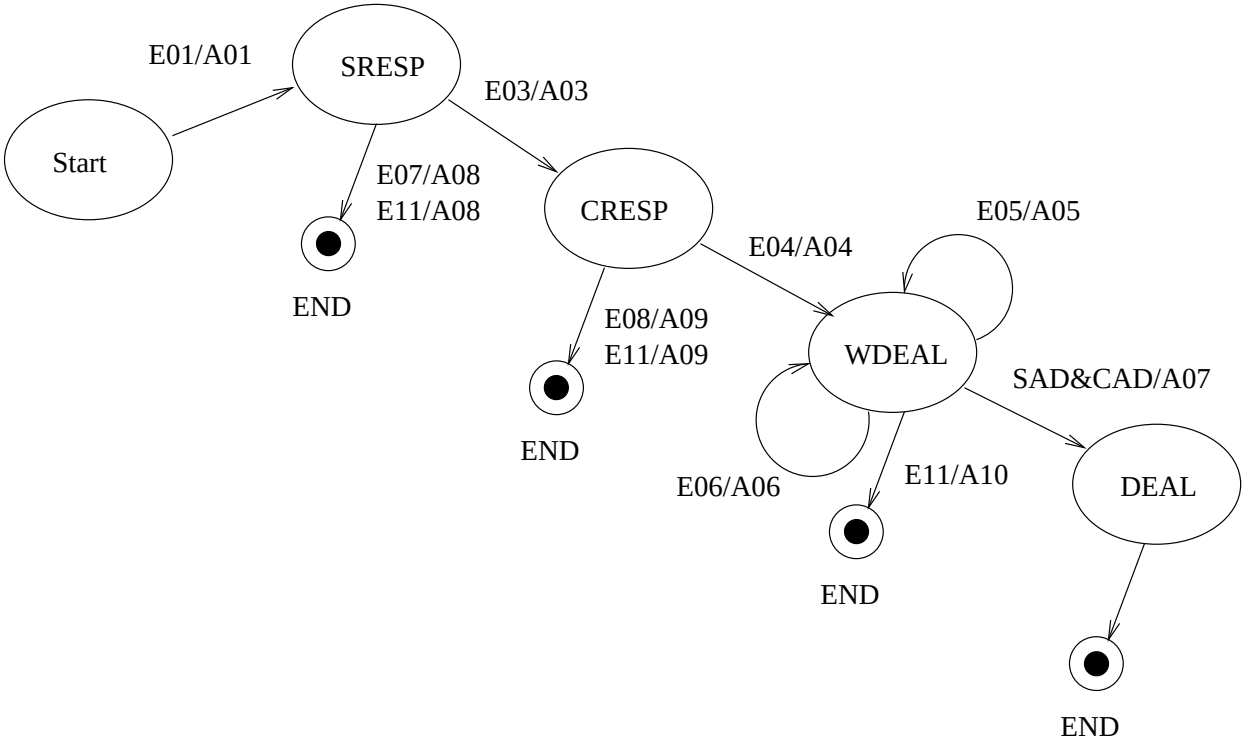

Figure 3.2: State-transition diagram showing negotiation.

3.3 Database

The database needs to hold information about customers, suppliers, orders, offers, matches, machines, history of negotiation and possibly more when extended in the future.
All data is persistent since it is stored by the database. Due to the fact that no authentication other than sending an ID exists, all operations on the Customer and Supplier tables should be done by an administrator. This is to prevent users from deleting the wrong record.

The relational database has proven itself to be very popular. We choose to use it since it is well known and there exists many products and standards that support it.

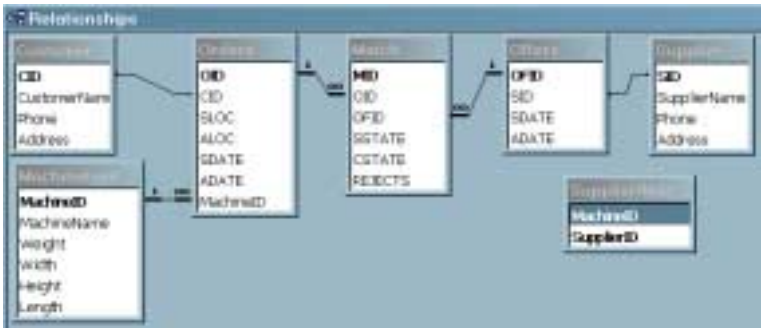

Figure 3.3: Relations between tables from MS Access

In fig 3.3 we see relations between tables in database. CID from Customer is a foreign key in Orders. MachineID is also a foreign key in Orders and has a one-to-many relationship, because machine can be involved into many orders, with it. OID from Orders is a foreign key in Match and has a one-to-many relation, because one order can have many matches. SID from Supplier is a foreign key in Offers. OFID is a foreign key in Match and has a one-to-many relation, because one offer can have match many offers. The primary key in each table is a counter.

3.3.1 Tables

Following presents the column properties of the database tables.

Customer

Primary key: CID

Name	*Type*	*Size*	*Description*
CID	Long	Increment	Number identifying the customer
CustomerName	Text	50	Name of the customer
Phone	Text	50	Phonenumber
Address	Text	50	Address

Table 3.3: Customer

The customer table holds information about the user named customer.

Supplier

Primary key: SID

Name	*Type*	*Size*	*Description*
SID	Long	Increment	Number identifying the Supplier
SupplierName	Text	50	Name of the supplier
Phone	Text	50	Phonenumber
Address	Text	50	Address

Table 3.4: Supplier

The supplier table holds information about the user named supplier.

Orders

Primary key: OID

Foreign key : CID

Name	*Type*	*Size*	*Description*
OID	Long	Increment	Number identifying the order
CID	Long	-	Number identifying the customer
SLOC	Text	50	Source location of machine
ALOC	Text	50	Target location of machine
SDATE	Short Date	-	Startdate of transportation
ADATE	Short Date	-	Deadline for transport
MachineID	Long	-	Number identifying the machine

Table 3.5: Orders

The order table holds information about an order that the customer has placed.

Offers

Primary key: OFID

Foreign key : SID

Name	*Type*	*Size*	*Description*
OFID	Long	Increment	Number identifying the offer
SID	Long	-	Number identifying the supplier
SDATE	Short Date	-	Available from this date
ADATE	Short Date	-	Available to this date

Table 3.6: Offers

The offer table holds information about the offer a supplier has placed.

Match

Primary key: MID

Foreign key : OID, OFID

Name	*Type*	*Size*	*Description*
MID	Long	Increment	Number identifying the match
OID	Long	-	Number identifying the order
OFID	Long	-	Number identifying the offer
SSTATE	Text	5	Supplier state of match
CSTATE	Text	5	Customer state of match
REJECTED	Int	-	Yes or No

Table 3.7: Match

The match table holds information about matches. Matches are orders and offers we think the customer and supplier wants to negotiate, because they fit in time, type or other criteria.

HistoryLog

Primary key: MID

Foreign key : OID, OFID

Name	*Type*	*Size*	*Description*
HID	Long	Increment	Number identifying the match
MID	Long	-	Number identifying the order
OFID	Long	-	Number identifying the offer
SSTATE	Text	5	Supplier state of match
CSTATE	Text	5	Customer state of match
REJECTED	Int	-	Yes or No

Table 3.8: HistoryLog

The historylog keeps track of every transaction made to a negotiation.

Machines

Primary key: MachineID

Name	*Type*	*Size*	*Description*
MachineID	Long	-	Number identifying the machine
MachineName	Text	50	Name of the machine
Weight	Text	6	Weight of the machine (kg)
Width	Text	3	Width of the machine
Height	Text	3	Height of the machine (m)
Length	Text	3	Length of the machine (m)

Table 3.9: Machines

3.3.2 Queries

Q01 Used when customer places an order. Inserts an order into Orders table.

Q02 Used when supplier places an offer. Inserts an offer into Offers table.

Q03 Used when supplier accepts negotiation offer. Retrieves e-mail address of supplier.

Q04 Used when supplier accepts a negotiation offer. Updates *SSTATE* in *MATCH* table to *CRESP*.

Q05 Used when customer accepts a negotiation offer. Retrieves e-mail for supplier with same *MID* as customer.

Q06 Used when customer accepts a negotiation offer. Retrieves e-mail address of customer.

Q07 Updates *CSTATE* in *MATCH* table to *CRESP*

Q08 eee

Q09 fff

Q10 ggg

Q11 ggg

Q12 ggg

Q13 ggg

Q14 ggg

Q15 ggg

Q16 ggg

4 Architecture and Middleware

4.1 Overview

As the world is moving away from client/server toward a "three tier model" (see [4] p208), or even "*n* tier models", it seems logical to build our system as a "three tier system". The system could easily adopt to a n-tier.

With the three tier model there is the advantage of separating *visual presentation* (on the client) from the *business logic* (in the middle tier) and the raw data (in the database). This makes it possible to access the same data and the same business logic from multiple clients, such as a Java application (in our case), or applet or web form.

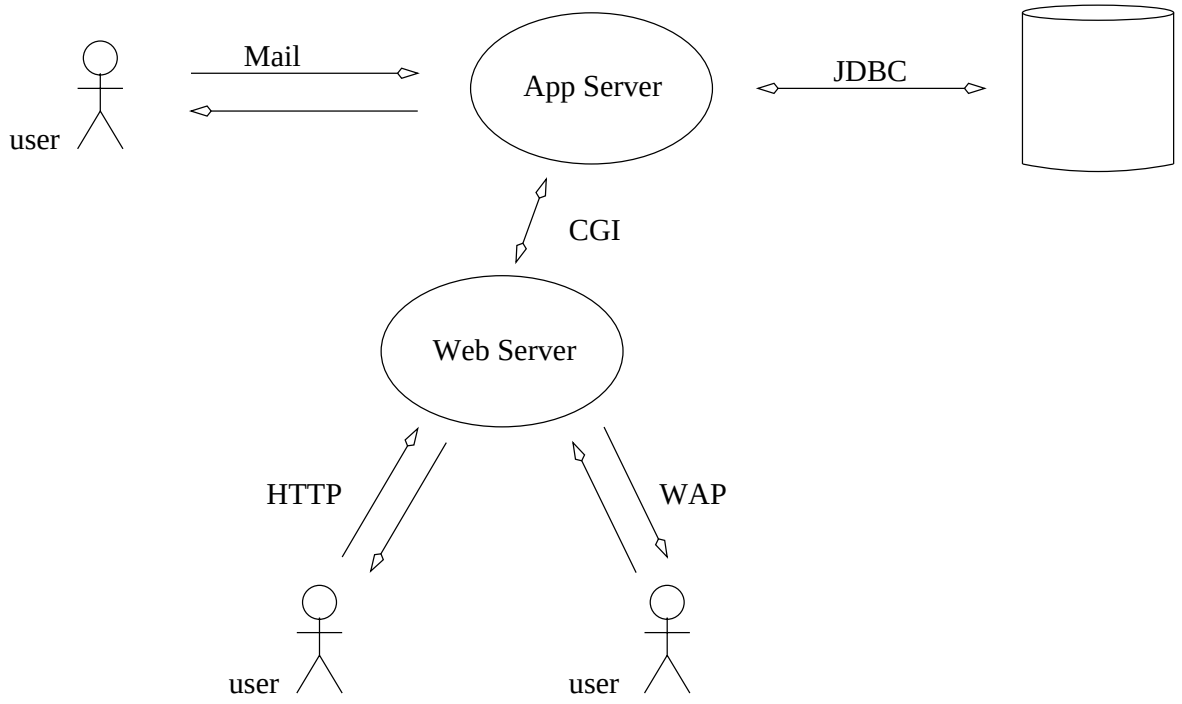

Figure 4.1: System overview

Figure 4.1 shows system architecture and middleware between tiers.E-mail, WAP or

HTTP can be used to exchange messages between actors and system. Messages are stored in a persistent queue, which is used by application server to retrieve messages and send appropriate responses. The system is using e-mail service as Message-Oriented Middleware (MOM) that supports the Queued-Message-Processing (QMP) paradigm.

Basic idea of QMP.
The client message is stored in a queue and the server works on it when free. The server stores the response in another queue, and the client actively retrieves the responses from this queue. This model, used in many transaction-processing systems, allows the clients to asynchronously send requests to the server. Once a request is queued, the request is processed, even if the sender is disconnected.

Analysis of QMP:
Asynchronous ("nonblocked") paradigm. The clients can put a message on the queue and then continue processing. Another segment of client code can continue to monitor the output queue for any responses and process the outputs received. Asynchronous processing has the following advantages:

- It is suitable when you do not want to have a constant session with the host.
- Failure of application server do not affect the clients to stop processing.
- Clients can be more efficient, not having to wait for server to respond. This is good for SMS.
- Clients and servers can use "deferred-message" or callback features if a response is not available.

Queue considerations.
The message queues can be persistent (stored on disk) or nonpersistent (stored in memory). We store them on disk for failure recovery. The queues can be at client machines,

at server machines, in a middle machine, or replicated on client and server machines. If queries generate large tables with millions of rows as responses to be stored in queues, then very large queue sizes need to be allocated. Queue overflows can be quite dangerous to the health of QMP-based client/server environments.

Benefits/risks.
The main strength of message-oriented middleware (MOM) is asynchronous processing, i.e.,the clients and servers are not blocked while waiting for responses from each other. Another strength of MOM is recoverability from failures, ,because the message queues on disk can be used to recover the system after failures.

The main limitation of MOM is that the overhead of writing/reading from disk queues can slow down a client/server application. In addition, queueing of unpredictably large responses can result in disk overflows. A major limitation of MOM is that it introduces some unique end-to-end security exposures because many security packages at present assume a direct connection between the communication parties("no middle man").

In balance, MOM provides a very powerful approach for providing reliable and asynchronous communication between very heterogenous applications.

4.2 Message protocol

Messagetypes that users should send to perform action are shown in table 4.1.

Following table explains the fields used in describing message protocol.
E-mail service is used to carry messages between user and system. These messages must follow this syntax:

Name	*Description*
COR	Customer places order.
SOF	Supplier places offer.
SAC	Supplier accepts negotiation offer.
CAC	Customer accepts negotiation offer.
SAD	Supplier accepts deal.
CAD	Customer accepts deal.
SRJ	Supplier rejects deal.
CRJ	Customer rejects deal.
SRM	Supplier removes previous placed offer.
CRM	Customer removes previous placed order.

Table 4.1: Message types

Name	*Description*
cid	Unique number that identifies the customer.
sid	Unique number that identifies the supplier.
systemid	Unique number that identifies the system.
msgtype	Predefined combination of letters see 4.1.
date1	For customer, the earliest date to start transportation. For supplier, the earliest date he can start taking orders.
date2	For customer, the date when the machine must be at location2. For supplier, after that date he can not take any orders.
location1	The place where customers machine is.
location2	The place customers machine is going to be transported to.
machinetype	Specifies what kind of machine should be transported.

Table 4.2: Message protocol

Between <, > sign is a value.

: marks a delimiter. Hard brackets [,] contain values that are optional.

Customer places order:

<cid>:<msgtype>:<location1>:<location2>:<date1>:<date2>:<machinetype>

Example 12345#COR#Karlstad#Hagfors#2000-05-23#2000-05-25#1

Supplier places offer:

<sid>:<msgtype>:<date1>:<date2>[:<machinetype>]

Example 12345#SOF#2000-05-23#2000-05-25

Supplier accepts offer:

<sid>:<msgtype>:<offerid>

Example 12345#SAC#47

Customer accepts offer:

<cid>:<msgtype>:<offerid>

Example 12345#CAC#42

Supplier rejects negotiation offer:

<sid>:<msgtype>:<offerid>

Example 12345#SRJ#47

Customer rejects negotiation offer:

<cid>:<msgtype>:<offerid>

Example 12345#CRJ#42

Supplier removes a previously placed offer:

<sid>:<msgtype>:<offerid>

Example 12345#SRM#47

Customer removes a previously placed order:

<cid>:<msgtype>:<orderid>

Example 12345#CRM#42

System sends timeout message:

<mid>:<msgtype>:<date>

Example 99999#TUT#2047-05-10

4.3 JDBC/ODBC bridge

JDK 1.2.2 has been used to implement this service. The JDBC/ODBC bridge [6] ch5, gives access to any ODBC data source for free!
However, you need to be aware of a few limitations. For each system that is going to use the Bridge, software needs to be installed and configured. This can be a timeconsuming task as it is not accomplished automatically. Depending on what operating system is being used, it could be hard or expensive to find ODBC drivers.

There will be limitations of the ODBC driver that you will be using. If the ODBC driver can not do it, neither can the Bridge. The Bridge is not going to add any value to the ODBC driver that you are using other than allowing you to use it via JDBC. In order for the Bridge to properly use an ODBC driver, it must be ODBC version 2.0 or higher. Also, if there are bugs in the ODBC driver, they will surely be present when you use it from JDBC.

We must not forget Java security considerations. From the JDBC API specification, all JDBC drivers must follow the standard security model, most importantly:

- JDBC should not allow untrusted applets access to local database data.

- An untrusted applet will normally only be allowed to open a database connection back to the server from which it was downloaded.

Trusted applets and any type of application, can use the Bridge to connect to a data source of any type. Untrusted applets on the other hand can only access databases on the server from which they were downloaded. Normally, the Java Security Manager will prohibit a TCP connection from being made to an unauthorized hostname; i.e. if the TCP connection is being made from within the Java Virtual Machine (JVM). In the case of the Bridge, this connection would be made from within the ODBC driver, outside the control of the JVM. If the Bridge could determine the hostname that it will be connected to, a call to the Java

Security Manager could easily check to ensure that a connection is allowed. Unfortunately, it is not always possible to determine the hostname for a given ODBC data source name. For this reason, the Bridge always assumes the worst. An untrusted applet is not allowed to access any ODBC data source. This means that if you cannot convince the Internet browser in use that an applet is trusted, you cannot use the Bridge from that applet.

4.4 Transaction support

The Java 2 SDK v1.2.2, includes the JDBC 2.0 core API and the JDBC/ODBC bridge. Support for distributed transactions has been added as an extension to the JDBC 2.0 API. This feature allows a JDBC driver to support the standard 2-phase commit protocol used by the Java Transaction Service (JTS).

A transaction consists of one or more statements that have been executed, completed, and then either committed or rolled back. When the method `commit` or `rollback` is called, the current transaction ends and another one begins. A new connection is in auto-commit mode by default, meaning that when a statement is completed, the method `commit` will be called on that statement automatically. In this case, since each statement is committed individually, a transaction consists of only one statement. If auto-commit mode has been disabled, a transaction will not terminate until the method `commit` or `rollback` is called explicitly, so it will include all the statements that have been executed since the last invocation of the commit or rollback method. In this second case, all the statements in the transaction are committed or rolled back as a group. The method `commit` makes permanent any changes an SQL statement makes to a database, and it also releases any locks held by the transaction. The method rollback will discard those changes. Sometimes a user does not want one change to take effect unless another one does also. This can be accomplished by disabling auto-commit and grouping both updates into one transaction. If both updates are successful, then the commit method is called, making the effects of both updates

permanent; if one fails or both fail, then the rollback method is called, restoring the values that existed before the updates were executed. Most JDBC drivers will support transactions. In fact, a JDBC-compliant driver must support transactions. DatabaseMetaData supplies information describing the level of transaction support a DBMS provides.

4.5 Middleware between user & application server

The EMWAC Internet Mail Services for Windows NT (known as "IMS") are a suite of server programs, which allow you to use Windows NT as a mail server for Internet mail. The main reason why I choose this is that it stores the messages as files in each users mailbox directory. This makes it much easier to retrieve, parse and check for new messages. An other reason is that it is free, keeping costs low. The components of IMS are:

SMTP Receiver Listens for incoming mail, and stores it for processing by the SMTP Delivery Agent.

SMTP Delivery Agent This is the core of IMS. It delivers mail addressed to local users into their "incoming" mailbox, and sends other mail out onto the Internet. It uses MX records[5] (see [8] p625) in the DNS for routing mail. It also supports aliases and mailing lists.

POP3 Server This component gives local users the ability to download mail from their incoming mailbox on Windows NT to their own computer, using POP3 mail clients such as Netscape Navigator Version 2.0, Pegasus, or Eudora.

IMAP Server With IMAP, mail is stored permanently on the Windows NT machine, and an IMAP client such as Pine is used to access it. The user can organize her mail into hierarchical folders. Not yet available.

[5]specifies the name of the host prepared to accept email for the specified domain

EMWAC IMS Control Panel Applet This applet allows you to configure the EMWAC Internet Mail Services.

4.5.1 Protocol Specifications for EMWAC Internet Mail Services

This describes some of the technical specifications of the SMTP and POP3 implementations. The *SMTP Receiver* is a server implementation of the SMTP protocol defined in RFC 821[6] and RFC 1123. The following SMTP commands are supported by this version:
`HELO, QUIT, MAIL, RCPT, DATA, RSET, NOOP, VRFY`

The *POP3 Server* implements the POP3 protocol defined in RFC 1725. The following POP3 commands are supported by this version:
`USER name, PASS string, STAT, LIST [msg], RETR msg,`
`DELE msg, NOOP, RSET, TOP msg n, QUIT, UIDL [msg]`
The following optional POP3 commands are not supported in this version:
APOP name digest

[6]Request For Comments, technical reports that specifies standards. [8] p.71

5 Maintenance and system evolution

5.1 Future

Trailer could have a builtin GPS-system, that can be connected to the DB for updating the current position and keep track of speed to get approximate times of arrival.
Trailer has builtin computer to receive orders during driving, similar to the
"Hector weighing system", sold by BIS. (see brochures from BIS).
The features above could be handled using mobile phone (WAP), SMS, email or voice.

Applets can be built, so that users can view negotiation progress on the Internet. The whole service could also be made available on the Internet on some site with a login system using a challenge-response system, see [7] p263.

Should there raise a need to spread the service on more servers, this is no problem due to the fact that we use a 3-tier system see figure 4.1 and accompanying explanations.

6 Assumptions

The user of the system has to be predefined, i.e. should be an existing user in NT with a mailbox in EMWAC. Users will have to apply to the service and someone (sysadm) will thereafter process their request.

7 Conclusion

The performance of a client/server application must take into account the network performance, the volume of computational processing, and the volume of database operations. To estimate the performance of a client/server application we need to compute the Response Time (RT). Without queuing, RT, the response time, is given by

$$RT = \sum_i N(i)S(i) \tag{7.1}$$

Where $S(i)$ = time needed for completion of service i, $N(i)$ = number of times service i is needed, and service i represents any activity needed to complete an application (e.g., transmission of the messages between clients and servers over networks, processing of the message to produce the results, and transmission of the results back to the user).
However we must to take regard to queueing. Queues are formed due to two reasons: The device providing the service may be busy or it may be locked by another activity. The first condition is an indication of workload (too many services requested) and the second condition is a result of resources being reserved (i.e., a file being updated) by one activity. We focus on queueing due to workload.
We introduce the parameter, $A(i)$, to handle queueing, $A(i)$ = arrival rate of messages for service i.

For one e-mail queue
Let us assume that system will receive 500 messages per hour.

$$A(i) = \frac{500}{60min * 60sec} \approx 0.139 \tag{7.2}$$

The following formula shows utilization *U(i)* of a server i:

$$U(i) = server\ i\ utilization = A(i)\ *\ S(i) \tag{7.3}$$

A rule of thumb in queueing calculations is that *U(i)* should be kept below 0.7 to avoid queueing. The theoretical foundation for this rule of thumb is the following well-known M/M/1 (Markovian arrival, Markovian service time, 1 server) formula see [5]:

$$Queue\ length\ at\ server\ i = Q(i) = \frac{U(i)}{1 - U(i)} \tag{7.4}$$

where *Q(i)* shows the number of customers in the system, including the one being served. Thus $Q(i) = 1$ if $U(i) = 0.5$; *Q(i)* reaches infinity if $U(i) = 1$. The basic assumptions of the M/M/1 queueing formula are:

- Arrivals at the server are independent of each other
- Service times are independent of each other

From equation 7.3:

$$S(i) = \frac{U(i)}{A(i)} = \frac{0.5}{0.139} \approx 3.6sec \tag{7.5}$$

With this estimation we get a service time of 3.6 seconds. That means that if the system performs processing of single message in time that is less than 3.6 seconds, we should not have a problem with overqueueing.

However the arrival rate of messages can increase leading to an increase of A (arrival rate of messages). In that case, to handle the queue, the system must be refined to handle this as follows:

Separation

Database server and application server could be separated. This way we free resources and allow the operations on database and filesystem to be asynchronous

Computability

Servers that have higher computational performance could be acquired. This will improve overall system performance, which will decrease response time.

Parallelism

Since every state change in the system is updated in the database upon every transaction, it would be possible to have multiple number of servers to process messages in parallell.

The three statements above will prevent overqueueing from an increase in the arrival rate of messages.

Bear in mind the chosen solution, with asynchronous operation on a 3-tier middleware system. It is clear that it would be easily managed to handle messages from different queues, which belong to different messaging services between users and system.

The system now works on Windows NT 3.51, NT 4.0, Windows 2000 with e-mail server EMWAC 0.86 and could be extended with different messaging protocols.

References

[1] Booch, Grady. *Object-oriented Analysis And Design 2nd ed.*
Addison Wesley 1994.

[2] Date, C.J. *An Introduction To Database Systems.*
Addison Wesley 1994.

[3] Foreby, Per. *Att skriva rapporter med LaTeX.*
PH:s kopieringsmaskin, 1994.

[4] Horstmann, Cay S. and Cornell, Gary. *Core JAVA2 Advanced Features vol II.*
Prentice Hall 2000.

[5] Kleinrock, L. *Queueing Systems, Vol. 2.*
John Wiley 1976.

[6] Patel, Pratik *Java Database Programming with JDBC*
The Coriolis Group 1996.

[7] Pfleeger, Charles P. *Security in Computing 2nd ed.*
Prentice Hall 1997.

[8] Tanenbaum, Andrew S. *Computer Networks.*
Prentice Hall 1996 3rd ed.

[9] Umar, Amjad. *Application (Re)Engineering.*
Prentice Hall 1997.

[10] Umar, Amjad. *Object-oriented client/server Internet environments.*
Prentice Hall 1997.

A Code

A.1 BISNet.java

```
/**
 * @author Stefan Sonesson
 * This is the main class of BISNet
 */
import java.util.*;
import MsgFinder;
import MsgParser;
import DBManager;
//import ContractWatcher;

/**
 * BISNet is the control application.
 *
 */
public class BISNet
{ static MsgFinder mf;
  static MsgParser mp;
  static DBManager dbm;
  //static ContractWatcher cw;
  static Vector tmp = new Vector();

  /**
   * Main method of the BISNet class.
   * <br>Creates four objects mf, mp, mm and ms.<br>A while-loop runs
   * forever to check for incoming messages and deal with them accordingly.
   * @param String[] args, can take the directory of the mailboxes
   * e.g. c:\\mailbox\\user1.
   * @return void.
   * @exception Exception description.
   */
  public static void main(String[] args) throws Exception
  { try
    { System.out.println("Started.");
      //"d:\\stefan\\final\\processed.txt"
      mf = new MsgFinder(args[0], "processed.dat");
```

```
      mp = new MsgParser();
      dbm = new DBManager();
      //mm = new MatchMaker();
      boolean test = true;
      String retlist;

      while(test)
      { System.out.println("i while");
        if(mf.beginSearch() >= 1) // Ret_list contains 1 or more elements.
        { tmp = mf.ret_list;
        }
        System.out.println("Going to parse...");
        int posts = 0;
        if((posts = mp.parseMsg(tmp)) > 0)
        { System.out.println("OK, parsed, POSTS = " + posts);
          tmp.clear();
          tmp = mp.posts;
        }
        if(dbm.process(tmp) > 0)
        { System.out.println("OK, inserted");
          tmp.clear();
        }
        /*if(cw.findMatch() > 0)
          { System.out.println("Match found");
            //tmp = mm.matches;
            //mid = mm.mid;
          }*/
        if(tmp.size() == -700)
          test = false;
        }
        System.out.println("Finished.");
      }
      catch(Exception e)
      { System.out.println("System error: " + e);
      }
      finally
      {
      }
    }
  }
}
```

A.2 MsgFinder.java

```
/**
 * @author Stefan Sonesson
 *
 * @return No returnvalue
 */
import java.io.*;
import java.util.*;
import MsgParser;

/**
 * Searches given directory for new messages.
 * Additional verbose description.
 * @return Size of <tt><b>ret_list</b></tt>, if the size is greater than 0
 * it means that a new message has arrived.
 */
public class MsgFinder
{ private String save_file;
  private String directory = "c:\\mailbox";
  private Vector processed = new Vector();
  public Vector ret_list = new Vector();

  /**
   * Constructs a MsgFinder object.
   * Assigns the file_name to the variable save_file and loads the file list.
   * @param file_name keeps the names of the processed messages.
   * @return description.
   */
  public MsgFinder(String file_name)
  { //System.out.println("Constructed a MsgFinder1-object");
    save_file = file_name; //new File(file_name);
    loadFileList();
  }

  /**
   * Constructs a MsgFinder object.
   * Takes the arg as directory where to start looking for new files.
   * That directory should not contain any files only folders.
```

```
 * Assigns the file_name to the variable save_file and loads the file list.
 * @param arg The directory to be searched.
 * @param file_name keeps the names of the processed messages.
 */
public MsgFinder(String arg, String file_name)
{ //System.out.println("Constructed a MsgFinder2-object");
  directory = arg;
  save_file = file_name; //new File(file_name);
  loadFileList();
}

/**
 * Loads the processed files.
 * Reads a list of already processed files.
 * @return void.
 */
private void loadFileList()
{ try
  { FileReader fr = new FileReader(save_file);
    BufferedReader in = new BufferedReader(fr);
    String line;
    //System.out.println("loadFileList()");
    while((line = in.readLine()) != null)
    { processed.add(line);
    }
  }
  catch(IOException e)
  {// System.out.println("File disappeared");
  }
}

/**
 * Adds to the file of processed messages.
 * Appends the filename of a processed file to the processed.dat file.
 * @return void.
 */
private void appendFileList()
{ try
  { String str;
    int offset, str_length;
    FileWriter fw = new FileWriter(save_file, true);
```

```
    BufferedWriter bw = new BufferedWriter(fw);

    for(int x = 0; x < ret_list.size(); x++)
    { str = (String)ret_list.get(x);
      //offset = 0;//(int)save_file.length();
      //str_length = str.length();
      bw.write(str);
      bw.newLine();
    }
    bw.close();
  }
  catch(IOException ie)
  { System.out.println("Can't save to file: " + save_file);
  }
}

/**
 * This method searches directories for newly arrived messages
 * @return Size of the Vector ret_list
 */
public int beginSearch()
{ String retlist;
  ret_list.clear();
  File file = new File(directory);

  if(!file.exists() || !file.canRead())
  { System.out.println("Directory '" + file + "' doesn't exist or" +
                       " you don't have access to it!");
    return -1;
  }

  if(file.isDirectory())
  { String[] dir_list = file.list();
    File[] file_list = new File[dir_list.length];

    for(int x= 0; x < dir_list.length; x++)
      file_list[x] = new File(directory + "\\" + dir_list[x]);

    for(int i = 0; i < file_list.length; i++)
    { // It is a directory!
      if(file_list[i].isDirectory())
```

```
        { String[] dir_list2 = file_list[i].list();
          for(int ii = 0; ii < dir_list2.length; ii++)
          { if(!processed.contains(file_list[i] + "\\" + dir_list2[ii]))
            { processed.add(file_list[i] + "\\" +  dir_list2[ii]);
              ret_list.add(file_list[i] + "\\" +  dir_list2[ii]);
            }
          }
        }
        else
        { // It is a file in the "c:\mailbox" directory.
          // No files are allowed in the mailbox dir, only dirs
          // If needed to check for files there, uncomment below.
          /*if(!processed.contains(file_list[i]))
            { processed.add(file_list[i]);
              ret_list.add(file_list[i]);
            }*/
        }
      }
      /*for(int s = 0; s < processed.size(); s++)
          System.out.println(processed.get(s));        */
    }
    if(ret_list.size() >= 1)
      appendFileList();

    // Fr att kolla att listan r rtt
    /*for(int pos=0; pos < ret_list.size(); pos++)
    { retlist = (String)ret_list.elementAt(pos);

      System.out.println("retlist= " + retlist);
    }*/
    // Listans storlek
    // System.out.println("retlistsize= " + ret_list.size());
    return ret_list.size();
 }
}
```

A.3 MsgParser.java

```
/**
```

```
 * @author Stefan Sonesson
 * This class parses email messages and stores info in a Vector
 *
 */
import java.util.*;
import java.io.*;

public class MsgParser
{ private Vector msg_to_check = new Vector();
  public Vector posts = new Vector();
  public Vector insert_this = new Vector();
  private BufferedReader in;

  public MsgParser()
  {
  }

  /**
   * @args Takes a Vector tmp
   * @return Size of Vector posts
   */
  public int parseMsg(Vector tmp)
  { try
    { String msg, s, value;
      String[] strArray;
      msg_to_check = tmp;
      Vector temp = new Vector();

      for(int pos = 0; pos < msg_to_check.size(); pos++)
      { //Extract messages to be parsed
        temp.clear(); // Clear the temp Vector to insert new stuff
        insert_this.clear();
        msg = (String)msg_to_check.elementAt(pos);
        System.out.println("MsgParser - " + msg);
        in = new BufferedReader(new FileReader(msg));

        //Get lines from the message
        //Line 1 - Tells where msg came from
        //Line 2 - Line 1 cont.
        //Line 3 - Line 1 cont.
        //Line 4 - Tells the date
```

```
        //Line 5 - Tells what MESSAGE-ID the message has
        //Line 6 - The DATA sent (what we use for our service)
        //Observe, Line 1 = Vector.elementAt(0)

        while((s = in.readLine()) != null)
        { temp.addElement(new String(s));
        }

        try
        { in.close();
        }
        catch(IOException e)
        { System.out.println("error in close");
          e.printStackTrace();
        }

        //Pick out the line with data, maybe not always Vector[5]...
        StringTokenizer st=new StringTokenizer((String)temp.elementAt(5),#");
        while(st.hasMoreTokens())
        { value = st.nextToken();
          insert_this.addElement(new String(value));
          System.out.println("value=" + value);
        }
        posts.addElement(new Vector(insert_this));
      }
    }
    catch(Exception e)
    { System.out.println("caught");
      e.printStackTrace();
    }

    System.out.println(posts.size());
    return posts.size();
  }
}
```

A.4 DBManager.java

```
/*
```

```
// header - edit "Data/yourJavaHeader" to customize
// contents - edit "EventHandlers/Java file/onCreate" to customize
//
*/
import java.util.*;
import java.sql.*;

public class DBManager
{ private Vector to_insert = new Vector();
  private Vector tilf = new Vector();
  private Vector result = new Vector();
  private String command;
  private String query;
  private String field;
  private static String lim = "','";

  static DBHandler dbh;
  static ContractWatcher cw;

  /**
   * @args String str showing type of message
   * @return Integer depending on type of message<br>
   *  1 - Customer places a request<br>
   *  2 - Supplier places an offer<br>
   *  3 - Supplier accepts offer<br>
   *  4 - Customer accepts offer<br>
   *  5 - Supplier accepts deal<br>
   *  6 - Customer accepts deal<br>
   *  7 - Supplier rejects deal<br>
   *  8 - Customer rejects deal<br>
   *  9 - Supplier removes a placed offer<br>
   * 10 - Customer removes a placed order
   */
  public int getInt(String str)
  { int value = 0;
    //System.out.println("str=" + str);
    if(str.compareTo("COR") == 0)
      value = 1;
    else if(str.compareTo("SOF") == 0)
      value = 2;
    else if(str.compareTo("SAC") == 0)
```

```
    value = 3;
  else if(str.compareTo("CAC") == 0)
    value = 4;
  else if(str.compareTo("SAD") == 0)
    value = 5;
  else if(str.compareTo("CAD") == 0)
    value = 6;
  else if(str.compareTo("SRJ") == 0)
    value = 7;
  else if(str.compareTo("CRJ") == 0)
    value = 8;
  else if(str.compareTo("SRM") == 0)
    value = 9;
  else if(str.compareTo("CRM") == 0)
    value = 10;

  return value;
}

/**
 * If the message was of the type <tt><b>COR</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>INSERT INTO Orders(CID, SLOC, ALOC, SDATE, ADATE)<br>
 * VALUES('cid','sloc','aloc','sdate','adate');</tt></p>
 */
public Vector customerPlacesOrder(String cid)
{ String command;
  Vector tmp = new Vector();

  String sloc  = (String)tilf.elementAt(2); // Source destination
  String aloc  = (String)tilf.elementAt(3); // Target destination
  String sdate = (String)tilf.elementAt(4); // Earliest time of departure
  String adate = (String)tilf.elementAt(5); // Deadline of arrival
  String type  = (String)tilf.elementAt(6); // Machinetype

  command = "INSERT INTO Orders(CID,SLOC,ALOC,SDATE,ADATE,MACHINEID) " +
            "VALUES('"+cid +lim +sloc +lim +aloc +lim +sdate +lim +
            adate + lim + type + "');";
  tmp.add(command);
```

```
  return tmp;
}

/**
 * If the message was of the type <tt><b>SOF</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>INSERT INTO Orders(SID, SDATE, ADATE)<br>
 * VALUES('sid','sdate','adate');</tt></p>
 */
public Vector supplierPlacesOffer(String sid)
{ String command;
  Vector tmp = new Vector();

  //Can take orders from this date
  String sdate = (String)tilf.elementAt(2);

  //Can take orders until this date
  String adate = (String)tilf.elementAt(3);

  command = "INSERT INTO Offers(SID, SDATE, ADATE)" +
            "VALUES('"+ sid + lim + sdate + lim + adate +"');";

  tmp.add(command);
  return tmp;
}

/**
 * Checks to see if state is correct.
 * Queries the DB for the states belonging to a specific MID.
 * @param query A <tt>String</tt> holding state.
 * @param mid A String telling which record in the Match table.
 * @return void.
 */
public boolean checkState(String mid, String state)
{ query = "SELECT CSTATE, SSTATE FROM MATCH " +
          "WHERE MATCH.MID ='" + mid + "';";
  String stat;

  // Get the current states from MATCH table
```

```
  dbh.retrieve(query);
  stat = (String)dbh.result.elementAt(0);

  if(stat.compareTo(state) == 0)
    return true; // Right state
  else
    return false;//Wrong state
}

/**
 * This method takes a <tt>Vector</tt> and extracts the data from it.
 * Each part of the <tt>Vector</tt> is itself a <tt>Vector</tt> that
 * contains the fields to insert to the DB.
 */
public int process(Vector tmp)
{ to_insert = tmp;
  Vector bat = new Vector();

  // Extract Vectors with fields from the Vector with posts
  for(int i = 0; i < to_insert.size(); i++) // Loop the posts
  { tilf = (Vector)to_insert.elementAt(i);

    //These do not change
    String uid = (String)tilf.elementAt(0);
    String msg = (String)tilf.elementAt(1);

    int val = getInt(msg); // Convert the message to predefined int
    switch(val)
    { // [Exx] means Event number xx, see the state-event matrix.

      case 1: // [E01] Customer places order
        bat = customerPlacesOrder(uid);
        dbh.insert(bat);
        cw.findMatch();
        break;

      case 2: // [E02] Supplier places offer
        bat = supplierPlacesOffer(uid);
        dbh.insert(bat);
        cw.findMatch();
        break;
```

```
case 3: // [E03] Supplier accepts offer
  if(checkState((String)tilf.elementAt(2), "SRESP"))
  { cw.supplierAcceptsOffer(tilf);
  }
  break;

case 4: // [E04] Customer accepts offer
  if(checkState((String)tilf.elementAt(2), "CRESP"))
  { cw.customerAcceptsOffer(tilf);
  }
  break;

case 5: // [E05] Supplier accepts deal
  if(checkState((String)tilf.elementAt(2), "WDEAL"))
  { cw.supplierAcceptsDeal(tilf);
  }
  break;

case 6: // [E06] Customer accepts deal
  if(checkState((String)tilf.elementAt(2), "WDEAL"))
  { cw.customerAcceptsDeal(tilf);
  }
  break;

case 7: // [E07] Supplier rejects negotiation offer
  if(checkState((String)tilf.elementAt(2), "SRESP"))
  { cw.supplierRejectsOffer(tilf);
  }
  break;

case 8: // [E08] Customer rejects negotiation offer
  if(checkState((String)tilf.elementAt(2), "CRESP"))
  { cw.customerRejectsOffer(tilf);
  }
  break;

case 9: // [E09] Supplier removes previously placed offer
  cw.supplierRemovesOffer(tilf); //No need check state
  break;                         //Can remove offer anytime
```

```
        case 10: // [E10] Customer removes previously placed order
          cw.customerRemovesOrder(tilf); //No need check state
          break;                         //Can remove offer anytime

        case 11: // [E11] Timeout
          cw.timeout(tilf);
          break;

        default :
          System.out.println("val= " + val);
          System.out.println("Message incomplete, send err msg to C or S");
          return 0;
      }
      tilf.clear();
      bat.clear();
    }
    return 1;
  }
}
```

A.5 ContractWatcher.java

```
/*
 * @author Stefan Sonesson
 * This class examines the DB to find matches order/offer.
 *
 */
import java.io.*;
import java.sql.*;
import java.util.Vector;
import DBHandler;
import MsgSender;


public class ContractWatcher
{ private String cstate;
  private String sstate;
  private String command;
  private String query;
```

```
private String field;
public Vector matches;

static String ACK     = "Your acceptance has been received.";
static String NACK    = "Failed to process your request";
static String SYSMAIL = "system@bisnet.se";
static DBHandler dbh;
static MsgSender ms;

public void ContractWatcher()
{
}

/**
 * Find matching orders/offers.
 * Inserts matches to the MATCH table and updates the state of these.
 * @return void.
 */
public void findMatch()
{ command = "INSERT INTO MATCH(OID, OFID) " +
            "SELECT Orders.OID, Offers.OFID " +
            "FROM Orders, Offers " +
            "WHERE Orders.SDATE BETWEEN Offers.SDATE " +
            "AND Offers.ADATE;";
  query = "";
  dbh.insert(tmp);
  //updateDB();
}

/**
 * If the message was of the type <tt><b>SAC</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>UPDATE MATCH SET SSTATE = 'CRESP'<br>
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * @param tilf
 */
public void supplierAcceptsOffer(Vector tilf)
{ String command;
```

```
  String query  = "SELECT EMAIL"+
                  "FROM SUPPLIERS"+
                  "WHERE SID = '" + tilf.elementAt(0) + "';";

  Vector tmp    = new Vector();
  Vector tosend = new Vector();
  String email  = "";

  String mid    = (String)tilf.elementAt(2);
  command       = "UPDATE MATCH SET SSTATE = 'CRESP' " +
                  "WHERE MID = '" + mid + "';";

  tmp.add(command);

  // Get address to reply to
  if(dbh.retrieve(query) > 0)
  { email = (String)dbh.result.elementAt(0);
  }
  else // Retrieving email failed, nobody to reply to...
  { // send message to ourself
    tosend.clear();
    tosend.add(SYSMAIL);
    tosend.add(command);
    ms.sendMail(tosend);
  }

  if(dbh.insert(tmp) > 0) // Insertion OK
  { tosend.clear();
    tosend.add(email);
    tosend.add(ACK);
    ms.sendMail(tosend);
  }
  else // Insertion failed
  {

  }
  // send msg to customer
  // update CSTATE to CRESP

}
```

```
/**
 * If the message was of the type <tt><b>CAC</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>UPDATE MATCH SET CSTATE = 'WDEAL'<br>
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * Retrieved message looks like this:<p>
 * #user_id#messagetype#match_id
 * @param tilf
 */
public void customerAcceptsOffer(Vector tilf)
{ String command;
  String state = "CRESP";

  Vector tmp    = new Vector();
  Vector tosend = new Vector();
  String mid    = (String)tilf.elementAt(2);

  command = "UPDATE MATCH SET CSTATE = 'WDEAL' " +
            "WHERE MID = '" + mid + "'";

  tmp.add(command);
    if(dbh.insert(tmp) > 0)
  { tosend.add(ACK);
    ms.sendMail(tosend);
  }
  else
  { tosend.add(NACK);
    ms.sendMail(tosend);
  }
}

/**
 * If the message was of the type <tt><b>SAD</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>UPDATE MATCH SET SSTATE = 'WDEAL'<br>
```

```
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * Retrieved message looks like this:<p>
 * #number#user_id#match_id
 * @param tilf
 */
public void supplierAcceptsDeal(Vector tilf)
{ String command;
  Vector tmp    = new Vector();
  Vector tosend = new Vector();

  String mid = (String)tilf.elementAt(2);

  command = "UPDATE MATCH SET SSTATE = 'WDEAL' " +
            "WHERE MID = '" + mid + "'";

  tmp.add(command);
  if(dbh.insert(tmp) > 0 )
  { tosend.add(ACK);
    ms.sendMail(tosend);
  }
  else
  { tosend.add(NACK);
    ms.sendMail(tosend);
  }
}

/**
 * If the message was of the type <tt><b>CAD</b></tt>, this method
 * constructs appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * <br><tt><b>command</b></tt> is an SQL expression with the following
 * syntax:<p>
 * <tt>UPDATE MATCH SET CSTATE = 'WDEAL'<br>
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * @param tilf
 */
public boolean customerAcceptsDeal(Vector tilf)
{ String command;
  Vector tmp    = new Vector();
  Vector tosend = new Vector();
```

```
  String mid    = (String)tilf.elementAt(2);

  //Change state to WDEAL
  command = "UPDATE MATCH SET CSTATE = 'WDEAL' " +
            "WHERE MID = '" + mid + "'";

  tmp.add(command);
  if(dbh.insert(tmp) >0 )
    return true;
  else
    return false;
}

/**
 * If the message was of the type <tt><b>SRJ</b></tt>, this method
 * constructs three appropriate <tt>String</tt>'s named
 * <tt><b>command</b></tt>.<br>The first <tt><b>command</b></tt> is an SQL
 * expression with the following syntax:<p>
 * <tt>UPDATE MATCH SET SSTATE = 'END'<br>
 * WHERE MID = 'mid'</tt></p>
 * The second <tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>INSERT INTO REJECTS(OFID,OID)<br>
 * SELECT OFID, OID<br>
 * FROM MATCH<br>
 * WHERE MID = 'mid'</tt></p>
 * The third <tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>DELETE<br>
 * FROM MATCH<br>
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>insert()</b></tt> is called between the first two command
 * strings.<br>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * @param tilf
 */
public boolean supplierRejectsOffer(Vector tilf)
{ String command1, command2, command3;
  Vector tmp = new Vector();
  Vector tosend = new Vector();
  String mid = (String)tilf.elementAt(2);
```

```
  //Change state to END
  command1 = "UPDATE MATCH SET SSTATE = 'END' " +
             "WHERE MID = '" + mid + "'";

  tmp.add(command1);
  //insert();

  //Update REJECTS, do not match those again
  command2 = "INSERT INTO REJECTS(OFID,OID) " +
             "SELECT OFID, OID " +
             "FROM MATCH " +
             "WHERE MID = '" + mid + "'";

  tmp.add(command2);
  //insert();

  //Delete the post from MATCH
  command3 = "DELETE " +
             "FROM MATCH " +
             "WHERE MID = '" + mid + "'";

  tmp.add(command3);
  if(dbh.insert(tmp) > 0 )
  { return true;
  }
  else
    return false;
  //if(insert(tmp) == 0)
    //System.out.println("Print to errorlog");
    //Write error message to a file, the database is probably broken...
}

/**
 * If the message was of the type <tt><b>CRJ</b></tt>, this method
 * constructs two appropriate <tt>String</tt>'s named
 * <tt><b>command</b></tt>.
 * The first <tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>INSERT INTO REJECTS(OFID,OID)<br>
 * SELECT OFID, OID<br>
```

```
 * FROM MATCH<br>
 * WHERE MID = 'mid'</tt></p>
 * The second <br><tt><b>command</b></tt> is an SQL expression with
 * the following syntax:<p>
 * <tt>DELETE<br>
 * FROM MATCH<br>
 * WHERE MID = 'mid'</tt></p>
 * <tt><b>insert()</b></tt> is called between the first two command
 * strings.<br>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 * @param tilf
 */
public boolean customerRejectsOffer(Vector tilf)
{ String command1, command2;
  Vector tmp = new Vector();
  Vector tosend = new Vector();
  String mid = (String)tilf.elementAt(2);


  //Update REJECTS, do not match those again
  command1 = "INSERT INTO REJECTS(OFID,OID) " +
             "SELECT OFID, OID " +
             "FROM MATCH " +
             "WHERE MID = '" + mid + "'";

  tmp.add(command1);
  //insert();

  //Delete the post from MATCH
  command2 = "DELETE " +
             "FROM MATCH " +
             "WHERE MID = '" + mid + "'";

  tmp.add(command2);
  if(dbh.insert(tmp) >0 )
    return true;
  else
    return false;
  //if(dbw.insert(tmp) == 0)
    //System.out.println("Print to errorlog");
    //Write error message to a file, the database is probably broken
```

```
}

/**
 * If the message was of the type <tt><b>SRM</b></tt>, this method
 * constructs an appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * The first <tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>UPDATE MATCH SET SSTATE = 'END'<br>
 * WHERE MID = 'mid'</tt></p>
 * The second <br><tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>DELETE<br>
 * FROM OFFERS<br>
 * WHERE OFID = 'ofid'</tt></p>
 * <tt><b>insert()</b></tt> is called between the first two command
 * strings.<br>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 */
public boolean supplierRemovesOffer(Vector tilf)
{ String command;
  Vector tmp = new Vector();
  Vector tosend = new Vector();
  //The meaning is to let the supplier remove an offer made by him
  String ofid = (String)tilf.elementAt(2);

  // Tar ACCESS bort poster som innehller OFID i andra tabeller???

  // Lade till referensintegritet i DB relationer. Det borde betyda
  // att om jag tar bort frn OFFERS s tar DB bort frn MATCH...

  //Change state to END
  //command = "UPDATE MATCH SET SSTATE = 'END' " +
  //          "WHERE MID = '" + mid + "'";
  //insert();

  //Delete the post from MATCH
  command = "DELETE " +
            "FROM OFFERS " +
            "WHERE OFID = '" + ofid + "'";

  tmp.add(command);
```

```
  if(dbh.insert(tmp) >0 )
    return true;
  else
    return false;
}

/**
 * If the message was of the type <tt><b>CRM</b></tt>, this method
 * constructs an appropriate <tt>String</tt> named <tt><b>command</b></tt>.
 * The first tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>UPDATE MATCH SET CSTATE = 'END'<br>
 * WHERE MID = 'mid'</tt></p>
 * The second <br><tt><b>command</b></tt> is an SQL expression with the
 * following syntax:<p>
 * <tt>DELETE<br>
 * FROM ORDERS<br>
 * WHERE OID = 'oid'</tt></p>
 * <tt><b>insert()</b></tt> is called between the first two command
 * strings.<br>
 * <tt><b>mid</b></tt> is the identifier of the specific MATCH.
 */
public boolean customerRemovesOrder(Vector tilf)
{ String command;
  Vector tmp = new Vector();
  Vector tosend = new Vector();
  String oid = (String)tilf.elementAt(2);


  // Tar ACCESS bort poster som innehller OFID i andra tabeller???

  // Lade till referensintegritet i DB relationer. Det borde betyda
  // att om jag tar bort frn OFFERS s tar DB bort frn MATCH...

  //Change state to END
  //command = "UPDATE MATCH SET CSTATE = 'END' " +
  //          "WHERE MID = '" + mid + "'";
  //insert();

  //Delete the post from MATCH
  command = "DELETE " +
```

```
              "FROM ORDERS " +
              "WHERE OID = '" + oid + "'";
    tmp.add(command);
    if(dbh.insert(tmp) >0 )
      return true;
    else
      return false;
  }
}
```

A.6 MsgSender.java

```
/**
 *
 * MsgSender.java
 *
 */
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import javax.swing.*;

public class MsgSender
{ private BufferedReader in;
  private PrintWriter out;

  public void sendMail(Vector message)
  { try
    { Socket s = new Socket("smtpds", 25);

     out = new PrintWriter(s.getOutputStream());
     in = new BufferedReader(new InputStreamReader(s.getInputStream()));

     String hostName = InetAddress.getLocalHost().getHostName();

     send(null);
     send("HELO " + hostName);
     send("MAIL FROM: system@bisnet.se");
```

```
     send("RCPT TO: " + (String)message.elementAt(0));
     send("DATA");
     out.println((String)message.elementAt(1));
     send(".");
     s.close();
    }
    catch (IOException exception)
    {  System.out.println("Error: " + exception);
    }
  }
  public void send(String s) throws IOException
  {  if(s != null)
    { //response.append(s + "\n");
      out.println(s);
      out.flush();
    }
    //String line;
    //if((line = in.readLine()) != null)
     //response.append(line + "\n");
  }
}
```