IT 08 044
Examensarbete 15 hp
December 2008

Incorporating an ISO 8583-service into an unreliable application

Alexander Funcke

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Incorporating an ISO 8583-service into an unreliable application

Alexander Funcke

If an existing service is to be made available to a broader audience, via for instance the Internet. The implementation of the service may thereby get many new requirements, such as support for a different access method or stricter authorisation and security policies.

This thesis discusses the implementation of an ISO 8583 to XML proxy. ISO 8583 is a widely used financial protocol, and was in this project used to communicate with the backing financial server in order to implement a complete web banking system. Hence the proxy encapsulates the already well-proved financial system into a module, that has taken appropriate security concerns and has a messaging schema that is compatible with the rest of the system.

The ISO 8583 and XML protocols are inherently incompatible, and hence the proxy can not be as simple as a translational proxy; it needs to keep state. How to handle this, and how to do it for many clients concurrently is the main focus of this paper. It does however also include a look at the broader picture of the full system, and how some of the implementation specific quirks was handled.

Handledare: Flavio Pescuma
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Anders Jansson
IT 08 044
Tryckt av: Reprocentralen ITC

Sammanfattning

Runt hela jorden, i så väl I- som U-länder, så utförs fler och fler banktransaktioner med hjälp av informationsteknik, så som Internet. Det finns många intressanta problem med datoriseringen av banktransaktioner, så som autenticerings, kryptografi och allmän system säkerhet.

Det är vanligt att man talar om olika lösningar som om de vore goda approximationer av den "bästa lösningen". Detta tror jag är en missuppfattning. Det är tvärtom så att en lösning måste ta hänsyn till faktorer som de rådande kulturella och infrastrukturella förutsättningarna när man avgör vad som vore den "bästa lösningen". Och lösningarna varierar verkligen. I delar av Asien och Afrika möter man efterfrågan på moderna banktjänster med hjälp av t ex SMS-kontrollerade överföringar, medans denna service inte saluförs alls i Sverige. Ett annat infrastruktur-exempel är den relativt säkra postala distributionen av autenticerings tokens som många banker brukar i Sverige. Dessa metoder är omöjliga att implementera i stora delar av t ex Latinamerika. Vilket gör många typer av autenticeringsmetoder i princip omöjliga att implementera där.

Motsvarande problem finns även i interaktionen med det existerande finansiella systemet. De baltiska bankerna som har förhållandevis moderna system såg sig bromsas när de svenska bankerna köpte upp dem och skulle harmonisera tjänsterna som erbjöds. Inte för att svenska banker hade lägre ambitioner, men deras existerande system var äldre och bar hade därmed andra förutsättningar vilket ofta betyder högre kostnader för att göra förändringar.

Det är många typer av investeringar som är stigberoende i denna bemärkelse, och att kunna återanvända ett välbeprövat och dyrt system för att möta den nya efterfrågan istället för att investera helt nytt är ofta lockande.

Ett sådant alternativ, som inte kräver några förändringar i de känsliga, men beprövade kärnan av de finansiella systemen är att utnyttja existerande gränssnitt och funktionalitet för att bygga de nya systemen.

Det existerande gränssnittet, i det aktuella fallet ISO 8583 standarden,

passar sällan direkt till det nya systemet, utan gör antaganden om den kommunicerande parten.

För att lösa det här problemet så kan man utveckla ett proxy som dels tar hand om översättning av de olika protokollen, men som även uppfyller de olika antaganden som klienten och servern ställer.

Det här pappret tittar närmare på ett par intressanta detaljer vid utvecklingen av ett sådant proxy mellan ISO 8583 protokollet och ett XML-baserat protokoll vid namn BMTML.

För att lösa skillnaderna i semantik så instansieras en ändlig automat för varje klient-initierad-transaktion. Automaten har bl a två olika loopar för att hantera ett avbrott från klienten, och för att göra uppslag som ger mer data än vad som rymms i ett ISO 8583 meddelande.

Det finns även ett par andra saker som behandlas i olika utsträckning i detta papper. En sådan är hur man kan integrera ett transaktionsförfarande i ett s k content-management system (CMS), d v s i ett system för enkel hantering av websidor. Ett annat är vilka överväganden som gjorts för att få en snabb mjukvara.

Contents

Chapter 1

Introduction

More and more of our bank transactions are made over the Internet, rather than in traditional bank offices. This is no longer only true in the industrialised world, but globally.

To satisfy this new demand for electronic services existing banks need to adopt new technical infrastructure, and to do so often find themselves forced to do modifications to their very core systems.

To avoid this they could try to utilise already existing interfaces for communication and transactions. One such interface is the messaging using the ISO 8583 standard. The ISO 8583 standard is used extensively in financial systems all over the world. It is carrying most of the worlds automated teller machine (ATM) and credit card transactions.

The main focus of this paper is the core component of a web-banking system developed to reuse the ISO 8583 interface of an existing financial system. The core component is hereinafter referred to as the isoproxy.

The isoproxy-component is briefly a proxy in-between a non-transactional XML-messaging, called BMTML (BM Transaction Mark-up Language) and the transactional ISO 8583 messaging. This is obviously not a one-to-one mapping between the two, and the details on how to design such a mapping and the BMTML-messaging as such is the main problem that this paper will explore. Another important topic that this paper will try to study is how to ensure high-performance in a proxy of this kind.

As preparations for the main topics we will take a look at the bigger picture of the project as a whole. However before looking anywhere else the two communication protocols central to this paper are to be examined, that is ISO 8583 and BMTML.

Chapter 2

ISO 8583 standard

2.1 Background

The International Organization for Standardization, also known as ISO, published the ISO 8583 standard in 1987 for usage in, and between, financial systems.

The standard introduces a protocol that has been used for financial transactions ever since, and whom is still used extensively in the financial sector today.

One application that uses the protocol extensively is the debit/credit card systems. Many, if not most, of the actors in the credit card sector uses ISO 8583 to communicate with their subscribing banks and vice versa.

Further, most of the automated-teller-machines (ATMs) uses this protocol to communicate with their respective bank's mainframe[18] in order to make the bank balance reflect the transactions conducted at the ATM.

The ISO 8583 specification exist in three revisions published in 1987, 1993 and 2003. The isoproxy was implemented using the original specification from 1987. This choice was due to the fact that all the already existing systems used the original specification to communicate with the customer's financial systems.

The standard is very focused on syntactical issues, but do set a few semantical communicative details as well. We will later see that some of the semantical details in practise often is regarded as conventions rather than definitions.

The ISO 8583 standard do not even fix any low-level details on how the messages are transported, encoded, nor does it set the standard for what information to carry where.

The details on this system's communicative patterns will therefore reveal themselves more in the next two chapters rather then this one.

2.2 Specification details

2.2.1 General

As just mentioned in the previous section, ISO 8583 does not specify any transport protocol, nor how the information in the messages ought to be encoded. There are a multitude of variations of encoding standards, e.g. ASCII and EBCDIC. The two protocol most commonly used as carriers for ISO 8583 messaging are SNA and TCP/IP[5].

The current version of the isoproxy only supports ISO 8583 messages that uses an ASCII encoding and are carried over a TCP/IP connection.

When the ISO standard, and hence this paper, uses terms like "digit" or "numerical character". These expression are represented as the number of bits that such a character spans in the currently used encoding. Since the isoproxy uses ASCII; a numerical character or digit uses 8 bits, or equivalently 1 byte.

An ISO 8583 message is essentially a portion of data divided into different fields. The fields are pre-defined for the specific implementation and then set up each fields characteristics. The presence of a specific field in a message is indicated by setting a bit in one of the mandatory fields, called the "Bitmap". This will be discussed in further details in the "Bitmap"-section below.

There is a set of fields defined by ISO as a common ground for applications to use (it is included as Appendix A). These definitions are as previously noted in practise often seen as conventions, and are often, altered as new needs arises.

In the following subsections we will look closer at some of the reserved fields in the standard that are central to how the messaging is conducted.

2.2.2 Message type indicator

The "Message type indicator" (MTI) is a reserved field in the ISO 8583 specification. It is the first indicator for the receiving end of what the nature of the message is.

The four initial digits of any ISO 8583 message is always the MTI field, and each digit encodes a message property.

The first digit declares what version of the ISO 8583 protocol the message intend to use. As mentioned in the introduction, there only exist three differ-

ent revisions of the protocol, 1987, 1993, 2003, and they are here represented by a 0, 1 or 2 respectively.

The second digit indicates what class the message is of. The available classes of messages are: authorisation, financial, file actions, reversal, reconciliation, administrative, fee collection, network management and a reversed class.

The third digit sets what function the message is suppose to serve. There is a digit allocated for each of the following options: a request, a response of a request, an advise, a response of an advise, a notation or an acknowledgement.

Finally the fourth digit gives the origin of the message. That is, if it is send from the acquirer, the issuer or an other source.

2.2.3 Bitmap

The "Primary bitmap" is a reserved field of 64 bits in the ISO 8583. Every bit represents a field. If the bit is set or not indicates whether the field will be present in the message. Since the bitmap is only 64 bits large one may specify the presence of up to 64 data elements in the primary bitmap.

There is however a "Secondary bitmap" field reserved in ISO 8583. It is of course only present and considered if the corresponding bit is set in the primary bitmap, and if it is, the secondary bitmap functions in the same manner. This gives a possibility of a maximum of 128 data elements (including the bitmaps).

The standard also mentions the possibility of further bitmaps, but I have yet to see it being used in any implementation.

2.2.4 Data elements definitions

Each field has a canonical index, and each index needs to share a data element definition between the communicating peers, fixing the type of data element the field will carry.

Below are the available data types for the data element, and an introduction to how they are utilised follows by example:

a - A string of alphabetic characters and blanks.

n - A string of numeric characters.

s - A string of neither alphabetic nor numeric.

b - Binary data.

z - Track 2 and 3 codes (in accordance to ISO 4909, ISO 7813)

For instance, the "Processing code" (field 3), data element is normally defined as *"n6"* . This indicates that it is a six characters long numeric string.

A data element may also be of a variable length. One example of that is the "Primary account number" (field 2) data element. The data element has the following definition, *"n..19"*, which means that it could store a numerical string of max 19 digits. The number of dots in the definition reflects the amount of data needed to store the size of the actual data element. In the case of "..", as in "n..19", two characters (16 bits in ASCII) is used, and for "...", as in "n..999", it uses three characters (24 bits in ASCII).

There is also the possibility to combine allowed data types so that you can define for instance alpha-numeric data elements or data elements that allow any character. Consider for example *"an16"*, that is, a static alphanumeric string consisting of 16 characters, and *"ans...999"*, a string of variable length of at most 999 characters.

2.3 The flow of a ISO 8583 message

A typical exchange of ISO 8583 messages gets initiated when a client sends a message with an "Message Type Indicator" indicating his purpose and a "Primary bitmap" indicating what fields will be present.

The third field, "Processing code", is often of special importance, since it is commonly utilised in the receiving server as a key to what part of the server-side code that shall be executed.

The server will reply with an ISO 8583 message as soon as it has handled the request which usually contains a "Return code" as well as other fields. Indicating the status, or retrieving information to the client.

If the flow for some reason gets interrupted, or the client for some other reason want to abort the action, the client will send a *reversal message*. It is essentially the same message as the client previously were sent, but the second digit in the "Message Type Indicator" is replaced with a "4".

The server will then abort the handling of the previous request and send back a *reversal acknowledge* indicating to the client that the server acknowledged the clients abortion. If the client do not receive an acknowledgement within a window of time it will resend the reversal message.

Chapter 3

The XML-based messaging

3.1 BM Meta Mark-up Language (BMMML)

The BM Meta Mark-up Language (BMMML) is used to propagate the settings for the system as a whole to its components, such as the isoproxy.

The setting most worthy of noting in regard to the isoproxy is the data element definitions of all possible fields and message definitions where different fields are associated to a tuple of "Message Type Indicator" and "Processing Code".

The BMMML messaging is central for the functionalities of the system as a whole, but not key to describe the isoproxy's role. Further details are therefore omitted. There is however an example of a fictional BMMML-message exchange included in Appendix C.

3.2 BM Transactional Mark-up Language (BMTML)

The BM Transactional Mark-up Language (BMTML) was specifically drafted for the purpose of this project. It is a human readable messaging language extended from W3C's well-known XML standard. The BMTML language was thus modelled to communicate the same sort of information as the ISO 8583 does in the system.

A fictional BMTML-message exchange is included in Appendix B as an example, and in order to give a more precise idea of what a BMTML message is, a descriptions of its constituting parts will follow.

A BMTML-message consists of a "BMTML"-root element containing a "query" or "result" element.

3.2.1 $< query >$ and $< result >$

The "query"-element indicates that the BMTML-message is send from a calling machine to the isoproxy. A "result"-element is the opposite, that is a isoproxy responding to a calling machine. The "query"- and "result"-elements has two attributes "mid" and "cid", who are abbreviations for "Message Identifier" and "Caller Identifier" respectively.

The first is used to identify the specific message. The second is used to identify the calling client.

Each "query"-element contains one or more "transaction"-element(s).

3.2.2 $< transaction >$

Every "transaction"-element corresponds to a transaction, or equivalently a complete ISO 8583 message exchange. The "name" attribute of the element is used as a key to look up the transaction definition, and hence sets the transaction's functionality. The "tid" attribute is an abbreviation for "Transaction Identifier" and is used just to identify the current transaction. The same "tid" will be used in the "query" and "result" BMTML message.

Each "transaction"-element may contain one or more "column"-element(s).

3.2.3 $< column >$

Every "column"-element correspond to a data element in the ISO 8583 standard. The correspondence is however not symmetric, hence there are data elements in ISO 8583 that will not be rendered as "column"-elements in a "result". The relations are given by the configuration retrieved from the metaserver.

The "column"-element has an attribute called "name", which is used as key to look up the data element definition, as well as to the data itself.

The "column"-element can be used in two different manners, a simple, and a complex.

In the simple case, the text found in the element is the content of the "column", who in most "query" cases will end up in the corresponding field in a ISO 8583 message, and in the case of a "result", origins from such a field in a ISO 8583 message.

The complex case contains one or more "row"-element(s).

3.2.4 $< row >$

A “row”-element is a element that contains one or more “column”-elements described as above. This gives the possibility to receive a matrix from the isoproxy.

This concept is not reflected in the ISO 8583 standard and it is also exotic in the sense that an BMTML reply from the isoproxy containing a “row” element may correspond more then one ISO 8583 request from the isoproxy to the backing system.

Chapter 4

The bigger picture

4.1 Purpose

The isoproxy was developed for two expressed purposes. First, to be a component in a larger web-banking system and secondly, to serve as the interface to the financial system of a credit card company by being a so-called virtual point-of-sale (vPOS) solution.

A web-banking system is the complete infrastructure that makes it possible for bank customers to do transactions from home via their computer's web browser application.

A point-of-sale (POS) is traditionally a card reader that can dial-up the credit card company and charge the card for a given transaction. This equipment is typically used by the clerk at every credit card purchase in a normal store.

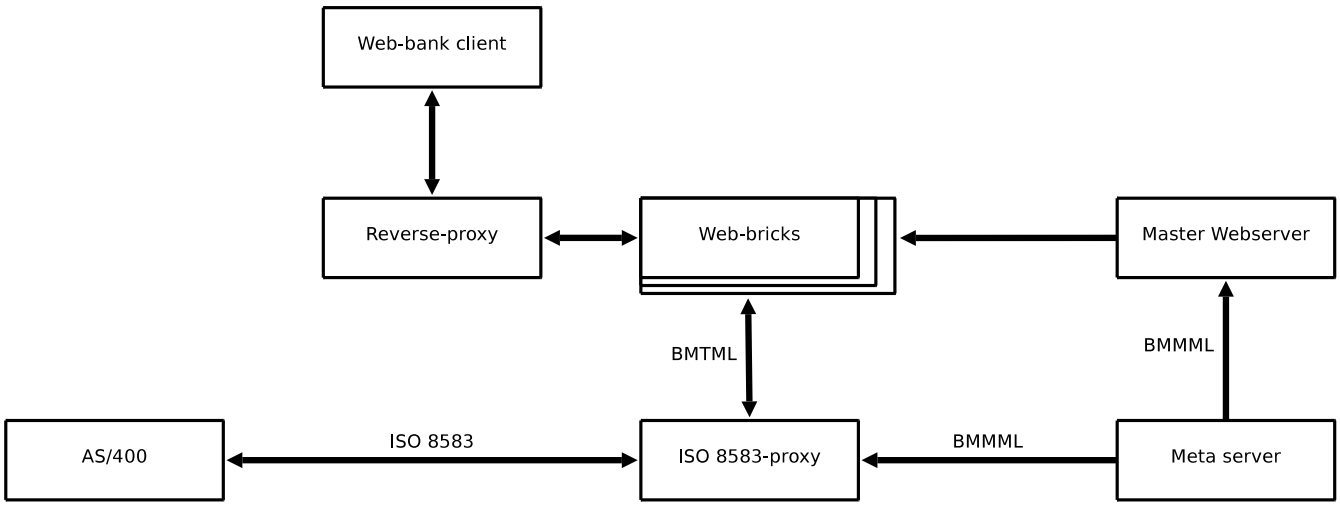
A vPOS is the virtual equivalent and hence the software that communicates with the credit card company to finalise a purchase in an online store.

4.2 Web-banking system

The diagram below shows the main components of the system and how they relate and communicate. Some components, such as those responsible for authentication and security, has been left out to preserve the element of "obscurity". Other details that for this study are of a more irrelevant nature, such as the mirroring of systems to ensure high-availability has also been left out for clarity.

In this case a transaction gets initiated by a end-user that accesses one of the web-bricks. The transaction is then forwarded to the isoproxy that

will communicate with the server-side system. The result of the transaction is then propegated in the reverse order back to the client.



We will briefly go through each component to get an idea of what role the isoproxy has in the larger system and how the other components relate to it.

4.2.1 Content-management system

The content-management system (CMS) is not a component represented in the figure above, since it is a component on another level. It is however crucial to present in order to understand the roles at the level of the figure. The CMS is a system to administrate the web pages served by the web-bricks. It makes both content and presentation easily managed by offering a structured graphical user interface to the administrator.

This implementation of a CMS is widget-based, that is web pages are structured using components (widgets) that have both presentational and functional aspects. Just as a window, or a button in a given window manager looks and work alike (windows and buttons are also called widgets in this context).

4.2.2 Reverse-proxy

The reverse-proxy keeps track of authentication and distribution of traffic in-between the web-bricks according to different parameters, such as the URL and the level of authentication.

4.2.3 metaserver

The metaserverstores and distributes the definitions of the available transactions, as well as other system-wide settings. It keeps the definitions of the ISO 8583 mappings to the BMTML equivalence and details for how the CMS is intended to render data of the various types. It uses the XML-based BMMML (BM Meta Mark-up Language) messaging to distribute its data to the web servers and isoproxy.

4.2.4 Master web server — The back-end

The web server runs on a well-protected traditional server and not a minimal appliance as the rest of the components do. It controls using the content-management system what content will be uploaded to the web-bricks. This machine also runs the user-interface of the CMS.

4.2.5 Web-bricks — The front-end

The role of the web-bricks is, put simply, to serve HTML-content to the end-user via the HTTP-protocol.

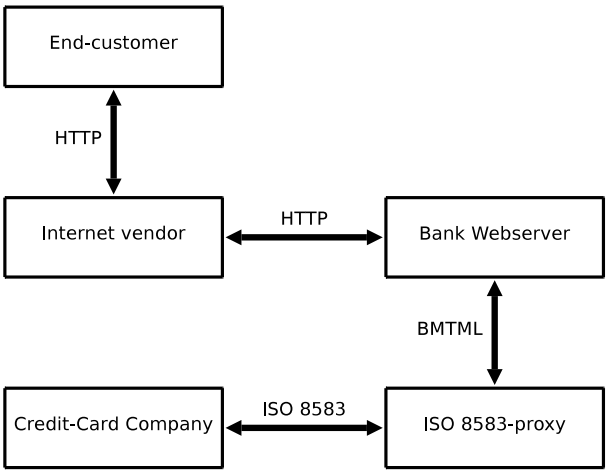
The deployment of web-bricks is actually more complex then what the diagram reveals. There is a differentiation in levels of protection on the different web-bricks set up depending on how critical the functionality served is.

The static and scripted content on the web-bricks is synced from the Master web server, who in turn has requested the meta definitions of the available transactions from the metaserverusing the BMMML-messaging.

As the end-user wants to execute a bank transaction, say a transfer in-between accounts, the web-brick in question will send an appropriate BMTML-message to the isoproxy whom will report back to the web-brick with a status or error BMTML message.

4.3 Virtual point-of-sale

The vPOS usage of the isoproxy is a simpler service than the web-banking system, and has only five components. The end-user, the Internet-based vendor, the bank's web server, the isoproxy and the credit card company, interconnected as in the diagram below.



As the end-customer filled his virtual shopping trolley at the Internet vendor's website, the end-user checks out. This action will redirect the end-user to the bank's website and a CGI-program. The CGI-program will verify that the request is a valid one (which actual incorporates the isoproxy too) and then send a BMTML-message to the isoproxy. The isoproxy will then connect to the credit card company using a private ISO 8583 channel.

The CGI will wait patiently for the isoproxy to complete its transaction, sending non-blocking spaces to the end-user in order to prevent a time-out. If a time-out does occur, the user will still be charged and the Internet vendor will of course be notified. It is then the responsibility of the vendor to contact the end-customer to inform about the successful purchase in a more reliable way, typically via e-mail.

Chapter 5

Problems addressed in the design

5.1 Introduction

Considering the isoproxy's role as described in the previous chapter "The bigger picture", it is quite obvious that the other components puts specific requirements, as well as constraints on the isoproxy.

There was no formal requirement specification drafted for the isoproxy nor for the system as a whole. There were however some intrinsic restricting factors, such as a financial system with limited possibilities to interface with external systems, and an existing web-banking system, whom the new system were to functionally replace and extend beyond.

The previous web-banking solution was running solely on an expensive type of machine that did neither scale cheaply or allowed to be maintained cost-effectively. A web farm, as in the new design, is easy and inexpensive to scale. The widget-based CMS ensures that there no longer is any need to hire an expensive technical consultant to update the content of the website.

A discussion of the implications of these specific requirements and their intrinsic design problems and solutions will now follow.

5.2 Linking an unreliable messaging to a robust financial system

The server-side application was running a financial system that had two principal methods of interface to external systems. These were "direct" database modifications and transactions that used the ISO 8583 protocol.

Since it would be far to expensive to reinvent all the already well-tested procedures developed to control the incoming request and to ensure that

the state of the database, such as locking, indexing and so forth is sound, there was only one option left. Hence, we choose to communicate solely via the ISO 8583 protocol, and for consistency all previously unimplemented transactions no matter how trivial was implemented using the same protocol and control mechanisms too.

To make it possible for a web-application written in PHP to communicate with the server-side application the isoproxy was drafted and implemented as a proxy that would handle the translation as well as the protocol design incompatibilities between the two.

The translational part of the functionality was implemented using two libraries: `libxml2` an open-source XML-library and `isolib` a proprietary ISO 8583-library written by external developers. At the time there were no open-source equivalent to the `isolib`. I note that today there are three such ones available[18].

The version of the `isolib` library we used did however lack support for all data types in the ISO 8583 standard, including some that we needed, hence, we develop a patch to complete the set of functionality.

The most obvious incompatibility, except for the lack of functions to generate valid ISO 8583 messages in PHP, is the level of care possible to take in the messaging. The reversal actions in the ISO 8583 standard is an illuminating example.

In web-applications there is not much care taken to ensure that the server can be confident that the message really reached the end-user at the client correctly. Whereas this feature is essential in the classical applications for the financial-system.

Even considering all these problems, the raison d'être for the isoproxy is security. Moving the possibility to send ISO 8583 messages to the financial-system from a machine running the web services, to an extremely minimal appliance who may only send predefined types of messages, obviously raises the level of security considerably.

5.2.1 The state machine

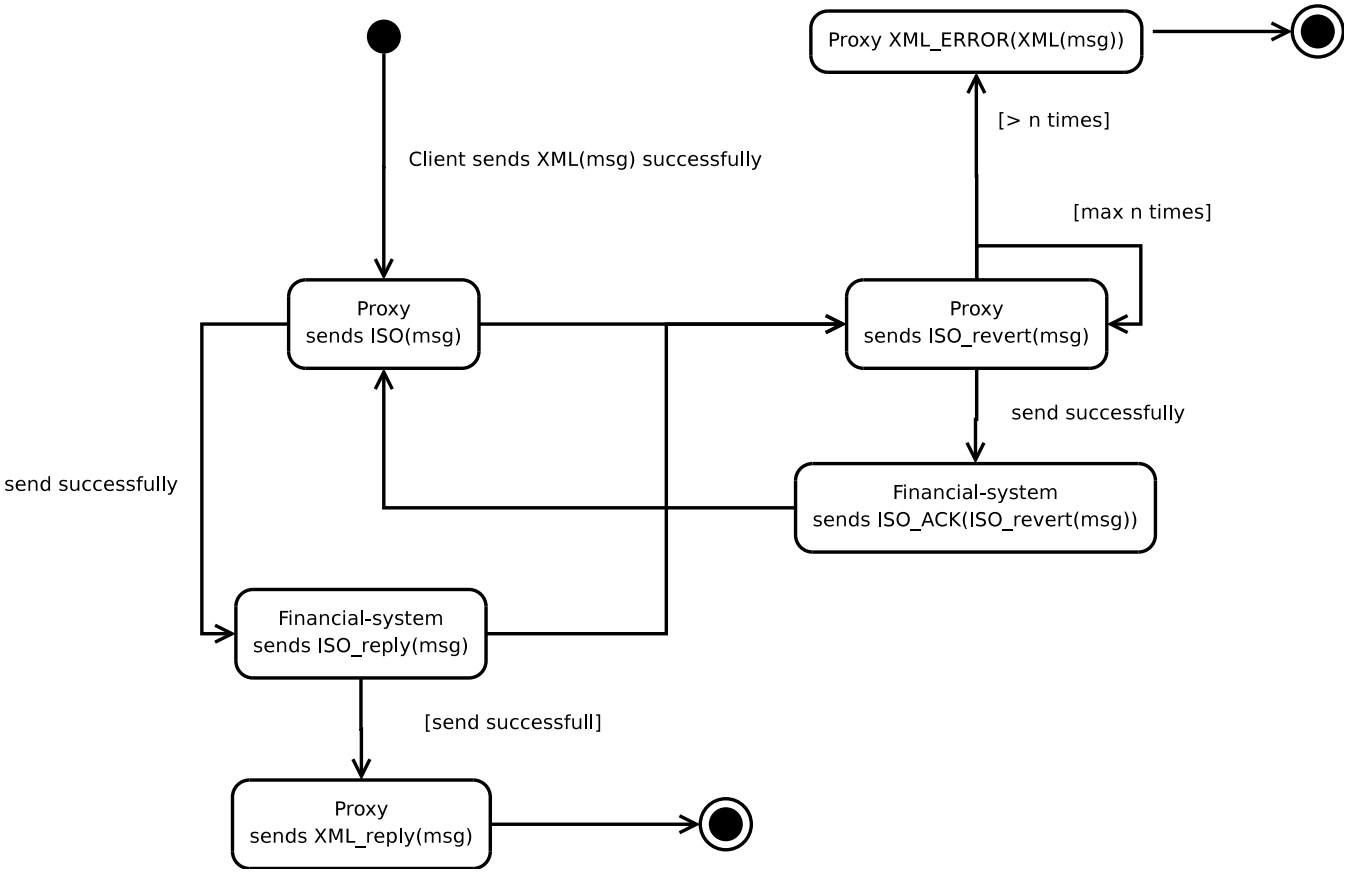
In order to give the isoproxy the reliable messaging that the server-side application assumes, the isoproxy requires more logic than just pure translation. This logic is mainly handled by a state machine that is instantiated per transaction as an initialising message is received from a client by the isoproxy.

If the exchange of messages runs without problems, and the reply is small enough to fit in a single ISO 8583 message, the state machine will not do anything exotic and the isoproxy will only conduct translation.

If however there is a problem in any part of the communication the isoproxy will enter a finite-loop trying to do a reversal of the transaction. More specifically it will send a reversal messages to the financial-system, and wait for the system to acknowledge, a timer is started and if the financial-system fails to respond within the window of time, it will try to resend. If the system has not acknowledged after a settable number of attempts the failure is verbosely logged and the transactional state machine will be cleaned up.

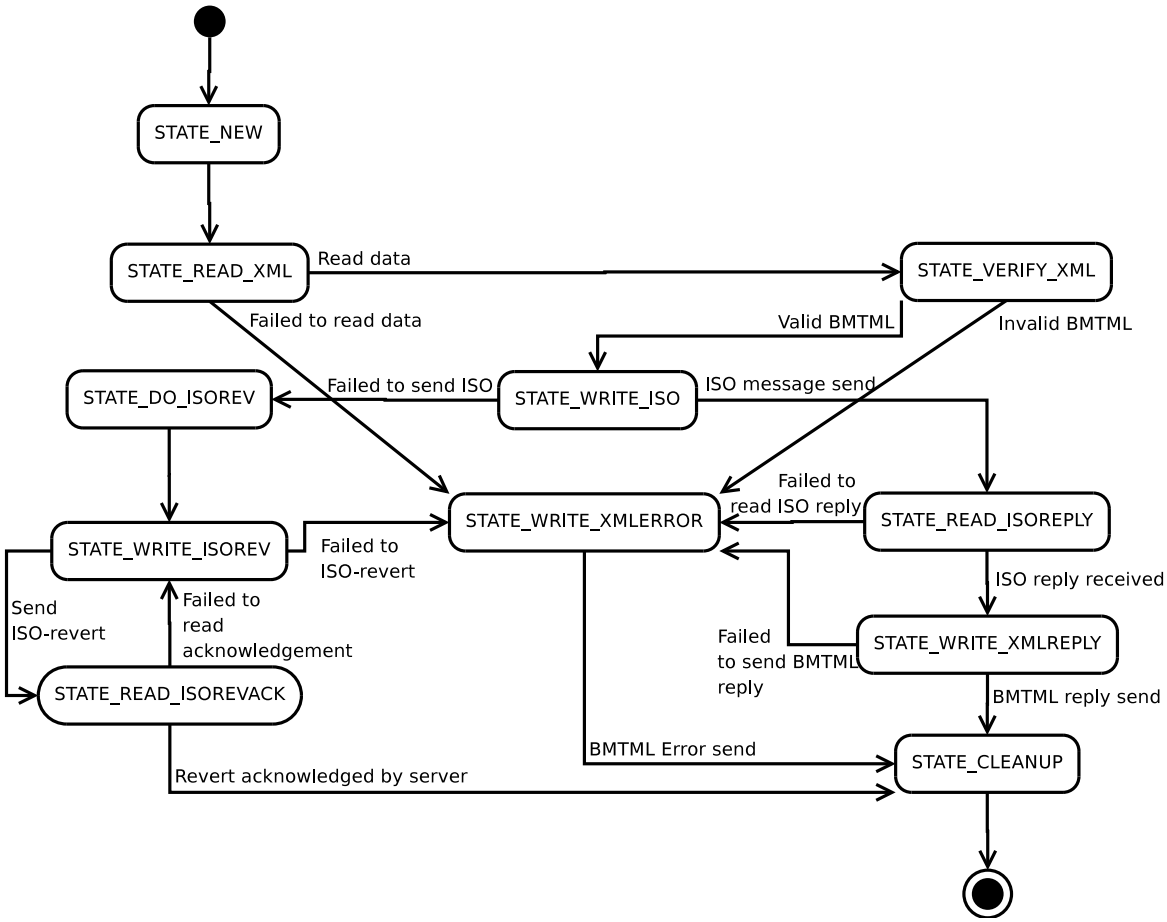
The BMTML messaging can handle arbitrarily sized messages, and this particular implementation handles potentially large database queries, that utilise this feature frequently. The isoproxy will in this case send multiple requests and collect all information before sending the information as a whole in BMTML to the client. This obviously needs support in the server-side application.

The figure below includes all the messaging patterns, normal, reversal, as well as multiple-message replies.



In order to get a more in-depth understanding of the isoproxy's functionality we will now take a closer look at the state machine and its states. The diagram below depicts the states and the possible transitions.

Every established client-connection allocates a structure, called the *transaction structure*. The structure contain a variable that is set to the current state, and various buffers and timers that are used in the different stages of communication. The structure is thereby the carrier of transactional data through state transitions.



STATE_NEW

Every newly allocated transaction structure will initialise its state variable to this state. As the transaction structure is allocated on the connection from the client the state machine will be initialised the not yet initialised

parts of the transaction structure and then perform a transition to the STATE_READ_XML state.

STATE_READ_XML

The entry action of this state is to register the interest to read data on the clients network socket, so that when ready the available data will be read. The first 32 bits read contains how much data are to be read, this is a connection specific to detail of the usage of TCP as the transport protocol. If it does not receive all data with a window of time it will time-out, fill the error buffer with an appropriate error message, and perform a transition to the STATE_WRITE_XMLERROR, and attempt to send an error BMTML message back to the client.

If the isoproxy could successfully read the BMTML-message send from the client it will store the data in a buffer allocated in the transactional structure and then perform a transition to the state STATE_VERIFY_XML.

STATE_VERIFY_XML

This state analyses the buffer allocated and filled in the STATE_READ_XML state using a specification in W3C's DTD and `libxml2`. If the XML is invalid it will fill the error buffer with an appropriate error message and perform a transition to STATE_WRITE_XMLERROR.

If the BMTML-message is a valid XML and more specifically a valid BMTML-message. That is, valid not only according to the DTD, but given the meta definitions received from the metaserver, it will perform a transition to the state STATE_WRITE_ISO.

STATE_WRITE_ISO

This state will now use an XPath expression and functions from the `libxml2` library to parse out the data needed to build the ISO 8583 message.

All the definitions of the available transactions has been read at the time of execution from the metaserver and are now applied together with the pin-pointed data to generate an ISO 8583 message.

Next, the state will register that the transaction would like to write to an available server socket (or in a non-persistent connection mode actually open one). It will now wait for a socket that is ready for writing, and send the message.

If the sending of the ISO 8583 message failed the state machine will perform a transition to the STATE_DO_REV. If the message was send successfully it will instead perform a transition to the STATE_READ_ISOREPLY.

STATE_DO_ISOREV

This state is functionally very similar to the STATE_WRITE_XMLERROR. That is, it will send a BMTML error message to the client. The error text send comes from the buffer that is set by the previous state that performed the transition to this one and hence indicates the problem that was experienced in the previous state. Next, the connection to the client will be closed. Unlike the STATE_WRITE_XMLERROR this state will perform a transition to STATE_WRITE_ISOREV instead of STATE_CLEANUP.

STATE_WRITE_ISOREV

In this state the client is no longer connected. The idea is that the transaction will try to do a revert. As previously described in the section that introduced ISO 8583, reverting a different message in ISO 8583 is done by sending the same message again, but with a message type indicator (MTI) that indicates a revert (e.g. 0400).

If it fails to send a revert it will time-out and retry a settable number of times. If it still fails after this number of attempts it will log the failure and then perform a transition to state STATE_WRITE_XMLERROR.

If the revert is send successfully it will instead perform a transition to the state STATE_READ_ISOREVACK.

STATE_READ_ISOREVACK

This state will register its interest to read from the transaction's server socket and will upon a successful read try to verify that what was received was an acknowledgement of the revert message sent in the STATE_WRITE_ISOREV state.

If the acknowledgement was read successfully a transition will be made to the state STATE_CLEANUP. If the read fails or times out it will increment the counter in STATE_WRITE_ISOREV on the number of attempts and then perform a transition to STATE_WRITE_ISOREV.

STATE_READ_ISOREPLY

The purpose of this state is, as expected, to read the reply from the server. The entry action is therefore to register the interest of reading from the transaction's server socket.

The data read once the socket is ready for reading will be stored in an allocated buffer in the transaction structure. If this operation time-out or the data is not valid the state machine will perform a transition to the state

STATE_WRITE_XMLERROR after setting an appropriate error message in the transaction's error message buffer.

If the reply however is read successfully a transition to the STATE_WRITE_XMLREPLY state will be performed.

STATE_WRITE_XMLREPLY

In this state the isoproxy will attempt to convert the reply stored in the buffer allocated and filled by the state STATE_READ_ISOREPLY into an BMTML-message using the transactional definitions received from the metaserver.

If the isoproxy fails to perform the conversion it will do a transition to the
STATE_WRITE_XMLERROR state. If it is successful with the conversion it will instead register its ambition to write the converted BMTML-reply to the client and wait for the client's socket to be ready for writing.

If the write fails it will again perform a transition to the STATE_WRITE_XMLERROR state, but if it is successful it will perform a transition to the STATE_CLEANUP state.

STATE_WRITE_XMLERROR

This state is functionally very similar to the STATE_DO_ISOREV. That is, it will send an BMTML error message to the client. The error text send comes from the buffer that is set by the previous state that performed the transition to this one and hence indicates the problem that was experienced in the previous state. Next, the connection to the client will be closed. Unlike the STATE_DO_ISOREV this state will perform a transition to STATE_CLEANUP instead of STATE_WRITE_ISOREV.

STATE_CLEANUP

This state does the de-allocation of all buffers previously used in the handling of the transaction. It also unregisters (or closes) the connections to client and server.

Finally it performs a transition to the final state STATE_DONE, which indicates to the event-loop's garbage collecting code that the whole transaction may be de-allocated.

5.2.2 Summary

There are in the messaging as already noticed two interesting cases where the messaging interaction done by the isoproxy goes beyond translation and where the number of BMTML-messages and ISO 8583-messages are not the same. The two loop cases already mentioned can now be identified with the loops between the states STATE_WRITE_ISOREV and STATE_READ_ISOREVACK for the reversal, and within the same state in the case of STATE_READ_ISOREPLY.

5.3 An web banking application that is easy to administrate

One of the main complaints that the bank had with their previous web-banking system was the cost of maintenance. They wanted the new application to incorporate tools that could enable bank clerks, with only minor training, to be able to update the less critical parts of the content and let in-house engineers to construct and maintain most of the more critical parts of their website.

The content management system (CMS) used was originally developed for another project, but was reused due to its sleek widget based design.

It was still a challenging task to integrate the web-banking functionality with the easy-to-use in-house widget based content-management system (CMS), and still allow it to be easy to use.

Some operations, such as the definition of new transactions, and how they will be rendered dynamically, is however controlled by the metaserver for security reasons.

In the first version of the system the integration of the transactions list of the available transactions for a given user was generated directly from the data retrieved from the metaserver and the server-side application's user data. The CMS then generates a menu using the information retrieved from the metaserver.

The list will contain links to self-generating pages with the corresponding forms, where caption, form definitions, help texts, etc. are all retrieved from the metaserver. A post of such a form will send a BMTML-message to the isoproxy. The isoproxy will then return an answer and refresh the browser in order to load a reply page.

This is a slightly more complex page if it is supposed to be dynamic and easily maintainable. The solution that was chosen was to have reserved page identifiers that will be associated with a form-reply and which is created and

edited with the ordinary CMS-tools. In order to include data from the reply a new concept needed to be introduced. As the reply message is in XML there is an existing standard to look up information in an message called XPath [11], we adopted and created an in-line expansion for XPath and it serves this purpose.

Given the following expression,

$${= `date` //`transaction`[@`name`]},$$

in the source of a page in the CMS it will expand the XML-element name using the pre-defined format `date`. There are a vast number of formats, for things like currencies, timestamps, escaped expressions, and so forth.

There are more complex expansions of expressions, such as,

$${= `table` //`transaction`[@`table`]},$$

this will not only consider the table XML-element, but all its sub-elements as well and include them in the rendering using the pre-defined format `table`, that uses the `TABLE, TH, TR` and `TD` tags in HTML[4].

Another problem particularly evident as we consider tables is how to handle large amount of data. A single ISO 8583 message may only carry a limited amount of data. The logic that handles this is in the isoproxy and hence hidden from the user of the CMS-components.

5.4 Connection pools with persistent connections

The engineer responsible for the server-side applications had only previous knowledge of a specific network stack implementation, and it had some major quirks. It would listen to a port in one process that never forks, this implies that the port will be busy for usage from other processes and on top of this the subsystem did not handle the closing and re-opening of a TCP/IP port gracefully which made it unfeasible to close a previously established connection.

The first problem of the implementation was side-stepped by having a number of instances of the program running on different ports on the server-side. As a work-around for the second issue, and the server-side's need for re-execution, we created what we call connection pools that gave the isoproxy support for persistent connections.

The connection pools is a structure of structures that kept track of the sockets and what transaction was currently using them. Hence a transaction

could reserve a socket from the pool for a certain read or write of data and then declare its usage as ended and free the socket for others usage.

One of the first thing that happens as the isoproxy starts executing in this mode is that it connects to n ports at the server-side application and allocates the structures for the pool. If the connection to the server-side application fails there is a very stubborn retry policy that will hammer the server every 10^{th} of a second.

5.5 High-performance

What the most efficient, scalable and maintainable design of an high-performance, high-concurrency network-application is, has been the subject of a lot of research and investigation.

The two fundamental design principles that are most commonly put back to back are the threaded and the event design. The isoproxy was implemented using an event-driven design.

This was however initially not the obvious choice and I will now give an overview of how the pros and cons of the two designs were weighted.

When threads are argued to be superior to an event-driven design, one of the most common arguments is the claim that events give an "obfuscated flow of execution" and what Adya et al. [1] called "stack ripping". That is, the separation (rip) of where in the code a call is made and the fact that it will not return immediately, and when it finally do it will be elsewhere in the code.

Another heads-up from threads advocates is the difficulty of handling exceptions in state machines. As the isoproxy was written in C, this was however never a problem[7] in this case. The issue of the so-called "stack ripping" and obfuscation of the flow is more of a matter of taste and experience, than actual facts. Personally I like the possibility to consider the flow of execution as a state machine diagram [2] and am not bothered by the so called "stack ripping".

von Behren, et al. [17] who set out to explain why events are a bad idea uses the above arguments to indicate that threads are superior to event-driven design. Prior to doing this they go through the things that are normally mentioned as reasons to prefer event-driven design. They mention five such topics which they try to deflate one by one, starting with performance. They agree that **performance** previously has been an issue for thread based applications. They however blame the quality of the old thread implementations and claim that performance issues "are not intrinsic properties of threads".

However, as John Outerhout points out in [10], event-driven designs do not need any expensive context-switches as threaded designs do—and hence, it has what I do consider to be an intrinsic property that hurts performance.

Next von Behren, et al. [17] take a look at the claim that threads have a limited **control-flow**. Here they do agree that threaded designs are limited in this aspect. They however suggest that the more complex control-flows that are possible in event-driven designs are to be avoided anyway because they lead to misunderstandings and an increased risk of race conditions.

I find the reasons for their suggestion weak, they say something about the complement of the six explicitly mentioned control-flow models and use only three implementations as reference for their findings.

The isoproxy does however only use control-flow models that are mentioned, so control-flow does not affect the scale in any specific direction.

Synchronisation is a regular suspect when it comes to finding bugs in threaded applications. The reason why synchronisation behaves well in an event-driven design is boiled down to its pre-emptive nature [1], hence the usage of pre-emptive threads on uni-processor architectures actually gives the same level of protection.

The **state management** in threaded application is another well-known weak spot of the threaded design. The design has to "face a trade-off between risking stack overflow and wasting virtual address space on large stacks" [17]. This could be solved by dynamic allocation of the stack as suggested by Adya et al. [1], but it does highlight a potential growth of the stack that does not have a corresponding phenomena in the event-driven case as it unwinds its calling functions immediately as they register a caller for a new event.

Finally von Behren, et al. [17] considers, **scheduling**, this is an important point and one of the reasons why the isoproxy did end up using an event-driven approach. In an event-driven design the scheduling of event-handlers are done on an application-level basis. It may therefore favour certain event-handlers over others. Here they refer to Lauer and Needhams empirical duality paper of the message respectively procedure approach to operative-system design[8]. The paper is very theoretical and the implementation details of threads in today's operative systems makes it unlikely to have any validity for real-world cases.

The major drawback that was identified with event-driven design from a perspective of the implementation of a proxy is however the *pre-emptive nature*, which implies that the particular application may not utilise multiple processors and therefore is harder to scale for performance.

This was not an important issue for the implementation of the isoproxy since it was to run on a dedicated uni-processor appliance-hardware. The alternative for scaling that we consider was to distribute the transactions over multiple such appliances and include active-active support which is relatively simple for this application.

If the nature of the project was different it is also worth noting that there are many strategies to make event-driven designed application span multiple processors [3].

One of the most cited papers when it comes to the implementation of highly concurrent network applications in GNU/Linux and the BSDs is "The C10k problem" [6]. It features an in-depth study of the various facilities available in GNU/Linux and the BSDs for usage in event-driven or threaded implementations. The isoproxy was implemented using non-blocking sockets and a `select()`-call. If the proxy were to be run on NetBSD or FreeBSD, a modification to support kqueue would be trivial.

Chapter 6

Conclusion

Most often when you find yourself listening to someone speaking about alternative systems, they have an interest in selling something. As they want to maximize their sales they often present a full replacement as the "optimal solution".

The reason why, is often phrased as, it would be "faster", "easier", "future-proof", "maintainable", "modular", and not least "modern".

All good things, but, and this a significant but, as a large number of old systems are very well-proved they have accumulated a special kind of worth, a history of function.

In the case of systems that handles huge sums of money, or even worse lifes, this is one of the most expensive features in a system. Today there are various tools to check for logical soundness in code, but it still do not make up for actual track history.

Many of these old systems are proprietary, and not at all what the sales person of the alternative calls "future-proof", but maybe it would not be a good idea to replace them just now.

The way we chose to contain an old system with a "modern" proxy given the old system's interfaces modularises the old system. This gives the opportunity to incorporate the module into a system that can be described with all the buzz-words mentioned above.

If the attribute "fast" no longer applies to the system as a whole the module constructed using the old-system one can replace this module, with a "modern" one, but that is when it is absolutely necessary, and not just to meet the requirements of a new "modern" system.

The implementation of a proxy like the isoproxy could easily be modularised and then have different functional modules to handle different systems and their particular set of quirks, and hence such a plug-in proxy could be truly cheap to tailor for a given system.

Appendix A

ISO-defined ISO 8583 fields

Data Element	Type	Usage
1	b 64	Bit Map Extended
2	n ..19	Primary account number (PAN)
3	n 6	Processing code
4	n 12	Amount, transaction
5	n 12	Amount, Settlement
6	n 12	Amount, cardholder billing
7	n 10	Transmission date & time
8	n 8	Amount, Cardholder billing fee
9	n 8	Conversion rate, Settlement
10	n 8	Conversion rate, cardholder billing
11	n 6	Systems trace audit number
12	n 6	Time, Local transaction
13	n 4	Date, Local transaction
14	n 4	Date, Expiration
15	n 4	Date, Settlement
16	n 4	Date, conversion
17	n 4	Date, capture
18	n 4	Merchant type
19	n 3	Acquiring institution country code
20	n 3	PAN Extended, country code
21	n 3	Forwarding institution. country code
22	n 3	Point of service entry mode
23	n 3	Application PAN number
24	n 3	Network International identifier
25	n 2	Point of service condition code
26	n 2	Point of service capture code

Data Element	Type	Usage
27	n 1	Authorising identification response length
28	n 8	Amount, transaction fee
29	n 8	Amount. settlement fee
30	n 8	Amount, transaction processing fee
31	n 8	Amount, settlement processing fee
32	n ..11	Acquiring institution identification code
33	n ..11	Forwarding institution identification code
34	n ..28	Primary account number, extended
35	z ..37	Track 2 data
36	n ...104	Track 3 data
37	an 12	Retrieval reference number
38	an 6	Authorisation identification response
39	an 2	Response code
40	an 3	Service restriction code
41	ans 8	Card acceptor terminal identification
42	ans 15	Card acceptor identification code
43	ans 40	Card acceptor name/location
44	an ..25	Additional response data
45	an ..76	Track 1 Data
46	an ...999	Additional data - ISO
47	an ...999	Additional data - National
48	an ...999	Additional data - Private
49	a 3	Currency code, transaction
50	an 3	Currency code, settlement
51	a 3	Currency code, cardholder billing
52	b 16	Personal Identification number data
53	n 18	Security related control information
54	an 120	Additional amounts
55	ans ...999	Reserved ISO
56	ans ...999	Reserved ISO
57	ans ...999	Reserved National
58	ans ...999	Reserved National
59	ans ...999	Reserved for national use
60	an .7	Advice/reason code (private reserved)
61	ans ...999	Reserved Private
62	ans ...999	Reserved Private
63	ans ...999	Reserved Private
64	b 16	Message authentication code (MAC)
65	b 16	Bit map, tertiary
66	n 1	Settlement code
67	n 2	Extended payment code

Data Element	Type	Usage
68	n 3	Receiving institution country code
69	n 3	Settlement institution county code
70	n 3	Network management Information code
71	n 4	Message number
72	ans ...999	Data record
73	n 6	Date, Action
74	n 10	Credits, number
75	n 10	Credits, reversal number
76	n 10	Debits, number
77	n 10	Debits, reversal number
78	n 10	Transfer number
79	n 10	Transfer, reversal number
80	n 10	Inquiries number
81	n 10	Authorisations, number
82	n 12	Credits, processing fee amount
83	n 12	Credits, transaction fee amount
84	n 12	Debits, processing fee amount
85	n 12	Debits, transaction fee amount
86	n 15	Credits, amount
87	n 15	Credits, reversal amount
88	n 15	Debits, amount
89	n 15	Debits, reversal amount
90	n 42	Original data elements
91	an 1	File update code
92	n 2	File security code
93	n 5	Response indicator
94	an 7	Service indicator
95	an 42	Replacement amounts
96	an 8	Message security code
97	n 16	Amount, net settlement
98	ans 25	Payee
99	n ..11	Settlement institution identification code
100	n ..11	Receiving institution identification code
101	ans 17	File name
102	ans ..28	Account identification 1
103	ans ..28	Account identification 2
104	ans ...100	Transaction description
105	ans ...999	Reserved for ISO use
106	ans ...999	Reserved for ISO use
107	ans ...999	Reserved for ISO use
108	ans ...999	Reserved for ISO use

Data Element	Type	Usage
109	ans ...999	Reserved for ISO use
110	ans ...999	Reserved for ISO use
111	ans ...999	Reserved for ISO use
112	ans ...999	Reserved for national use
113	n ..11	Authorising agent institution id code
114	ans ...999	Reserved for national use
115	ans ...999	Reserved for national use
116	ans ...999	Reserved for national use
117	ans ...999	Reserved for national use
118	ans ...999	Reserved for national use
119	ans ...999	Reserved for national use
120	ans ...999	Reserved for private use
121	ans ...999	Reserved for private use
122	ans ...999	Reserved for private use
123	ans ...999	Reserved for private use
124	ans ...255	Info Text
125	ans ..50	Network management information
126	ans .6	Issuer trace id
127	ans ...999	Reserved for private use
128	b 16	Message Authentication code

Appendix B

Examples of BMTML messages

Example of a transactional message, part of the registration phase:

```
<BMTML>
    <query mid="message-12312-232-2" cid="server-12">
        <transaction name="RegistrationPhase"
                tid="123125543-2007-01-03-13-41">
            <column name="PAN">6950000358229</column>
            <column name="Username">test</column>
            <column name="Password">secret</column>
            <column name="Seed">jlkjlkj</column>
            <column name="RIF">0011256572</column>
        </transaction>
    </query>
</BMTML>
```

Another example, an account balance enquiry:

```
<BMTML>
    <query mid="message-12312-244-5" cid="server-12">
        <transaction name="Saldo"
                tid="123643689-2007-03-02-00-41">
            <column name="PAN">6950000358229</column>
            <column name="Account">000114151</column>
        </transaction>
    </query>
</BMTML>
```

Appendix C

Examples of BMMML messages

Example of the most common BMMML query:

```
<BMMML>
    <query action="getAllMeta"
        mid="message-78592039-237"
        uid="server-12" />
</BMMML>
```

Example of how an answer may look like:

```
<BMMML>
  <result mid="message-78592039-237">
    <transaction name="getSaldo">
      <column name="cardNo"
          spec="2n"
          description="Card umerico"
          help="Porfavor..."
          optional="1">
        <column name="cardNoSUB"
            spec="2n"
            description="Card umerico SUB"
            help="Help SUB"
            optional="1">
          <column name="cardNoSUBSUB"
              spec="2n"
              description="Card umerico SUBSUB"
              help="Help SUBSUB"
              optional="1" />
        </column>
```

```
        </column>
      </transaction>
    </result>
</BMMML>
```

Bibliography

[1] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management, 2002. `http://research.microsoft.com/~adya/pubs/usenix2002-fibers.pdf`.

[2] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design.* Addison-Wesley Professional, 2005.

[3] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002. `http://citeseer.ist.psu.edu/525358.html`.

[4] Arnaud Le Hors, Ian Jacobs, and David Raggett. HTML 4.01 specification. W3C recommendation, W3C, December 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`.

[5] ISO 8583:1987. *ISO 8583:1987, Bank card originated messages – Interchange message specifications – Content for financial transactions.* ISO, Geneva, Switzerland, 1987.

[6] Dan Kegel. The C10k problem. Technical report, 2006. `http://www.kegel.com/c10k.html`.

[7] Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 1988.

[8] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979. `http://portal.acm.org/citation.cfm?id=850658`.

[9] Donald A. Lewine. *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1991.

[10] John Ousterhout. Why threads are a bad idea (for most purposes), January 1996. http://www.cs.ubc.ca/~norm/508/2007W1/ouster95threadsbad.pdf.

[11] Jérôme Siméon, Jonathan Robie, Don Chamberlin, Mary F. Fernández, Michael Kay, Scott Boag, and Anders Berglund. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[12] C. M. Sperberg-McQueen, Jean Paoli, Tim Bray, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (third edition). first edition of a recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[13] W. Richard Stevens. *Advanced programming in the UNIX environment.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[14] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[15] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[16] John Viega and Gary Mcgraw. *Building Secure Software: How to Avoid Security Problems the Right Way (Addison-Wesley Professional Computing Series).* Addison-Wesley Professional, October 2001.

[17] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea for high-concurrency servers, 2003. http://www.usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren_html/.

[18] Wikipedia. ISO 8583 — Wikipedia, the free encyclopedia, 2008. [Online; accessed 2008-06-13].