17

Charging for information services in ODP systems

M.R. Warner

Telecom Australia Research Laboratories, 770 Blackburn Rd, Clayton, Vic, 3168, Australia.

This paper describes a banking service which enables charging for information services in open distributed systems. A novel two phase payment protocol is proposed which overcomes many of the shortcomings of other distributed accounting systems. The banking service is specified using the ODP viewpoint languages. The behaviour of the banking service is formally specified in the information viewpoint using the Z notation, while the interface signatures are defined using ANSA's IDL.

Keyword Codes: H.1.m, H.3.5; H.4.0; K.6.m
Keywords: Charging; Accounting; Online Information Services; Viewpoint Specifications

1. INTRODUCTION

The development of ODP standards should allow a wide variety of information services to be provided over public networks, without the need for specialised hardware or software for the clients of such services. Individuals and organisations will make use of information services provided by others, thereby eliminating the need to duplicate these services. Charging for service usage in such a system will be of great importance, particularly in stimulating the market for such information services.

In theory, any user in an ODP system may provide information services to any other user by developing applications and publishing the interfaces to them. Therefore the number of potential clients and servers in the system is extremely large. Servers must be able to charge for service usage by any client in the system, most of whom will not be known to the server in advance.

2. DISTRIBUTED ACCOUNTING SYSTEMS

Distributed accounting systems have received comparatively little attention, perhaps reflecting the lack of emphasis given to management and operational aspects in experimental distributed systems. Two distributed systems that have included accounting functions are Amoeba [1,2] and the Distributed Academic Computing Network Operating System (DACNOS) [3]. These systems introduce the notion of a trusted banking service which maintains bank accounts for clients and servers in the system. Both meet most of the requirements for resource control in a distributed system; however, neither offers sufficient flexibility nor security to allow automatic charging in a commercial distributed system, particularly one which spans multiple administrative domains.

Both systems require the client to make funds available to the server before a service request is carried out, either by depositing funds into the server's account (Amoeba) or creating an

K. Raymond et al. (eds.), *Open Distributed Processing*

account from which the server can withdraw (DACNOS). Either way, there is nothing to prevent the server taking the money and running. The client can limit the potential loss by only depositing sufficient funds for a single transaction, although this increases the overheads since funds must be transferred before each transaction. Furthermore, if the price of an individual transaction is high, the client's potential loss will also be large.

Another problem occurs where the price of the service is not known in advance. If the client deposits too little the server must either trust the client to deposit additional funds, or withhold service until the funds have been transferred. If too much is deposited, there is no way for the client to retrieve the excess, since both systems prevent the client from withdrawing funds deposited into the server's account. Were this not the case, the client could defraud the server by depositing funds before the request and withdrawing them again before the transaction is completed.

Since the concept of service covers a prolonged binding with potentially many operation invocations, the combination of high price, which is not fixed *a priori* is the norm rather than the exception. Were separate payments made for individual invocations, the overhead associated with the charging system would become excessive.

Many of the problems associated with charging, particularly where the price is high and/or unknown before service is completed, can be overcome by introducing an intermediate stage to the payment process, analogous to the authorisation process used in credit card transactions. Essentially it provides a half-way point from which neither party can withdraw without the other's consent, although the transaction can be stalled, thereby ensuring that neither party obtains an advantage by dishonouring the initial terms. Stalled transactions can then be settled manually.

3. ENTERPRISE SPECIFICATION

The enterprise specification of an ODP function identifies the objects which participate in the provision and use of that function, defining the roles that the various objects can take with respect to the function, and the activities which they perform. Deontic relationships (ie. permission, obligation and forbiddance) should be specified between the different roles and activities. Some of these relationships will be generic, while others may be specific to a particular management policy.

3.1 Community rules

The purpose of the banking community is to allow servers to charge clients for the use of their services. The basic roles within a banking community are:

Purchaser: an object which is the client of the service being produced, and which instructs a bank to transfer funds from its account into the vendor's account to pay for the service.

Vendor: an object which is the owner of the resource providing the service, and has funds transferred into its account as payment for the service. The vendor may not necessarily provide the actual service.

Administrator: an object which manages (part of) the account space of the banking service.

Bank: an object which provides the banking service.

The purchaser and vendor roles are relative to a particular transaction. An individual object may act as both a purchaser and vendor, perhaps even simultaneously if the object is engaged in multiple transactions. The additional role of Account Holder may be identified as being a super-type of the Purchaser and Vendor roles. As the name suggests, this role includes any object which holds an account with the banking service, regardless of the role it takes with respect to particular transactions.

It is assumed that the objects which perform the administrator role for a bank are identified when the bank is created. These administrators then assign roles to different objects with respect to particular accounts.

3.2 Activity rules

The basic activity within the banking community is that of purchasing a service. In addition there will be a number of activities for managing the account space, including activities to create and delete accounts, deposit and withdraw funds, and to obtain information about the state of accounts. These will be not be addressed in this paper; however, they are described in [4].

Purchase Service may be initiated at any time by a purchaser, but relies on the co-operation of the vendor and the bank.

The purchase service activity is a composite activity which is performed by a number of different objects in different roles co-operating with each other. The composition of the purchase activity is defined by a contract between the purchaser, vendor and banking service. In general, the sub-activities which are required to implement the contract are:

Request Contract: The purchaser requests the formation of a contract by sending a signed authorisation request for an amount A to the vendor. Guidelines for choosing the value of A will be included in the service offer exported by the vendor to the trading service.

Check Contract: The vendor examines the contract to determine if A is sufficient. This will depend on the charging policy of the vendor, the level of trust between the vendor and purchaser and the expected cost of the vendor's service. If A is sufficient, the vendor may forward the authorisation request to the banking service in order to obtain an authorisation number to guarantee the creditworthiness of the purchaser. This indicates acceptance of the contract.

Authorise: The banking service checks the balance of the purchaser's account. If the purchaser has sufficient funds the banking service deducts the amount A, and issues an authorisation number to the vendor. Details of the outstanding authorisation are kept until payment is made. The receipt of an authorisation number obliges the vendor to provide service up to the value of the authorisation.

Request Service: The purchaser, after requesting the formation of the contract, may begin issuing service requests to the vendor.

Serve: After receiving an authorisation number from the banking service, the vendor must service the requests it receives from the purchaser, maintaining an account of the value of service provided so far. Service is provided up to (or exceeding) the value of the authorisation amount A, or until the purchaser requests that service be terminated.

Check Service: The purchaser checks the service results as they are returned to ensure that the service has been performed satisfactorily.

Bill: The vendor computes the final price *B*, and issues a bill for this amount to the purchaser.

Request Payment: If satisfied with the service, the purchaser must send a signed payment request for the amount *B* to the vendor.

Check Payment: The vendor checks that the details of the payment request, including the amount, are correct. It then forwards the payment request to the banking service, including the previously obtained authorisation number.

Payment: The banking service adds the initial amount *A* back to the purchaser's account, and transfers *B* to the vendor's account before confirming the successful payment to the vendor.

Any of the above sub-activities may fail, either as a result of an active decision by the entity performing that activity, or because of some equipment or other failure. Examples of the former are the vendor's decision that she can no longer provide service, or the banking service's refusal to provide an authorisation because the purchaser has insufficient funds in her account. Whenever a sub-activity does not terminate successfully, the entire sequence should be backtracked, reverting the system to its state prior to the initiation of the activity. This backtracking may not always be possible as the service may already have been performed, and requires the co-operation of all the entities involved in the transaction.

Where the failure was caused by a dispute an entity may refuse to backtrack, thereby causing the entire transaction to stall. In general, a stalled transaction will provide neither party with a gain, since the vendor will not receive payment, while the amount previously authorised will not be available to the purchaser. Stalled transactions may be identified by the banking service when outstanding authorisations reach a certain age. The procedure for recovering from stalled transactions cannot be completely automated as the parties involved may well dispute the terms of the original contract, as well as the actual fact of what service (if any) was provided. Where any automatic procedures fail to resolve such disputes they must be resolved by a human arbitrator (perhaps a magistrate) according to applicable policies and, perhaps, the common law of restitution.

3.3 Policies

The terms of the contract for purchasing a service will be determined by the following policies:

Charging Policy: The charging policy of the vendor will determine what value of authorisation *A* is acceptable. This policy must take into account the degree to which the purchaser is trusted. Where this level of trust is low, the vendor will demand that the value of the authorisation is high, perhaps exceeding the expected service price *B*. A high authorisation ensures that the purchaser will gain little from withholding payment, since the authorised amount will be unavailable until the dispute has been resolved. If the vendor trusts the purchaser then the charging policy may allow the authorisation phase to be omitted. In such cases the vendor accepts the risk of the purchaser defaulting on payment.

Payment Policy: The payment policy of the purchaser will determine whether a particular service offer is accepted. Both the amount demanded as an authorisation, and the expected

service price, must be considered when determining whether an offer will be accepted. If the purchaser trusts the vendor, the policy may allow for pre-payment for service. In this case the purchaser pays for the service before obtaining it, thereby accepting the risk of the vendor defaulting on the service.

Credit Policy: The credit policy of the bank will dictate whether or not the bank allows authorisation and/or payment operations whose value exceeds the purchaser's current balance.

Security Policy: The bank's security policy will dictate a method for generating signatures for authorisations and payments.

3.4 Security

The proposed system relies on a number of generic security services to prevent fraudulant transactions. These services include data origin authentication and integrity services which prevent an attacker (including the vendor) from modifying or replaying messages signed by the purchaser. Note that since payment requests from the purchaser are passed to the banking service via the vendor, a generic digital signature facility must be available in addition to link by link data origin authentication.

4. INFORMATION SPECIFICATION

The information specification defines the functionality of the banking service by specifying the information elements of which it consists, and the transformations that occur to these elements as a result of the activities described in the enterprise specification. The Z language [5] is used to provide a formal description of the information specification. Activities performed by the purchaser or vendor are not specified.

4.1 Information elements

We begin by introducing the primitive information elements which are needed to describe the banking service. Unique account numbers are used to distinguish between the many different accounts, and authorisation numbers are needed to identify particular authorisations. Time-stamps may be used to detect replays of transaction requests and to identify stalled transactions. The way in which these elements are represented is not considered in the information specification, but will be covered in the computational viewpoint. They are therefore introduced as primitive types:

$[AcctNum, AuthNum, TimeStamp]$

Money is represented by a natural number, which precludes negative balances. Fixed amounts of credit may still be incorporated into the system by depositing funds into an account before they have been paid for.

$Money ::= \mathbb{N}$

As described earlier the transaction protocol is based on the concept of authorisations. The details recorded about the outstanding authorisations must include the value of the authorisation, the time it was issued, and the account into which the payment will eventually be credited. An outstanding authorisation may be represented by a simple schema with the following three components:

Authorisation

$$\begin{array}{|l} value : Money \\ payee : AcctNum \\ date : TimeStamp \end{array}$$

Using these basic types, we may now define an account. An account will consist of the balance which is an amount of money. The set of outstanding authorisation numbers is maintained, and a function is defined which maps them onto the details of the authorisations.

Account

$$\begin{array}{|l} balance : Money \\ outstanding : \mathbb{P}\, AuthNum \\ auth : AuthNum \pfun Authorisation \\ \hline outstanding = \mathrm{dom}\ auth \end{array}$$

4.2 Dynamic schemata

Activities in the enterprise specification are associated with information transformations in the information specification. The following schemata define the transformations of a particular account. Note that in addition a number of initialisation and error checking schemata are required. These are omitted here for brevity and clarity. They may be found in [4].

We begin with the operations deposit and withdraw. These operations are used to define the payment operation. They are not normal banking operations (and therefore do not appear in the Teller interface in the computational specification) since money cannot be created or destroyed, but only transferred between accounts. Apart from the balance, the remaining components of the account should remain unchanged as a result of these schemata. A pre-condition of the withdrawal schema demands that the existing balance in each currency be no less than the amount to be withdrawn. In addition the enquiry operation which yields the state of an account may be defined.

Enquiry

$$\begin{array}{|l} \Xi Account \\ balance! : Money \\ \hline balance! = balance \end{array}$$

Deposit

$$\begin{array}{|l} \Delta Account \\ amount? : Money \\ \hline balance' = balance + amount? \\ outstanding' = outstanding \\ auth' = auth \end{array}$$

Withdraw

$$\begin{array}{|l} \Delta Account \\ amount? : Money \\ \hline balance \geq amount? \\ balance' = balance - amount? \\ outstanding' = outstanding \\ auth' = auth \end{array}$$

The authorisation schema must add the new authorisation to the set of outstanding authorisations. The only restriction placed on the authorisation number issued is that it is not already in use.

Authorise

$\Delta Account$
$amount? : Money$
$dest? : AcctNum$
$time? : TimeStamp$
$number! : AuthNum$

$balance \geq amount?$
$number! \notin outstanding$
$balance' = balance - amount?$
$outstanding' = outstanding \cup \{number!\}$
$auth' = auth \cup \{number \mapsto \langle\!| amount?; dest? ; time? |\!\rangle\}$

Two separate cases of payment may be distinguished depending on whether an authorisation has already been obtained. The simpler case of an unauthorised payment may be specified simply as a withdrawal and a deposit operation. Component renaming [5, page 18] is used to allow the same variable (*amount?*) to be used in the withdrawal and deposit schemata which have been decorated with subscripts to distinguish between the two different accounts on which they operate.

UnAuthPay

$Withdraw_1[amount?/amount?_1]$
$Deposit_2[amount?/amount?_2]$
$amount? : Money$
$number? : AuthNum$

$number? = null$

The authorised payment is more complex since the authorisation must be checked, the authorised amount (*value*) added back to the account, and finally the authorisation number must be removed from the list of outstanding authorisations. Due to these additional changes to the state of the source account, the authorised payment operation cannot conveniently be based on the withdrawal schema.

AuthPay

$\Delta Account_1$
$Deposit_2[amount?/amount?_2]$
$amount? : Money$
$dest? : AcctNum$
$number? : AuthNum$
$Authorisation$

$number? \in outstanding$
$number? \neq null$
$\Theta Authorisation = auth_1(number?)$
$dest? = payee$
$balance_1 + value \geq amount?$
$balance_1' = balance_1 + value - amount?$
$outstanding_1' = outstanding_1 - \{number?\}$
$auth_1' = auth_1 \lhd \{number?\}$

The two pay operations may now be combined into a single operation:

$Pay \triangleq AuthPay \wedge UnAuthPay$

4.3 Banking service

So far we have considered operations on particular accounts. The actual banking service will consist of a set of accounts referenced by their account numbers.

BankingService

$account\_space : \mathbb{P}\, AcctNum$

$bank : AcctNum \pfun Account$

$account\_space = \mathrm{dom}\ bank$

Again, initialisation and management schemata have been omitted here, but may be found in [4].

A promotion schema [7, pages 24-25] may be defined which allows us to refer to a particular account in the banking service, while leaving the remaining accounts unchanged.

ΦBankingService

$\Delta BankingService$

$src? : AcctNum$

$\Delta Account$

$src? \in account\_space$

$\theta Account = bank(src?)$

$bank' = bank \oplus \{src? \mapsto \theta Account'\}$

$account\_space' = account\_space$

Using this, the previously defined operations may be performed on the appropriate accounts within the banking service as a whole.

$BankAuthorise \triangleq \Phi BankingService \wedge Authorise$

$BankEnquiry \triangleq \Phi BankingService \wedge Enquiry$

The payment operation affects two accounts in the banking service. The promotion of these two accounts is incorporated with the actual operation to produce the following schema:

BankPay

$\Delta BankingService$

$src?, dest? : AcctNum$

$\Delta Account_1, \Delta Account_2$

Pay

$\{src?, dest?\} \subseteq account\_space$

$\theta Account_1, = bank(src?)$

$\theta Account_2 = bank(dest?)$

$bank' = bank \oplus \{src? \mapsto \theta Account_1', dest? \mapsto \theta Account_2\}$

$account\_space' = account\_space$

4.4 Error checking and security

As mentioned earlier, the error and security checking schemata have been omitted here. In [4] error checking is incorporated by defining special error reporting schemata which are combined with the operational schemata via schema conjunction. Security checking is incorporated in a similar way, although additional components must be added to the Account schema to describe digital signatures and an authorisation function. Again, more detail can be found in [4].

5. COMPUTATIONAL SPECIFICATION

The computational specification of the banking service defines the interfaces to which clients of the banking service bind. The banking service is offered via the teller interface. This interface is defined below using ANSA's Interface Definition Language (IDL)[8]. In addition, banks will provide an administration interface through which the account space can be managed[4].

As defined in the enterprise specification, all authorisation and payment requests from the purchaser are passed to the banking service via the vendor. Therefore only the vendor need form a binding to the banking service's teller interface during the course of a transaction. In addition the vendor will provide a service interface to which the purchaser will bind. This interface, or a separate accounting interface, will require a number of operations to exchange accounting information such as the signed requests. Although it may be possible to standardise these accounting operations, this is considered outside the scope of this work. Furthermore, since the purchaser and vendor must define a common service interface, it is reasonable for them to also define an appropriate accounting interface, particularly since its form may depend on the nature of the service provided.

5.1 Type definitions

We now define how the information elements identified in the information specification are represented using the basic types defined in the IDL. Only those information elements which are passed across interfaces need to be defined. The way in which accounts and authorisations are represented is an implementation issue.

Account numbers will be composed of two parts: one indicating the "branch" of the bank, and the other a unique account number within the branch. Cardinals are used to represent amounts of money since the information specification defined only positive sums of money. Since authorisation numbers are always used in the context of a particular account, a simple cardinal representation may be used. The null value used for unauthorised payments may be represented by the value zero. Finally the success or failure of an operation may be defined as an enumerated data type. If the IDL supported exceptions these could be used to report the failure of an operation.

```
BankTypes : INTERFACE =
BEGIN

   BranchName      : TYPE = STRING;
   Name            : TYPE = STRING;
```

```
   AccountNumber : TYPE = RECORD [branch : BranchName,
                                  number : CARDINAL];
   Money         : TYPE = CARDINAL;
   AuthNumber    : TYPE = CARDINAL;
   Report        : TYPE = {Ok, InvalidAuthorisation, PermissionDenied,
                           AccountOperational, InsufficientFunds,
                           UnknownAccount, AccountExists}
END.
```

In addition a number of types are required for security. These include a signature used to validate authorisation and payment requests. This could be constructed using a symmetric key algorithm such as DES to generate a message authentication code for the concatenation of the parameters in the operation. Under such a scheme, signatures would be represented by a 64 bit block, conveniently represented by an array of octets. A time stamp may be represented by a cardinal.

```
SecurityTypes : INTERFACE =
BEGIN
   Signature    : TYPE = ARRAY[8] OF OCTET;
   TimeStamp    : TYPE = CARDINAL;
END.
```

5.2 Teller interface

The teller interface is used by the vendor to initiate authorisations and payments after having received the appropriate request from the purchaser. The operations supported by this interface are enquiry, authorise and payment. Each of these operations requires a number of security related parameters indicating the identity of the user requesting the operation, a time stamp and a signature. Note that in the case of the authorise and payment operations, the identity and signature refer to the purchaser rather than the vendor who actually invokes the operation. Any security service between the vendor and bank may require additional parameters. Other parameters are as indicated in the information specification. These comprise a source account number, a destination (ie. the vendor's) account number and an amount for the authorise and payment operations, as well as an authorisation number for the payment operation. In addition to the results described in the information specification, a status result is returned, indicating success or the reason for a failure.

```
Teller : INTERFACE =
NEEDS BankTypes;
NEEDS SecurityTypes;
BEGIN
   Enquiry : OPERATION [account : AccountNumber,
                        id : Name,
                        time : TimeStamp,
                        sig : Signature]
             RETURNS [result : Report,
                      balance : Money];
   Authorise : OPERATION [account : AccountNumber,
                          dest : AccountNumber,
                          amount : Money,
                          id : Name,
                          time : TimeStamp,
                          sig : Signature]
               RETURNS [result : Report,
                        auth : AuthNumber];
   Payment : OPERATION [account : AccountNumber,
```

```
                    dest : AccountNumber,
                    amount : Money,
                    auth : AuthNumber,
                    id : Name,
                    time : TimeStamp,
                    sig : Signature]
            RETURNS [result : Report];
END.
```

5.3 Distribution of banking service

A single, centralised bank can provide all the functionality specified above; however, apart from in relatively small, isolated systems, such an implementation is not practical. In a large scale ODP environment, it is inevitable that a decentralised banking service must be provided by a number of co-operating banks. There are several reasons for this. Since the provision of a banking service for charging is likely to be profitable, it must be assumed that a number of different organisations will compete to provide the banking service. For account holders with one bank to have access to servers whose accounts are held with another bank, the two banks must co-operate in some way, yet they must be able to maintain their autonomy. The way in which multiple banks interact to provide the complete banking service is described in [4].

7. SUMMARY

This paper has provided the outline of a specification for a banking service which allows charging for information services. The banking service makes use of a novel two stage payment protocol, analogous to the authorisation and payment phases of a credit card transaction. The proposed system has a number of significant advantages over both Amoeba and DACNOS. These include:

- reduced risk of fraud for both client and server, particularly in cases where the price is high or is not known prior to service completion;
- no need to create a sub-account for every client-server pair;
- fewer steps to protocol when client uses a new service;
- fewer steps before service completion, thereby reducing delay;
- fewer communicating parties therefore reducing the binding and communications overheads.

Furthermore, by allowing the client to pay for larger amounts of service usage without the risk of fraud, the authorisation mechanism may reduce the overheads involved with the charging system in comparison with either Amoeba or DACNOS.

The banking service was specified using the ODP viewpoint languages. In particular, enterprise, information and computational specifications have been provided. As such this paper also provides an example of the structure and content of the viewpoint specifications. The Z notation is used to provide a formal specification of the behaviour of the banking service. The full specifications of the banking service (of which only a subset are included in this paper) have been formally verified for completeness and consistency[4]. Interface signatures are defined using ANSA's IDL.

An initial demonstration system has been implemented over ANSAware. This implementation allows authorisation and payment operations to be performed over an account space partitioned into an arbitrary number of banks. The banks interwork via the same teller interface used by their account holders. While the initial implementation only includes rudimentary security features, a more elaborate implementation based on DCE is being considered. This enables the use of DCE's secure RPC to provide additional security.

ACKNOWLEDGEMENTS

The permission of the Director, Telecom Australia Research Laboratories, to publish this paper is hereby acknowledged.

REFERENCES

1. Mullender, S.J., "Accounting and Resource Control", *Distributed Systems*, S.J. Mullender (ed), ACM Press, New York, 1989.
2. Mullender, S.J. and Tannenbaum, A.S., "Protection and Resource Control in Distributed Operating Systems", *Computer Networks*, Vol. 8, pp. 421-432, 1984.
3. Harter, G., and Geihs, K., "An Accounting Service for Heterogeneous Distributed Environments", *Proceedings of 8th International Conference on Distributed Computing Systems*, 1988.
4. Warner, M.R., "Charging and Resource Control in Open Distributed Systems", *PhD Thesis*, Cambridge University, 1993.
5. Spivey, J.M., *The Z Notation - A Reference Manual*, Prentice Hall International Series on Computer Science, 1989.
6. Hayes, Ian (ed.), *Specification Case Studies*, Prentice Hall International Series in Computer Science, 1987.
7. Macdonald, R., "Z Usage and Abusage", *Royal Signals and Radar Establishment Report 91003*, February, 1991.
8. Architecture Project Management Ltd, *ANSAware 4.1: Application Programming in ANSAware*, Document RM.102.02, February 1993.