MAC TR-128

THE COMPUTER UTILITY AS A MARKETPLACE FOR COMPUTER SERVICES

Robert M. Frankston

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

THE COMPUTER UTILITY AS A MARKETPLACE FOR COMPUTER SERVICES

by

Robert Matthias Frankston

Abstract

Computers are unique in their ability to be programmed for a wide variety of applications. This is in contrast with hardware dedicated to specific tasks such as the telephone system. Because of its flexibility, a computer system can support, concurrently, many diverse services that do not require dedicated hardware. Conversely, these services act to bring the capabilities of the computer to the consumer who might otherwise find the operational difficulty of running computer programs too formidable.

Since the computer is supporting many services which are sold to consumers it is natural to model the system as a marketplace for these services. Most contemporary computer systems are oriented towards users who run programs. The environment for services puts different requirements on the computer systems than do the needs of programmers, so as to permit all the participants in the market to make effective use of its facilities without requiring dedicated facilities and without interfering with each other. As with any marketplace, it must be convenient to do business within its framework.

The requirements of such a marketplace are not satisfied in contemporary computer systems. However, the marketplace can be evolved from some existing computer systems without fundamental changes. Presently the use of a computer requires considerable expertise on the part of the user. The evolution to a marketplace is necessary if the capabilities of computer systems are to be made more widely available than they are now.

THESIS SUPERVISOR: F. J. Corbató
TITLE: Professor of Electrical Engineering

Acknowledgements

I would like to thank Professor Corbató whose insights were invaluable in identifying many of the issues discussed in this thesis. I would also like to thank graduate students and staff at Project MAC who read and commented on earlier drafts of this thesis: Doug Wells, Doug Hunt, Austin Henderson, David Reed and Vic Voydock.

Table of Contents

CHAPTER ONE

INTRODUCTION

Computer systems have evolved from simple calculators to more sophisticated processors capable of running complex programs with large data bases, to information systems serving the diverse needs of many users. Despite the evolution of hardware and software in the past twenty five years, computers are still regarded by many as esoteric devices for solving well-defined problems. One approaches the computer with a problem, programs the machine (or uses an existing program), feeds in data and the computer prints the results. This mode of operation is being superseded by one in which the roles of the computer and the user are not as well-defined, where the user works with the computer to find the solution to a problem. Even the idea that one must necessarily have a specific problem in mind is being replaced by the view that a computer offers a collection of programs which one may call upon (through a command language) to perform tasks when requested. The tasks may be one of those traditionally associated with the computer -- compiling, statistical analysis and report generation -- or they may be used to communicate with one's bank, library or colleagues, to manage household finances, recipes,

scheduling, or shopping.

These latter applications are not common for a number of reasons. Some of these are technical -- we often do not have the knowledge to construct programs of the required sophistication. Many of the difficulties are not technical, but are the result of designing computer systems for running programs. But being able to run a program is not equivalent to being able to make effective use of a computer. The vast majority of computer users, as opposed to programmers, should not need to be concerned with the details of running the programs. By concentrating on the computer as a home for programs one misses the forest for the trees. This thesis will distinguish between the use of programs and the use of services on a computer system. The aim is to examine computer systems as bases for providing services.

The evolution of computers from programming systems to service systems is similar to the earlier evolution from the bare machines to those with the support of sophisticated operating systems with their associated file systems, libraries and other resource management facilities. The service environment provides the vendor with the facilities needed to enable him to present his

services to his users without requiring knowledge of the complexities of the computer system. The telephone system is an example of a complex system dedicated to a limited set of services. When placing a telephone call, a subscriber is using a very complicated communications network, but he need only state his request in terms of the service he is asking for. Normally this is done by simply "dialing" the number of the phone he wishes to be connected to. In addition to being shielded from the technical complexities of the phone system, he is also shielded from the financial complexities and the politics of dealing with multiple telephone companies. All he sees is a bill stated in terms of the calls that he made. A computer system has much in common with the telephone system in providing services to its subscribers. However, while the telephone system is organized around a limited set of services, the computer system is an environment in which many diverse services can be offered. As the telephone companies have found solutions to the problems of providing their specific services, the computer systems must provide an environment that permits solutions to be applied to a large variety of services.

To investigate the general problems of providing a variety of services within a computer system, this thesis

provides an enhanced machine. The most important characteristic of this environment is the availability of other services that may interact with each other. In fact, the computer system can be viewed as simply a collection of these services; some being basic, such as the computer hardware and file system, and others being more sophisticated.

A description of an environment for offering computer based services is incomplete without any discussion of the financial or resource management aspects of the environment. In the design of computer systems the problems of buying and selling computer services have largely been dealt with in an ad hoc fashion. In many projects such as the ARPA network, they have simply been deferred. But the viability of a computer system as a marketplace for services depends largely on the ability to transact the business of buying and selling the services. To date, much of the discussion of accounting for computer usage has been concerned with the allocation of costs for the computer system to its users and with the use of pricing structures to distribute the computer resources according to given pclicies. This is not, however, the aim of a financial system for a marketplace. In the marketplace, the problem is one of facilitating business transactions, not that of

devising appropriate pricing policies, except to note that prices should be based on measures meaningful to the user of a service. It is important to permit conventional buying and selling to be extended to the marketplace. There is another necessary aspect to these transactions -- the settlement of disputes. In typical computer systems there may be many parties to a dispute, the installation management, the programmers, and the user of the system. In a marketplace, each vendor can make use of many other services. Unless guidelines are established for assigning responsibility when service is not satisfactory, the market becomes unattractive to the user who does not enjoy the details of litigations.

To demonstrate the feasibility of the computer as a marketplace for services, an implementation will be described. Multics, a system developed cooperatively by MIT and Honeywell, will be used as the basis for such a system. Multics is appropriate because it was designed to serve as a basis for a computer utility. The implementation of the marketplace will serve to demonstrate techniques that can be applied to other computer systems. The principle extensions to Multics will be the creation an environment for services as opposed to programs including a financial system for conducting business in the marketplace.

Once the framework of the marketplace has been applied to a single computer system, it can be extended to a distributed computer system such as the ARPA Network. By applying the framework of the marketplace to the ARPA network, some of the practical problems encountered in trying to offer services via the network can be resolved.

CHAPTER TWO

SERVICES IN A COMPUTER ENVIRONMENT

The goals of a computer system as an environment for services and those of structured programming have much in common in that they both represent attempts to control the complexity of computer systems by decoupling the services or modules of the system. Each module is viewed in terms of its interfaces with other programs. In analysing one program the internal operation of the individual programs it calls upon is not important. Each program module is considered to be fully responsible for the service it performs. Though one module may make use of other services, the user of the module is insulated from these secondary services. While this approach has been taken in some large programming projects, the facilities required for properly protecting programs from unwanted interaction are inadequate in most computer systems.

The computer marketplace and structured programming share the basic goal of making more effective use of the computer system. The desire for modularity stems from a need to debug a large program by analysis of local

relationships between programs. For the marketplace, modularity means that each vendor takes responsibility for his service thereby relieving the user of concerns about the details of the operation of the service. The ability to assign responsibility is important in localizing bugs in programs. One can determine whether a bug is in one's own programs or not. If not, then the bug is either in another module used in which case, the person responsible for the module can be informed, or it is in the interface to the other module. In the marketplace, the clear assignment of responsibility is necessary for isolating deficiencies in services and for resolution of disputes.

If the computer is to serve as a basis for providing services, an adequate framework must be developed for the financial transactions associated with the buying and selling of services among the participants in the marketplace. More than simply making it possible to transact business, the marketplace must make the sale and purchase of its services very convenient. The financial system is, effectively, a system of resource allocation simplified by the use of a common measure (money) for exchanging resources. Unlike many systems for accounting for the use of computer resources, the financial system of the marketplace is not meant simply as a means of

allocating costs of the computer hardware, it must permit the assignment of value to resources created by vendors and embodied in the services provided.

The marketplace puts requirements on services other than those strictly dictated by structured programming. If services are to be sold to consumers (as opposed to specialists and sophisticated users) in a competitive market, it is important that the services be attractive to users. In addition to the characteristics of specific services, general factors can contribute to the attractiveness of services. The include:

- Convenience: What is required of the user to take advantage of the service? What must the user know outside of his particular application area?

- Completeness: Must the user access multiple services to accomplish his goal? Can he accomplish the goal through the use of a small number of services that can communicate in a common language?

- Relevance: Can the user "speak" to the service in terms meaningful to him, or must he learn some special purpose language? Is the service interface oriented toward the

user's problem or must the user make the problem match the services available?

- Safety: What risks must the user take in order to use the service? Is the service so powerful that a minor error can be catastrophic? Will the user be able to predict the effect of his request? Does the "principle of least surprise" apply -- does the behavior of the service conform to that which the user intuitively expects? What assurances does the use have that the service will be satisfactory? If the service is unsatisfactory, what appeals does the user have?

The aim of creating an environment for computer services is to simplify the offering of services. Some of the techniques have been mentioned above. The primary technique is the decoupling of services by requiring that the vendor of a service take complete responsibility for his service at the interface to the service. This is a prerequisite for the criteria listed above. The service is convenient if the user need not be aware of the details of implementing the service. By taking full responsibility for his services the vendor gives the user a well defined means of finding who to appeal to when any aspect of the service is unsatisfactory. The degree to which a user or vendor can

be liable in such a situation can be limited by contractual arrangement between the user and the vendor prior to accessing the service.

The financial system of the marketplace also supports these marketplace criteria and is discussed in the next chapter. The problems of developing standards and design criteria for service is a much broader problem than the design of the environment for services and is beyond the scope of this thesis.

The availability of many services, possibly in competition, is necessary to provide a "critical mass" of service wherein one is likely to find a service appropriate for a particular task or one can find a set of services suitable for the task. Services must also be available to be combined for the creation of more powerful services. The competition between services in the marketplace is a primary motivation for making services as attractive as possible. To provide an environment in which all services can be made available to the consumer in a uniform manner, it is necessary that the users and services share a common pcol of resources. This can be done by sharing a single computer system or by establishing protocols for communicating between services within a distributed

ccmputer system. In this latter case, the marketplace still appears to the user as a single shared computer system.

Since competing services are sharing a common computer system, it is necessary that they can be assured that no service can interfere with any other service either maliciously cr accidently. Interference can occur whan a database is altered, when a progras is stolen, or even when service is unavailable because one user is exhausting the capacity of the system. The elimination of unwanted interfaces must not prevent normal interactions between users and services. Both the vendors and the users should be able to control these interactions so that the vendor can selectively deny access to his service, so that the user can prevent the vendor having unauthorized access to the user's data or money, and so as to protect the privacy of the subscribers.

The computer system as a marketplace differs from the computer system as a programming environment. To a user not doing his own programming the system must provide an interface to its services. The purpose of this interface, which is similar to conventional command processors, is to simplify the accessing of services and to permit business transactions associated with these services to be made

conveniently. Conceptually the user can be viewed as accessing the services from a typewriter-like terminal, though specialized terminals and interfaces are also possible. It is not, however, reasonable for each service to have its own unique mode of access because the user would be required to learn the idiosyncracies of accessing each service. Thus the computer installation must provide a uniform access interface to services. The user of a service can also be another service so that the interfacing protocol must also be suitable for use by a program without human intervention.

CHAPTER THREE

THE FINANCIAL SYSTEM

As explained in the introduction, a viable financial system is key to the successful operation of the marketplace. The purpose of the financial system is to permit standard business practices to operate within the framework of the marketplace. This differs from the goal of many existing computer resource accounting systems as the emphasis is on the transactions between the subscribers to the computer system as opposed to recovering the costs of the computer system.

The primary requirement for the financial system is that users have confidence in it. Confidence requires not only that the system be secure and trustworthy, but also that the system be convenient and simple enough to use so that the user can be at ease with the system and not worry about being at the mercy of a system that he does not understand. The degree to which the user can trust the financial system depends, in part, on permitting the user to apply his existing experience with the conventional financial system to the new environment as well as on the correctness of the algorithms used and the integrity of the

computational methods used within the financial system and within the services offered.

It is inevitable that disputes will arise between the users and vendors of services. Unless disputes are resolved fairly, the participants do not have any recourse within the marketplace in the case of unsatisfactory service. If one must pay for a service even when it has not been provided satisfactorily, the risks involved in using services ma outweigh they potential benefits. Central to the resolution of disputes is the determination of who the responsible parties are, for if responsibility cannot be assigned no action can be taken to resolve a dispute. For the vendor, the recourse in a dispute may be to deny farther access to his service. The user may apply financial leverage and deny payment until the issues are resolved. The financial system must be able to protect the user's right to withhold the payments for services. In other words, the financial system is a servant of the participants; payments are not made and services are not provided without this authorization by the interested parties.

In day-to-day business transactions a number of conventions have been established to facilitate common

occurances such as the buying and selling of mechandise and to protect the interests of participants. When the marketplace for service is within a computer system the methods for financial transactions differ from conventional methods because the transactions do not involve physical assets or produce written records. It may be argued that one can continue to apply current practices of using written notes even though the services are within the computer system. This is analogous to receiving a separate bill for each telephone call one makes. The burden of handling the bills for many small transactions would exceed the advantages of such tangible records. In fact, in conventional transactions, the problem of processing many paper transactions is leading to the development of electronic fund transfer systems.

The advantages of a financial system designed for the marketplace is that it can greatly simplify the buying and selling of services. In designing such a system one must provide counterparts to traditional financial safeguards such as giving the subscriber control over his own assets (i.e. no payments without subscriber's permission) and adequate records to give the subscriber the equivalent of such conventional records as receipts, check stubs and itemized bills. Above all, of course, the financial system

In the marketplace's financial system, one pays for a service by transferring money from the user's account with the installation to the vendor's account. This transfer is really authorization to the installation manager to consider money formerly being held on behalf of one user to thereafter belong to another user. A record of the transaction is kept by the financial system and functions as a receipt. A bill can be considered the converse of a payment in that it is a request made by a vendor for a transfer of money from the user's account to the vendor's account. For simplicity, we define a standard transaction as consisting of a bill and it associated payment. In practice, the bill may not be paid, or there may not be an explicit bill associated with the payment.

As pointed out above, the burden of processing the bills from many services can be quite heavy. It is therefore necessary to minimize manual processing of the transactions. Most transactions are fairly standard and can be dealt with in a predictable, or automatic manner. It is only in the exceptional cases that intervention is necessary. Thus one must be able to specify what action is to be taken in the normal cases and what conditions are to be considered exceptional and thereby require human attention. Simple examples of automatic decisions are

default limits for spending and authorization to pay a bill when it becomes due.

If the specification of how to process each transaction is itself too complicated, the advantages of automatic processing of transactions is lost. The basic interface to the financial system should permit a simple specification of which transactions require manual processing. The user has the potential of programming his own more sophisticated screening of the transactions, but the specifications for payments associated with the standard transactions should be sufficient for most users.

A standard transaction record was described above as consisting of a request for payment (or a bill) and the corresponding payment. In addition to these two items, the transaction record contains a specification of conditions under which payment is to be automatically made. A simple specification includes the (future) date on which payment is to be made and the maximum amount of payment. This amount is normally the price the vendor quotes when the users accesses a service. The user can thus request a service and not have to consider the bill individually unless it exceeds his limits. If the service is not satisfactory, he has until the payment date he specified to

cancel payment.

Grouping transactions together into accounts provides the ability to consider classes of transactions. Rather than considering all payments to be made from one large account that the subscriber has with the installation, smaller accounts for specific purposes can be maintained. This has the advantage of permitting limits to be placed on each of the accounts so that an error will not threaten the user's total balance. Associated with each account are default values for transactions and total limits for the account. More importantly, the user can delegate authority for managing an account. A user can permit a trusted vendor access to an account to collect periodic payments. He can, however, still limit the total amount that the vendors with access to the account can withdraw. (Of course, this is against a background of contractual arrangements which may give the user legal rights to still reclaim his payments if the vendor has not satisfied the conditions of the contract). Another advantage of having multiple accounts is the structuring of the user's records so that he may more easily evaluate his financial position.

The specification of who can do what with a user's account must be formalized so that the user can delegate

some of the responsibility for managing the account without giving up full control. The ability to limit the amount of responsibility delegated to another subscriber simplifies the user's task of monitoring the use of his accounts. We can define five roles which a subscriber can have with respect to an account. The subscriber's access to an account depends on the roles he is permitted to play.

Owner — May specify a subscriber's access to the account.

Clerk — May examine the status of an account by listing the balance and any transactions associated with the account.

Paymaster — May authorize payments from the account.

Receiver — May accept payments to the account.

Requestor — May make requests for payments from the account.

Roles must also be defined with respect to transactions within accounts. A subscriber may be given the ability to change the amount requested (i.e. specify the amount of the bill), cancel the bill, authorize payment (assuming one has

appropriate access to the account with which the transaction is associated), or to simply examine the status of the transaction.

The system of accounts, transaction records and controlled access permits the user to simplify his own processing of his financial transactions by being able to delegate some responsibility for their management while still maintaining overall control. For example, even when a user delegates authority to make payment on his behalf, he can limit the total amount that can be spent by simply limiting the balance in the account to which he is giving access. Typically he may give access to accountants or others providing financial management services.

CHAPTER FOUR

AN IMPLEMENTATION OF THE MARKETPLACE

An important criterion in the design of the marketplace is feasibility. To demonstrate the feasibility of the requirements of the computer services and the framework for the marketplace an implementation will be presented. A detailed implementation requires that points be clarified that are vague in the general description, such as the exact method of requesting a service and the exact capabilities of the financial system. The Multics system developed cooperatively by the Massachusetts Institute of Technology (Project MAC) and Honeywell Information Systems will serve as the basis for the implementation. See Appendix A for a description of Multics.

Multics was designed to serve as the basis for a computer utility and thus comes closer to meeting the requirements of the marketplace than other systems generally available. Multics can be extended without great modification to serve as the basis for the marketplace. There are other systems that could also serve as basis for a utility. These include IBM's VM/370 and OS/VS2-2/TSO

systems and BBN's Tenex system.

In specifying an implementation one must first find representaticns for the elements of the model. Specifically, what are the representations for the installation manager, the subscribers, services and the financial system?

The installation manager takes on the responsibilities of the Multics system administrator. The subscribers are registered Multics users.

Services on Multics

A service is essentially any sellable commodity on the computer system. One may simply sell access to a data base by subscription and then provide the service by giving the user access to the data. More interesting are services which actively perform a function and must be represented as a program for the computer. The requirements on services include the following:

- One must always be able to determine responsibility. There must be no unadvertised side effects. Such side effects are typically the result of sharing resources

without clear assignment of responsibility.

- The subscribers must be protected from each other so that one would not normally be affected by the other's mistakes or attempts at penetration.

- The user should consider the service to be "black box" and just view it as something that takes requests and returns results.

- The representation should not prevent a service from using other services.

- The vendors of services should be able to control access to their services.

- Services should be able to operate asynchronously.

In Multics, these requirements can be met by running each subscriber's programs in separate processes. Each process is identified with the subscriber's principal identifier (or access name). Thus the identity of the subscriber running a given program can be determined at all times.

Conventions must be established for communicating with services. Since, ideally a service can be viewed as simply taking requests and returning a result, we can associate with each service a queue of requests. For each queue there is (at least) one process that reads the requests and performs the service. The details of the implementation of services as processes associated with queues is described in more detail in Appendix B.

Process creation and communication between processes via queues is not very expensive in Multics, costing approximately a second of processing time for each process created. For many services, however, this expense is too high. Instead of requiring its own process, a service may be a subprocedure in the user's process. There are a number of drawbacks to this approach:

- Since the procedure is being run in the user's process, it is difficult to determine who has responsibility for usage of resources.

- Since the service and the user's programs share a common memory (more precisely - a common address space), they can easily communicate via shared storage. This use of side effects can obscure the

interface and thus make it difficult to determine whether unsatisfactory service was the fault of the vendor

- The user must be willing to trust the vendor's program since it will run with the same access as the user himself and may thus perform actions in the user's name and without his agreement or knowledge. This is often referred to as the Trojan Horse problem.

- Because the user has access to examine the vendor's program he can easily make a copy of the program removing the calls to the financial system to avoid being charged for using the vendor's program by using only the modified copy.

There are some partial solutions to the problems raised by attempting to run a vendor's program within a user's process. The problem of billing for a program which may be tampered with can be reduced by selling the services of the program on a subscription basis rather than charging for each invocation.

The use of the rings of protection permits one party to run the other party's programs in a restricted

environment and thus remove the problem of mistrust in one direction. For example, if the vendor's service is provided as an inner ring procedure, the user may not circumvent the vendor's protection, but the vendor may still misuse the user's access privileges. Since it is very difficult to provide more than one inner ring procedure in a given process, this mechanism is of only limited utility. In his doctoral thesis, Schroeder proposes a mechanism for generalizing rings to permit the use of multiple mutually suspicious subsystems in a given process. While Schroeder's domains provide a viable means of providing some services, they have not been implemented and will therefore not be explored further in the description of the implementation within the context of the existing Multics system.

The Financial System on Multics

The financial system is a special service in that it is offered by the installation management and is used by nearly all services and thus must be efficient. The representation chosen is an inner ring (or privileged) procedure available in all processes. In Multics there are eight rings. The innermost are reserved for use by the system and installation and can can therefore be used in all processes even those that are already using multiple

rings. The use of an inner ring has the virtue of being simple and efficient. It also provides a secure financial system though it doesn't protect that subscriber against errors in programs in the financial system, a protection that is superfluous since, in general, one cannot protect against the installation management. Appendix C gives sample sections of the Multics Programmers' Manual Accounting System Supplement and can be referred to for more detailed description of the interfaces referred to below.

The entry points for programs calling the financial system can be divided into three main groups according to whether they deal primarily with accounts, transactions or services. For accounts, the operations consist of creation, examination, modification of access and termination. For transactions the operations are: creation, authorization, payment processing and requesting, examination and assignment to a service request. Service requests pass the specified parameters to the service. Such a request might also specify a transaction which the service may use for its bill. The transaction specifies an amount authorized for payment so that the payment can be made immediately if the vendors price is within the authorization range.

Accounts are collections of data representing a set of balances and relationships with other accounts. Associated with an account is a set of access privileges. The following items are maintained for each account:

1) balance This is a set of pairs of amounts and units representing the balance on deposit with the installation manager.

2) payables This is a list of transactions representing payments from this account to another account. They may be pending or processed.

3) receivables This is a list of transactions representing payment to this account from another account. The transactions may be pending or processed.

4) access list This is a list of subscribers names and the access that each has to the account. The access may be:

owner The owner may set other's access to the account.

paymaster The paymaster may authorize payments from

	the account and change existing authorizations if payment has not yet been made.
requestcr	A user who has access to request payment may create transactions to transfer amounts from this account to another account to which he has receiver access.
receiver	Access to transfer payment to this account.
clerk	A clerk may list the status of an account.

A copy of each transaction record is kept with both the account to which and the account from which payment is to be made. It can be retrieved by using the name of either of these accounts and its unique identifier. The transaction record consists of a bill and its associated payment.

When creating a transaction, one specifies a vendor's account, a user's account and an intelligible (i.e. Enlglish) description of the transaction. The vendor may then specify the amount of the payment he is requesting. He may change his amount so long as the transaction is open. He freezes the requested amount when he closes the transaction. The user specifies the amount to be paid to

[illegible]

requested, the amount authorized and the remaining balance in the account for the given units. Payment will only be made if the transaction is closed. If the account is open on the specified date, the transfer will be made when it is closed. Once this date is passed, the authorization amount cannot be decreased because the money has already been transferred.

6) amount paid — The amount paid sofar. This represents payment which has been transferred.

7) description — An intelligible description of the purpose of the transaction.

8) whether open — A status flag. As long as the account is open, the amount of payment requested may be modified.

9) access — The name of the subscriber who may change the amount of payment requested.

The low level entries into the financial system include entries to create accounts, create transactions, specify access, specify quantities for payments and requests, and reading the contents of transactions and accounts. These are described in appendix C. Appendix C also includes the entries used to request services since the financial system acts as the intermediary between services.

It isn't sufficient to specify the interfaces to the financial system, one must also be able to make effective use of them. The aim of the financial system is to do more than permit transfers of money between users. It is also a means of providing the subscribers to the marketplace with control over their financial dealings. In an environment where there are many transactions being made, effective control means not only that the subscriber be able to decide the fate of each transaction, but also that he is not swamped with details that would render him unable to deal effectively with important transactions. Some examples of simple transactions:

Case one: a user purchases a service and pays in advance. This is similar to a conventional cash purchase except that a record is made of the transaction. The user

asks the financial system to make a payment to the vendor's account. The user then sends the service request to the vendor along with the identifier of the transaction.

Case two: a user purchases a service but does not pay immediately. The user may simply pass parameters to the service and the service will in turn make a request for payment to the user's account. In order to do so, the user must give the vendor access to make payment requests and then must consider the request to decide whether to make the payment. A simpler procedure is for the user to create a transaction and then give the vendor access only to the particular transaction. If the user expects the service to be satisfactory, he may specify a maximum payment for the transaction and a date on which the payment is to be automatically made. Thus, unless the service is unsatisfactory or the bill is higher than expected, the user need not be bothered with deciding on payment for the transaction.

Case three: A user wishes to pay a group of vendors automatically but limit the total payment. This is useful in paying utility bills such as gas and electricity. The user can create an account for this purpose and give it a balance equivalent to the maximum allowable payments. He

can then give the appropriate vendors access to make payments to themselves and does not need to check transactions individually.

To make most effective use of the marketplace, higher level routines than the interface specified in the appendix would be necessary. These interfaces would take care of many of the details associated with standard transactions. For example, instead of explicitly specifying a date for each transactions, the routines would normally supply a default date such as the end of the user's next billing period. Protocols for more complicated arrangements such as the use of credit from a third party must also be developed.

Multics as a service system.

The standard user interface to Multics is a command processor that is used to invoke programs. To the user of the marketplace this is unsuitable and a more service oriented interface is necessary. In using the service interface, the user need only specify the services he wishes to use and the parameters for the service. This interface also includes the higher level accounting aids mentioned above. In practice this interface might also be

used in conjunction with special consumer (as opposed to programmer) oriented terminals.

To illustrate the characteristics of a consumer's interface to the marketplace, a script will be presented. The purpose of this script is to present the reader with an overview of the system as described so far; the example is not meant to demonstrate an ideal interface. For simplicity in presenting a script, a typewriter like terminal will be assumed for the scenario below. The scenario is very limited in that it does not explore the modes of user interaction that might make the use of service much more convenient, but is meant as a small example of the use of a service through the system.

The user's typing is underlined and the computer system's output is not underlined. Commentary on the scenario follows "//".

Welcome to the Scenario Marketplace
Date and time: January 1, 1975 11:00 am edt
Default account: "Misc Bills"

// Most transactions have much information in common which need not be specified explicitly each time.

Default Payment date: 2 months (March 1, 1975)

// The date on which payment will normally be made if no further action is taken. The amount of the payment will be restricted to the default limits.

Request: account summary

// This is a service which helps manage the user's accounts.

Bills paid recently:

// Amount	Service	Description
$50.00	Newsmagazine	1975 Subscription
$18.97	Electric Co	November 1974 Service
$ 2.34	Marketplace	Service charge November 1974

Bills maturing soon:

// These are printed as reminders to the subscriber so that he can cancel payment for those services that were unsatisfactory or are incomplete.

$34.32	Reference Library	January 3, 1975
$32.42	General Programming	January 4, 1975

No receivables overdue.

// To keep track of payments due to the subscriber in his role as a vendor.

Request: read mail

No mail.

// A typical use of the marketplace is communication with other subscribers. This is an example of a service which is small enough so that the user is rarely concerned with managing the payments directly but instead sets a predefined limit and does not request any notification at all of the transaction.

Request: FoodMarket

// Shopping that does not require actual handling of goods can be done via the computer system if it is convenient enough. In this example, some explicit interaction with the financial system must take place since the amounts of money involved exceed the user's default limits. For the purposes of this scenario, lines beginning with "*" are commands to the financial system as opposed to the service.

Do you wish to change your standard shopping list? yes
change, add or delete? change
item? milk 4 liters

// Remember that the purpose of this scenario is not to demonstrate an ideal interface, but just to demonstrate how one might deal with services.

Warning: Default authorization is: $32.00,
Cost of items is: $34.39
Transaction is "groceries:01.01.73.134"

*authorize groceries:01.01.73.134 $34.39

// User authorizes the increased amount. This does not necessarily authorize immediate payment. In fact, the user is able to change the amount at any time until the money is actually transferred to the vendor's account, but chooses to specify the amount now so that he will not need to examine the transaction again. (The "*" is used in the example to mean that the command is directed to the financial system, rather than the servite be accessed.)

Request:

Since the financial system is so central to the marketplace, it must be very reliable. It must also provide convenient facilities for subscribers to monitor their own use of the market. In addition to the file system integrity features of Multics, records of transactions are kept in duplicate, one copy with the account from which payment is being made and one with the account receiving payment. In each record is sufficient information to locate the other copy of the transaction. Since the two copies are on opposite sides (i.e. credit and debit) of the account, the

total amount of any measure is constant in the system, thus providing a checksum for the financial system. A journal can also be maintained of all events effecting the financial system. This journal is important for updating accounts from earlier version in the case of catastopic failure and for resolving disparities between transaction records in two accounts.

The user is safeguarded by the availability of records of all his transactions. To be intelligible, they all contain description of the transactions. Since all the transactions are online, the user may easily use programs for this management process. For example, he can summarize his payments by categories cr other criteria. Having all the records available is a mixed blessing since unauthorized disclosure can be a serious invasion of privacy. Multics offers the technical means of protecting the records, but legal protections must also be provided for misuse of the records by the installation manager and by others who may have access to read them.

CHAPTER FIVE

DISTRIBUTED SYSTEMS

A network of computer systems linked via communication lines can be considered as a single distributed system. Lesser degrees of distribution are also possible, ranging from a multiprogramming system where the processing is distributed by the software over a number of logical processes, to separate computer systems sharing storage facilities, to separate computer systems with autonomous administrators who cooperate through a small set of well defined protocols. So far we have considered only the first case, a single multiprogramming (and possibly multiprocessing) system that is viewed as a homogeneous environment for computer services.

To the user requesting a service, it is not important where the program for the service is being run, as long as the interface to the service is well defined and the parameters can be passed without complications. Before considering the ramifications of extending the marketplace to a distributed system, we must first examine the reasons for having a distributed system.

To the consumer, it is inconvenient to use multiple markets. The computer system can be compared with the telephone system. A telephone subscriber doesn't purchase services directly from each phone company. Instead he subscribes to one phone company which provides him with access to the subscribers of the other phone companies. To the subscriber, the telephone system appears to be one large network without the complications of many interconnections and differing tariffs. In the same manner, the subscribers local market can serve as his entry into the larger system of marketplaces.

Unlike two subscribers of a telephone system, the communication between a user and a vendor is not restricted to a single set of conventions. In addition to parameters explicitly presented by the user to the vendor and results explicitly returned as messages, the two parties may communicate via shared resources outside the scope of the restricted protocol for accessing services. For example, the user may present data to the vendor by passing a memory address to the vendor. The vendor would then access the data directly in this shared memory. While this may be a simple operation within the confines of a single computer system, it is more difficult to extend this mode of communication to a distributed system. For the purposes of

discussion, we will restrict communication between the user and the vendor to an access protocol wherein the user passes messages to the vendor via a queue of requests for the vendor's service and the vendor replies by sending messages to the user via a similar queue used for replies. It should be noted that the name of a service may be passed as a message.

As in a single system marketplace, financial services must be provided. As long as one can provide a secure environment, interfaces similar to those specified in chapter four can be provided for a common financial system. When there are autonomous administrators, the approach of having a single financial system cannot be used since the participating installations are mutually suspicious. The relationships between the installations is analogous to that between services at a single installation. Each installation maintains an account representing its status with respect to each other installation. Periodically the installations must make actual cash payments to resolve imbalances.

When a user authorizes payment to the vendor, he is telling his installation to authorize payment by the vendor's installation to the vendor. If the vendor's

installation makes this payment it is, in effect, trusting the user's installation to make the payment when the accounts are resolved. This requires that each pair of installations establish a credit relationship. The number of such relationships grows with the square of the number of installations. The maintainance of many such relationships is cumbersome. To reduce the number of relationships, an intermediary can be introduced as a clearinghouse for the transactions between installations. This clearinghouse is fulfilling similar functions to the installation manager at a centralized marketplace, with its associated benefits. Just as the presence of a marketplace financial system does not preclude special arrangements between vendors, use of the clearinghouse is voluntary, but is a requirement for keeping dealings with other installations manageable.

The ARPA Network is an example of a distributed computer system. It is composed of autonomous computer systems linked together through a message switched communication system. The importance of the network is in its attempt to define protocols to permit the cooperation among users of these diverse computer systems. These protocols have so far been mainly concerned with the use of resources at one computer installation by users at another

installation. The network has been used by some subscribers to purchase services from remote installations.

In order for a user to access a resource on a computer system he must first make arrangements to pay for the use of these resources beforehand. This requires that he either have an account with that computer installation or that the installation provide a common account for network users without charging each one individually. For many users the difficulties of making special arrangements outweigh the advantages to be gained from such sharing. It is also difficult for a service to make transparent use of the network, i.e. if a user calls upon a program at his local installation, the program cannot make use of a service at a remote installation without the user having made prior arrangements -- the user cannot treat the service as a "black box" but must be aware of the services it invokes in turn.

A financial protocol can be established to facilitate the use of and payment for resources at remote installations. Each installation on the network is, in effect, a marketplace offering services to its subscribers. The financial protocol permits financial systems at two installations to request and authorize payments at the

other installation on behalf of their subscribers. The protocol provides both installations with capabilities equivalent to those provided to the subscribers within a single installations. The messages passed between the installations correspond to the financial system entries described in appendix C. The account and service names would, of course, be expanded to include the identification of the installations at which the name is defined.

CHAPTER SIX

CONCLUSIONS AND COMMENTS

The aim of this thesis has been to explore some of the difficulties that have inhibited the growth of computer-based services and the converse problem of how to make the capabilities of computers available to users. The solution to the problem of making more effective use of computers involves a combination of computer technology and human engineering. The thesis thus addresses both requirements for a consumer oriented computer system and for the technology needed for its implementation.

Supporting services involves more than just writing computer programs to perform specific applications. The services must be provided within a context that permits the user of the services to be unencumbered by the complexities of computer systems without giving up the conveniences that he is accustomed to with more conventional services. It is important that he have confidence in the services and in the marketplace. This confidence depends on a user's ability to predict the effects of his actions and on his ability to protect what he considers to be his rights or at least to limit his risks.

For the vendor, the marketplace must provide an incentive for creating services. The marketplace, by permitting the vendors to sell services for a profit, supplies some of this incentive. Competition within the marketplace helps ensure that the services sold will be attractive to users.

Disputes will inevitably arise and must be easy to resolve in common cases. Fundamental to this resolution is the assignment of responsibility. A vendor must take complete responsibility for his service both to provide a simple (and thereby attractive) interface and to give the user someone who can be held accountable for unsatisfactory service, even if the ultimate cause of the problem may be a secondary vendor who supplies a basic service to the primary vendor.

Basic to a free market is a trustworthy financial system. Unless the subscriber can have confidence in the financial system, he cannot make effective use of the marketplace. As stated above, confidence requires predictability and safety. By building upon the user's experience with conventional business transactions, the financial system of the marketplace is intended to allow him to have enough of an intuitive knowledge of the

financial system so as to be able to predict the effects of his actions. To protect his rights, a user must be able to selectively deny payments to vendors as a way of protesting unsatisfactory service. But it isn't enough for it to be possible for the user to exercise control, it must also be easy to do sc. To prevent the user from being swamped by a large number of transactions, the financial system provides capabilities that permit him to automate his financial management and restrict his attention only to the exceptional situations.

The marketplace, as described in this thesis, is feasible to implement within present-day technology. To demonstrate this feasibility, an implementation has been described based on Honeywell's Multics system. While changes are required to the system, these are relatively minor and do not affect the basic structure of Multics. The changes need not even interfere with existing methods of using the system. The implementation has been extended to a distributed system since a large marketplace would exceed the capabilities of a single computer installation.

The thesis has, by necessity, been limited to those topics immediately relevent to the implementation of the marketplace. Many of the topics that were mentioned were

explored only to the degree necessary to establish that the difficulties they presented were surmountable, but the solutions were examined in only a cursory fashion. The prime omission has been a full discussion of the interface to the consumer except where used in examples and where used to define basic requirements for the low level interfaces. The omission has been intentional because our understanding of the relevant issues has been limited by the currently available environment for such services. Given the environment of the marketplace, research in the design of the services themselves can be carried out.

Other topics omitted include the legal aspects of the marketplace, the detailed implementation of the financial system, and the limitations of a system such as Multics as a basis for a marketplace. The legal questions faced in the marketplace involve the problems of rules governing the market as a whole -- should it be regulated, what constraints are there on the installation management; and the problems of relationships between users -- what contractual arrangements are appropriate, what implicit or common law obligations do the parties have? The financial system must be very robust (i.e. immune to catastophe), even though the environment in which it runs may not be very reliable. Multics has been designed as a basis for a

computer utility but suffers from some of the same errors of omission as this thesis. For example, is the system sufficiently reliable; is the security system appropriate for the "real world" when the value of breeching it is high?

Beyond the detailed issues of the marketplace are the questions of the implication of the marketplace as a microcosm of an economy and the relationship between the marketplace and society as a whole. What are the effects of mapping existing institutions such as banks, credit bureaus, insurance, communications systems (including the post office), etc, into a computer system's marketplace. Some of the problems will be intrinsic to any marketplace, but others will arise from the nature of the computer system. For example, the consumer would not be constrained by geographic considerations in the selection of services. He will also be able to use the computer as an intermediary between himself and the services he is using. He may, for example, have a program which gives him a personal record of each transaction with a given service. An important consequence of the low capital necessary to create a new service is the opening of the marketplace to hordes of small entrepreneurs. What are the advantages of such a flea market? What are the disadvantages? What regulations are

appropriate?

The social effects of such a marketplace are hard to foresee at this time. We can only speculate. It is possible that such a marketplace can become a significant social factor since it provides a very good environment for many services especially those that involve communication between people where some intelligent processing is necessary. The current system of banks and credit agencies is an example. The marketplace is also suitable for services that require simple, but personalized processing such as catalog shopping or library services. Both of these also rely on the ability to communicate with a shared pool of information. The effects of lessening geographic and national restrictions on information sharing must not be overlooked. This is an extension of the information flow made possible by telephones and airplanes. Finally, what are the implications of globally available technical expertise, as supplied by computer based services?

The marketplace model of the computer system is an idea whose time has come. The costs of computer hardware are decreasing so that the necessary computational power is becoming available. In fact, the major cost of computer hardware today is the overhead of new research and support

personel. As more hardware is sold, this cost can be spread thinner. Software represents a large portion of the expense of a computer system and the expertise for developing computer services is a scarce resource. The marketplace framework eases the burden of development of computer services since it permits the development to be decentralized among individual entrepreneurs, although although the availability of the marketplace will not cause the problems of implementing service to disappear.

APPENDIX A

Description of Multics

Multics, as a system, performs two main functions: it distributes computer resources among the users of the system and it provides an environment for running programs. The resources consist of the hardware and the software. The hardware is made up of processors, memory, mass storage, communications facilities and other peripherals. Minimal distinction is made between the system software and the user software. The system consists of a privileged supervisor which supports the virtual memory environment and manages resources, and a non-privileged portion.

The running programs are organized into processes. A process is equivalent to a job or a task in common operating systems. It is defined by an execution point (i.e. a location counter) and a mapping of names in the storage system to addresses in the processes memory. The address itself consists of two parts, a segment number and an offset within the segment. A segment is, as its name implies, a piece of memory and generally corresponds to a single program or a data file; Multics makes no distinction between the two. Two processes referencing the same segment

may know the segment by different numbers, but will reference the same data; if one process changes a word in the segment, the other processes will be able to read the new value immediately.

The storage system corresponds to more traditional file systems. The important difference is that once a segment in the storage system has been initiated ("opened") by a process (either explicitly or implicitly) it becomes part of the directly addressable memory of the process. The storage system consists of segments, and catalogs of segments called directories. A directory may also contain other directories and thus the storage system forms a tree structure with the leaves being directories and segments.

Segments of memory form the basis for the controlled sharing of information. Each process has associated with it a principal identifier (or access id) that is formed from the user's name. Each segment has a list of these identifiers and what access each has to the segment. Thus one user may be designated as having access to read a segment while another may also write data into it.

Multics extends the concept of supervisor and problem states found in many operating systems to a multilevel

system of rings. There are currently eight rings in Multics. The most privileged or innermost ring is ring zero. In designating access to a segment one may specify the rings in which the access is available. If one, for example, designates a procedure to be readable in rings less then four, but only writable in rings less than three, the user in ring three may then use the segment normally as long as he is not modifying it. To modify it, he must call a program in ring two or below. This ring two program may then check the user's request for validity and, in effect, extend the hardware access control mechanism by providing arbitrary mechanisms via software. For example, access to a data base may be via an inner ring procedure which will return aggregate values, but not permit an individual item to be examined. data item. The limitation of inner ring procedures is that, if they are mutually suspicious, not more than one can be used at a time in a single process.

Multics administration consists of three levels: the system administration, the project administration and users. The system administration is in charge of maintaining the system integrity, creating projects and registering users. The purpose of registering a user is to guarantee that his user name will be unique throughout the system so that it may be used as an identifier.

Registration, per se, does not give the user access to the computer system. This is done by the project administrator. The project administrator can designate a list of users who may login using his project's name. The project administrator is responsible for the resource usage by those on this project. He may also specify individual spending limits for his users. Users may excercise control over access to their resources by specifying which users may have what access to them. For example, the access control list associated with a segment specifies which users (or projects) may read, write, or extend the segment. The access may be different for each user.

APPENDIX B

Implementation of Service Queues on Multics.

Chapter four presented a model of a service as a queue of requests and a server for the requests. Such a facility can be implemented with minimal change to Multics. The following three facilities provide a sufficient basis:

1. Message segments. Message segments are queues managed by inner ring system procedures. The are implemented using segments in the storage system, but users may only access them through these system procedures. Users may have access to add entries to a queue; list, read and delete their own entries; and to list, read and delete all entries. Associated with each entry is a secure identification of the user creating the entry.

2. Multiple processes. Multics users may currently have processes created on their behalf. To use this facility, the user sends a description of the process to be created to an absentee process coordinator via a message segment. The absentee coordinator schedules the process to be created as

if it were a job on a batch processing system. Process creations are deferred until the number of processes created in this manner is below the maximum allowable. To make effective use of services, it is necessary to ease the restriction on the number of processes that may be created on behal of users. This restriction is present mainly to simplify resource scheduling.

3. Interprocess communication. Interprocess communication permits coordination of processes by enabling processes to send signals to each other. Processes may go blocked (i.e. stop running) and wait to be awakened by such a signal. For example, a process might go blocked waiting to read a message from a queue. When another process puts a message in the queue, it can wakeup the waiting process who can then read and process the message.

By relaxing the restriction on the number of process that a user can create, and providing better tools for managing these processes (such as the ability to kill a runaway process), processes can be created as necessary to perform services. A service can be requested by putting a

message in a queue. Associated with the queue is at least one server process. When the service request is put in the queue, a wakeup signal is sent to the server associated with the queue. If the server is not processing any other requests at the time, it would then read and begin to process the users request. If it is busy, the request will be processed when the server is finished with the request it is busy processing. By combining message segments and interprocess wakeups into one facility, the entering of the request and the notification to the server is a single operation.

After system initialization, there are no active servers associated with a queue. Thus, if there are any pending requests, or when the first request is entered, a process must be created to act as a server. The description of the process is associated with the queue and together they form the representation for a service in the Multics storage system. More than one process can be associated with the queue so that the service may handle more than one request simultaneously.

An example of a service in the current Multics system is the output daemon which takes requests for bulk printing and punching. There is one server process for each output

device. In the current implementation, these processes are created when the system is initialized, or must be created explicitly by a operator. It is not, however, necessary to create these processes before they are requested. If there are not processes available to handle a request (and the maximum number of processes specified for the queue have not already been created), a new process can be created.

APPENDIX C

Functional listing of entries into the financial system.

The financial system is available to all other services of the marketplace. Since it is so central to the operation of the marketplace and is required for most transactions between services, special consideration must be given to its implementation. For this reason, the system is implemented as a set of inner ring procedures within Multics. To the programmer, this means that the financial system is accessed by using the standard Multics subroutine call mechanism. Since the system is in an inner ring, the user cannot make unauthorized references to the financial system databases.

The interfaces (or entry points) of the financial system can be roughly divided into three categories: those that deal primarily with managing and accessing services, those which permit management of accounts, and those which are used to manage individual transactions.

The entries for requesting services are included in this appendix because the financial system acts as the interface between the services.

Tables

Name: account_status

This is the prototype for the structure in which the status information for an account is returned. This information can be retrieved using the entry: get_account_status_.

Format

```
declare 1 account_status based,
          2 version_number fixed binary,     /* is 1 */
          2 number_of_payables fixed binary,
          2 number_of_receivables fixed binary,
          2 length_of_access_list fixed binary,
          2 balance fixed decimal(15,2),
          2 payables
              (0 refer(number_of_payables)),
            3 transaction_id bit(72),
          2 receivables
              (0 refer(number_of_receivables)),
            3 transaction_id bit(72),
          2 access_list
              (0 refer(length_of_access_list)),
            3 subscriber_name char(32),
            3 access_flags aligned,
              4 owner_access bit(1),
              4 paymaster_access bit(1),
              4 request_access bit(1),
              4 receiver_access bit(1),
              4 clerk_access bit(1);
```

Name: assign_transaction_

This entry is used to give a vendor access to set the requested payment amount in a transaction. It is called automatically by the request_service_ entry for the transaction id specified (if it is not zero).

Usage

```
declare assign_transaction_ entry( char(*), fixed
        bin(71), char(*), fixed bin(35));

call assign_transaction_ (account_name,
     transaction_id, assignee, status);
```

1) account_name — The name of the account containing the transaction to be assigned. (input)

2) transaction_id — Unique identifier of transaction being assigned. (input)

3) assignee — The name of the subscriber, or service to which the transaction is being assigned. (input)

4) status — Standard Multics status code. (output)

Name: cancel_service_request_ Name: assign_transaction_

[illegible] the [illegible] service
requested payment amount is a transaction. It is
automatically by the request_service_ entry for the
Usage transaction id specified (if it is not zero).

```
declare cancel_service_request_ entry(char(*), fixed
        bin(71), fixed bin(71));
declare assign_transaction_ entry( char(*), fixed
        bin(71), fixed bin(71));
call cancel_service_request_(service_name,
     request_id, status);
call assign_transaction_(account_name,
     transaction_id, status);
```

1) **service_name** [illegible] (input)

[illegible] transaction [illegible] assigned. (input)

2) [illegible] transaction [illegible]
[illegible] (input)

3) assignee the name of the subscriber, or service
to which the transaction is being
assigned. (input)

4) status Standard Multics status code. (output)

Name: close_transaction_

This entry is used to close an open transaction. Closing the transaction enables payment to be made and prevents further modification of the amount of payment being requested.

Usage:

```
declare close_transaction_ (char(*), bit(72), fixed
        bin(35));

call close_transaction_ (account_name,
     transaction_id, status);
```

1) account_name The name of the account in which the transaction can be found. (input)

2) transaction_id The unique id of the transaction. (input)

3) status Standard system status code. Is zero if the transaction is closed normally. (output)

Name: create_account_

The create_account_ procedure creates a new account with owner's access to the specified user. The owner would then use additional financial routines to initialize the account as desired.

Usage

```
declare create_account_ entry(char(*), char(*),
        char(*), fixed bin(35));

call create_account_ (account_name, billing_account,
     owner_name, status);
```

1) account_name — The name for this account assigned by the subscriber. This name must be unique for the accounts created by the subscriber. (The Multics procedure unique_chars_ may be used to create unique names if necessary). (input)

2) billing_account — The account to which is to be billed for the service charges for the creation and maintenance of this account. (input)

3) owner_name — The identification of the subscriber who is to be given owner's access initially. (input)

4) status — Status returned. It is zero if the account has been created normally. (output)

Name: create_service_queue_

This entry is used to create a service. Associated with each service is a segment in the storage hierarchy that is used to store the description of the server process and as queue of requests for the service.

Usage

```
declare    create_service_queue_ entry( char(*),
        char(*), fixed binary, fixed bin(35));

call       create_service_queue_ (service_name,
     initial_procedure, number_of_servers, status);
```

1) service_name The name of the service. It is the pathname of the service queue in the Multics storage system. (input)

2) initial_procedure The first procedure called when the service's process is created. (input)

3) number_of_servers The maximum number of processes that are to be available simultaneously for the service. (input)

4) status Standard status code. Is normally zero. (output)

[redacted]

Name: create_transaction_ [redacted]

[redacted]

[redacted]

2) **from_account** [redacted]

3) **transaction_id** [redacted]

4) **description** [redacted]

5) **status** [redacted]

[redacted]

Name: financial_system_error_table_

This is the data base in "error table" format for status codes from the financial system. Their meanings are generally obvious from their names. The status code returned is zero if the subprocedure completed its task normally.

account_name_invalid
to_account_invalid
from_account_invalid

An account name specified in the parameter list is invalid. If the calling program is not permitted to know that an account exists, the status code for invalid account will be given even if the account is otherwise valid.

account_already_exists

Attempt to create a new account with the same name as an existing account.

no_owner_access
no_clerk_access
no_access_to_transaction
no_paymaster_access
no_receiver_access
no_access_to_service

The user does not have the appropriate access to complete the request.

transaction_not_found

The specified transaction cannot be found in the given account.

transaction_already_closed

Attempt to close a transaction that is already closed or to change the amount being requested after a transaction has been closed.

area_too_small_to_return_data

The area specified for the return of the data structure is too small to contain all of the data. The initial portion of the structure

containing the size of the rest of the structure will, however, be filled in so that the user may call the entry again with an area of the appropriate size.

area_too_small_to_return_size

The area could not even contain the header of the structure. The user should supply a new area and attempt to call the routine again.

service_not_found

The specified service cannot be found in the directory hierarchy.

service_queue_full

There is no room left in the queue of requests for the service.

service_out_of_order

The service is unavailable for an unspecified reason.

insufficient_balance

The balance of the specified item is insufficient to satisfy the request.

service_request_not_found

The specified service request could not be found in the service queue.

Name: get_account_status_

This entry returns a complete status of an account.

Usage

```
declare get_account_status_ entry( char(*), pointer,
        pointer, fixed bin(35));

call get_account_status_ (account_name, area_ptr,
     status_ptr, status);
```

1) account_name — The name of the account whose status is being requested. (input)

2) area_ptr — The area in which the status is being returned. (input)

3) status_ptr — Pointer to the structure within the area containing the status. The structure is described in "account_status". (output)

4) status — Standard financial system status code. This is zero if the request completed normally. (output)

Name: get_payment_status_

This entry is used to retrieve information from a transaction record. To get this information, the subscriber must have either access to the transaction or clerk access to the specified account.

Usage

```
declare get_payment_status_ (char(*), bit(72), fixed
        decimal(15,2), fixed decimal(15,2), fixed
        decimal(15,2), fixed binary(71), bit(1),
        char(*), char(*), fixed binary(35));

call get_payment_status_ (account_name,
     transaction_id, amount_requested,
     amount_authorized, amount_paid_sofar,
     payment_date, from_account, to_account, status);
```

1) account_name — The name of either account containing this transaction. (input)

2) transaction_id — The unique identifier for this transaction. (input)

3) amount_requested — The amount of payment that has been requested. (output)

4) amount_authorized — The amount of payment that has been authorized. (output)

5) amount_paid_sofar — The amount that has actual been transferred to the receiving account. (output)

6) payment_date — The date on which the payment authorization matures. That is, the date the authorized amount is to be transferred to the receiving account. (output)

7) open_indicator — If the value of this indicator is '1'b, the account is still open. This means that the amount being requested is subject to change. (output)

8) from_account — The account from which payment is to be transferred. (output)

9)	to_account	The account that is to receive payment. This may be blank if it has not been specified yet. (output)
10)	status	Standard financial system status code. Value is zero if data returned normally. (output)

Name: get_transaction_status_

This entries returns the status of a transaction.

Usage

```
declare get_transacton_status_   (char(*),   bit(72),
        ptr, fixed bin(35));

call get_transaction_status_          (account_name,
     transaction_id, transaction_ptr, status);
```

1) account_name — The name of an account containing the transaction. (input)

2) transaction_id — The unique name of this transaction. (input)

3) transaction_ptr — Pointer to a structure described in "transaction_status" in which the information is returned. (input)

4) status — Standard financial system status code. Is zero if request completed normally.

Name: list_accounts_

This entry permits a user to get a complete list of accounts he owns.

Usage

```
declare list_accounts_ (pointer, pointer, fixed
        binary(35));

call list_accounts_ (area_pointer, list_pointer,
     status);
```

1) area_pointer — Area into which the list of accounts is retuned. (input)

2) list_pointer — Pointer to structure containing list of accounts. The format of the structure is:

```
declare 1 accounts based(list_pointer),
        2 number_of_accounts fixed binary,
        2 account_names
          (* refer(number_of_accounts))
          character(32);
```

(output)

3) status — Standard Multics status code. Is normally zero. (output)

Name: list_pending_requests_

This entry returns a list of all requests a user has pending for a given service.

Usage

```
declare     list_pending_requests_ entry(char(*),
        pointer, pointer, fixed bin(35));

call        list_pending_requests_ (service_name,
     area_pointer, request_pointer, status);
```

1) service_name — The name of the service for which the requests are to be listed. (input)

2) area_pointer — Area in which the returned list is to be allocated. (input)

3) request_pointer — Pointer to structure of the following format in which the list of requests is returned:

```
declare 1 request_list based(request_pointer),
        2 number_requests fixed binary,
        2 request(* refer (number_requests))
          fixed binary(71);
```

(output)

4) status — Standard Multics status code.

Name: make_payment_

This entry is used to transfer money between two accounts without a prior request for payment. This means that there is no requested payment amount associated with the transaction.

Usage

```
declare   make_payment_ entry( char(*), char(*),
          bit(72), fixed dec(15,2), char(*), fixed
          bin(35));

call      make_payment_ (from_account, to_account,
     transaction_id, amount, description, status);
```

1) from_account — The account from which the money is to be transferred. (input)

2) to_account — The account to which the money is to be transferred. (input)

3) transaction_id — The identifier for the record of this payment. (output)

4) amount — The amount of money to be transferred.

5) description — The description of this transaction. (input)

6) status — Standard status code. (output)

Name: request_service_

This entry is used to request a service. The internal format of the parameter field varies according to the service.

Usage

```
declare request_service_ entry( char(*), fixed
        bin(71), char(*), fixed bin(71), fixed
        bin(35));

call request_service_ (service_name, transaction_id,
     parameters, request_id, status);
```

1) service_name — Name of the service being requested. (input)

2) transaction_id — The identifier for the transaction record to be used for billing purposes. If not desired, transaction is to be associated with the request, a value of zero is passed. (input)

3) parameters — The parameters for the request. The format of the parameters is defined by the vendor of the service. (input)

4) request_id — The unique identifier for the request. (output)

5) status — The standard Multics status code. It is normally zero. (output)

Name: set_access_

This entry is used to specify the priviliges subscribers have with respect to an account. The subscriber must have owner access with respect to an account in order to set access.

Usage

```
declare set_access_ entry( char(*),.1 like
        access_structure, fixed binary(35));

call set_access_ (account_name, access_structure,
     status);
```

1) account_name — The name of the account for which the access is being specified. (input)

2) access_structure — Is a list of subscribers and their access with respect to an account. The form of the structure is:

```
declare 1 access_structure,
        2 length_of_access_list fixed binary,
        2 access_list like
            account_status.access_list;
```

(input)

3) status — Standard Multics status code. Is normally zero. (output)

Name: set_authorization_

This entry sets the authorization parameters for a transaction. It cannot set the authorization payment to a value less than that already paid. When the date specified is reached, the specified amount will be transferred from the "from" account's balance for the transaction's unit of payment.

The user must have paymaster access relative to the "from" account to use this entry.

Usage:

```
declare set_authorization_ (char(*), bit(72),
        char(*), fixed bin(71), fixed dec(15,2),
        fixed bin(35));

call set_authorization_ (account_name,
     transaction_id, amount, date, status);
```

1) account_name — The name of an account containing the transaction. (input)

2) transaction_id — The unique identifier of the transaction. (input)

3) amount — The amount of payment to be authorized. (input)

4) date — The date when payment is to be made. (input)

5) status — Zero unless there is a failure, in which case it is the reason. (output)

Name: set_request_amount_

This entry sets the amount of payment requested for a transaction. It is valid as long as the transaction is open. Modify access is required on the transaction to use this entry.

Usage

```
declare set_request_amount_ (char(*), bit(72), fixed
        dec(15,2), fixed bin(35));
call set_request_amount_ (account_name,
     transaction_id, amount, status);
```

1) account_name — The name of an account containing the transaction. (input)

2) transaction_id — The unique id of the transaction. (input)

3) amount — Amount of payment being requested. (input)

4) status — Standard financial system status code. (output)

Name: specify_receiving_account_

This entry is used by a vendor who has been assigned a transaction to specify the account to which payment is to be made.

Usage

```
declare   specify_receiving_account_ entry(char(*),
          fixed bin(71), char(*), fixed bin(35));

call      specify_receiving_account_     (account_name,
          transaction_id, receiving_account, status);
```

1) account_name — Name of the account containing the transaction. (input)

2) transaction_id — Identification of the trsaction. (input)

3) receiving_account — The name of the account which is to receive payment. (input)

4) status — Standard Multics status code.

Name: transaction_status

Structure in which the status of a transaction is returned by get_transaction_status_.

Format

```
declare 1 transaction_status based,
          2 transaction_id bit(72),
          2 paying_account char(32), /* from acct */
          2 receiving_account char(32), /* to acct   */
          2 amount_requested fixed dec(15,2),
          2 amount_paid_sofar fixed dec(15,2),
          2 amount_authorized fixed dec(15,2),
          2 payment_date fixed bin(71),
          2 payee_name char(32),
          2 receiver_name char(32),     /* for reference */
          2 open_flag bit(1),
          2 request_change_access char(32),
          2 description_length fixed binary(35),
          2 description char(* refer(description_length));
```

Annotated Bibliography

- Corbató F. J., Saltzer J. H., Clingen C. T.
Multics -- The first seven years
AFIPS - Conference Proceedings, Volume 40, Pages 571-583
AFIPS Press; Montvale, N. J. 07645

A review of the history of the Multics project. It contains a comprehensive list of references for further information about Multics.

- Dennis J
A Position Paper on Computing and Communications
Communications of the ACM, May 1968, Volume 11, Number 5, Page 370

Explores the implications of generally available information systems. The emphasis is on the social issues as opposed to the technical problems of implementing the appropriate software.

- Frank, R A
Banks' Power Questioned (EFTS vs. Citizens' Rights)
Computerworld, 11/7/73, Vol VII, Number 45, Page 1

Presents some of the problems with the Electronic Fund Transfer Systems as they are being implemented. The current systems seem to be reducing the individual's ability to protect his interest. Further, discussion of this topic can be found in the following issue of the newspaper. Computerworld has been following these issues in many articles and is generally a good source for information about computers' threats to individual rights.

- Nanus, B.; Wooton, M.; Borko, H.
The social implications of the use of computers across national boundaries
AFIPS - Conference Proceedings, Volume 40, Page 735

A summary of questionnaire results on the topic.

- Roberts, Lawrence; Wessler, Barry
Computer Network Development to Achieve Resource Sharing
AFIPS - Conference Proceedings, Volume 36, Pages 543-549

An introduction to the ARPA network as a method of sharing computer resources. Is a source of further references on the ARPA network.

- Schroeder, M. D.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility
Project MAC Technical Report 104; Massachusetts Institute of Technology; Cambridge Massachusetts

A discussion of some of the problems with implementation of a class of services. A method for temporarily granting a service access to a user's memory is presented.