The Pennsylvania State University
The Graduate School
Capital College

A Methodology to Provide and Use Interchangeable Services

A Master's Paper in Computer Science by Brian Fenicle
Submitted in Partial Fulfillment of the requirements for the Degree of Master of Science
February 2001

Abstract:
Computing today often requires the infrequent use of many software packages. This infrequent usage pattern does not justify purchasing full licenses and therefore motivates a need for a more flexible way to use and pay for the usage of software. This paper describes a design philosophy, built on CORBA, where similar services provide the same interface to clients. Services based on this design are interchangeable, allow payment per use, handle payment conveniently, are platform independent, and frequently do not require local installation. Clients can therefore easily utilize resources based on application needs and services available at the time that the application is executing. An example implementation using this methodology is also discussed.

Introduction:

Statement of the problem:

In today's computing environment people need access to many software packages and separately purchased components of software packages in order to meet personal or business needs. Many times users find themselves faced with needing a software component only one time or a small number of times. In these situations, the expense of purchasing a full license is not justifiable. Currently, there are a number of possible remedies to this problem, but each of them has limitations that make them impractical, especially for solving business needs.

Consumers could just purchase the software. Much of the trouble of purchasing software has been eliminated with the increased number of local retailers and the ability to download software after purchasing it with a credit card. However, this still requires the consumer to install the product, wasting both time and hard disk resources. Futhermore, credit card purchases are not a viable solution at companies where purchase orders are required to buy software. Consumers are not as likely to buy software needed infrequently because of the cost and extra effort required; therefore both the customer and software vendor lose.

Pay per usage software is an idea that is currently gaining popularity as a means to purchase access to software. Based on my experience working with software licensing models, I assert that in most cases the pay per usage paradigm will be implemented as a simple count down license (also known as metering). The pay per use software on the market today uses metering [16,8,10]. Metering has long been supported by major licensing software vendors and is not much of a change from how the majority of software currently works. If the software purchase is available online, then much of the normal trouble associated with purchasing software is gone in many cases. Although the software is more easily accessible, it still requires local installation.

In some areas one can go to a store that sells customers time on a computer to use certain software packages. The concept of kiosk machines is an improvement over purchasing the package in the typical fashion, because we can now pay for the software only when we need it. However, there are a few limitations here, too. The kiosk machines must be physically located near by and must also be available late at night when people are working against a deadline. Additionally, software availability is likely to be based on the needs of the majority and often might not meet less mainstream needs. Although installation and local resources are no longer a concern, the drive to the store may outweigh these benefits. Currently there does not seem to be many useful applications available in kiosks; they are limited mostly to internet access and network game play.

The software usage models presented above mostly pertain to using whole software packages. But, there might be more need for quick access to components of a larger software package. For instance, one may almost never need to convert Adobe FrameMaker documents to something WordPerfect can understand. Assume that Adobe sells a module to do this conversion separately from the rest of FrameMaker, but FrameMaker has a menu option to make this conversion. In this case FrameMaker users clearly do not want to purchase the module for a one-time need. Nor is FrameMaker likely running on the local kiosk machines. It is easy to see that any type of file conversion follows the same idea. This type of usage dictates that such software components are available in a form that can be used by the program without requiring too much intervention from the user. Again, we see that current models for obtaining software are not convenient for potential users.

Software that requires frequent updates by the user is another good reason people should be able to access software in a more convenient fashion. Consumers should be able to access the newest functionality without the administration of making software updates a few times a year, especially when they may only use the software a few times in a year.

Additionally, consumers may need access to some powerful computing resource that is used too infrequently to purchase. For example, 3D CAD models of electrical connectors are created in a CAD package. At certain points in the creation of new connectors, an electrical properties analysis is required. The electrical analysis package is expensive and requires vast amounts of computing resources. This scenario motivates the need to pay for such service when used and to allow that service to run on high end machines not owned by the customer (high end machines might be owned by the customer, but would not be required at the desk of each user). This saves the company money in both hardware and software. Furthermore, multiple companies may provide electrical analysis using the CAD model from a particular CAD package. A consumer should be able to select the vendor based on particular details of the service provided. For example, a cheaper service might provide the results in 4 hours while a more expensive service might return the analysis in 1 hour. Certain services might provide a higher level of precision in the calculations. Interacting with these services in a consistent manner allows the customer flexibility and promotes competition as a side effect.

Additionally, certain low-level services should be available in such a way that they are accessible by other programs. Imagine a personalized electronic subscription to a magazine. A customer gets only the articles of interest. It should be easy enough to retrieve articles by entering identifying information about the magazine and user identification credentials. Traveling users might prefer to view these articles from a word-processor when at home, but might be forced to view them from a web browser when they are out of town. The idea is that there are services that require some simple information to access and return a basic text (or html) stream. The returned data can be handled by any number of existing products and should not be restricted to certain applications. Even if there is a special application that handles electronic magazine subscriptions exceptionally well, there is no reason it should be limited to one particular magazine. Obtaining the articles is a simple function that should be very similar for all electronic subscriptions, so this application should be able to work with all magazines. Additionally, customers should not need to subscribe on a yearly basis. Customers could pay per article or per issue they choose to access.

Clip art could be accessed similarly. Users may have access to a number of clip art repositories, accessed in a standard manner, that charge per image downloaded. Again, a more convenient paradigm for using computer-based resources is needed.

Characteristics of a good solution:

A solution to the above problems should have all of the characteristics listed below. Quite simply, customers should be able to purchase software on demand and use it. Customers should not be required to enter user name and password to access this service (we all have too many names/passwords now, who needs more?). If users can access services in a generic manner there is no need for special installation or configuration by the user.

Users of such a system should be able to pay for services in some familiar manner. This means that creating some form of e-money is not a reasonable option. Credit card transactions seem to best meet the needs of the system while remaining familiar to most users.

In many cases the functionality offered can be used without installing software locally. Many common services require little input and generate little output, and are therefore reasonably run on remote machines. Admittedly, certain packages may require local installation, but could still be licensed using a pay per use model that is largely transparent to the user.

The user should not be required to interact directly with the service, but the service should be able to describe itself when requested by the user. This becomes especially important when multiple services are available to do the same job, as in our electrical properties analysis example. The idea that a customer would have a choice further demands that services are easily comparable to other services that provide the same functionality. The consumer will get the best selection, service, and price when competition is supported.

Services should be interchangeable. This means that a client should be able to talk to all software that provides the same service using the same interface. The interoperability of services helps prevent customers from being limited to a particular vendor and promotes competition. Furthermore, an interchangeable service can easily be replaced with a comparable service, based on availability. This provides some level of fault tolerance, since users have the ability to choose another vendor when their preferred vendor is having system trouble. The idea of interchangeable services also dictates that users don't care what platform the service is running on.

When financial transactions are involved, users demand security. Furthermore, some services might consume or generate sensitive information. These services should be able to provide a level of security as required by the nature of the exchanged data. Users of the system will also have varying interest in security of the data exchanged, so the services should be able to negotiate the details of the security with the user. The actual financial exchange should be safely conducted through an existing tool that is readily available and familiar to prospective customers.

The design goals of such a system listed above, are restated here in order of importance.

1. Services should be interchangeable, which implies they should have a consistent interface. This means that using a service should require little user interaction. Any interaction should be simple. This also means that the transactions related to obtaining a service should be largely transparent to the user.
2. Payment for using a service should be convenient. This requires a system with no additional passwords and a convenient method of payment. This means that paying for a service should be largely transparent to the user.
3. Software and components should be available on a pay per use basis.
4. Clients should be able to use a service regardless of the platform the client software runs on.
5. For basic services no local installation should be required.

Some of these goals are met by existing systems and are discussed more in the 'Related Work' section. In the following, we describe a solution that meets all these goals. This description has been split into three primary parts. Security is an issue throughout this work, but is discussed separately after the methodology is clear. A breif section on the business motivation completes the section.

The first part of the section 'Problem Solution' covers the definition of services so that they can be used interchangeably. The paradigm, described by the goals above, is referred to as the Interchangeable Service Model (ISM). As described in the ISM, service implementations follow a simple development methodology that promotes a stable system in which programs can access services in a well-defined manner.

The second section describes a sample implementation, for the sake of clarity, but standard CORBA and Java provide most of the functionality. The primary goal of interchangeable services promotes hiding transaction details from the user. Hiding the details allows users and larger software packages to leverage existing functionality in a convenient and consistent manner. Since the primary goal is interchangeable services, this implementation is called Interchangeable Service System (ISS). ISS allows programs to use a type of service with little or no vendor specific interaction from the user (clearly the user will still need to make a menu selection or enter criteria for selecting the service).

The third section of the 'Problem Solution' relates to how a client should pay for using a service in a non-obtrusive manner. This section describes a technical solution to payment that is modeled on the use of credit cards in retail purchases. This solution for payment services gains flexibility by drawing on the implementation methodology for interchangeable services. The availability of a financial service and easy access to services combine, forming a system in which users can conveniently access software and pay for services on demand.

The fourth section covers some details of security required by ISM. Since security is not the main goal of this project, we address primarily the security measures that should be in place for such a system to be feasible. This section also covers the example security implemented as part of ISS.

The fifth and final section addresses the important issue of why companies would choose to make software available with the restrictions imposed by this methodology. This section is not intended to provide full

coverage of the business issues involved, but attempts to persuade the reader that such a system is practical.

Most computer users could benefit from software that is convenient and flexible in the manner described above. Such an extension to the current software licensing paradigms allows more flexibility and therefore increased productivity for users. Customers could more quickly satisfy their business and personal needs. Software built on these principles, such as ISS, adapts easily to changes in the environment (i.e. the introduction of new service providers or unavailability of a certain service provider) and promotes the reuse of existing components.

Background:

CORBA (Common Object Request Broker Architecture) is a specification that abstracts communications between software modules. In technical terms an implementation of this specification is considered middleware. This abstraction frees the software module from concern over the programming language or the hosting architecture of the object it communicates with. CORBA implementations allow methods to be invoked on remote objects and handle the details of the required communications; remote method invocations look just like local method invocations in the program's source code. CORBA also specifies additional services for objects to find exporters either by name or object type. Implementations of CORBA are well suited for enterprise and internet computing, since they are able to find other objects and use those object regardless of platform[2,6].

In practice CORBA implementations are purchased from an ORB vendor. The core services as defined by CORBA are either purchased from the ORB vendor or from another vendor. These services are not strictly required, but they provide important functionality when developing applications. Visibroker for Java from Visigenic/Inprise[11] provides the ORB used for ISS and Prism Technologies provides the OpenFusion suite

that contains the Trading Service [17]. Other core services include the Naming Service, Transaction Service, Event Service, Time Service, Collection Service, and Security Service.

Objects in a CORBA system must understand other objects in the system. We need a common way to describe objects to promote mutual understanding, even if objects are written in different programming languages. IDL (Interface Definition Language) is a programming language-independent wayfor programmers to describe their objects in CORBA. IDL syntax is very similar to C++ syntax. Most CORBA implementations allow the programmer to define their objects in CORBA IDL, then generate some of the required source code in the language chosen for the implementation. This code combined with the ORB do almost all the work required to enable objects for networked use.

To understand ISS we can assume that the network communication portions of Visibroker work; however, we need more understanding of the Trading Service. In general the trading service allows objects to register their existence in the system. Objects register some key information allowing other objects to query the Trading Service for objects meeting certain criteria. This is a more powerful choice than the Naming Service where all objects must be located based on the name of the object. There are a number of important details that must be considered to understand Trading Service.

A service type defines a logical grouping for use by the Trading Service. Any object in the system may register as a member of a group as long as the service type has been defined. A service type defines a template with one or more property names that serve to describe members of that group. In programming, this concept is analogous to defining a class in C++, then requiring all objects to fill in the values for the data members. Also as in C++, a service type can be extended through inheritance. Each object that registers as a member of a service type, known as an exporter, sets the property values that correspond to the property names defined by the service type. Property values can be set only once, any number of times, or can be dynamically defined. For example, we could define a service type of VideoRental. This service

type might have a property name ‘cost’ which has a corresponding value of type double, since it will represent a dollar value.

An object that registers with the Trading Service as a VideoRental service type must fill in values for that service type according to the rules specified . In our example, a specific video object might register as having a cost of $3.75. When an object correctly registers, the result is called a service offer. Objects use information in the service type to find the property names they are interested in, then use the property values from the service offer to find an object of interest.

A service type may be defined to contain dynamic properties. An exporter does not supply a value for a dynamic property when it registers with the Trading Service. Instead the Trading Service waits until an importer requests that value for some service offer. At that time, the Trading Service asks the exporter for the value and returns it to the client object. One can see where dynamic properties might come in handy for a greedy VideoRental that wants to raise the prices on weekends.

Now that we understand how objects would register with the Trading Service, the next topic to consider is how to find a particular object from the myriad of objects registered in the system. The most logical way to focus a search would be to search only for objects of a certain service type. This type of search is supported by the Trading Service and is a realistic choice for static CORBA programs. So, in our example we would search for all objects registered as VideoRental service type. Since this is likely to produce more than a single match, we can further constrain our search based on property name and value pairs and order the returned matches according to property values. So, bargain shopping objects might search the system looking for VideoRental (service type) with a cost (property name) < 2.00 (property value). Furthermore, the Trading Service can sort the results of a query. For example, a bargain hunting service may request the Trading Service return Video Rental objects listed from least to most expensive for the VideoRental with the title A River Runs Through It. Property names and values can be used as search criteria without the restriction on object type, but searching by object type is the only search used in ISS.

Problem Solution:

The design goals, listed in "Characteristics of a Good Solution", are conceptually very simple and appealing. However, there are many tools and approaches that could be used to create a reasonable solution. Java and CORBA have been selected for this implementation. These tools supply the building blocks for a methodology that provides access to services in a generic manner to applications running on non-homogenous platforms spread across a network. The combination of these tools leads to a solution that is complicated to develop, but has many advantages. Services are also responsible for registering themselves as available and supporting a registered service type. Once created, services of the same type can be accessed generically by objects interested in that particular type of service.

Methodology for Creating Interchangeable Services:

Since creating and using objects in an interchangeable fashion is the most important component of ISM and the example implementation, we will consider it first. We generally understand, from the background section, that services should be created and registered with the Trading Service such that importers can access them exactly as they access all offers of a given service type. This means that each service implemented must offer the same interface and register under an equivalent service type. In this case, the property names will all be the same, as defined by the service type, but the property values can be used to distinguish among available services. Therefore, each service that corresponds to a given service type will register as such and provide the property values specific to this service offer. Now the client can access a number of services (of the same service type) in a consistent manner and can select a service based on the property values. Once a client obtains a reference to a specific object via a service offer, all the methods defined in the interface can be used. The example client and service, discussed later, will help clarify how these concepts are implemented in practice.

It is important that all offers that provide the same type of service register with the Trading Service under the same service type. This simplifies the search for matching services and provides stability in the

properties a client should consider when selecting a service provider. A service provider might like to introduce a new service type that extends an existing service type so that they may provide some additional functionality not provided by the original service type interface. Although this is acceptable in the CORBA model, this sort of behavior detracts from the consistency of the Interchangeable Service Methodology.

As mentioned in the design goals a service should be available on a pay per use basis. This simply means that designers of systems should provide code to manage the payment transactions. However, all services should handle the financial transactions using the same interface. A good solution would be for the financial transaction to be handled by another service. This way all services become clients of a banking service. Handling the process of purchasing software in this manner makes paying for software more transparent.

By basing any solution on CORBA, we enable clients to use services without concern for the computing platform of the service provider. We are also able to use CORBA services without installing each service locally. Java adds more platform independence to ISM (and therefore ISS), but clearly the important platform independence is provided by CORBA.

The example implementation for this paper concerns a Translation Service. This example service and other translation services implement a simple interface to translate from one human language to another. This is a good example of a service the may occasionally be useful, since there are times when e-mail would need to be translated to communicate precisely with another person. This functionality may be available as a menu or button selection in a favorite e-mail editor. This menu selection might return a list of choices to the user, may use the user's configurations to select a preferred vendor, or may select the translation provider based on some default behavior. In this way a user can take advantage of any new service providers as well as provide some level of failover if the preferred service offer would not be available (assuming another service offer existed for the desired translation). The example implementation, ISS, will clarify the details of the methodology.

Example Implementation of Client and Service:

Small details of the implementation will be addressed as they are required. For the sake of simplicity the topics will be ordered as they are in the source code.

Defining A Service:

To make exporters of the same type interchangeable, each must inherit from the same type. Our example centers on the idea of a service that can translate from one human language to another. So, each object providing a Translation service should inherit from the TranslationServiceInterface. This assures clients that each translation service object will provide access to at least this set of methods. The IDL definition of the base class that should be used by all implementations of translation services is listed in Figure 1 and the IDL definition as used in the example implementation follows in Figure 2.

```
// base class interface definition
interface TranslationServiceInterface {

        string translate(in string inText, in ServiceContract svcContract,
                        in string creditCardNumber);        //this method actually does the translation
        string textDescribingService();      // may provide more information about the service
        ServiceContract getServiceContract();      // handles the price & time details of a translation
        long serviceContractIsValid(in ServiceContract svcContract);     //allows customer to see
                                                                        //how long until the contract expires

};
```

Figure 1: The IDL definition for the TranslationServiceInterface

---

```
// a particular implementation where the : indicates inheritance
interface TestTranslationService:TranslationServiceInterface {

        // no new methods
};
```

Figure 2: The IDL definition of a particular TranslationService called TestTranslationService that inherits from the base class of TranslationServiceInterface

---

The development tools bundled with the ORB will take this definition and create most of the source files required to support these objects. The ServiceContract object details are addressed later. The important thing to understand at this point is that a client can now treat a reference to a TestTranslationService object as if it were a TranslationServiceInterface object. As expected the inheritance heirarchy show in Figure 1 and Figure 2 allows us to manipulate objects of the subclass as instances of the baseclass object.

Registering a Service with the Trading Service:

When an object providing a service starts, it must first find the Trading Service. Since the normal method of finding the Trading Service (using resolve_initial_references("Trading Service")) was not incorporated into the Java ORB provided by Inprise, we must find the Trading Service using the Interoperable Object Reference file (this is a minor coding detail that has no real impact on the problem solution). Once the Trading Service is found, the service must first check if the service type is defined. If the service type is not yet defined, it must be defined before the service provider can register with the Trading Service. Once the Trading Service understands the service type, the service provider can set the property values as required by the service type definition. Once the service type is registered and the values are set for this service offer, the object providing the translations is available to importers.

A utility class provides methods that check if the Trading Service has a service type registered. If that type is not yet registered, the method registers it. Now the method registers the service offer using the arguments provided by the caller. These methods are quite simple. They provide all the functionality needed for this example implementation, but would need to be more robust to handle the complexities of a real implementation. However, they do help a service writer register a service, with the side effect of forcing the service to register as the preferred base type.

Property Name	**Property Type**	**Mandatory or Optional**
Translate_from	String	Mandatory
Translate_to	String	Mandatory
Cost	double	Mandatory

Figure 3: Property Structure for a TranslationService
Mandatory parameters must be defined by the exporter so that the Trading Service will present the service offer to other objects.

The TranslationServiceInterface type is defined such that a service wanting to register a service offer must fill in values for the following properties (as seen in Figure 3):

- Translate_from: this property is the language that should be input to the service.
- Translate_to: this property should have a string value that represents the language output from the service.
- Cost: This property is the advertised price for the service.

Discounts may be offered based on customer or volume, but that will be finalized during the ServiceContract negotiation. The price for a service can quickly change, since the cost property can be updated at the service provider's discretion. In some cases dynamic computation of the service offer's price would be ideal, however this feature is not supported in the example implementation. The client locks in a price during ServiceContract negotiations.

An inheritance hierarchy can be defined in the trader. Then each service extends the base class and registers as the base class. If a hierarchy was defined, a request for matching services based on a class,

would also return any matching subclasses. The standard CORBA approach to service registration leverages this object-oriented functionality. Since any subclass must provide at least the same interface as the parent class, we can be assured that the service provider will understand the client's method requests.

The ISM method is a slight deviation from the standard CORBA approach described in the previous paragraph. Each service registers as the base type using a utility class to simplify this process. Even though a class may have extended the original functionality, it will still be known to the clients (through the Trading Service) as a base class object. This masking of the actual type is somewhat superficial, since a dynamic client can interrogate the object after obtaining a reference. However, this approach lends stability to the system and discourages coding to a non-standard version of the service type definition. This standardization is important to stability over time. Now, all clients can code to the standard interface (this interface should be sufficient, since in theory it would have been defined by a consortium),which prevents frequents changes in client code. Furthermore, clients cannot take advantage of the additional subclass functionality in static CORBA without causing exceptions.

So, at this point we have implemented the sample service, TestTranslationService as a subclass of TranslationServiceInterface. As expected no additional public methods are provided by the service. We proceed to register with the Trading Service as a TranslationServiceInterface type, which has the ramifications discussed above. The distinction between the actual object type and the registration type should be clear. If we actually knew the object type, we could use all the methods it provides (in this case none beyond the base class). However, the clients will believe that this service is an instance of a TranslationServiceInterface (because of its registration with the Trading Service) and will limit their use to the base class methods. Now that the service is available, it will respond to requests from clients as coded. In the example implementation TestTranslationService is a facade, and does no actual translating.

Code Discussion for a TranslationService:

The methods defined in the base class do not have implementations. Therefore, all subclasses must define the specified methods.

The translate() method should do the actual translation from one language to another language. For now, it will be enough to understand that a client passes a string and some other data in, and is returned the results in another string. The last two arguments pertain to the agreement on price and payment for use of the service. These two items will be covered when we consider non-obtrusive payment for services rendered in a later portion of this paper.

The textDescribingService() method takes no arguments and returns a string. This is intended to allow the service provider a chance to explain any specialties, promotions, pricing, or any other useful information. This method is intended to provide information to a human user. This allows a person to select a service using more than just the information listed in the service type. For example, a person could pick between the two services that matched his selection criteria. However, this is provided only for the benefit of the user; it is not required to select and interact with service providers.

The getServiceContract() method allows the client to request a contract that contains start and expiration times, the service price, and any other useful information. The simple ServiceContract object allows limited contract negations and options and is covered in the section related to payment for services rendered.

The serviceContractIsValid() method allows a possible client to verify a contract. Knowing that agreeing on time is impossible in a distributed system and understanding that time constraints play an important part in contract negotiations, we see that we must be able to verify a contract is still valid with the ultimate source. Since the contract will be measured against time as defined at the service provider, we allow the client to easily check if a contract is still valid by returning the number of seconds until expiration.

Example Implementation of a Client Program that Uses a TranslationService:

In ISS, an importer has a few simple steps to take so that it can find and use a service. As always this client should be able to use any service (having the same service type) in a simple consistent manner. Before a service offer can be used the client must be able to find it using the Trading Service. It is important that all

similar services register with the Trading Service using the same service type. By using the same service type, we can be assured that the property names and interface will be consistent. Knowing that a consortium defined the service type, we believe it will be stable over time. These two conditions allow clients to find services in a simple and stable manner, which was a primary design goal. Once a client selects a service based on availability and application specific code, the object reference can be used. Since the selected service extends the TranslationServiceInterface, a client can be assured any Translation Service will support the appropriate method calls. Clearly, the client will need to know the base class definition at compile time. This is as expected when using static CORBA. However, it is important to note that a client will not need compile-time access to the subclass definition, since it is only accessing base class methods.

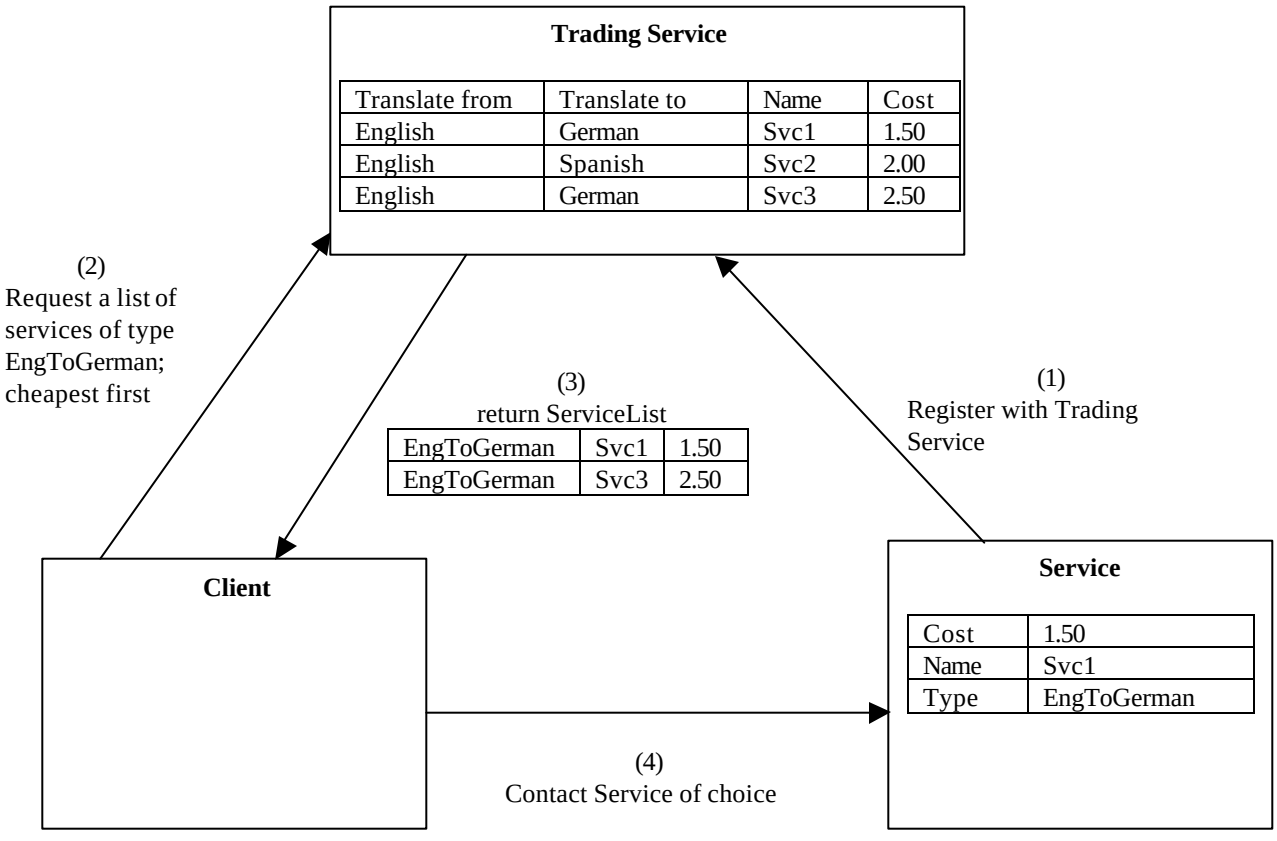


Figure 4: This diagram illustrates the events required for a client to find a service of interest using the Trading Service.

As seen in Figure 4, a client will first find the Trading service. Next the client will need to find an acceptable service. Acceptable is defined by the requirements of each specific application. In this case, we search for all TranslationServiceInterface services. Searching restrictions use the properties to restrict matches to only English to German translation services. The services matching these criteria will be sorted from least expensive to most expensive, at the client's request. In this case the cheapest service is contacted to obtain a service offer.

Once a client has chosen a service offer it can access the methods provided. Again, we are temporarily ignoring the ServiceContract negotiations and payment for the service. In our case the client passes a string of English text and gets a string of German text in return.

In the section above we have seen how a service is offered in a generic way. Consequently, clients can access a service in a consistent manner. It should be clear that if a client program needs a translation service it can access any of the available services using the same interface. So, a person who wants a translation done could select this function from the menu in his e-mail editor. A few inputs may be required from the user (alternately we could find their preferences stored on disk). A list of matching services might be returned to the user, but is not required. Once the service provider is selected, the translation can be completed without further user interaction. The e-mail editor can clearly access all translation services in this manner. The system, ISS, as described above clearly meets design goal 1 listed in the "Characteristics of a Good Solution". Furthermore, the TestTranslationService is an example of a basic service that requires no local installation, which meets goal 5 as well. A local installation would still be required in the case where we are using an additional module from a larger software package or when the software has a complex user interface, but the idea is that there are some basic services that can be used without an installation step.

Non-obtrusive payment for services rendered:

Until now, we have ignored how a client and service agree on the terms for a service and how that service provider receives payment for the service. Both of these details are critical to a successful community where clients can access services in a predictable way. First, the client and service should agree on a price. The details of contract pricing and expiration are at the discretion of the service provider. This means that one service may offer discounts to preferred customers and another may not, but both will use the same objects and interfaces. Once client and service have agreed on a price there should be a convenient way to pay for that service. In ISS, financial transactions are modeled after retail credit card purchases. A BankService fills the role of the credit card company in typical credit card transactions.

Service Contract Negotiations in ISS:

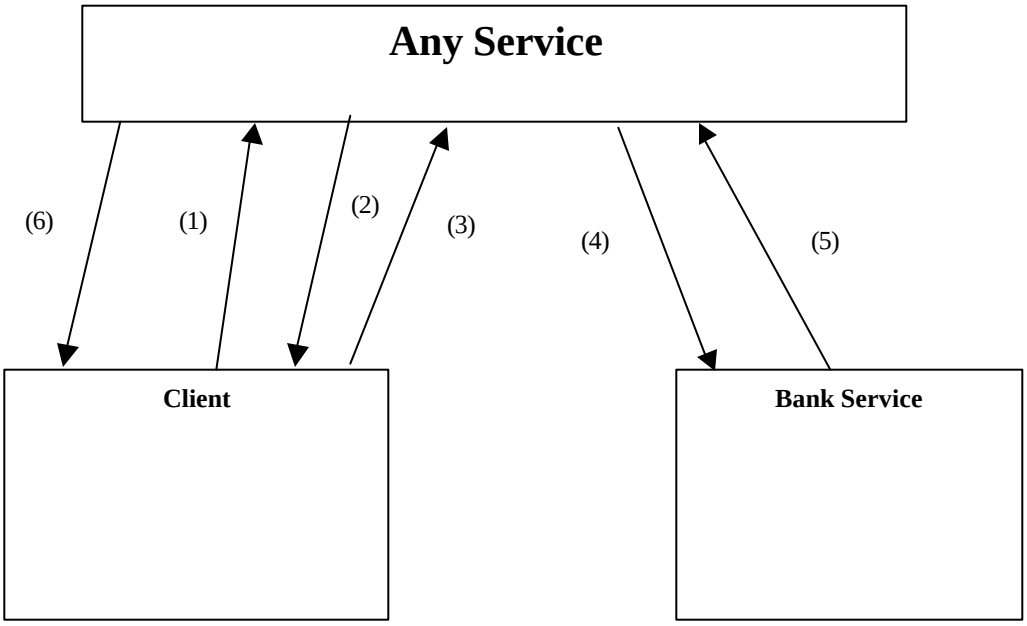
It is logical that an object should be used to represent all the information relevant to a contract. An object simplifies the client/service communication and fits well in the Java/CORBA approach. This object should be common across all service offers in the service type, but may be different to meet the needs of different service types. An example ServiceContract object has been defined for use with translation services. The IDL for this object is shown in Figure 5.

```
interface ServiceContract {
    void setContractBeginDate(in long long bd);     //this is just a normal long
    void setContractExpireDate(in long long ed);
    void setContractNumber(in long long cn);
    void setContractPrice(in double cp);

    long long getContractBeginDate();
    long long getContractExpireDate();
    long long getContractNumber();
    double getContractPrice();
};
```

Figure 5: The IDL for a service contract object.

The long values are to represent milliseconds since 1970. Milliseconds were used for simplicity. The double value holds a dollar value that represents the price of a transaction. The first four methods are for use by the service program creating the object and the last four are for use by the client program. At this point there is no security to prevent the client from changing the ServiceContract object other than security measures implemented in the service provider that issued the ServiceContract. In ISS the service contract is validated and is expired to prevent contract modification and reuse.



(1) Client requests service contract.
(2) Service contract is returned.
(3) Client requests the service according to the contract.
(4) The service provider requests a funds transfer.
(5) The Bank confirms/denies the transaction.
(6) The results of the service are returned to the client.

Figure 6: This diagram shows the way client, service, and bank service interact with each other during contract negotiations and payment for the service.

Figure 6 shows the steps involved in contract negotiations. This example definition of a service contract only allows one round of negotiation. The client requests an offer and can either choose to accept or

decline. Furthermore, the contract and pricing information are limited to US dollars. In a real life situation, this limitation would not be acceptable. Currently, contracts must be handled by each service. Since the ServiceContract should be the same for all services of the same type, client code can easily able to deal with these contracts. This constancy also enables all services to reuse library code that deals with the processing involved with contracts.

Since price changes are such a simple matter, we can expect prices to change frequently. ServiceContracts allow the price of a transaction to be fixed. Now that a customer has a fixed price, he can make his final decision on purchasing the service. If the customer would like to purchase the service, we expect him to pass the Service Contract object back to the service provider, pass his credit card information, and any other application specific data structures. In the running example, the client will pass only a string in addition to credit card and contract information. We expect a service to bill the client's account when the transaction is complete to prevent a client being billed for a job that did not complete as expected (i.e. a crash occurred).

Payment for the Service:

Now that we have a client and service provider that agree on a price for a service, we need a banking service to provide the banking transactions. In our normal lives as consumers, we frequently use a credit card to pay for goods and services. We establish identity and credit limit with this one company. All participating vendors honor our payment via that banking service. This situation allows us to purchase goods without having to reveal much about ourselves to retailers and is familiar to most people. It would be desirable for software to be purchased in the same way.

We have a need for an online credit card service. In ISS, the interface for the credit card based transactions is defined in the BankServiceInterface class. The example implementation of the BankServiceInterface is called FunnyMoneyBankService. An interface is defined for all BankServices, so that all access is standardized. Clearly, this is just another service implementation based on ISM. It seems likely that there will be more than one credit card-like service available in the world, so the interchangeableness makes sense. The BankServiceInterface defines a rudimentary banking interface. All credit card-like services in an

implementation, like ISS, would inherit from the BankServiceInterface IDL definition presented in Figure 7.

The IDL for FunnyMoneyBankService, which inherits from BankServiceInterface, is listed in Figure 8.

```
interface BankServiceInterface {

        exception TransactionDeniedException{};
        string creditCardTransaction(in string amount,
                                     in string fromAccount,
                                     in string toAccount)
                                     raises (TransactionDeniedException);


}; //end of bankService decl
```

Figure 7: The IDL for a Bank Service that will provide credit card-like transaction between clients and vendors in the system.

```
interface FunnyMoneyBankService:BankServiceInterface {

        // no new methods, see base class

};
```

Figure 8: The IDL for an example bank service called FunnyMoneyBankService that has no additional public methods beyond those dictated by the base class. The bank service might contain other classes, like Account, that are used in processing. These methods and objects are private and therefore not exposed in the interface.

The BankServiceInterface defines the methods that a minimal credit card-like service might offer. The BankService in ISS works similarly to a normal credit card. The customer and merchant account numbers are supplied with the amount of the transaction. Figure 6 shows the steps involved in payment for service. In this case, an exception has also been defined to notify the caller of error situations. The TransactionDeniedException in Figure 7 lets the caller know that the transaction was denied, but gives no further explanation. A more powerful solution would have more precise exceptions, but this serves as an example of the correct approach. A credit service like the FunnyMoneyBankService will have lots of

information about its customers and many rules about transactions. In the example implementation, the example bank service will not allow a person to exceed their credit limit. It will not allow manipulation of funds in a non-existent account, and so on. The business rules and the data stored in the example bank service are not important for this example, so we can assume it works reasonably.

This section has described how a customers pay for a service they want to use. The idea is simple and mostly transparent to the user. Using credit cards leverages existing financial infrastructure and financial tools already possessed by the customer. Since customers do not have an account with the vendors, there is no need to remember extra passwords or handle billing from each individual service. Clearly this payment arrangement meets the second design goal of the system, convenient payment, and at the same time supports the pay per use paradigm, which was design goal 3.

Security Features of the System:

CORBA traditionally controls who can access a particular object through an access control list (ACL). This administrative burden is not acceptable for the system described in this paper. The idea of a service contract places the burden of authorization on the exporting object, so we no longer need the ACL to enforce such rules.

A distributed system such as ISS without secure communications is not feasible. In the proposed system, there are two separate needs for security. First we must secure the information involved in the financial transaction. Second, we must secure the information exchanged between a client and a service provider. The final two sections on security discuss the CORBA Security Service offerings and a way that all communications could be secured.

Financial Exchange Security:

A number of reasonable options are available for securing the exchange of financial information. In ISS, credit card information is sent out in the clear to the vendor. Unfortunately, it is all too easy to dis cover the

account information while it travels on the network. One reasonable solution would be to encrypt the credit card number using SET (Secure Electronic Transaction), a financial industry standard, or some other standard encryption scheme [5]. In this manner, neither the vendor nor hackers would be able to "see" the credit card number, but the bank could still verify the accounts in question to make the funds transfer. Public key encryption could also be used, so that the bank could be certain that the credit card belonged to the supposed user.

Security of Data Exchanged During Service Transaction:

So far we have ignored the security of data exchanged between the client and service provider. Some data might not require any security at all, while other data may represent proprietary information and require very stringent security measures. The ISS implementation uses public key encryption combined with symmetric encryption to securely transfer the text input and output of the translation service as an example of a possible solution. The steps involved in securing data in the example implementation are shown in Figure 9 below. In the example translation, only the text string is encrypted. The financial information is transmitted insecurely. For application specific data, simply securing all communications would be better than no security at all. However, a variable level of security negotiated between the client and service would be the best solution.

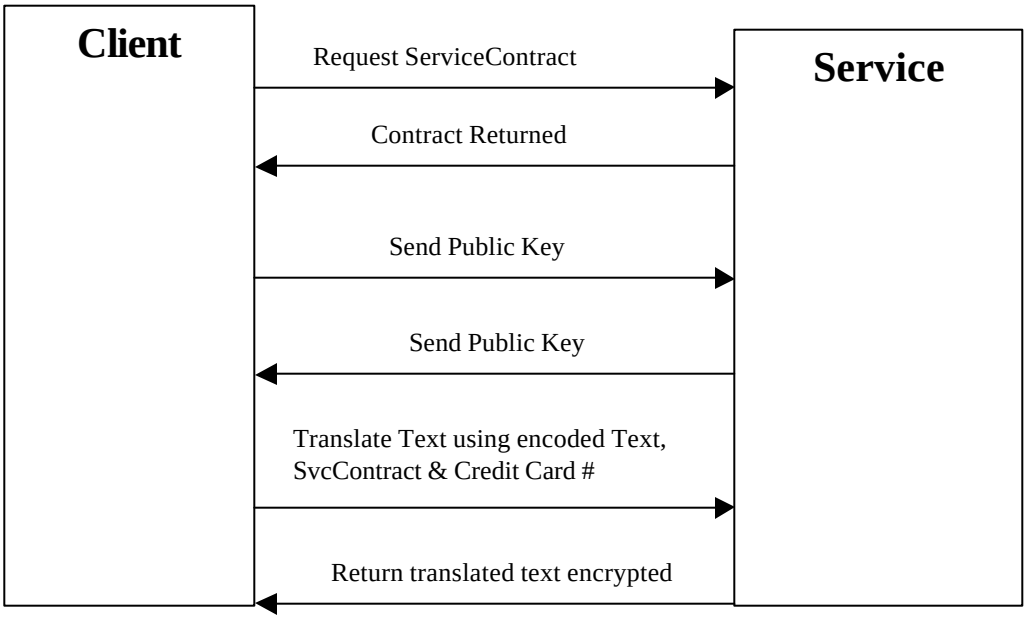


Figure 9: This diagram illustrates a sample exchange of information between a client and the TestTranslationService. Note that not all communications are secure, particularly the credit card information.

CORBA Security Service:

SECIOP (Secure IOP) is now available as part of the CORBA Security Service. This allows all communication to be secured, but does not allow the level of security to change based on user needs. Although this would be better than no security it is not the best solution for the system described. When the project began, there were no implementations of the Security Service freely available.

Additional Ways to Secure All System Communications:

Visibroker has hooks built in so that programmers can insert code to encrypt all inter-object communications. NetCrusader has used these hooks to deliver a product that does just that [9]. These are

different security approaches available, but using any of these is outside the scope of this project. Additionally, the NetCrusader product was not freely available during the implementation of ISS.

Any solution for securing the inter-object communications should be flexible. The client and service provider should agree in the type and strength of security that will be used for the exchange. Secure Sockets Layer (SSL) is based on this principle, and therefore might be a good choice. SSL over IIOP (Internet Inter-ORB Protocol) exists and is now provided separately from the Security Service, for the Visibroker product. This product was not freely available during the implementation of ISS

Ultimately the security features of the system as it is implemented are insufficient. However, one of the above methods could be employed to transform ISS in to a secure system. At this time, it seems that SET should be employed to manage the financial portions while SSL should be the solution to provide the flexibility desired for client/service communications.

Practical Application in Business:

For service providers, ISM makes the most sense in service areas where there are few providers of a particular service, and in cases where a large company dominates, but there are many other smaller service providers.

Translation is an example of a service where there are currently few providers, although, there are some free rudimentary services available on the Internet. In this case the involved parties all benefit from the standardization, because the services become more visible and accessible to prospective customers. Since there is clear motivation for consensus and the interface should be technically simple, we can conclude that a standard interface will be agreed upon. The group might form a consortium, working together to produce a simple user interface to access translation services. Translations available will likely range from those done quickly by machine to those done precisely by bi-lingual humans. Although many of the service providers will offer the same service, speed, cost, and quality will differentiate them from one another.

In the situation where there are many small companies and a few large companies in a market, the larger companies often control the market share. In this case the smaller companies are clearly motivated to work together to take market share from the larger providers. Thus, the smaller companies will have a strong motivation to find a common interface, while the larger companies will likely not participate in the interest of protecting their advantage. The larger company might then be forced to participate, as their competition is providing a desirable feature to customers.

In the above ways, communities will form. Others will choose to join to make their product available to another market segment, while others will participate at the request of their customers and potential customers. Participating in such a community becomes another marketing tool. Indeed, an infrequent customer may eventually use the product enough to purchase a full license, the financial mainstay of the company.

Related Work:

There are a number of systems in use or proposed that share certain features with ISM and/or ISS. The first section covers ideas similar to ISM, but not close enough to merit detailed discussion. The final two sections more completely cover implementations that are most similar to ISS.

Similar Ideas:

The systems described below demonstrate that the pay per use paradigm has found a place in the software industry. Most of these are simple proprietary examples of a concept less generic than the goals of ISS. In most of the examples below one must install the software first, then pay for the usage. This type of pay per use model meets only one of the design goals of this system. In these cases, where the software package must be installed locally, the proposed paradigm for purchasing usage would save users the work of calling the vendor to purchase more usage.

Pay Per Clik is a pay per use Application Service Provider. The company promotes the idea of paying per use instead of the monthly-based fees typical of Application Service Providers. They simply offer the service of billing the customer for fine-grained usage, for which they take a percentage of the profit. They enable metering of software to prospective customers. Pay Per Clik shares the pay per use goal, but does not attempt to provide services that are interchangeable [16].

e*ECAD Inc. is offering its chip design software for use by pay per use and pay per hour licensing. This will allow customers to access the tools for minimal cost based on usage rather than purchasing the software for thousands of dollars. This is a good example of how one might install software locally, then pay for only the time it is used. This is a proprietary solution that does meet some of ISM's design goals, but it does not meet the requirement that services are interchangeable [8]. Face2Face Inc. provides their facial motion analysis software similarly. In this case, the customer pays based on the amount of video processed by the software [10].

Lacert provides programs to accountants for calculating state and federal taxes. The state tax returns can be purchased for a one time unlimited usage fee or in a pay per use model. The easy segregation of the components by state and infrequent need for an accountant to do an out of state return makes this model appealing for both parties. This practice has been going on for a number of years in tax software. Lacert may have implemented the software so that all state tax returns have a standard interface to obtain the license. Since the processing is done locally, one objective of interchangeable services has not quite been met. This model is like purchasing separate components of a larger system. Lacert's system requires a phone call to purchase a license, which could be more convenient. Currently, there are efforts to allow customers to purchase licenses online, but it will not be transparent to the user [3].

Kala:

Kala [4] is a system that is more similar to ISS. Kala is quite different in implementation strategy, but shares enough design goals that further discussion is appropriate. Kala is an API to create licensing schemes.

Therefore, it can be used to license software in the standard way, pay per user, or to license software using the pay per use paradigm. Components and resources in Kala are very similar to services as presented here.

Kala Overview:

When using Kala a user negotiates pricing with the resource. This price is "provisional", which means estimated, since the component might use another component and incur higher charges than expected. There is a maximum and minimum provisional charge provided before the client makes the purchase. After the two parties agree on the provisional charge access is granted to the component. Charges are only incurred if the resource is used. The actual charges are handled by the resource manager once both parties agree (it also does some sanity checking on the real charges against the estimated charges). Each component is responsible for computing billing information based on its particular billing algorithm. All interactions between user and resource are conducted in meter units, not normal currency. Billing reports are generated by the resource manager using meter units. Conversion to real money and actual billing must be handled by a bank-like entity. Users may have credits added to their account through the exchange of cookies with the vendor. Using cookies, a customer can securely purchase meter units to be used at a later time.

Some Specific Points:

The resource manager is responsible for uniquely identifying the resources and the clients (clients are also resources, but we will call them clients to be clear). Storing all the resource information in one place is subject to a single point of failure. Maintaining a list of clients is an extra administrative step that can be avoided if users do not need local accounts. In a CORBA system, the Trading and Naming services handle the identification of resources. It is preferable to use the existing functionality of CORBA rather than re-inventing code to handle services for joining and leaving a community.

Service contracts in ISS have no user accounts associated with them, since customers have no local accounts by which to reference them. However, service contracts apply to a specific service and maintin expiration information. Licenses in Kala maintain the expiration and related resource, but also maintain the

user information about the user who was issued the license. There is a distinction to be made. Service contracts by themselves are only a promise to provide a service at a particular price until some expiration time. The ServiceContract must be submitted with payment to actually use a service. A license is prepaid and entitles the bearer to use of the service within the restrictions imposed.

Cookie exchanges in Kala allow the user to obtain enough meter units to access a resource multiple times without requiring a financial exchange for each use. Meter units are exchanged for each use of a resource. Although this reduces the number of financial transactions, it requires users to have accounts with each resource to allow tracking of meter units purchased.

Kala handles the revenue collection by moving the credits from the user account to the supplier account. There is a separate billing step where meter units need to be converted and billing must be done. The entity that does the conversion and periodically pays those resources with a positive balance is nebulously described. ISS does not deal with meter units for two reasons. First, this is a big responsibility; savvy users should be wary that this account transfer is indeed safe and correct. Furthermore, reusing existing financial solutions is preferable to creating a new non-standard solution.
Second, there are plenty of financial institutions that exist just for securing financial transactions and handling billing. Why reinvent the wheel?

As in ISS, Kala intends to make paying for each use transparent to the client. Exchange of metered units is transparent, but the actual financial exchange is not. In [6], the financial exchange appears to be an offline process. This model may be acceptable to users who purchase all their software from a few vendors, but is not as transparent as in ISM.

Much of [4] walks through examples of how Kala can handle various forms of licensing. Much of that discussion focuses on installing software locally, then paying per use. This usage model is addressed by ISS as well. In ISS, an installed application might have menu options that allow the user to pay for some

amount of usage and results in a license file being downloaded to their machine. Software that anticipates repeated use by individual clients might support a slightly different contract model or may handle the details of the service contract somewhat differently than in the TranslationService example.

Partly because the authors intend for components to use other components as needed during execution, Kala involves a complex system of estimated and actual charges. This means that the cost for a "run" is not known until execution is complete. A simple system of paying a clear fee up front is preferable. In ISS, the advertised fee for a system could be dynamically generated based on the nature of the client's request resulting in a sum of the cost of services used indirectly. However, the paradigm described in this paper is limited to static up-front charges and no provisions have been made for variations during service execution. The authors of [4] point out that the system should be recursive, meaning that resources can use other resources. It should be clear that ISM supports a service using other services. For example, all services use a Bank Service.

Kala is a reasonable system to provide licensing for software a user already has and also works nicely to activate specific separately purchased features of the software. The ability to activate specific components was a design goal this project shared with Kala. However, it does not manage interchangeable resources in a generic reusable manner, which is an important design goal of the described system. Ultimately the distinction is that Kala is an API for licensing software and components, while ISM primarily focuses on a methodology to allow the purchasing of services that are interchangeable.

JINI:

ISM shares a number of design goals with JINI, although the implementations use some different technologies. JINI is a layer on top of Java provided by Sun Microsystems Inc [1]; ISS combines the use of Java with CORBA.

In both systems we have the idea of obtaining a contract with another resource in the system. The two parties involved work out the details of this agreement. In JINI these leases may be obtained to hardware

and software resources. The idea for ServiceContracts is similar to the JINI leasing approach. In JINI leases can automatically renew themselves. However, leases have no concept of paying for resource usage as in the Service Contract model. JINI is intended to allow resources to find each other on an internal network, so charging for usage is not an important consideration. Leases in JINI are defined for the entire system. This means that all services use the same lease object. This model is an advantage for JINI because all resources use the same lease object. ISS provides the flexibility for different types of services to have different contracts when required.

In both systems programs look for available resources matching the program's needs at run time. We will use printing to illustrate the similarities between JINI and ISS. When one needs to print a document, the word processor finds all printers available. From that list, clients select the service that is closest or has the features needed. By not relying on a specific printer, a user can avoid problems when that particular printer is not available. In JINI, this is referred to as the lookup and discovery protocol. In a CORBA based system, the Trading Service functions similarly to the look-up service in JINI.

JINI programs, both client and service, are limited to Java (with the exception of wrapping other programming languages, like C). Clearly a CORBA based system has the advantage of not limiting the programming language of the client or the service provider.

In both systems we have the idea that there are standard components that should have a common interface. If we continue the printing example, we realize that all printers do the same basic thing and have some common features. For example they have a location in the building, they may support double-sided printing, may support color printing, may support certain file formats, and may support a number of printing resolutions. So, all printers should have a common interface that can be accessed by other programs. Through the interface we should be able to ascertain the features of a particular printer, and send our print job to the printer we have selected. In both systems a common interface must be defined and each service must implement that interface.

Ultimately, JINI and CORBA share a number of ideas, many of which were critical to meeting the design goals of this project. However, JINI limits users to the Java programming language where a CORBA based system does not. Additionally, JINI does not directly support the pay per use model.

Suggestions for Further Work:

This example Bank Service accepts all charges that are proposed by the vendor. In normal credit card transactions we have signatures and shipping addresses to provide some checks on legitimacy. When working with computing resources, we have eliminated both these checks on the system. A reasonable solution would be for the bank to contact the client for approval of the proposed charges. A better alternative would be for the client to notify the Bank Service that it will accept charges of some amount from a specific vendor. Then the Bank Service can be sure the charges are legitimate if it can positively identify the customer (via public key or some other security means).

The security model could be extended as described in the section 'Security Features of the System'. Since Security was not a main focus of ISS, the example service implements encryption on a small part of the exchanged data, simply to show that it could be done when required. As mentioned, a combination of SET and SSL should be adequate. Using an implementation of the SET standard would require significant changes to the current architecture of ISS. Using public key encryption for just the credit card number would be simpler to integrate, but requires that we have a key server. A reasonable security model would eliminate the need for the suggestion in the paragraph above, since we could now assure the legitimacy of both parties in the transaction.

This system focuses on using services infrequently and generically. Certainly there is a need for pay per use software that will be used more frequently. This system requires the negotiation of a contract and a financial transaction for each use of a service. While this model is appealing in its simplicity, it results in

unnecessary overhead when the service is accessed frequently. The paradigm used by Kala might prove to be a better solution when the service is accessed routinely. If a customer could purchase some number of meter units, we could avoid the extra financial transactions and the two parties involved would simply exchange meter units. In this scenario, ServiceContracts would be negotiated using meter units instead of dollars. Unfortunately, this requires extra support for a protocol for purchasing meter units and more importantly forces the client to have an account with the vendor. Avoiding a local account and related login credentials was a design goal of this system.

A more complete ServiceContract might have the user's credentials optionally supplied. This would allow the service provider to offer particular users special deals or more personalized service. Additional methods to support multi-round contract negotiations would also be an improvement to the service contract negotiation process.

Conclusion:

The introduction proposed five goals for this system. They are summarized here for convenience.

1. Services should be interchangeable, which implies they should have a consistent interface.
2. Payment for using a service should be convenient.
3. Software and components should be available on a pay per use basis.
4. Services should be available on all client platforms.
5. For basic services no local installation should be required.

The primary goal of ISM was to provide access to similar services through a consistent interface. The implementation of ISS requires services to register under a common service type and therefore similar services share an interface. This consistency allows client programs to easily access services, even if different vendors provide them. This design feature is a slight deviation from the normal CORBA approach. The JINI model also supports consistent interfaces, but does not meet some of the other goals. The other systems discussed do not meet this important need.

Payment for use of services in ISS is modeled after the credit card transaction in retail purchases. This model is convenient and familiar to most users. This payment model eliminates the need for accounts with

each service provider. Kala and many other systems require the overhead of accounts with each service provider, which results in unnecessary administration and user interaction. As in retail transactions, the customer has very little involvement in the funds transfer when using ISS.

Goal three suggests that a reasonable system should support a method to pay for services when used. Infrequently used services or components motivate this goal. The pay per use paradigm is not new to computing. As seen in the related work section, there are a number of companies providing software using this billing model. Clearly pay per use software is an idea that is gaining support. However, these proprietary systems are at disadvantage when compared to a system based on CORBA.

Goals four and five are inherent to ISS since it uses Java and CORBA. Client programs can be written in Java to gain a platform independence advantage, but Java is not required to work with the CORBA services. Hence, client programs can be written in any CORBA-enabled language as required for the particular application, or in Java to maintain platform independence. Other systems restrict the platform or client program implementation language, or require local installation.

Throughout this paper we have seen examples of how the flexibility of ISM, built on the five goals, benefits the service provider and the customer. Although a number of existing systems address a subset of these goals, ISM is unique in that it satisfies them all. Thus, the power and flexibility of ISM surpasses that of existing systems for licensing software and for providing interchangeable services.

Bibliography:

[1] Edwards, W. Keith, *Core JINI*, Prentice-Hall Inc, Upper Saddle River, New Jersey, 1997
[2] Orfali, Robert et al, *Instant CORBA*, John Wiley and Sons Inc, New York, New York, 1997
[3] Park, Robert, *On Lacert Tax Returns*, private conversation, Jan 2001
[4] Simmel, Sergiu et al, *Metering and Licensing of Resources: Kala's General Purpose Approach*, www.cni.org/docs/ima-ip-workshop/Simmel.Godard.html, Nov 1997
[5] Stallings, William, *Cryptography and Network Security – Principles and Practice (second edition)*, Prentice-Hall Inc, Upper Saddle River, New Jersey, 1999
[6] Vogal, Andreas et al, *Java Programming with CORBA (second edition)*, John Wiley and Sons Inc, New York, New York, 1998
[7] Chamberworks Inc., http://www.chamberworks.com
[8] EE Times, http://www.eetimes.com (article on pay-per use e*ECAD, Nov 2000), http://www.eoenabled.com/edtn/out.asp?a=EET&i=e%2Aecad&n=33586385&tid=0&url=http%3A%2F%2Fwww%2Eeet%2Ecom%2Fstory%2FOEG20001228S0014&title=Extraction+tool+features+per%2Dhour+licensing
[9] Entegrity Solutions, http://www.entegrity.com, February 2001
[10] Face2Face Inc., http://www.f2f-inc.com , February 2001
[11] Inprise http://www.inprise.com or http://www.visibroker.com, February 2001
[12] Kiosk Software Inc., http://www.kioskco.com, February 2001
[13] Netshift Software, http://www.netshift.com, February 2001
[14] Netstop Kiosk Software, http://www.netstop-kiosk.com, February 2001
[15] OMG, *Security Services Specification*, OMG, http://www.omg.org/technology/documents/formal/corba_services_available_electro.htm, May 2000
[16] Pay-Per-Clik, http://www.payperclik.com, February 2001
[17] Prism Technologies, http://www.prismtechnologies.com, February 2001
[18] Programmer and Reference Material for Visibroker, http://www.inprise.com/visibroker/download, February 2001
[19] Saltmine LLC, http://www.saltmine.com, February 2001