# The Client Utility Architecture: The Precursor to E-speak

Alan H. Karp, Rajiv Gupta, Guillermo Rozas[1], Arindam Banerji
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-136
June 1st , 2001*

distributed computing, e-services, peer to peer

The Client Utility project at HP Labs was started to explore ideas needed to build a scalable, secure, and manageable peer-to-peer computing environment. The results of that research formed the basis for e-speak, an open source project that HP also released as a fully supported product. This report is the most complete description of the Client Utility architecture at the time the research work was transferred to the newly formed Open Services (soon to be renamed E-speak) Operation. It is reproduced here for its historical interest, showing the thinking that led to e-speak and the concept of e-services.

* Internal Accession Date Only Approved for External Publication
[1] Present address is Transmeta Corporation, Santa Clara, CA

# The Client Utility Architecture: The Precursor to E-speak

Alan H. Karp
Rajiv Gupta
Guillermo Rozas*
Arindam Banerji
Hewlett-Packard Labs
Palo Alto, California

## Abstract

The Client Utility project at HP Labs was started to explore ideas needed to build a scalable, secure, and manageable peer-to-peer computing environment. The results of that research formed the basis for e-speak, an open source project that HP also released as a fully supported product. This report is the most complete description of the Client Utility architecture at the time the research work was transferred to the newly formed Open Services (soon to be renamed E-speak) Operation. It is reproduced here for its historical interest, showing the thinking that led to e-speak and the concept of e-services.

* Present address Transmeta Corporation, Santa Clara, CA

## Introduction

Before there was e-speak[1], there was the Client Utility project at HP Labs. Work began in late 1995, and a first prototype was ready by March 1996. That prototype was refined over the next 6 months, and a set of use cases were put together to demonstrate the vision. Once a commitment to go beyond the prototype stage was made by HP management, the entire system was rearchitected based on what we had learned from the prototype. Eventually, these ideas were implemented as the first prototype of e-speak and announced to the world in May 1999. The motivation for, and ideas behind, e-speak have been described elsewhere[2].

As you will see from reading this document, the architecture describes a peer-to-peer system, one of the earliest to be based on this concept. One of the other ideas that form the basis of the architecture is that everything can be thought of as a service, leading to the e-services vision introduced by HP and that has

recently captured the imagination of the computing community as web services. The first figure, which we used as the logo for the project, shows examples of many of the features we were designing for. Today, we see the same features listed for peer-to-peer infrastructures and web services environments.

There are three principal ways that the Client Utility architecture anticipated the environment needed for peer-to-peer computing. The first was treating everything as a service or resource that could be linked into a dynamically composed value chain instead of using a specific web service from a client applet. The second was the need for a flexible way of describing services and a powerful means of finding them, vocabularies as discoverable resources. The third was recognizing the need for advertising services so that those looking for services could find them, even without knowing ahead of time who might be providing them.

As with many research projects, the Client Utility was rich in ideas but weak in documentation. This report is the most complete specification of the architecture. Most of the ideas were present in the open source releases through December 1999, although some pieces were left for later implementation. After that, the product was targeted at the B2B portal space, and substantive changes were made. However, many of the concepts presented here survived.

Since this document is to be treated as a historical document, only minor changes were made in producing this report. For example, the "Confidential" label was removed and dead web links were replaced with explicit references. No attempt has been made to finish any of the incomplete parts of the architecture or to update them with what we learned during the implementation. In particular, the issues of specifying security policy and interfacing to management systems are particularly weak. The failure limitation work is still a subject of active research. What follows is the document in its original form.

---

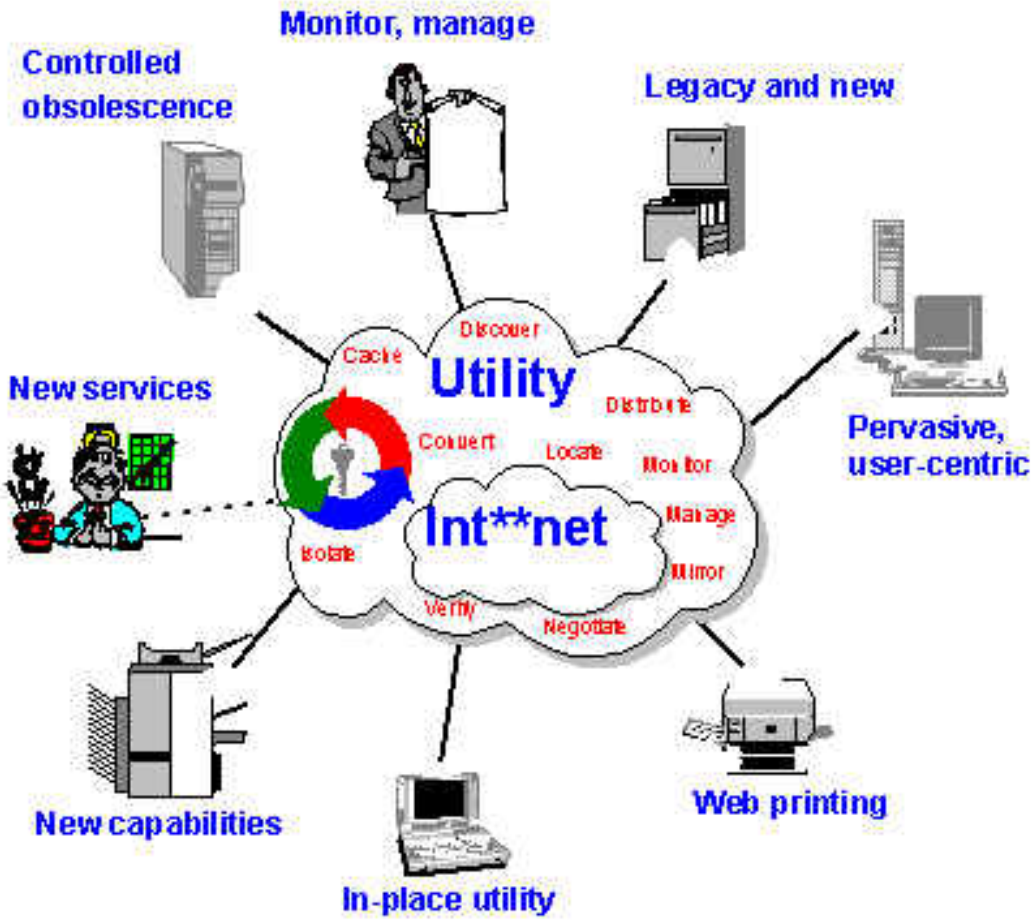
1. http://www.e-speak.net

2. "E-speak E-xplained", HP Labs Technical Report HPL-2000-101, available from http://www.hpl.hp.com/techreports/2000/HPL-2000-101.html.

DISCLAIMER

This document is a work in progress. It reflects our current thinking on the issues, but some of the details remain to be worked out. Expect frequent updates. If you find any errors, confusing descriptions, or inconsistencies, please contact the author.

The Client Utility



# Table of Contents



---

# 1. THE VISION

Why the world needs the Client Utility



---

## 1.1. Why Client Utility

Today at our homes and work we do not think about power, how it is generated or where it comes from. We simply plug into the wall and use the facilities provided by our local power utility company. Client Utility computing intends to create a paradigm for end-user computing where users simply plug into the wall and use the facilities, services and applications provided by their local utility. This form of computing is realized by a middleware infrastructure that facilitates ubiquitous/pervasive computing.

For information technology to become truly pervasive, it must transcend being merely manufacturable and commonplace. It must become intuitively accessible to ordinary people and must deliver sufficient value to justify the large investment needed in the supporting computation infrastructure. The consumer's expectations from the computation infrastructure or utility will be substantial. Just as people pick up a phone and expect a dial tone, so too will people expect the infrastructure to be available, ready and waiting.

In a high bandwidth, digital, multimedia world -- in which clients (general-purpose computers or appliances) connect to a utility - people could pay for their computing by usage, modulated by guaranteed response requirements. This enormous paradigm shift, changes what is now a capital investment into a competitive service, like electricity and water. It will do for computing what the Web did for data. It is not the ultimate revenge of time-sharing, which was proprietary, fixed resource, and usually location-dependent. In this form of computing, called ***Client-Utility Computing***, standards-based open resources, located arbitrarily, are combined as needed for a particular job. It will no longer matter where the computers operate or which manufacturer makes them.

The suicidal obsolescence schedule of today will be replaced by the capacity requirements of the service provider, with upgrades undetectable by the end-user within a given performance envelope. The trend towards such utilities is likely, since the end-user benefits from greatly decreased cost of ownership, the manufacturers gain a relaxation of insane time-to-market demands, and a new lucrative service industry should emerge. If this new model of computing becomes practical and reliable, history tells us that current manufacturers who do not adjust in time will not be able to satisfy customers and will suffer dire consequences.

---

## 1.2. Enabling Infrastructure

Client-Utility enables ubiquitous services over the Int**net (where "**" is "er" or "ra") - making existing resources such as files, printers, Java objects, and legacy applications, or for that matter any software or hardware resource, available as services. In this way, the Client Utility fundamentally lowers the barriers for providers of new services.

The infrastructure provides the basic building blocks for service creation, billing, management, and access - dynamic discovery of new providers and

capabilities, security of access to resources and data, negotiation, monitoring, and management of resources and data, replication to maintain reliability, caching to enhance performance, coherency and conversion of data, *etc*. These capabilities are enabled transparently for both legacy and new applications, OSes, and systems.

The architecture is built on a stringent security model and on a small number of fundamental abstractions. These abstractions are inherently extensible to allow for dynamic composition and customization. The implementation protocols have been architected specifically to scale cleanly from utilities that include a few machines to one that includes all machines on the internet.

The Client Utility is built on a scalable inter-machine protocol that seamlessly extends the resources available to applications to include remote resources, such as disk space, compute cycles, and database records, to name a few. There are a few important points to note about the infrastructure. First, it is a peer-to-peer protocol architected specifically for the kind of scalability required by the Internet and Web environments of today. All basic interactions in the infrastructure involve two individual machines at a time; there is no notion of multicasts or broadcasts in the underlying architecture, although these mechanisms can be implemented above it.

Second, the infrastructure reifies all resource accesses, thus making resource access protocols of any kind, such as file accesses or bank transactions, highly extensible. In short, resource access protocols become programmable entities that can be extended to add features of manageability, reliability, replication, accounting, payment information and the like, without in most cases affecting clients or resources. Third, the infrastructure does not prescribe the rewriting of applications to be beneficial in legacy environments. Applications such as Lotus123 or Microsoft Word seamlessly benefit from the infrastructure as long as Client Utility emulation has been integrated into the run-time environment. Fourth, the infrastructure acts in conjunction with existing operating environments and builds upon them, rather than replacing or modifying them. In fact, it builds the next level of abstractions above existing distributed environments and protocols, such as DCOM, COSS, HTTP, and Java RMI.

Finally, a word about security. Security is a critical deciding factor for any large scale distributed system, especially one that can have scale of the Internet. Here, malicious hackers, inadvertent configuration and implementation errors are the norm, not the exception. Issues of scale dictate that a single security policy for all users, dependence upon physical security of machines or centralized information banks are simply not viable solutions. The Client

Utility infrastructure ensures that each machine or user in a utility can have a different security policy, without allowing single machine security breaks to affect the security of the entire system.

Client Utility computing is the form of end user computing enabled by this infrastructure. In this form of computing, users have competitive but seamless access to distributed resources. Access to resources can be modulated by the user's requirements of cost, performance or functionality. No longer is a user tied to only resources on his/her machine or have to bend over backwards (thanks to the highly non-intuitive nature of existing distributed resource access protocols) to get access to resources from elsewhere.

Resource access does not mean owning the machine, configuring the resource (such as through file system mounting), or even managing the resource itself. All these activities can be automated and outsourced, thus opening up the possibilities for new types of service industries. It is important to note that such a paradigm does not require the users to give up control of their machines or have to compromise on the safety of their data. This fundamental change in end-user computing allows for the creation of paradigms where the users deal with computing and computing systems much like they deal with power utilities and paper. In short, users may pay for it and use it is as an integral part of their daily lives, but in general they do not notice or pay attention to its existence.

---

## 1.3. Scenarios

Gives examples of how the Client Utility changes the way people use computers.

### 1.3.1. Introduction to Scenarios

Client Utility computing makes possible many different scenarios which would otherwise only stay on our technology wish list. Many of these scenarios ameliorate existing ingrained problems, but quite a few of them open up the possibility of entirely new markets and businesses. In fact, there are few restrictions to the kind of utilities that may be built. For example, a printer manufacturer may want a printing utility to deploy and provide innovative printing solutions; a large enterprise such as a FedEx may want to deploy an enterprise utility to simplify (read lower the costs) of the management and administration of its IT infrastructure; a company like GM that needs large Web sites can significantly increase their flexibility and lower costs through a Web Utility and so on. The next few sections present utilities that could be of interest to various HP businesses and divisions.

### 1.3.2. Enterprise Computing

A whole enterprise or simply parts of it may use a utility for the IT infrastructure. Users in an enterprise get simultaneous access to a myriad of applications from different operating systems, transparently through the utility. Type "winword" on a UNIX machine and the utility will find the application, run it on an appropriate machine and forward the display to the end user's machine, without any prompting. Incompatible file formats from different versions of software are seamlessly handled; the utility uses attribute matching to find and handle differences between file formats.

System administrators may reset and move storage around for best utilization of network disks. Thanks to the utility this does not affect active users or running applications. Rolling upgrades of operating systems and application software do not affect any users and applications in the enterprise; the utility virtualizes resource accesses so that name transformations, or application availability can be completely hidden. During peak processing requirements or heavy business seasons, compute resources from outside vendors are automatically and seamlessly brought on-line to ramp up capacity. In short, a utility environment in an enterprise drastically reduces the cost of ownership without rapid obsolescence or complete rewrite of application programs. Enterprise Intranets and systems work as they are supposed to - smoothly - not as a disjoint system requiring immense amounts of caring and feeding.

### 1.3.3. Internet Service Provider

Internet service providers can change the very model of service that they provide to their users. ISPs provide users with a virtual personal computer. This customizable personal computer may contain applications from any number of different systems and the users may configure the environment, directories as they choose. No matter where in the world the users log in from they see the same set of files, directories, preferences, and resources. The ISP company dynamically maps this virtual personal computer to resources on a large number of different machines - changing this mapping as required by system loads, user location or network bandwidth.

None of this mapping and remapping affect the active users or running applications. So, a user may start a large computation on his virtual personal computer and fly to the next state, login and check on progress of his computation. Naive users do not have to worry about installing applications or configuring them. They simply boot their machines and use a pre-configured virtual computer provided seamlessly by their ISP utility. At the end of the month, users pay one additional bill - to their ISP utility. Simply, put none of this service model would be technologically feasible without having a utility infrastructure.

ISPs can also manage their plethora of machines more easily via the management infrastructure provided by the Client Utility. Centralized management is provided by the same mechanisms used to provide transparent access to users. All that is needed is to give the administrator a different set of permissions than those given other users. In addition, a hacker breaking into one of the ISP's machines will have a difficult time compromising more of the system.

---

### 1.3.4. Print Services

Print Utilities extend the notion of what a printer can do, how tightly it may be integrated with other compute resources, and allows printer manufacturers to think beyond the printer hardware to alternate commercializable print solutions. A large rendering job that reaches a printer is automatically forwarded by the Print Utility to the nearest machine with adequate horsepower. Such an operation cannot be implemented simply through remote management of

printers but needs a utility-like infrastructure to allow the printer to locate seamlessly remote compute cycles and subsequently access them.

A new non-standard format document reaches a printer, which upon not recognizing it uses the utility to find a resource on the Internet and to render automatically the document to a recognizable format. Travelers in an airport walk up to kiosk printers and identify themselves through a swipe of their smart credit-card. The printer utility contacts the traveler's home machine automatically, extracts all urgent email messages and displays them on a small display attached to the kiosk printer. The traveler points to one or more items and they are automatically printed out; charges being automatically sent to the traveler's credit-card. Here the print utility enables an entirely new business opportunity for printer manufacturers and resellers.

In a similar vein, a travelling executive could walk up to an Ethernet tap or phone line in an offsite office and hook up his portable. He opens up an appropriate document, suggests the kind of printing desired and prints it - without having the appropriate drivers or having to configure the appropriate printer queues. Here the utility locates the machine with the appropriate drivers, send over the document to it. That machine in turn locates the nearest appropriate printer and sends the print job to it.

---

### 1.3.5. Commercial Services

Utilities that allow for the on-line representation of business processes, inter-company financial transactions and other Internet services are very much in the realm of short term possibilities. Of course, many companies are marketing business management solutions, but each is a point solution to a specific problem. The Client Utility leads to an environment where such solutions are constructed by composing a number of services instead of writing a specific program. It is the Client Utility infrastructure that makes such compositions feasible.

The change is significant. When in need of parts, a computer manufacturer's inventory management program puts out an on-line request for bids with appropriate specifications for both the parts as well as the supplier businesses. Supplier businesses with the required credentials (supplied by an online certification authority) bid for the part dynamically. The utility ensures that the bids are matched with what the supplier companies claim to provide, uses an on-line negotiation service to provide a fair price for both. Finally, the

manufacturer's inventory management client contacts an online representation of a packaging service through the utility and arranges to have the parts delivered. Negotiation for services, contract maintenance, and payment support can be provided as available services, while the matching and mapping of bids and requirements, deployment of services, and real-time location of appropriate services is done by the utility itself.

This sample is only one possibility in the realm of commercial services utilities. It is also possible to use the same model to create new software businesses. A deployer of a new service in a commercial services (e-commerce) utility need only describe the components needed by the new service. If these components are provided by other providers in the Client Utility, the workflow completely specifies the service. When a customer request comes in, the service provider gets bids from its providers of the individual components, selects the appropriate ones to do the work, and returns the final result to the customer. In this way a software service is no different than a build to order computer made up from parts delivered by a just-in-time inventory management system.

The Client Utilty provides the infrastructure the provider needs to make money from the service; this includes service specification, dynamic installation and execution as a result of client accesses, and security. In each of the steps the utility plays a significant role. In the service specification step, the new service may use one or more pre-existing utility services; the utility here provides the basis for thinking and reasoning about value-added services. The glue for the installation step is the utility infrastructure's APIs, which automatically subsume advertising and informing clients if necessary. During the execution step, the utility actually acts as a composition service allowing for the interposition of payment and contract maintenance options, as well as allowing the new service to easily access/use pre-existing services. In every step of the process, the Client Utility protects the integrity of all the participants.

---

### 1.3.6. Web Management

Large, complex websites of today can be reorganized into highly manageable web utilities. An image processing request comes into a large web site; the utility automatically finds the least loaded machine and runs the image processing application. In fact, if the requestor of the processing needs a print out, the utility could find an appropriate printer geographically near the requestor (say at the nearest Kinkos) and automatically send it there after informing the requestor.

As requests increase beyond a certain threshold, the utility automatically moves the overflow to the computer center utility. To enable this the administrator simply had to modify the capacity constraints at the utility management console for the website. Adding machines, changing software, and failures are handled without affecting incoming requests. As machines come up and go down the utility ensures continuity and resource visibility. To support better availability for a particular set of files, the web-site administrator simply changes the replication policy at the utility management console. The utility ensures that the appropriate replication module gets interposed.

---

### 1.3.7. Compute Center

Compute center utilities rent out hardware to other commercial concerns in very novel ways. They do not rent out disks; they rent out disk space. Even more, they rent out file services such as version control, backups, *etc*. They do not rent out computers; they rent out compute cycles. Even more, they rent out reliable access to cycles by rolling work over to new machines as others get overloaded. They do not rent out memory simms; they rent out the use of RAM. Even more, they can rent out non-volatile storage for a fee.

A package handling company during Christmas season could automatically offload overflow computation to machines in the compute center utility. To the running applications that need these extra cycles, this is completely transparent. Due to a contract with the compute center, the package handling company's machines can seamlessly access the cycles on the compute center machines. Interestingly enough, if the compute center wants to ensure that the package handling company only uses 100,000,000 cycles/second and no more; it can do so through the utility infrastructure's fine-grained resource access checks.

Through similar mechanisms, small businesses can rent out out disk space with varying constraints, such as storage that is backed up once a week or remote storage access times less than 1ms, from a compute center utility. Such on-line outsourcing of hardware resources is not practical with current technology. In fact, a Client Utility can evolve from a conventional client-server environment. Here the existing infrastructure itself becomes a utility and finally links with an external computer center utility to get additional computation resources.

It is important to note that these utilities are not carved out from a disparate and disjoint sets of technologies. In fact, designing and constructing such utilities would be impossible without a structured way of thinking about the technology

for utility infrastructures. By solving such problems as naming, security, and location independence in the underlying infrastructure, the Client Utility makes a large number of new approaches feasible.

---

# 2. TECHNOLOGY OVERVIEW

Provides a brief overview of the technology and the key abstractions.



---

## 2.1. Introduction to the Technology

At the core of our technology lies the ability to virtualize resource accesses on host systems (which could be appliances, workstations, or large servers) and map these accesses to the uniform resource abstraction presented by a Distributed Resource Interchange Protocol (DRIP). Virtualization of resource accesses allows for such characteristics as mobility, distribution, availability, manageability, security, and reliability to be added seamlessly to the resource being accessed, without affecting the client applications.

Virtualization paves the way for domain specific protocols such as file access interfaces and the X-protocol to be mapped to a generic interchange protocol like DRIP. The use of DRIP as a protocol creates a highly extensible software bus that can connect any set of resource provider-accessor pairs without extensive *a priori* arrangement. This software bus is programmable and may be associated with properties such as secure access, negotiation for services, attribute-based location of appropriate services and replication to name a few. This generic interchange protocol allows fine-grained manipulation of inter-module (or client-resource provider) interactions. It also forms the basis for creating higher level resource and management abstractions such as the virtual personal computer or the web management utility, to name a few.

An application may choose to be directly aware of this software bus or be completely unaware of it. In case it is aware of the Client Utility software bus, it can exercise fine-grained control over the properties associated with the software bus. On the other hand, an unaware application can still use the facilities provided by the software bus. However, control and management of the properties must be done through management APIs (interactive or programmable). The control of these properties allows an application to negotiate for the cheapest service, locate the nearest service, ensure that its data is safe, replicate the use of certain services to ensure reliability, *etc*. There is also a third possibility depending upon the success of such utility infrastructures. The infrastructure may be subsumed into the local operating system, thus extending the nature and functionality of operating systems. This merger is a possible future step, but in no means necessary for the use and success of the utility.

The Client Utility architecture is a framework that can be used to describe, control, manage, and match the interaction between any resource provider and a client of the resource. The resource provider could be a virtual memory manager or a search engine - the principles used in the framework still apply. The client could be an OS's process management subsystem, a legacy NT application, a newly minted Java applet or an Active-X control - the intention at least is to provide appropriate mappings for most of the important categories of applications. This framework not only acts a "software bus" for controlling resource interactions but also acts as a substrate that facilitates higher-level resource aggregations and management abstractions, such as information utilities and Web-site control for large distributed sites.

---

## 2.2. Abstractions

The implementation of the framework may be viewed as an internet OS; it does for networked applications what standard OSs do for standalone applications. Just like standard OSs, it provides abstractions for building applications (files in standard OSs), provides a core set of services for all networked applications/services (comparable to memory management in standard OSs) and a set of powerful support software that aids in realizing a complete system (*libc*, *login,* and *inetd* in standard OSs). It is important to note that this Internet OS does not replace existing OSs such as NT, HP-UX or VxWorks, but instead acts in conjunction to provide an inter-machine protocol that allows for the creation of scalable ubiquitous services and interactions.

The Client Utility is based on just 5 fundamental abstractions. The fact that so few abstractions are needed makes the system manageable; the fact that so much can be done with them gives us confidence that we have chosen wisely.

Resource: Fundamental abstraction that may encapsulate any functionality or service that needs to be directly accessed across protection, machine, or geographic boundaries. Everything is a resource, be it a file or a complex combination of services, which clients access by name. The Client Utility associates meta-data with each resource. Access permissions and attribute-based descriptions are built into the abstraction as opposed to being disparate mechanisms.

Attribute-based specification: Abstraction that is used for resource discovery and lookup. Clients can easily specify the attributes and constraints for the resource lookups in a large-scale distributed system through this abstraction. They are not required to agree on names in order to work on resources, making it easier to deal with heterogeneity in a dynamic environment.

Resource Proxy: Abstraction that provides a contact point for managing and interacting with one or more resources. Acts as the advertising and controlling agent for a set of resources or services. Understands, or delegates responsibility for understanding, resource semantics simplifying the Client Utility implementation.

Client Interface: Client side abstraction that acts on behalf of the clients to provide a range of functionality such as error handling or dynamic compositions of resource accesses. This abstraction is geared towards simplifying the client program interactions in a highly componentized and distributed system.

Intermachine Agents: Entities that handle all inter-machine interactions. They implement the DRIP protocol, act as proxies for clients on the other machine, and look like local resource proxies to their machine, thereby hiding the distributed infrastructure from the Client Utility resource management implementation on a machine.

These abstractions dramatically simplify the model that implementors have to think about when deploying or building clients or services for a large scale distributed system. Also, they subsume a large number of issues such as resource discovery, security matching, caching, and error handling. Furthermore, the traditional ties that clients have to services and resources have been made explicit so that it is easier to implement extensions.

This approach allows for various ways of componentizing applications and services. The kinds of functionality that had to be built into clients or servers, such as conversion between two incompatible interfaces, can now be provided as separate intermediary services implemented by independent organizations. This degree of componentizing significantly changes how we think of application construction and service providers in distributed systems. All these drastically lower the barrier to creating, deploying, and using networked services in a large distributed environment - the fundamental goal of a utility.

---

## 2.3. Core Services

The core services subsume the kind of mapping, matching, and interposition needed to facilitate differing requirements and specifications of clients and services. This part of the infrastructure is intended to be a white box that provides a small but powerful set of services and can be configured/fine-tuned through appropriate management APIs. The core services include matching, mapping and binding of names, resource specific data such as permission sets, and resource attributes. The services facilitate a number of useful features.

Personalizable name spaces: Every client or service in the utility has complete control over what its name space looks like. Thus, to a client on an UNIX machine, Word97 could look like

```
/usr/bin/winword,
```

whereas the service actually exporting Microsoft Word from another machine in the utility could actually have it stored as

```
c:\winnt\system32\WinWord.exe.
```

The infrastructure does not mandate a global namespace, although one could be created using the abstractions of the infrastructure. In systems that could possibly have millions or billions of machines, this degree of virtualization of names is necessary.

Attribute based resource lookups: In large scale distributed systems, unlike single machine systems, specific resource names are often not an appropriate way for finding resources. In a single-machine system, knowing the name `/dev/fd0` is good enough to identify the exact resource. However, the list of resources that provide the same service in a large distributed systems is volatile

and may be very large. Hence, it is best to locate and find resources based upon descriptions of what the client needs. The Client Utility infrastructure achieves this end by supporting the resource abstraction mentioned above and incorporating attribute-based and constraint-based searches when a client binds a name to a resource.

Seamless interposition: Every request made for a service, regardless of type, it intercepted by the Client Utility enabling redirection and interposition. Thus, a service handler may create a proxy that is close to the client, an invocation stream may be forwarded to a transducer, or a client-specific workflow engine or error handler can be invoked. Thus, a display interposer could take the X-message protocol and display it on a handheld device that only understands a proprietary graphics device protocol. It is not just that the model allows interposition, but it can often be provided dynamically as a value-added service. Thus, an independent vendor could dynamically add a commercial service that transducts the X-protocol to Java AWT.

Secure Interactions : Security and safety of client interactions with resources and resource providers is ensured by the infrastructure. Capability based mechanisms, which have long been demonstrated to scale well, are used to secure access and use of resources. The infrastructure does not itself handle the chore of authenticating individual clients and users, but instead allows for "plugging in" various services that provide the required authentication and certification. The core services only recognize capabilities; conversion of passwords or smart-card accesses to capabilities is provided through the authentication and certification services, mentioned above. These mechanisms allow the core to restrict client visibility of resources to only the set that it has capabilities for or could get capabilities for indirectly. It is important to note that the infrastructure only provides the mechanisms for maintaining security; the actual policies are outside its domain. Configuring the permissions for resources in the right way makes it easy to support roles (Sometimes a user is a manager, other times an engineer.), compartments (When you are working on project A, you can't see anything related to project B.), military style security like that in the "Department of "Defense Trusted Computer System Evaluation Criteria", commonly referred to as the *Orange Book*. (Someone with Secret Clearance can not read a Top Secret document.), *etc*.

Introspection: Since every service request is intercepted by the Client Utility, the interaction between clients and services can be examined closely and manipulated by higher level APIs. A system administrator of a large utility may decide to change dynamically a file mirroring site, modify security policies, and so on. All these are enabled by allowing trusted entities (programs and end

users) to use the Client Utility APIs that deal with resource meta-data, such as names, permissions, and the like. It is expected that higher level management APIs that construct various management tools or aggregation abstractions such as the virtual personal computer, will be the main customers of these APIs.

The entities that the core services deal with are components of all application-specific (or domain-specific) interactions. These entities are dealt with by the core services in a generic manner, and consequently higher level management abstractions can reason about and manipulate client-service interactions uniformly though the use of introspection APIs. Also, the conversion to a generic interaction protocol relieves clients and services of dealing with naming mismatches, interface variations, and the like, that invariably occur in the interactions between independent components. In this sense, the OS not only acts as a substrate for building higher-level paradigms, such as utilities and virtual personal computers, but also simplifies the construction, deployment, and management of networked clients and services.

---

## 2.4. Support Software

Support software in the Client Utility architecture falls into three specific categories. Emulation software allows existing legacy environments to benefit from the functionality provided by the client utility middleware implementation. For example, emulation may enable Microsoft Word '97 to access seamlessly remote UNIX files. Similarly, NTFS files may be seamlessly exported to the utility and made accessible to client applications on non-NT machines.

General form a small list of meta-services such as metadata repositories, directory, and authentication services which are necessary to build complete systems. They allow the client-utility systems to bootstrap and use existing industry-wide standard functionality as far as possible. Domain specific services include all the services and functionality needed to implement specific forms of utilities. Hence, a utility supporting e-commerce could include contract maintenance services.

---

## 2.5. Intermachine Protocol

The Client Utility would be useful but quite limited if it didn't include a means to deal with other machines. In fact, the main reason to implement the Client Utility is the secure, transparent sharing of resources that it provides. A utility can be as small as one machine, but it has been designed to scale to the size of the Internet. A very large scale system can only perform well if it doesn't rely on any centralized services. Hence, the Client Utility is designed so that all connections are pairwise, all scheduling decisions are made dynamically at resource request time, and no connection hierarchy is imposed, although one can be constructed if desired.

It is unlikely that a very large utility will be made up of machines of the same type. Indeed, we expect a utility to be made up not only of computers of incompatible architectures, but to include devices of various capabilities. Today, these devices are likely to be printers and scanners; tomorrow they are likely to be cellular phones or even light switches. In order to deal with this diversity, the Client Utility only requires that communication between machines obey certain conventions. No machine ever attempts to look inside another.

If a light switch makes a request of a clock, there is no reason for the clock to know what operating system is running on the light switch, what user is making the request, what accounting structure the light switch uses to manage its billing policy. All the clock needs to know is that the light switch wants to know what time it is and if the clock has agreed to give the light switch the time of day.

An important implication of this decision is that a machine can choose how much of the Client Utility infrastructure it runs. If all it wants to do is use resources from other machines, it only needs sufficient code to send the proper requests and handle the replies. If it wants to let others use some of its resources without using the Client Utility security infrastructure, it can respond to requests as specified in the protocol. This approach also means that machines can be using different versions of the Client Utility protocol; all they need is a common dialect to enable them to share resources.

Since machines can connect to each other in a completely unstructured way, what constitutes a utility? We believe a utility is defined by the set of machines that share an administrative domain. Since, as we'll see, each machine decides exactly what part of its resources it will let other machines use, it can choose to

give administrative control over certain resources to another machine. A set of machines sharing a common administrator constitutes a utility. This administrative domain is completely independent of the way machines connect to each other. Machines within a utility may choose to share more resources with other machines in the same utility than with those in other utilities, but this decision is one of policy, not architecture.

The protocol used by Client Utility machines has several stages - connection, authentication, exchange of resource descriptions, use of remote resources, and failure limitation and recovery. Each of these components was designed to be consistent with the scalability, heterogeneity, and security requirements of a large, distributed system.

---

### 2.5.1. Connection

The first step in participation in the Client Utility is to find one or more machines to connect to. The list of machines could be provided by a system administrator, a list of participants could be kept by some directory service such as Yahoo, or the machine could broadcast a request. However it is obtained, the machine now has a list of other machines to talk to. Since all connections are pairwise, we need only describe a single connection.

The information needed to establish the connection is contained in a Connection object that was prepared by the machine being contacted. It specifies everything needed to communicate with this machine, such as communications mode (HTTP, TCP, UDP, ...) and contact port (if appropriate) as well as things about the machine, such as its endiannness.

A machine running the full Client Utility infrastructure will start a new task running in a very restricted protection domain, a sandbox that makes available only the resources the task needs. This task is only allowed to talk to the other machine over the agreed upon channel. All communication from a specific machine is mediated by its proxy. These proxies will negotiate a dialect of the Client Utility protocol that both machines understand. In this way machines running different versions of the protocol can still communicate. If they don't have a dialect in common, they will terminate the connection.

---

### 2.5.2. Authentication

The first thing the machines will do once they have determined a common dialect is to authenticate each other. What constitutes authentication determines the context. Two machines within an enterprise can prove to each other that they do indeed belong to the same company. Once that is done, they can share resources they wouldn't want seen by outsiders. If they are communicating over an insecure line, they can also establish a session key so that all messages can be encrypted.

In a commercial setting, authentication may not require identifying the machine or owner of the machine. Instead, it may make more sense to establish an *ad hoc* contract. For example, a customer may supply a credit card number or prove possesion of a digital wallet acceptable to the vendor. In these cases, the service provider is not authenticating the buyer, only the party guaranteeing payment, a scheme that greatly simplifies the model. It is also reasonable to defer the authentication, allowing the other party to see only publically available resources such as advertising.

---

### 2.5.3. Exchange of Resource Descriptions

Once the machines know who they are talking to, they can decide what resources they are willing to make available to users on the other machine. This list constitutes a policy determined by the administrator of each machine. Few resources will be exposed to machines that haven't established a high degree of trust; more resources will be shared with those that have.

The proxy on the machine that owns the resources will ask the Core to add name associations for the resources to be exported to its protection domain. It will then send a description of each resource to the other machine. When it receives a description of a remote resource, the receiving proxy will validate whatever part of the resource description it understands. Next, the receiving proxy will ask its Core to register these resource naming itself as the resource proxy.

When the exchange is complete, the sending proxy will have a protection domain consisting of the minimum set of resources it needs to do its job, all the resources imported from the other machine, and all the resources exported to the other machine. The Core will have added entries for all the resources its

proxy imported. Any use of these resources will be forwarded to the proxy as the designated resource proxy.

---

### 2.5.4. Use of Remote Resources

Tasks add resources to their protection domains by telling the Core the attributes of what they want. There is no need for the applications to know the source of these resources. They may be local, or they may be remote. If they are remote, any request to use one will be forwarded to the proxy for the machine that owns the resource. Even the Core is not aware that the request is for a remote resource. As far as the Core is concerned, the proxy is just another resource proxy. When a proxy executes a command on behalf of a task on the other machine, the Core sees a local request. This time the proxy looks like any other local client.

The proxy for another machine can be intelligent and partially process the request, or it can merely forward it across to its counterpart on the other machine. That proxy can also do some processing of the request, or it can attempt to execute the request. When it does, it talks to the thread that is acting as its client proxy for the Core. The request gets marshaled into the proper format and forwarded to the Core. At this point, the Core sees a normal request from a local task. It can do all the checking it normally does without knowing anything about another machine.

The proxies not only isolate the Cores from each other, but they also provide the opportunity to do some filtering. For example, it may be corporate policy to do certain functions only on machines within the enterprise. Since the proxy sees the payload, it can redirect the request to a proxy connected to an internal machine. In this way it acts like a firewall.

Some requests may involve resources from a variety of places. For example, if a task runs a word processor on another machine with a file from the first machine, we need some way for the word processor task to have permission to read and write the file. Clearly, the resource description was exported to the machine, but the task running the word processor needs to have a name for the file. We handle this situation by having the proxy transfer parts of the protection domain to the proxy on the other machine.

When a task wants to start a job on another machine, it transfers resources to the proxy for the other machine. (Of course, we use our security mechanisms to

make sure that the proxy doesn't get access to resources that shouldn't be available to users on the other machine.) The proxy forwards the request across the wire along with the resources the requester transferred as part of the request. The receiving proxy registers these resources with its Core and adds them to its protection domain. Finally, it executes the command. If the program needs resources from both machines, the two proxies must impersonate the requesting task.

A single proxy can represent many tasks by getting a number of distinct protection domains from the Core. Each one will have the set of resources common to both the requester and the machine running the command. The Core need not be aware of this schizophrenia of the proxy or any other task, for that matter.

---

### 2.5.5. Failure Limitation and Recovery

Components will fail. These may be hardware failures, such as network links and disk drives, but software failures, such as operating system kernel panics and memory leaks, are also common. It is important to discover such failures and limit their effects. Well-behaved components are expected to report failures they detect to a designated agent. This agent will make inferences about the cause of the failure using information collected from many sources. If this agent can determine the cause of the failures, it can direct components to take action to limit the spread of the failure. Well-behave components are expected to accept such instructions from their designated monitor.

As an example, consider a machine that provides a copy of a particular file that is used by a number of applications. If this machine fails, applications asking for the file will experience a failure. They may have an alternate source for the file, so they need not elevate the problem to the user. However, the extra overhead involved in identifying the failure may be substantial. For example, they may have to wait for a time-out to expire before going to the alternate source. If these applications report the failure, the monitor receiving the information can infer the source of the problem and tell other applications to go directly to the alternate source. Further, the monitor can tell the non-responsive machine to take corrective action such as rebooting.

The goal of the failure limitiation architecture is to use information from a number of sources to limit the affect of a failure in the system. In many cases, a failure that would be elevated to the user can be repaired without adversely

affecting running components. At the very least, the number of components affected by the failure can be limited.

---

# 3. CORE ARCHITECTURE

Architectural principles that apply within a machine

- 3.1. Introduction to Core Architecture: Introduces the nomenclature used
- 3.2. Walk Through: Stepwise introduction to the concepts
- 3.3. Naming: How tasks can share a resource without having a common name
- 3.4. Resource Access Control: How access to resources is controlled
- 3.5. Messaging: How a request gets from an application to a resource proxy and back

---

## 3.1. Introduction to Core Architecture

The Client Utility consists of a small set of services that communicate with user tasks and resource proxies via messages. These services are denoted by the term Core, a term chosen to avoid confusion with the term kernel often used to describe the trusted part of a conventional operating system.

The only services provided by the Client Utility Core are name resolution, extraction of resource specific data, message routing, and a means to monitor and manage the system. All other services are provided by resource proxies written to understand the management of specific kinds of resources, such as files, memory segments, processes, *etc*. These resource proxies can be separate user-space processes, but trusted services can run in the same address space as the Client Utility Core.

In order to avoid confusion with terms such as System, Machine, or Processor to designate the domain of control, we will use the designation Core. Resources within a Core have unique designations; each Core has a Repository listing all resources that it knows about; each Core has the 4 basic components.

Although it is most commonly the case that there is a single Client Utility Core running on a single machine, the architecture is more general. We can have one

Core running across several machines, whether they form a multiprocessor or are network connected. We can also have more than one Core running on a single CPU. This latter configuration is useful when we need to repair a machine yet present the view to the users that it is running.

This overview provides an introduction to the Client Utility architecture. Many of the special cases have been ignored in order to keep the discussion straightforward. In particular, we will describe only a single Core running on a single machine with only one CPU.

Fundamental to the Client Utility is the fact that virtually everything in the system is treated as a named resource. These resources include such familiar things as files, processes, and memory segments, but also may include resources that are not usually named. For example, it is common to name a data base, but the Client Utility also allows naming of data base records. The Client Utility also names resources that don't exist in other systems, such as name spaces and key rings. The Client Utility Core maintains a Repository which lists all resources known to this Core. Each resource has a name unique to this Core, a Core Repository Handle (CRH). A Core repository handle is never used outside the Core.

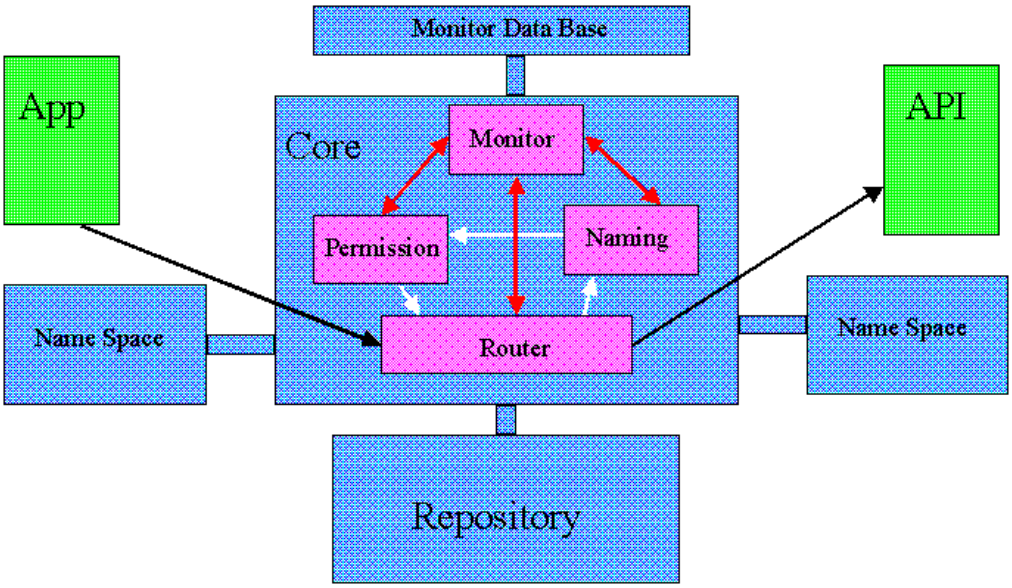
The Core can only deal with resources having entries in its repository. However, the Core, as a policy decision, may delete repository entries any time it wishes. Hence, a small machine can continue running even if it is too small to hold all resource descriptions exported to it. The Client Utility makes provisions for external agents to provide resource descriptions that the Core has chosen not to keep. Hence, an application referring to a resource description that the Core has deleted can still continue running, although it may suffer a preformance penalty.

Each request from a task for a system resource is made via a message from the task to the Core. A message consists of an outbox envelope, which is read by the Core, and a payload, which is read by a resource proxy, a task which will process the request. The router component of the Core constructs an inbox envelope and forwards the rewritten message to the resource proxy which is responsible for interpreting the message payload. A task can grant access to a resource to the recipient by including it in the message envelope. Unless there is strong authorization on the resource, a corresponding entry will be put into the inbox envelope. In this way, the sender can give the receiver a place to send replies; it includes a resource naming the sender as resource proxy. The same mechanism allows the sender to pass parameters to the receiver even though they don't share names for the resources.

Since the payload is defined by a convention established between a task and a resource proxy, we say that each resource speaks a specific language. (A language in this context is simply an API plus Client Utility specific information.) Thus, a file resource proxy speaks the file language; a scheduler speaks the scheduler language; *etc*. There is also a Client Utility Core language used to manipulate Core data structures. The Core **never** looks into the message payload unless the message is in the Core language. The Core knows nothing about languages other than the fact that they are unique within a Core. Any task can register a resource that represents a language and give itself ownership rights. The Core will make sure that this resource can be uniquely identified.

The Client Utility separates the concepts of naming and permissions. Each task works within a name space that it is free to define. Each name in this name space is bound to zero or more Core repository handles and an optional attribute description. Permissions are defined by the set of keys held by the task. Thus, access to a resource is controlled both by associating a task specific name to a Core repository handle and by controlling which keys are transferred to the task.

Three kinds of errors can occur when a task attempts to manipulate a resource. If no name association exists in the task's name space for the specified name, the task receives a Does not exist error. If the name exists but can not be bound to a repository handle, the task receives a No resource with that name error. If the name association exists, but the task's permissions indicate that it shouldn't see the resource, a Does not exist or an Access denied error is returned depending on the reason for the denial. Of course, the proxy responsible for this resource can return a variety of errors, as well.




The messaging is straightforward. Consider an application that needs a service from a resource proxy, say to open a file. It sends a message to the Core. The Router examines the message envelope to find the named resources. For each, it asks the Name Manager to find the repository handle associated with the name in the application's designated name space. Next, the Core repository handler looks up the resource metadata in the repository, including permissions that correspond to keys on the application's designated key rings. If all has gone well, the Router constructs an inbox envelope which it forwards to the proxy for this resource along with the payload of the original message. Meanwhile, the Monitor has been logging what's been going on.

---

## 3.2. Walk Through

A first look at the architecture

- 3.2.1. Introduction to Walk Through: How this tutorial is structured
- 3.2.2. Getting a Protection Domain: How a tasks initializes in the Client Utility
- 3.2.3. A Typical Request: The flow followed for every request
- 3.2.4. Structure of a Name Space: Giving a name space structure
- 3.2.5. Lookup Procedure: How the Core resolves the name association
- 3.2.6. Resource Discovery: Getting more resources into a protection domain

- 3.2.7. Advanced Access Control: Implementing enhanced security policies
- 3.2.8. Inheriting Resources: Making sure all needed resources are available
- 3.2.9. Positive and Negative Permissions: Controlling who can discover a resource
- 3.2.10.Bids: Declaring the cost of using a resource
- 3.2.11. Delegation: Getting help from another task
- 3.2.12. Resource Metadata: Core's view of a resource

---

### 3.2.1. Introduction to Walk Through

A simple walk through of the features of the Client Utility architecture will make it easier to understand the details. We'll use the example of opening a file since it's familiar to most readers. However, any access of any resource of any type will be handled in the exact same way.

For pedagogical reasons, we're going to hide details at each step along the way. We'll introduce them as the proper context is established. Two facts are needed to get started.

1. Everything managed by the Core is a named resource.
2. Every named resource has an entry in the Core Repository that carries information about the resource needed by the Core.

In addition, every entry in the Core Repository has a handle which we call its Core Repository Handle (CRH). Note that repository handles are unique within a single Core. No coordination of repository handles is needed between Cores, whether on different machines or on the same machine. A Core never accepts a repository handle from the outside.

The Client Utility uses a mailbox metaphor to describe the way applications make requests to access resources. The actual implementation may use mailboxes or not. In one prototype implementation, the messages are actually constructed by a thread running in the Core based on data passed from the application.

A task makes a request to the Core by putting a message in the task's outbox. This message consists of an envelope, the outbox envelope and a payload. The outbox envelope is defined by the Client Utility protocol; the payload is a

convention between the requester and supplier of the service. Because of this convention, we say that the message is in a specific language. The Core will never look at the payload of the message unless it is in one of the Core languages, languages used to manipulate resources owned by the Core, such as mailboxes.

---

### 3.2.2. Getting a Protection Domain

Any task can run on a Client Utility machine as long as it can do its job with the default resources. Hence, a typical Java applet will work without interacting with the Core. However, any task that needs more resources can get them only from the Core.

Most operating systems have facilities that allow the Core to start all tasks. In this situation, the Core will establish a default protection domain for every task. For example, users will log on using a logon task that the Core has started with this default protection domain. Communications with the Core are via anonymous pipes that the Core created when it started the task. The Core uses the pipe it reads from as the task's unforgeable identity.

A two step process is needed on operating systems on which the Core can not start all tasks. The first thing the task needs to do is contact the Core by connecting to the Core's portmapper. The Core will respond by returning to the task a port to use to contact the thread acting as the task's client proxy. (Note that these ports need not be socket ports; they are merely a means to exchange messages.) The Core will also return a token for the task to present on each request. This token is an unguessable 64-bit number used as the unforgeable identity of the task for the task's lifetime. The Core will associate a protection domain with this token.

There appears to be a security hole here because there is a time interval between when the port is assigned and when the task first uses it. A malicious task could come along and steal the port at this time. However, nothing is gained in this attack because any task will be given the initial resources, malicious or not. Once the connection is established, the token is used to authenticate any future requests.

The differences between these two means of connecting to the Core is hidden in the library routines provided on different operating systems. The client always uses the API to connect to the Core, and the library routine uses the

proper messaging scheme. In this way, a single application can run on Cores that implement different messaging layers.

The initial protection domain built by the Core will include exactly enough resources to allow the task to request more resources from the Core. The task will then begin by adding resources to its protection domain, either by making general requests or by authenticating itself and getting resources belonging to its account.

---

### 3.2.3. A Typical Request

Let's look at a typical request, say to open a file, made by a task that has been running for a while. I call this file `my.addresses`, but I may not want anyone else to know the name. After all, my name might be `the_boss_is_dumb`. Hence, I want file system to see an alias, say `the_boss_is_smart`.

Back to reality. The task will construct a payload in a language known to the file system, say

```
open read address.book
```

and an outbox envelope. One of the things the task puts in the envelope is a name field for the resource. A name field contains the task's name for the resource, `my.addresses`, and additional information described later.

When the message is delivered to the Core, it looks in the envelope to extract the task's identifier token and to see the name of the resource being accessed, `my.addresses` in this case. The task's identity, determined either by its communications channel or its token, is associated with a protection domain which, in turn, specifies a name space. The Core looks in this name space to find the repository handle associated with the task's name `my.addresses`. If no name association is found, an error is returned to the task stating that the resource doesn't exist.

If a name association is found, the Core looks in the Repository for the state information associated with this resource. One such piece of information is the resource proxy for the resource. This proxy is designated by a mailbox attached to a task that understands the language of the payload.

Another field in every repository entry contains resource specific data. This field has two parts, one private to the resource proxy and the other available to

all tasks. The private data contains any data the resource proxy may need to deal with requests. It usually includes a set of permission fields made up of a lock and an associated permission. Each request contains a list of key rings, each of which holds a number of keys. The Core will match up the keys on this key ring with the locks in the permission field. Any permission associated with a lock that gets opened will be forwarded to the resource proxy. Note that the Core never looks at the resource specific data, it just passes it on to the resource proxy. In our example, the core might extract the strings `read` and `write`.

The public part of the resource specific data field contains any data that the task registering the resource feels tasks might need to know. For example, the public field might contain a digital certificate identifying the supplier of the resource. It might also contain a piece of code, like a Corba stub, that the task can use to invoke an operation on the resource.

The Core will now construct a message to be delivered to the resource proxy. This message consists of the unmodified payload and an inbox envelope. The Core will put a name for this resource in the inbox envelope. In addition, if the outbox envelope specified a label for the resource, say `address.book`, the Core will insert that value into the label field of the inbox envelope. This label can be used by the task and resource proxy to mutually identify the resource.

The inbox envelope will tell the resource proxy the resource specific data and the name for each of these resource. The association between these names and the fields in the payload is part of the language specification agreed on by the requester and resource proxy. The semantic content of the payload and the resource specific data is the business of the resource proxy, not the Core.

The resource proxy can now do its job, except for one thing. How does it know what file in the underlying system the requesting task is talking about? Fortunately, there is no problem if the repository entry is configured properly. The resource simply gets registered with this information encoded in the field containing resource specific data. In our example, this name could be

```
/u/joey/addresses.
```

The resource proxy now knows that the request is to open file `/u/joey/addresses` and that the requester has the permissions associated with the strings `read` and `write`. It can now access the file system to open the file, but what does it do with the file handle? How does it get the handle back to the requester? It could ask the Core, but the Core doesn't keep any information about message-reply status. Instead, we use a different scheme.

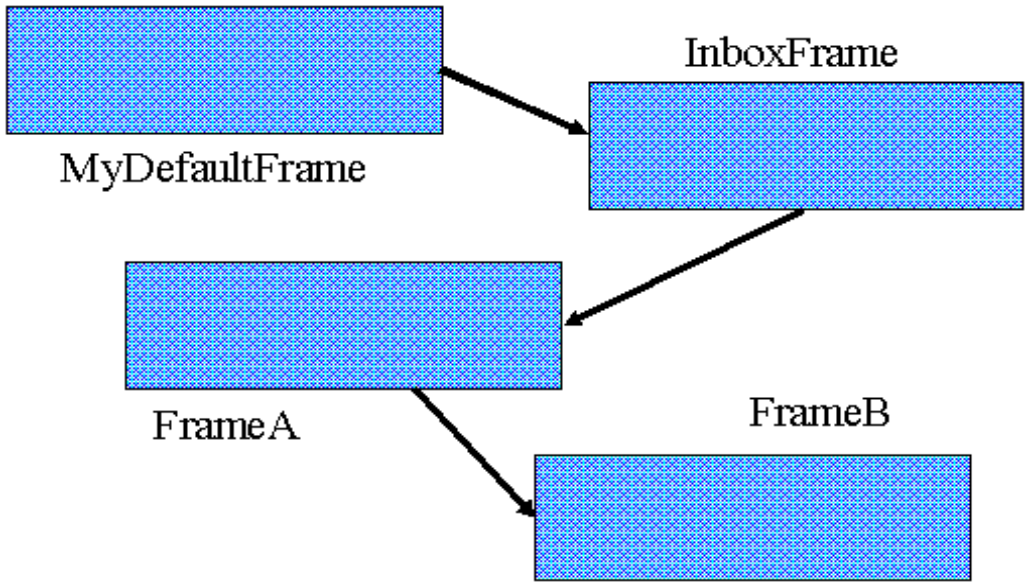
Another field that can be included in the outbox envelope constructed by the requester includes a name association to be transferred to the message recipient. Except for being used to find a resource proxy for the message, this name field is identical to the primary one. It has the requester's name for the resource and an optional label.

In our example, the requester would specify a resource for which the requester is the resource proxy. By convention, the resource proxy knows that the second name field in its inbox envelope refers to a resource to be named to send a reply. The resource proxy can now put the file handle into the payload of a message and the outbox envelope can specify the name associated with the reply resource as the primary resource. The Core will then use the same procedure as for the original message to deliver the reply.

---

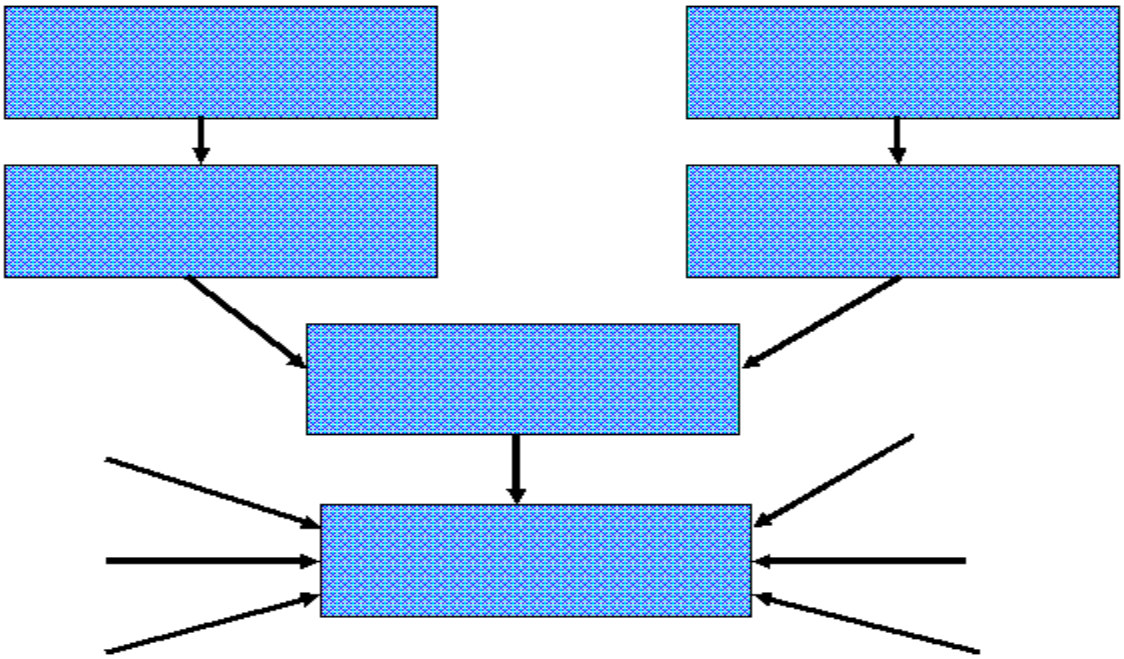
### 3.2.4. Structure of a Name Space

Since the Core decides what name goes into the recipient's name space, it can not be sure it won't specify a name already in the recipient's name space unless it searches the entire name space, too time consuming to do on every message. There is another problem. What if one task creates a file to be used by another task? Must one explicitly transfer a name for the resource to the other? The solution to both these dilemmas is the same - give the name space some structure.




MyNameSpace=(MyDefaultFrame,InboxFrame,FrameA,FrameB)

A name space consists of an ordered list of frames. Each frame contains the association between the task's name for a resource and a specification for the resource. Both name spaces and frames are named resources, so they can be manipulated in the same way as any other resource. Once a name association is put into a frame, the name is available to any task with permission to use the frame.



The sharing problem is solved by defining a frame that both tasks can use. This frame can be a global frame available to all tasks, which gives us the file visibility of conventional systems. A global frame is usually built during system start-up and put into the name space of every task. On the other hand, this frame could be shared by all tasks in a single login session. It is built as part of the login process and given to all tasks in the session. Finally, the frame can be constructed at the request of a task and shared by the usual means. Once both tasks share the frame, any name put into the frame is immediately available to both tasks.

We also use frames to avoid the name multiplicity problem. Each mailbox has a frame associated with it. When a message is delivered, the names of the resources being transferred are put into the frame attached to the mailbox. The recipient can use these names by including the frame in its name space, or it can copy the entries to another frame with the same names or different ones.

Frames are useful for handling other kinds of name multiplicities. If a task is collaborating with two users, say chuck and sally, it can decide which task's

names take precedence. By putting sally's frame first in the name space, a task can assure that it will see a name association in chuck's frame only if it doesn't exist in sally's.

A task can construct many name spaces. An entry in each name field in the outbox envelope is used to tell the Core the name of the name space to use for that resource. Hence, a set of resources can be hidden for the purpose of a particular message by not including the frame containing their name associations in the name space.

Building name spaces can be quite onerous if the task has to list explicitly every frame to be included. In particular, we can envision getting a large set of frames from a name mapping service. Building a name space by listing all these frames is prone to error and makes it difficult for the provider to make changes. We solve this problem by having each frame specify an ordered list of child frames. A task can now build a name space by simply listing a modest number of frames and a set of traversal rules, say depth first, and stopping criteria, such as not including child frames from certain sources. Once built, the name space can be used many times.

---

### 3.2.5. Lookup Procedure

All this is great, but where does the Core start the lookup? If everything is a named resource, where does the name recursion end? It turns out that this bootstrap is straightforward because each mailbox has an associated frame. We call the frame associated with a task's outbox its bootstrap frame. Some names are put into this frame when the task checks in. Among them are names for a mandatory key ring and a default name space. The task is free to change these names, but these resources will still be used to begin the name look-up. The mandatory key ring is presented to the Core on every request; the default name space is used whenever a name specification does not designate a name space.

When a message is delivered to the Core, it looks into the bootstrap frame associated with the mailbox. There it finds the RCHs for the mandatory key ring and default name space. This name space is used to find the list of frames that contain the name associations specified in the message envelope that don't designate a name space. The keys on the mandatory key ring are used to check the permissions to these resources. That's all there is to it. The Core can now search through the frames in the designated name spaces using the keys on the designated key rings.

We now see the only resource in the entire Client Utility that isn't accessed by specifying a name, the task's outbox. This resource can't have a name because the Core has no place to look for the name association. It is this feature that grounds the name lookup recursion.

---

### 3.2.6. Resource Discovery

So far we've acted as if the application's protection domain automatically contains names for all the resources the task will ever need. Clearly, this isn't the case. We need some way for a task to add name associations to its protection domain. One approach would be to ask another task to transfer the name association, which we allow, but we also allow a different approach.

Each entry in the repository has an optional set of attributes. A task can add name associations to its protection domain by asking the Core to give it associations for all the resources that have certain properties defined by their attributes. All resources that match the request get bound to a single name in the requester's name space.

Defining a single attribute vocabulary for all uses and all times is not a good idea, even if room is left open to extend it. Instead, we'd like to let the attributes, like the resource specific data and message payload, have a syntax and semantic content that the Core does not in general understand. We provide for this case by allowing any task to create a new attribute vocabulary that the Core will use when machine requests against resource attributes. These vocabularies are named resources and can have attributes of their own. The recursion is grounded by making a Core vocabulary available to all tasks.

A vocabulary consists of a set of rules. There are a set of name-value pairs. Associated with each name is a value type. For example, TITLE might be associated with a string while SIZE is associated with an integer. The vocabulary also contains a matching rule for each field. For example, a shoe size vocabulary might match a request for a 10C shoe with a 9D. It is also possible for a matching rule to denote other vocabularies that it can match. For example, a US shoe size vocabulary could match entries in a European shoe size vobabulary.

Since the Core will be using the vocabulary to match the attributes of resources registered in the Core repository, we can't let it run code provided by client tasks. Instead, we provide a toolkit consisting of basic data types, comparison

operations, and logical combinations. A vocabulary specification language is part of the Client Utility specification.

There may be times where we want to do a look-up in a different vocabulary. In this case, we can provide an attribute vocabulary matching and/or translating service. The request takes the form of a message that is routed to the task doing the translation with a payload containing the request. The translator can now return a new look-up request in the new vocabulary.

It is important for security reasons that information not be leaked between the owner of the resource that defines its attributes and the requester. We don't want a malicious task to know all the attributes on a resource because the information can be used for an attack. We don't want a malicious provider to garner information on the task's resource requests. In addition, we want the requester to decide what constitutes a match to protect against an unethical supplier that gets its resource heavy usage by saying it matches every request. On the other hand, we want the resource owner to determine what constitutes a match in case one of the attributes specifies a password. The Core, the only component that sees both the attribute description in the repository and in the look-up request, does the matching. The matching rules require that both the matching rules of the look-up request and those of the resource specify that a match has occurred.

Attribute descriptions can be included in a name association along with repository handles. Hence, a name association can be

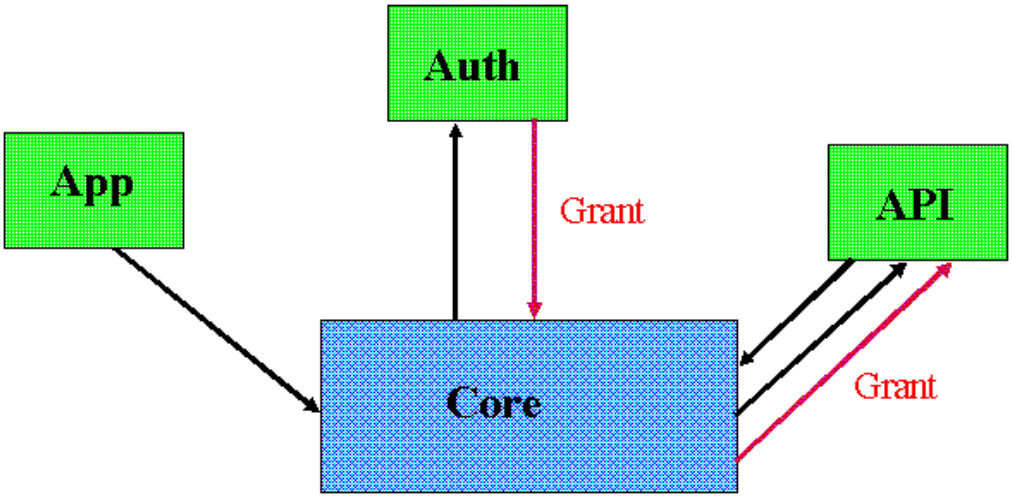
- Explicit: associated with one or more repository handles,
- Implicit: associated with an attribute description,
- Hybrid: associated with both repository handles and attributes,
- Unbound: associated with neither repository handles nor attributes,
- Partially bound: needs task to complete association.

If a particular type of association is needed in a request, the task must first contact the Core to return a new name association of the desired form. For example, the request could be to use one of the listed repository handles, but do an attribute based lookup if none of the repository handles is still valid. This new name can then be used in a subsequent request.

---

### 3.2.7. Advanced Access Control

What if we want some more advanced forms of security? For example, audit trails are one of the easiest and most critical steps in dealing with attackers. Also, you may have noticed that the Client Utility Core doesn't have the concept of authentication, not even something as simple as passwords.




We allow for these extensions by having a field in the repository entry that lists resources that act as authorizers. If the authorizer is just building an audit trail, the Core forwards a message to the authorizer when it puts a name association for the resource into the frame associated with the recipient's mailbox. We call this a notify authorizer. We can also designate the authorizer as being a grant authorizer. In this situation, a name association for the resource is not delivered. Instead, the recipient is told that delivery is pending. The partial association given the name does not allow the name to be used, but it does contain information on how to complete the association. Only a grant authorizer for such a resource can forward a name association for it to another task.

Authorizers let us implement some interesting functions. As noted, we can build an audit trail of what task was granted what name association when. We can also use a notify authorizer as an interface to a system monitoring tool. For example, if memory segments are treated as named resources, we can display the memory usage on a task by task basis by tracking the transfer of name associations to memory segments to the user tasks.

Here's a rather extreme possible use. Say that we make a time slice a named resource. When the scheduler swaps a task back in, it gives the task a time slice. If this task needs a service, say to do a message/reply with another task, it

can transfer a name association for the time slice along with the message. Making the scheduler a notify authorizer allows it to know that the message recipient should be scheduled next. The reply can send back the name association for the time slice. When the time slice finally expires, the scheduler can create a new time slice and give it to some other task.

Grant authorizers also have many uses. If a resource is to be protected with an access control list, the grant authorizer can assure that the resource is only made available to tasks in the list. The requesting task's identity can be determined in many ways, one being authentication carried in the message payload. We also use a grant authorizer to handle password protected resources; the grant authorizer gets notified when the partial association for the resource is to be completed. The authorizer can then check the message payload for the proper password. Actually, the authorization can be arbitrarily complex, including challenge response sequences.

Grant authorizers can also be used to synchronize tasks in a parallel program. For example, the authorizer registers a barrier resource with a particular attribute description. When a task is ready to wait at the barrier, it asks the Core to deliver a name association for a resource with the designated attributes. Since this resource has a grant authorizer, the name association is delayed. The requester sends a request to complete the binding and waits for a reply. Once the final task has requested a name association for the barrier resource, the authorizer can transfer the name association in a reply to all waiting tasks. The tasks now know that they can proceed.

---

### 3.2.8. Inheriting Resources

Some applications need a large number of resources to run. For example, a word processor might need a long list of font files for correct execution. The word processor might also use other applications for certain functions, such as graphics editing, that also need a large number of resources. It would be inconvenient to have to access all these resources individually. Instead, we can set up a field in the repository entry of a resource that lists the other resources that should be delivered to the message recipient. If this inheritance is recursive, the requester will end up with all the necessary resources just by requesting a single resource.

Inheriting a resource guarantees that the resource has a specific name in the frame containing the parent resource. Hence, the word processor not only

knows that the task has a name for the resource, it knows the name of the resource. Hence, programmers can write applications with hard-wired names. Note that the inherited resource can be a name frame that contains the names, which makes it easier to deal with a large number of resources.

---

### 3.2.9. Positive and Negative Permissions

Our security was pretty tight until we got to resource discovery. After all, if there was no name for a resource in a name space, there was no way to access the resource. However, resource discovery opens a hole. How can we construct a set of attributes so that only authorized tasks can get a match? We could put a required field in the attributes that must be specified exactly. However, relying on secret information is dangerous (what if the secret gets exposed), so the Client Utility uses a different scheme.

Every entry in the repository has a visibility field that holds two collections of locks. One is the allow field, and the other is the deny field. Whenever a task attempts to access a resource, either for resource discovery or as part of a message envelope, these fields are checked. If the specified key rings have at least one key that opens a lock in the deny field, the Core behaves as if the name is not bound to this resource. Then, if the key ring does not have at least one key that opens a lock in the allow field, the Core acts as if the task does not have a name association for the resource. Denial takes precedence.

It is now a simple matter to keep unauthorized tasks from discovering resources that need protection. Simply put a lock in the allow field and control who gets the key. Any task that doesn't have this key never knows that the resource exists. There isn't even an attack based on guessing attributes.

Using the allow and deny fields in combination makes it simple to enforce compartmentalization. This form of security says that you can not see items from one compartment when you're in another. For example, you can't see the submarine blueprints while looking at the bomber blueprints. All that is needed is make the allow key for the submarine be the deny key for the bomber and *vice versa*. The mere fact of presenting the key needed to see one of them makes it impossible to see the other.

The allow and deny fields also give us a way to implement roles. A role is used when a single task has many different jobs. For example, sometimes an engineer is also a manager. When acting as an engineer, only engineer

resources need be available; when acting as a manager, only manager resources are needed. While we can implement roles by having different key rings for each role, all the names for both roles would still be visible. However, it is possible to set up the allow fields of the engineer resources with one key and the allow fields of the manager resources with a different key. Now, when only the engineer key is presented, the manager names can not be accessed. In other words, we're dealing with just the engineer name space. On the very next message, the task can present just the manager key making all the engineer resources invisible.

The deny field has an important role in the security model. Recall that resource lookup begins by having the Core identify the task's mandatory key ring. This key ring is used on every resource lookup in addition to those specified in the message envelope. We can get very strong protection by putting a key on this key ring that opens a lock in the deny field of a resource we want to protect. By not giving the task a name for this key, we ensure that the key is presented on every access which, in turn, ensures that the task can never know that the protected resource exists.

---

### 3.2.10. Bids

Many resources can be used with no accounting being done. However, each resource use bears a cost that must be absorbed by someone. Often, this cost is bundled together with the cost of accessing the machine. Some resource use, however, must be accounted for separately. Pay-per-use software is one example of such a resource. If it is going to cost you to use a resource, you'll need a means to know the cost before deciding to use the resource. Also, if there are several providers of a particular service, you might want to pick the least expensive. On the other hand, cost isn't everything, and you might want to pick the provider with the best response time or the most reliable service. Factoring all these considerations into a resource access decision sounds like it needs a complex mechanism. Fortunately, we already have something in the architecture to handle it - attribute vocabularies. The bid field of the resource metadata contains an attribute description, just like one used to support attribute based look-up. Even the mechanism for determining a match fits the requirement. Recall that an attribute based look-up returns a match only if the matching rules of both the look-up request and the attributes pass the test. Such agreement is exactly what we need to make a contract, both the buyer and the seller must agree to the deal. The mechanism just described works for static bids, those that are independent of context. However, other bids need to be

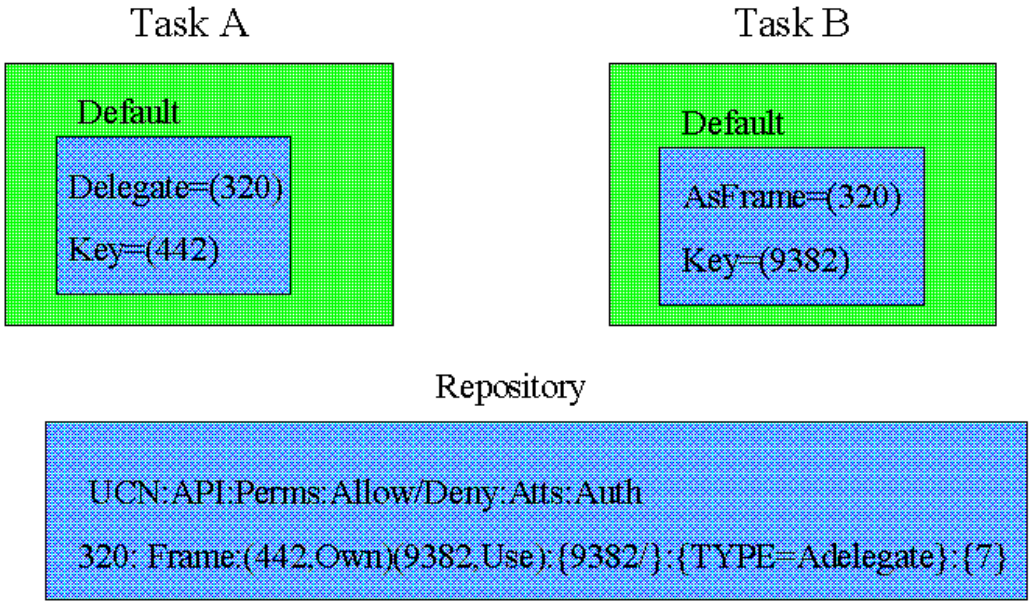
dynamic. For example, it might cost more to use a resource in heavy demand, such as a particular disk drive or CPU. Slowly changing bids can be dealt with by periodically updating the bid in the resource's metadata. Truly dynamic bids, such as one based on instantaneous CPU load can be handled using the features of dynamic attributes provided by the attribute vocabulary toolkit.

---

### 3.2.11. Delegation

There are times that one task needs to take on the abilities of another. For example, when a request involves a resource coming from the Core of another machine, the proxy on the machine that provides the resource must have the permissions of the original requester plus some of its own. We could have the requester transfer name associations for all of its resources, but the overhead would be large. Instead, we have the requester transfer a name association for one of its name spaces to the proxy on its machine. The proxy converts the exportable resources to resource descriptions which it sends to the proxy on the machine providing the resource. The receiving proxy registers these resources and executes the requested command in a name space containing the imported resources.

This scheme works quite well unless we want to be able to revoke the delegation, perhaps because the proxy isn't responding and we want to get the resource from another site. Revocation can work if the proxy trusts the requester enough to give it write permission to parts of its environment, but we can avoid this potential security problem.



When a requester wants to delegate authority to another task, it can set up a new frame containing a name space and a key ring. A name association for this frame is passed to the delegate as part of the request. The proxy can now use this name space and the key ring in making its own requests. The delegation can be revoked simply by unregistering the frame or removing one or more of the names in it.

We still have a problem. What if the delegate needs to use keys of its own for its requests? It could put these keys on the key ring supplied by the requester, but now the requester has access to the keys. The requester can include names for the keys on the key ring so the delegate can copy them to its own key ring, but now the access to the keys can't be revoked. The delegate can put these additional keys on its mandatory key ring since that key ring is implicitly used for all requests. However, this approach has too much overhead in moving keys on and off the mandatory key ring.

Our solution to this problem is to have each request include a list of key rings. The list can be empty, in which case only the mandatory key ring is searched. The delegate can now include the key ring specified by the requester as well as one or more of its own when making requests of the Core. Since any task can specify multiple key rings, the number of key ring manipulation requests to the Core will be reduced leading to more efficient operation.

---

### 3.2.12. Resource Metadata

Each resource managed by the Client Utility Core has an entry in the repository. If the resource managed by a Core is intended to be permanent across Core start-ups, such as a file, its repository entry is maintained in the persistent state of the Core. When the Core initializes, it first restores any existing entries fron its persistent store. Then, it creates a unique identifier, the Core Repository Handle (CRH), for each resource it manages. Resources supplied by other Cores also have repository entries. Some may be preserved across Core start-ups, but others may not be. Since the repository handle is arbitrary, it can be structured to make resource look-ups more efficient.

The preceding parts of this walk-through have introduced all the fields used by the Core to completely describe a resource. These fields are

- Language
- Resource proxy

- Resource specific data
- Attributes
- Security field
- Bid

The security field is made up of the

- Permissions
- Authorizers
- Visibility field

More detail is given in the detailed description of the Repository.

---

## 3.3. Naming

Describes the interpretation of names and the structure of a name space.

- 3.3.1. Introduction to Naming: Why names are local to a task
- 3.3.2. Name Spaces: Structure of a name space
- 3.3.3. Building a Name Space: How a name space is built
- 3.3.4. Name Association: What's in a name association
- 3.3.5. Name Multiplicity: Resolving name multiplicity
- 3.3.6. Name Visibility: Sharing a resource without sharing its name
- 3.3.7. Getting a Name Association: Adding name associationss for resources

---

### 3.3.1. Introduction to Naming

The Client Utility is designed to work in a distributed environment in which a task can be started on a different Core from the one on which it was invoked. In addition, a task that is written to be mobile can be migrated from one Core to another. It is critical that the task be able to name the resources it needs in such an environment.

In the Client Utility, each task is free to assign any name it likes to any resource. This name is the only means the task has to refer to the resource. When a message containing this name is sent to the Core, the Name Manager looks in the task's name space to see what resource is associated to this name. If

there is no name, an error is returned. Hence, there is no way for a task to tell the Core to do something with a resource that hasn't been put into its name space. Security is enhanced because there is no way to express an action against a resource a task is not allowed to see. You can't hurt what you can't name.

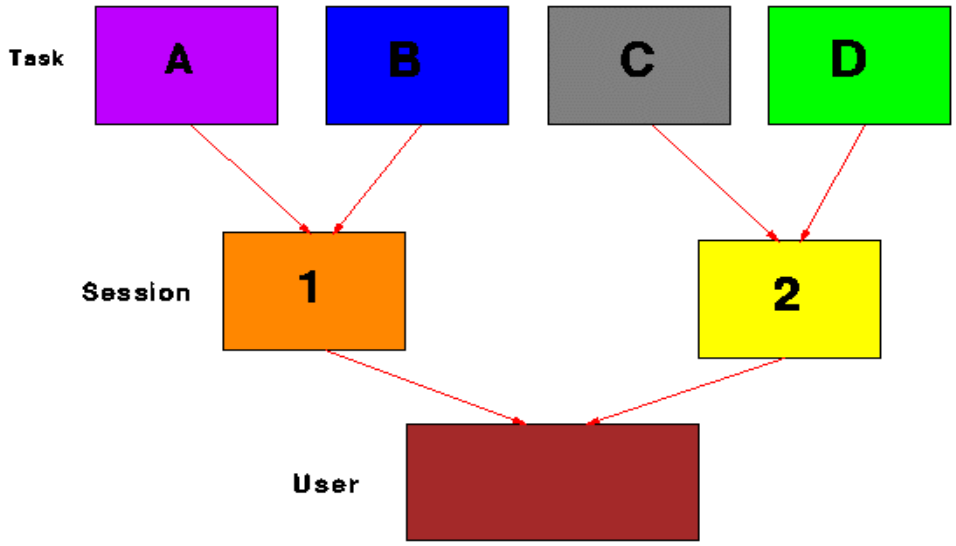
---

### 3.3.2. Name Spaces

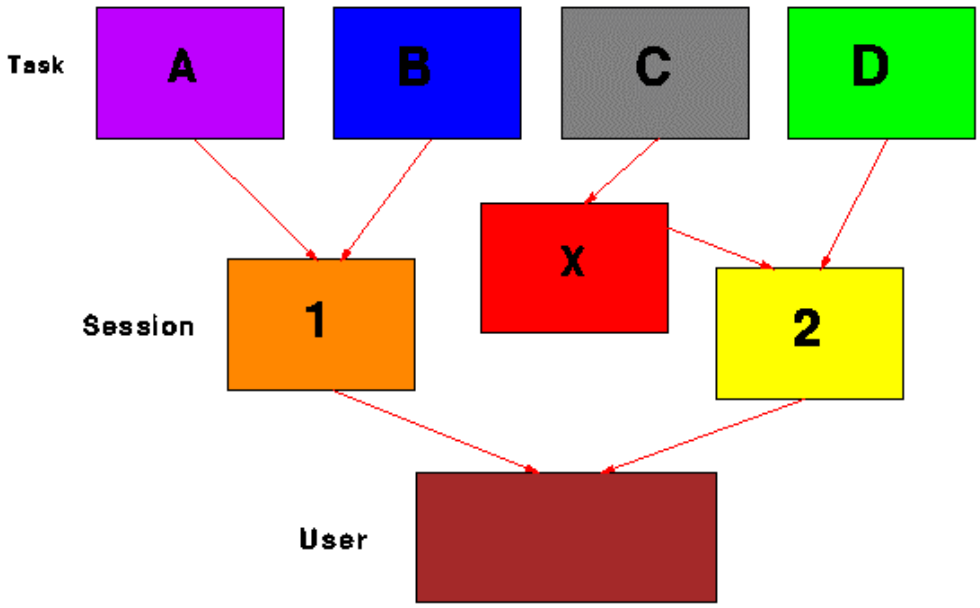
A task's name space consists of an ordered list of frames. Each frame contains a set of associations between the task's logical name for the resource and an association specification. The frames themselves are also named resources in a language owned by the Core which a task can connect into an ordering called a name space. Name spaces are also named resources in a language owned by the Core.

When a task requests an action on a resource it specifies its logical name for the resource. This logical name may optionally specify the logical name for a name space in which to resolve the name. If no name space is specified in the logical name, the name manager looks up the name in the task's default name space found in the task's bootstrap frame. It then searches the frames in the designated order looking for a name association. The name manager will stop when it finds the first association for the designated name. If this name is associated to more than one resource, the Core will invoke a mechanism for resolving name multiplicity.

Tasks can share names by agreeing on a convention, but the Client Utility provides a more flexible scheme. Since frames are named resources, one task can create a set of name associations in a named frame and give a name association for that frame to another task. That task can insert this frame into its name space and thereby share the name associations. The standard user naming scheme allows all tasks belonging to a given user to share a common set of names.




We can now see how these name spaces can be used. If Task A wants to create a new resource that only it can see, say a temporary file, it will put it into its Task frame. If it creates a resource that it wants to be seen by a sibling in the same session, say a shared memory segment, it puts the name into its Session frame, frame 1. This resource is now visible to both Task A and Task B. Those resources common to all tasks running on behalf of a given user appear in the User frame. This hierarchy can be extended to groups and global resources, but this extension is by convention only. Each task decides on its own name space.



Here we see that a task, say D, has created a new frame X, which has been put into task C's name space between its Task frame and its Session frame. Now, when C uses a name that is in frame X and its Session and/or User frames, the association in frame X will be used. More examples will be given when we walk through more scenarios.

A parent can control a child's name space by putting names for certain resources into the child's frames. In our example, task D might not want task C to access accidentally a particular resource in the User frame. One approach is to put the name appearing in the User frame into frame X but not associate the name with an association object. If task C doesn't have permission to change its name space, it won't be able to see the resource its parent wants to hide.

---

### 3.3.3. Building a Name Space

A name space is an ordered list of frames. It would be inconvenient to have to list all the frames in order to create a name space. Such a strategy would make it difficult to compose name spaces from components collected from others. We could compose name spaces from other name spaces, but, as we'll see, this approach is too static.

We make name spaces composable out of smaller components by having each frame specify an ordered list of frames as its children. A task can now construct a name space by specifying a list of frames and the traversal rules through its children. Traversal rules can include instructions like "depth first" or "don't look at children of FrameZ". Once defined, this name space can be used many times.

This approach lets someone provide name associations as a service using name associations provided by others as components. For example, Service A provides a set of name associations in a particular frame, call it FrameA; service B provides its associations in FrameB. Service X can now provide a unified set of associations by providing FrameX with children FrameA and FrameB. FrameX could contain any number of name associationss, but it need only contain name references to FrameA and FrameB. Any task that builds a name space that includes FrameX, either directly or as a child of another frame, will also see the name associations provided by Services A and B. Service X can later change the order of the child frames, add more, *etc*, without having to inform potential users of the internal structure and allowing different users to

traverse the tree differently. Simply sharing name spaces would lose this last bit of flexibility.

---

### 3.3.4. Name Association

What's in a name? Or more precisely, what's in a name association? The Client Utility supports 6 states for a name. It is an error to request an operation on a resource for which there is no name association.

1. Nonexistent: The name does not appear in the task's name space.
2. None: The name is not associated with any resource or description.
3. Implicit: The name is associated with one or more resource specifications but no CRHs.
4. Explicit: The name is associated with one or more CRHs but not a resource specification.
5. Hybrid: The name is associated with both one or more resource specifications and one or more CRHs.
6. Partial: Further action is needed to complete the name association.

An explicit name association is connected to one or more resouces by specifying an ordered list of repository handles. This type of association is used when the task wants to connect to a specific resource.

There are times when it makes sense to associate a name with a specification of the desired resource rather than to a specific resource. One such case is when the name is used to access a service provided by one of several different providers. As machines go down and back up, the application doesn't have to identify a specific resource on a specific machine. When the name is resolved, the Core will find any resources that match the specification.

The hybrid association connects the name to both a collection of repository handles and a resource description. Such an association allows the task to specify that the Core should first try to find a valid resource among those specified by repository handles. Should none be valid, the Core will then treat the name as if it were an implicit association. A hybrid association can also be used to tell the Core to update the collection of repository handles.

A primary resource must be bound to a single repository handle or a set of repository handles that all have the same resource proxy. If it isn't, the task will have to ask the Core to resolve the name before it can be used. Requests to

resolve the name carry a flag telling the Core how to interpret the name association. Recognized options are

- Pick one of the CRHs if available, description otherwise,
- Pick one of the CRHs (ignore descriptiong),
- Look up using attributes (ignore CRHs).

A second option indicates whether the Core should update the repository handles with resources that match the descriptions when doing a look up.

---

### 3.3.5. Name Multiplicity

There are times when a task needs to know that a single logical name refers to more than one resource. For example, the task may be trying to decide among a number of providers of a given service. We allow the task to see this multiplicity by associating a single logical name to more than one resource, both repository handles and resource descriptions.

Name multiplicity will be a common occurrence. For example, a user might well use the same name for a machine and the display attached to that machine. If the name is for a secondary resource, the resource proxy handling the message will have to resolve the name. However, the resource proxy can't be identified unless we know whether the payload is intended for the machine resource proxy or the display resource proxy.

It is the programmer's responsibility to make sure that the name association used as the primay resource in a request correctly identifies the desired resource or resources by asking the Core to produce a new name association specified by a designated arbitration policy. The allowed arbitrations are:

1. Use first resource,
2. Report an error if more than one resource,
3. Forward all to recipient,
4. Parameter field is a logic expression,
5. Parameter field is a task to be contacted.

In each case the designated policy can be applied to all resources in the name association or only those in a designated language.

The first choice is useful when a task wants to protect itself against accidental name multiplicity and makes the most sense for a explicitly bound name; the second, when it the name must be attached to a single resource. The third choice is used when the recipient is to decide what to do, as might be done with a mirrored file. In this case, it is an error if all the resources matched for the primary resource don't have the same resource proxy.

The last two choices allow the task to specify the selection via an algorithm. The logic expression option is used when the selection algorithm is very simple, such as making a random choice; the specification is built out of operations defined by the Client Utility. More general operations must be done by forwarding the request to a task, perhaps the sender itself, which will run code to make the choice. This method is used when we want multiple providers to bid for the work. In this case, the designated task gets a partial name association for the resource.

---

### 3.3.6. Name Visibility

If a task has no name association for a resource in any of its frames, it can't access that resource in any way. (You can't hurt what you can't name.) However, unlike most security mechanisms, this one provides an additional benefit to the user. A utility with a very large number of machines has more resources than any individual user wants to see. (Imagine asking for a directory listing on a million machine utility.) The user wants to see the resources relevant to the task at hand and nothing more. The name scoping provided by the Client Utility naming scheme gives the user the desired degree of control.

An important question remains. How do two tasks that don't share a naming convention talk about a resource? The answer is to use the Core to provide the name translation.

When two tasks wish to take action on a specific resource, they use the message envelopes to make the connection between their names. The Name Manager looks in the sender's name space to find the resource's association. The router puts a name and the label field given by the sender into the receiver's inbox envelope. It also puts this name association into the name frame associated with the recipient's mailbox. The router then delivers the message to the recipient. In order to avoid the problem of a task being tricked into using a name association that was transferred from another task, the Core makes sure that the this frame is empty before delivering the message.

### 3.3.7. Getting a Name Association

There are no naming conventions imposed by the Client Utility architecture, not even for tasks running on the same machine. How do tasks tell the Core what resources they want added to their protection domains? The answer is that it is done the same way you find a florist. Specify a set of attributes - close to my home, open at 6 PM, etc. - and look through a directory such as the Yellow Pages. The Core Repository serves the role of the Yellow Pages in the Client Utility.

The Core Repository is where a cache of the resource descriptions is held. Its internal structure is not part of the architecture. It must support attribute based look up, and we expect it to do fast look ups of resource metadata when given a repository handle.

Each entry in the Repository has a field containing a set of attributes. When a task asks the Core to add a resource to the task's name space, the task specifies the attributes. If a match is found, a name association for the requested resource will be added to the task's name space unless authorization is required or unless the name visibility restrictions are not met. In the first case, the name association will be partial until the authorizer does the transfer; in the latter, the lookup acts as if there was no attribute match. If more than one resource is identified that matches the request, the arbitration policy included with the request is used to decide with of these resources to bind to names for the requester.

Unlike most directory services, such as ORB Trader or LDAP, the Core Repository doesn't impose an attribute grammar. Instead, each request and each resource description include a designation of its grammar. The grammar is a named resource. Of course, there is a default grammar understood by the Core. If the request and description are in the same grammar, the grammar interpreter can decide if there is a match. If they are in different grammars, the request will fail to find a match unless it specifies a task that can translate the attributes to a common grammar.

## 3.4. Resource Access Control

Describes how access control is done.

- 3.4.1. Introduction to Permissions: Enforcing access rights without understanding them
- 3.4.2. Restricting Use of Names: Hiding names that exist in the name space
- 3.4.3. Authorization: Building audit trails and enforcing more strict access control
- 3.4.4. Key Management: Controlling access to keys
- 3.4.5. Security Level: Limiting which tasks can communicate

### 3.4.1. Introduction to Permissions

Having a name association for a particular resource is only part of the access control. The rest is what a task can do with the resource it has named. The Client Utility uses keys (not cryptographic keys) and locks to determine the permissions a given task has for a resource. Each message passed to the Core designates a set of keys, described by a set of key rings, that are used to determine the permissions.

Keys are named resources that are kept on key rings, which are also named resources. Keys are different from other resources in one important respect; mere possession of the key is sufficient to use the key. All other resources need an additional lookup to determine permissions. Of course, keys do have permissions for other operations such as unregistering, copying, *etc*.

Keys are used differently from other resources in that it often makes sense to put a key onto a task's key ring without giving the task a name for the key. This feature makes it possible to grant a task a set of permissions without letting the task forward them to another task. (You can't manipulate what you can't name.) Control of resources other than keys, including key rings, is managed with permissions.

Each resource in the repository has some resource specific data, including permissions, that may get passed to the resource proxy. This information has been inserted into the repository by the Core on behalf of the task that

registered the resource. Some of this data is always passed; other pieces are passed only if the requester has a key that opens the lock associated with that permission. The Core interprets these parameters only if the resource is in a Core language.

Consider the entry for a file for a Core running on top of a conventional operating system. The name of the file in the underlying file system is a parameter that is always passed to the resource proxy for the file system. There will also be permissions for different kinds of access, each with a different lock. For example, the permission that the file system will interpret as granting read access will have one lock, while the permission for write access may have a different lock.

Once the Name Manager has determined the Core Repository Handle for the resource specified in the message, it forwards the request to the Permission Manager. This component looks at the repository entry for this resource. First, it validates that the visibility rules are enforced. Next, it finds permissions with locks that match keys on the designated key rings. The resource specific data, including all the permissions with matching keys, get passed to the resource proxy in the order they appear in the resource specific data field.

In the most general case, each resource can have an arbitrarily large amount of resource specific data. Fortunately, there will be very little defined most of the time. For example, to provide Unix or NT file access semantics we need the name in the underlying file system, at most 3 permissions (read, write, execute) for the world, 3 permissions for each group, and 3 permissions for each user, independent of the number of files in the system. More commonly we'll need even fewer permissions. For example, a Unix file with access code 664 (rw-rw-r--) needs only 2 permissions.

Permissions are split into 2 sets. One set contains permissions associated with the resource; these permissions are meaningful to the resource proxy. The other set contains permissions associated with the resource metadata in the repository; these permissions are meaningful to the repository manager. They include such things as permission to change various fields in the description.

---

### 3.4.2. Restricting Use of Names

Keys can be used to hide names when it is convenient. The security field of each resource description has a visibility field made up of a set of allow and

deny locks. A name will appear to be undefined unless the request includes keys that open at least one lock in the allow field and opens no locks in the deny field. This behavior is in addition to the limitations imposed on lookup requests.

The allow and deny fields make it possible to control who discovers a resource. These visibility fields also make it possible for a task to control what resources are visible on a given request. For example, all resources related to the production version of a piece of software can have a particular lock in their allow fields. A test version can be run without this key which guarantees that none of the production resources are included by accident.

Another application is protecting critical resources, such as system configuration files. In this case, a particular key is put on the mandatory key ring of all general users. If this key opens a lock in the deny field of system critical resources, general users can never find out that the resource exists. If the general user does not have a name for this key, there is no way to remove it from the mandatory key ring.

Using both allow and deny fields makes it easy to implement compartments. If a lock appears in the allow field of one resource and the deny field of another, a task can never see the two in the same request. Either the allow key is absent, which hides the former resource, or it is present, which hides the latter.

---

### 3.4.3. Authorization

There are times when we need other kinds of control. For example, we may need an audit trail to track which tasks have had access to certain resources. Very secure systems may also want to control granting of keys and/or individual resources, perhaps to enforce access control lists or to implement military Orange Book type security.

A field in the Repository entry for a resource designates authorizers. For each, there is a bit to indicate whether the authorizer is to be notified on the transfer of a name association, a notify authorizer, or if only the authorizer can transfer a name association for the resource, a grant authorizer. The former is used when all we want is an audit trail; the latter, when we want tighter control over resource distribution.

When a name association of a resource with one or more notify authorizers is about to be put into a message recipient's mailbox frame, either as the result of a look up or because the resource is named in an outbox envelope, the message is delivered to both the appropriate resource proxy and to the notify authorizers. The name association is inserted into the frame associated with the receiving mailbox.

If the resource has one or more grant authorizers, the message is delivered, but only a partial association is put into the mailbox's frame and this fact is noted in the inbox envelope. This partial association contains a point of contact to complete the name association. Until then, it is treated as if the name does not exists. It is the responsibility of the message recipient to complete the name association before specifying the resource in a message envelope.

The authorization mechanism is general enough to support some functions that are normally part of an OS kernel. For example, some resources are password protected, but the Core doesn't know anything about passwords. Instead, a language owner can list an authorizer that will look in the payload for a password and decide whether or not to complete a name association for the resource. (An alternative approach is to start a task that registers itself as the resource proxy for password resources and send explicit messages to it.) Notice that this scheme also allows us to use an arbitrary challenge/response protocol to authenticate the requester because the names of the sender's and receiver's mailboxes can be exchanged.

The authorization mechanism can also be used to implement some useful functions not found in conventional operating systems. One such function is inheritance of resources. Say an application, such as a word processor, needs the user to have access to a number of files. We don't want to require the user to add explicitly each and every resource the application needs. If the names don't depend on the state of the system, we can use the inheritance field of the resource description to give the application names for all the resources it needs. Sometimes, though, the resources to be inherited depend on recently created resources. In this case, we can point the user to a single resource, an inheritance resource. When a partial name association for that resource is to be completed to the task's name space, the authorizer is notified. It can construct a name space, populate the frames with name associations to resources, and transfer a name association for the name space to the task. If any of the resources specified in the name space are inheritance resources, their designated authorizer will make sure the task gets name associations for the desired resources.

Most resources can be shared; there is no reason two tasks can't name the same file. However, there are other resources that we don't want shared. If we've started a task with a limited amount of memory, we don't want it to get access to more by simply starting a child process. Since the memory allocation is a named resource, we can control how it is used. When the parent starts the child process, it will transfer part of its memory allocation to the child. If the memory allocation is a named resource, a notify authorizer can remove a corresponding amount of memory from the parent's allocation.

If we're not careful when transferring name associations for aggregates, resources that contain other resources such as key rings or frames, we can lose audit trails and grant authorization. For example, if task A creates a frame with names for some resources and transfers a name association for that frame to task B, B can use any of those names with no audit having been done. Even worse, a grant authorizer might be enforcing an access control list that B is not on.

One solution would be to check every resource contained in an aggregate to see if it is authorized. The problem with this approach is the overhead it imposes on aggregates that don't contain resources that invoke authorizers. A better approach seems to be to make the aggregate resource itself one that needs authorization if it contains any resources that require authorization. When a name association for a resource that has a notify authorizer is to be put into another task's mailbox frame, the resource's notify authorizer is sent a message. The aggregate's authorizer can then forward messages to the notify authorizers of the individual resources.

If any of the resources in the aggregate has a grant authorizer, the Core will deliver a partial name association and a contact point. This contact point will complete the name association only if the requester is allowed to complete the name bindings of every component of the aggregate.

How does the authorizer decide whether or not to grant access? There are a number of possibilities.

1. The task asking for the name association to be completed can have previously registered with the authorizer. The request specifies a handle supplied by the authorizer that is used to deliver the name binding.
2. The payload of the request carries identification information, such as a password.
3. The authorizer and requester can go through a challenge response dialog.

The authorizer of the aggregate will do its job differently depending on which scheme is used. In the first approach, the request to the aggregate authorizer can include the handle; in the second, the identification information; in the third, the mailbox for the challenge. In each of the first two schemes, the requester will have to trust the aggregate authorizer not to misuse the information in the payload. This trust is not a problem because the aggregate authorizer is a component shipped with the Client Utility Core.

---

### 3.4.4. Key Management

Most tasks will have a single key ring, their Mandatory Key Ring. However, there will be times when one task will have to specify an alternate key ring. For example, a proxy for a resource being supplied to another Core will have to make requests to the local Core as if it were the task running on the other Core. A task may also want to construct key rings for specific purposes, such as running a command with a restricted or expanded set of permissions. In particular, a file can be protected from accidental erasure by removing the key corresponding to delete permission from the active key ring. Since keys and key rings are named resources, keys can be moved or copied from one key ring to another. Key rings can be forwarded to other tasks or their names added to or deleted from frames.

The mandatory key ring is special in one regard; its keys are checked on each message. This feature is particularly useful in controlling access to certain resources. If an unnamed key is put on a task's unconditional key ring, and that key appears in the deny field of a resource, there is no way for the task to access the resource; the system behaves as if the task were not allowed to name the resource. This mechanism can be used to restrict user access to a resource even when its password has been compromised. All that is needed is for a task with write permission to another task's unconditional key ring to put on it an unnamed key that opens a lock in the deny field of the resource to be protected. Since the task has no name for the key, it can't move it from the unconditional key ring, and can't get access by specifying an alternate key ring.

We can use the visibility (allow/deny) field of resources in the repository to implement some sophisticated security models without needing to involve authorizers. Compartmentalization is used to make sure that data from one project doesn't get mixed up with that from another. In general, we can access the resources associated with one project only if we are explicitly denied access to the other. Simply putting a key from the allow field of resources associated

with one project in the deny field of resources associated with the other projects prevents the user from seeing both simultaneously.

By properly defining the permissions associated with specific keys, we can easily implement military type, Orange Book, security. For example, a user with secret clearance can read both unclassified and secret documents and write both secret and top secret documents. Enforcing this so-called *-property is complicated in many systems but simple with the Client Utility. All we need to do is put an unnamed key on the user's unconditional key ring. If this key represents a secret clearance, it can be associated with read permission for unclassified and secret documents and write permission for secret and top secret documents. The Core doesn't even know that it's enforcing the *-property. Note that there is no overhead on systems that don't need to enforce this type of security, and almost no overhead, just another key to manage, on systems that do.

---

### 3.4.5. Security Level

The mechanisms just describe provide a means to control access to resources. However, there are times when we need to control messaging. For example, military installations put strict controls on the transfer of information. Control is so tight that the issue of covert channels with data rates as low as a 100 bytes per second are a concern.

The Client Utility provides a security level field in the resource's metadata. It enforces the required control by comparing the security level of the sending protection domain with that of the receiving mailbox. If the protection domain security level is higher than that of the mailbox, the message is not sent. The only way to communicate with a task at a lower security level is to lower the security level of the sender's protection domain. This explicit, auditable action is permitted by the military rules.

Note that every resource has a security level, but the level is checked only for protection domains and mailboxes.

## 3.5. Messaging



### 3.5.1. Introduction to Messaging

Sending a message is the only way for Client Utility components to talk to each other. We use a mailbox abstraction, but the implementation is not specified. For example, a mailbox can be just a memory buffer shared by some task and the Core. However, it may be more convenient to implement the messaging with pipes or sockets. The important thing is that a mailbox is a named resource.

There is one mailbox that does not have a name, the one used by a task to talk to the Core. (The Core can't do a lookup for this name since there is no way to get a message through.) Instead, a task's outbox is part of the task's fundamental structure. It can be a pointer to the shared buffer in a shared memory implementation, or it can be the handle to a pipe or socket. Regardless of how it is built, each mailbox has a frame associated with it.

A mailbox can be a persistent resource and can be named as the connection to a resource proxy when registering a resource. If the resource being registered is persistent, specifying this resource as the primary resource will result in a message being delivered to this mailbox. However, the task acting as the resource proxy is transient; it does not survive across reboots, for example. How then, does the Core associate a mailbox in the metadata to a task acting as the resource proxy?

Since the mailbox is a Core managed resource, the Core can act on messages listing the mailbox as the primary resource. When a task starts, it can ask the Core to connect the mailbox to the task. If the task has presented a key that unlocks the corresponding permission, the Core will build the data structure necessary to communicate with this task and will associate this data structure with the designated mailbox. From that time until the task ends or asks the Core

to disconnect the mailbox, messages sent to the mailbox will be received by the task. Messages sent to a disconnected mailbox are undeliverable, and an error is returned to the sender.

Only one task can be connected to a mailbox at any time. (It is technically feasible to connect many tasks to a mailbox to achieve multicast, but the semantics of doing so is not clear.) However, one task can disconnect the mailbox, pass its name to another task, and have that task connect the mailbox. In this way, a resource proxy that is having a problem can delegate its duties to another task.

Although three kinds of messages are sent, all messages are sent in the same way. The only difference is that some fields are needed for one kind and not another. There are messages to the Core, anonymous messages, and messages to a specific task. The first is used to implement Client Utility specific functions like creating name spaces; the second, most standard operating system operations such as memory allocation or file operations; the third, for direct message passing between tasks.

Some messages are meant only for the Core. For example, name spaces are resources that the Core manipulates. Declaring a new language, manipulating an entry in the repository, or creating a new key ring are other examples. Many fields in the standard message format are ignored in this mode.

The most common form of messaging is when the sender is unaware of the identity of the receiver. For example, the sender need not know the identity of the resource proxy for the file system when doing a file operation. The Core, knowing the Core Repository Handle of the resource, will forward the message to the correct task.

There are times when two tasks need to communicate with each other. For example, when a resource proxy needs to send data back to the requester. In this case, the sender transfers to the resource proxy a name association for a resource for which the sender is the resource proxy. Now, the resource proxy can send a message listing this resource as the primary resource. The conversation can continue if the reply transfers a resource to use for additional messages.

### 3.5.2. Outbox Envelope

The Outbox Envelope contains information relevant to the Core. It begins with a field that contains the unguessable token the Core passed to the task during the check-in process. It also has a name field for the primary resource, a list of name fields of key rings, the name field of a mailbox the Core is to use if an error occurs, and name fields for additional resources for which name associations are to be transferred to the recipient. The primary resource is used to determine which resource proxy is expected to process the request.

An outbox name field can contain a label and a name field for the name space in which to resolve names and the task's name for this resource. The name space name is also a name which can have a name space field. If no name space is specified, the name is resolved in the task's default name space. For convenience, a name field can also designate a name frame rather than forcing the task to construct a name space containing a single frame.

The Core first identifies the sender and associates the request with the corresponding protection domain. The Core then uses that protection domain to look up the task's mandatory key ring and default name space. These are used to look up the names specified in the outbox envelope that do not list name spaces. If the mandatory key ring or default name space can't be identified, an error is returned to the designated return mailbox. If this mailbox cannot be identified, the request is ignored. Once the default name space has been identified, the Core can look up the name of the primary resource. It is an error if the Core can not identify at least one repository handle for the primary resource or if all the repository handles it identifies don't have the same resource proxy. Finally, the Core looks up the remaining resources. It is not an error if one or more of these is not found.

If the name of the primary resource isn't found in the designated name space, an error is returned to the specified error mailbox. If one or more repository handles having the same resource proxy is found, processing is forwarded to the Permission Manager. If the association is to an attribute description, the Core will return an error message to the designated Core error mailbox. The task will have to resolve this name and try again.

If the name resolution identifies more than one suitable resource, arbitration will be needed. The arbitration *must* fully identify which resource proxy is to receive the message. Most of the arbitration policies can be evaluated by the

name manager. However, sometimes we need to ask a task to do the arbitration. For example, if the request is to run a job being offered by two providers, an external arbiter will collect bids and select a provider. It will then send a message to the original requester and include an association for the specific resource in the frame associated with the requester's mailbox. The requester can then try again naming this resource in a name space specifying this frame.

Note that the error mailbox specified in the Outbox Envelope need not be a mailbox belonging to the requester; it can belong to a designated error handling process. Several novel functions can be implemented by delegating the exception handling. For example, a task could understand an exception model different from that of the Utility; the designated exception handler would do the translation. Of course, if the handler doesn't understand the exception, it can put a message into a mailbox designated by the requester. Other scenarios are discussed later.

---

### 3.5.3. Inbox Envelope

Once the name translation and extraction of the resource specific data is complete, the router can construct the Inbox Envelope. This envelope contains an inbox name field for each name field listed in the outbox envelope. Each name field contains the name the Core put into the inbox frame, the label specified in the outbox envelope, the public resource specific data, and the private resource specific data if the resource has this task as resource proxy. Except for the primary resource, it is possible to pass a partial binding or a name bound only to a look-up request to the resource proxy.

The mapping between fields in the payload and names in the inbox envelope is part of the language spoken by the requester and resource proxy. Also, the resource proxy is the only task that needs to understand the semantic content of the private resource specific data, including the permissions.

---

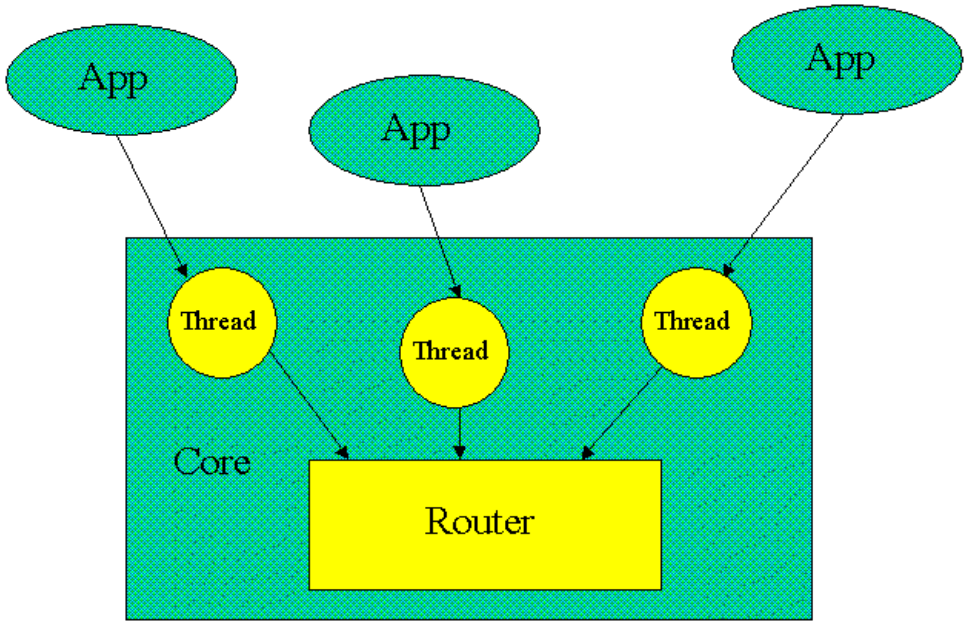
### 3.5.4. Message Delivery

The message will be put into the mailbox designated as the proxy responsible for the primary resource. A separate inbox envelope will be constructed for every notify authorizer listed for every name association transferred. We don't forward the same envelope to all notify authorizers because we want to limit

their knowledge of resources they are not authorizing. However, some of the authorizing functions will need to know what's in the payload.

Note that we always provide resource specific data for the primary resource and other resources that designate the same proxy. Otherwise, we wouldn't be able to handle requests like `copy foo bar` which requires that the resource proxy know the requestor has read permission for `foo` and write permission for `bar`. Private resource specific data for resources serviced by other tasks will be unintelligible to the resource proxy, so there is no reason to send them along.

Private resource specific data gets forwarded if the resource proxy is the same as the primary resource, not if the languages are the same. This way a single proxy that understands multiple APIs can handle requests involving resources in different languages. In our example, `foo` could be a Unix file while `bar` is an NT file. Since the resource proxy has registered itself as the handler for both, it understands both APIs, including the meaning of the resource specific data. By extracting this data for all resources sharing a handler, we guarantee that the task owning the resources gets the information it needs.




One key aspect of the architecture is the way tasks communicate with the Core. There is a lot of information in the message envelopes that the application may not want to deal with. Legacy applications may not even know that this format exists. Also, the internal structure of the envelope may change with time. We don't want to require the applications to change when this happens. Our solution is to have the application talk to a thread in the Core that marshals the message into the proper form. Hence, each application talks to a client proxy.

The Core associates the protection domain established for the task with its client proxy. When a task first checks in with the Core, it specifies its dialect of the protocol, and a thread that speaks that dialect is started on its behalf.

---

### 3.5.5. Events

Events are a special kind of message. They tell the recipient to look at the message now instead of later. The recipient may choose to defer any action, or even to ignore the event entirely, but the task is supposed to decide immediately.

We can consider the arrival of any message to be an event. The corresponding action is to put the event on the end of the incoming message queue. Since incoming messages are normalling processed by a thread running in an event loop, we only need a flag on the message to tell this thread to take a specific action other than enqueing the message.

The Client Utility messaging model provides a particularly effective way to manage events. Recall that a task wishing to receive messages creates a reply resource naming itself as the resource proxy. It can also provide resource specific data to denote an event. If the task wishes to control who can send it an event of a certain type, it can put the event information into a permission in the reply resource metadata. Now, only tasks holding the key that unlocks the event have the ability to send this event to the task.

The event itself can be a callback to a routine in the task's address space. The thread running the messaging event loop can start another thread to run the callback when an even arrives.

Events are most useful if tasks can subscribe to them. The Client Utility supports this model by defining a set of resources called distributors. Each distributor accepts subscribe and unsubscribe requests. Each subscribe request specifies a particular event and a set of rules specifying the conditions under which the distributor should send the event to the task. These rules are specified as a set of attributes in an event vocabulary. Each event that arrives at the distributor also carries a set of attributes in this vocabulary. The distributor applies each subscriber's filter to the event's attributes to decide if the event should be published.

---

# 4. REPOSITORIES AND PERSISTENCE

Storage mechanisms for registered resources and finding new resources



---

## 4.1. Introduction to Repositories and Persistence

In order to have a resource managed by the Client Utility, the resource must be registered. The Core will record metadata for this resource in the Core repository and assign it a repository handle, a CRH, which is unique to this Core. The metadata includes all information the Core needs to enforce the policies specified by the task registering the resource. Among these are the task responsible for managing the resource, a specification of the security restrictions, and an attribute description so tasks can discover the resource.

Some resources, such as files, are persistent; their resource descriptions should be persistent across machine reboots. Other resources are transient, such as open socket handles; their resource descriptions need not be saved across machine reboots. When a resource is registered, it can be declared persistent. The Core will make sure that its description is kept in non-volatile storage of some sort, most likely disk. Descriptions of transient resources may be written to disk, for example when the memory cache of the repository gets full, but they need not be.

## 4.2. Core Repository

The Core repository contains everything the Core knows about resources. This repository is accessed on every resource use, every look-up, every time a resource is registered, every time one has its metadata removed, and every time a resource metadata is changed. These are also the only operations supported by the Core respository.

Most implementations will support an in-memory repository. This approach works for transient resources, those that are not expected to be reinstantiated when the machine reboots. For example, open sockets accessed as named resources will not survive a power failure; there is no reason their resource descriptions need survive. Other resources, such as files, are supposed to persist across machine failures. This persistence is maintained by backing the Core repository with permanent storage. Unlike most systems, the Client Utility does not select some particular form of storage, such as disk. Instead, the backing store is a resource like any other. Most often, this resource will be a disk, but it could be a service provided by some other machine. Since the backing store is a service, it can invoke another service it may need. Hence, we can get hierarchical storage management by having each level invoke the appropriate service.

One issue involving persistence is how the Core repository deals with resources from another machine that have been declared persistent. What should be done if the exporting machine fails or becomes unreachable? If the proxy for that machine takes no explicit action, the resources will stay registered. An access to any one of them will be rejected by the proxy when it finds the machine supplying the resource is unavailable.

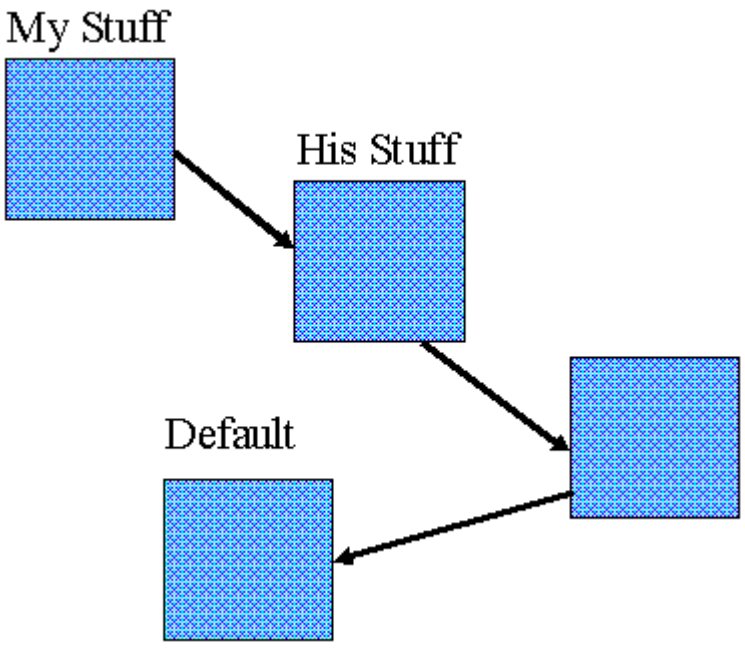
What happens when the exporting machine reconnects? It is quite likely to export its resource descriptions. The Core repository has a very simple way to deal with this problem. If there is a resource with metadata already in the repository that is identical in every way to the one being registered, including having the same resource proxy, the request is ignored. The reason is simple. If the resource were registered again, and one tasked accessed it through the first entry and another task accessed it through the second, the exact same thing would happen. Since the results are indistinguishable, there is no reason to register the same resource again.

## 4.3. Repository Views

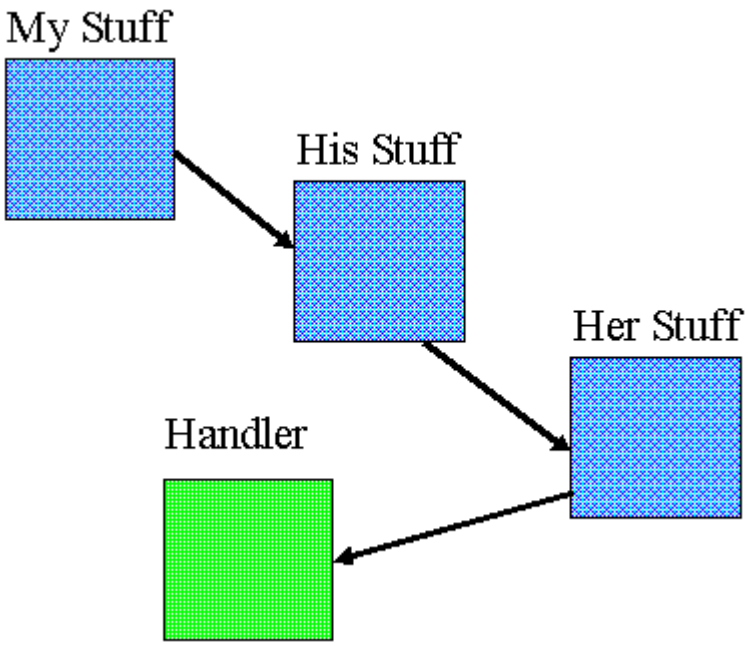
The Core repository can become quite large, and searching it for resources can become slow. This problem is addressed by allowing any task to define a view into the Core repository, a repository view. A repository view is a resource like any other and is accessible by specifying its name. By convention, every task is given a name for the default repository view which is a view that includes everything that has been registered with the Core. A given resource may appear in any number of repository views, but its resource description is always contained in the Core repository.

Searches can be bounded by specifying a repository view with a limited view of the Core repository. For example, a task could create a repository view and populate it with TrueType fonts. Any application looking for such fonts would only have to search this repository view for its fonts. As we'll see, repository views can also be used to manage different degrees of persistence as well.

Repository views are more than an optimization. They provide a way to avoid accidental matches on look-ups, a particular problem in the Client Utility because there are no global standards for attributes. Resource owners may give any attributes they like to the resources they register. I can be assured that I will only find resources from my own set by naming only a repository view constructed for such searches.



Repository views are also useful in dealing with resources from other machines. My machine will export resource descriptions to your machine and ask the proxy acting on my behalf to add them to its own repository view. If my application running on your machine specifies only this repository view, it can be assured that it will find a resource matching its requirements from those that came from my machine, not from those on yours. This way the application won't have to worry about an accidental attribute match to an unrelated resource; it simply searches the repository view containing the set of attribute descriptions it wants to use. Specifying an ordered list of repository views is similar to specifying a search path for executable files, a familiar and useful construct.




We can't let repository views grow without bound; a machine with limited resources will need to limit the amount of data it accepts from another machine. Also, one machine may not wish to export all the resource descriptions it is willing to make available. Instead, it might wish to have these descriptions pulled when needed. The Client Utility supports these needs by allowing each repository view supply an extended look-up handler.

If a look-up request fails, no name association can be returned. However, if any of the repository views searched has an extended look-up handler, a partial association will be returned. The contact point to complete the binding will be the set of extended look-up handlers belonging to the specified repository views. The client can ask any or all of the handlers to complete the name

association. The look-up request can include a time-out as a hint to the extended look-up handler.

---

## 4.4. Persistence

There are many classes of persistence. Some resources, such as files, survive across reboots. Other, such as socket connections, don't. Some will survive across machine disconnections; some across logins. Other resources will exist for a certain amount of time and others for a specific number of uses. While we could include a persistence field in the resource metadata, the Core repository understands only two classes, persistent and transient. Other persistence classes are left up to the task registering the resources.

When a resource is registered with the Core repository, it can be declared to be persistent. In this case, the Core will guarantee that the resource description will be associated with the same repository handle until the resource is explicitly unregistered. If the resource is not declared persistent, the Core need not restore the entry the next time the Core starts. Note that if the resource description was written to non-volatile storage as part of the repository cache management, the description of a transient resource may still be restored.

Other persistence policies are managed by the task registering the resource. For example, a proxy for another machine may register the resources it imports from another machine as transient. Should the importing machine fail, the resources will be registered with different repository handles the next time the machines connect. Any name associations to the old repository handles will be invalid. On the other hand, the proxy could register the imported resources as persistent. Now the repository handles will be the same after the importing Core restarts. Should the other machine fail, the proxy, at its discretion, can unregister the imported resources. Should it do so, all name associations to these resources will be invalid. Hence, the proxy may attempt to reconnect without first unregistering any resources.

The simplest way for a proxy, or any task for that matter, to manage a collection of resources is to add them to a repository view. Now, the entire collection of resources can be deleted by telling the Core to unregister all resources contained in a given repository view. Of course, the requesting task must have unregister permission on each resource being removed from the repository.

## 4.5. Attribute Grammars

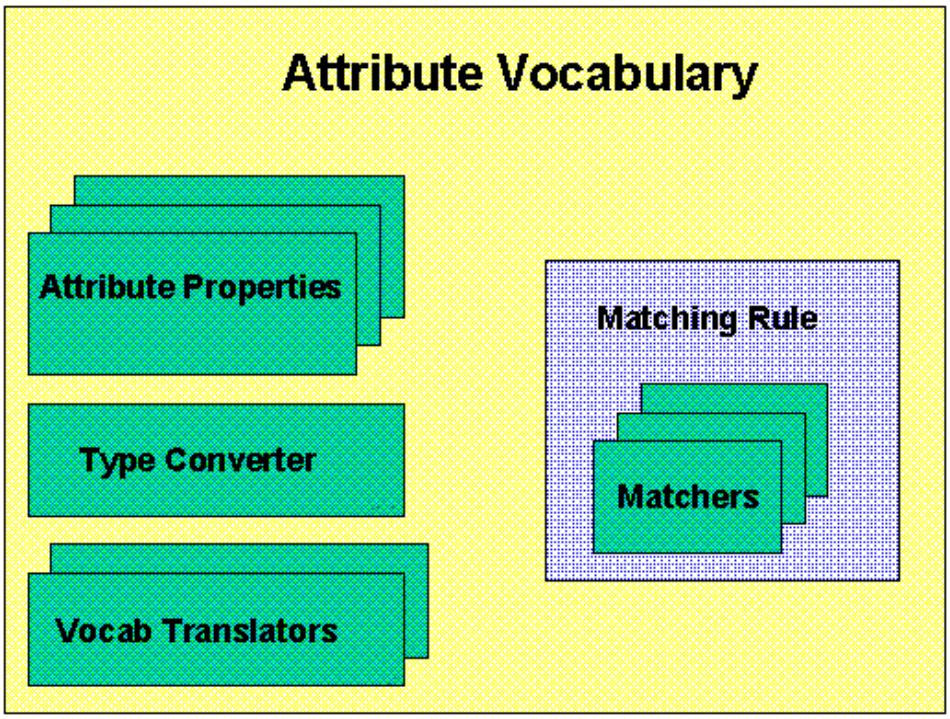
Identifying resources in a large scale distributed system is a problem. One solution is to provide a global name space as is done for URLs. In this scheme, the names are partitioned into a strict hierarchy with a well-defined entity controlling names issued at any point. Network Solutions, Inc., for example, is responsible for handing out top level domain names such as `hp.com`. In turn, whoever owns `hp.com` gives out names such as `hpl.hp.com`. This procedure continues down to the level of individual files.

While global name spaces are convenient, they have some problems. First of all, they require that everyone conform to the naming convention, a problem when different operating systems have different rules for forming names. Secondly, name resolution is a problem. To find a name within `hp.com` a user at IBM would have to traverse the tree to `.com`. Heroic efforts at caching have ameliorated this problem but not eliminated it.

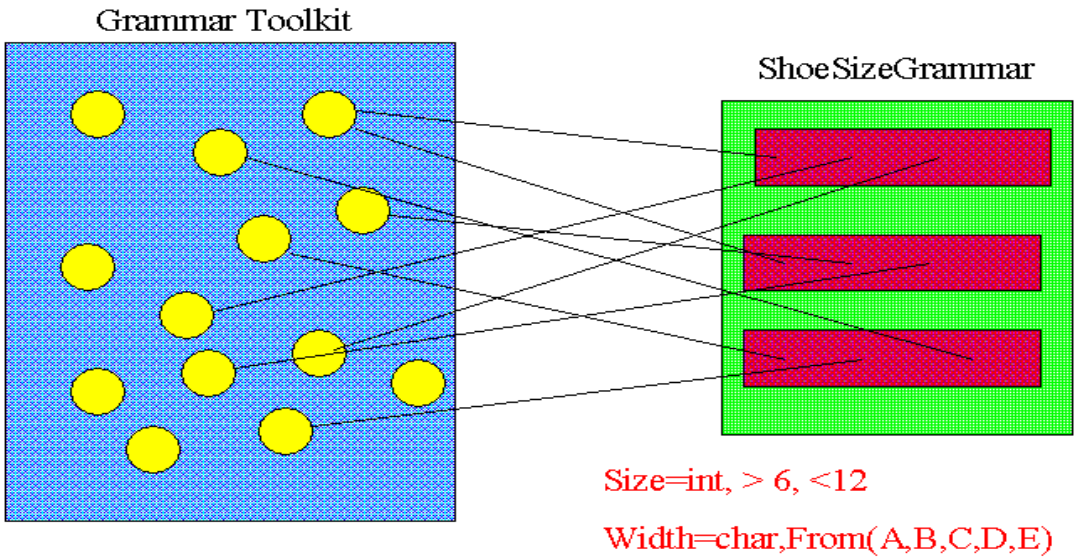
A bigger problem is the coherence of names. Rename a file on your machine and anyone who attempts to access the file by the old name is told the file no longer exists, the dreaded 404 error return by browsers. Since it is virtually impossible to track all the places where the name of a resource might appear, global names have no effective means to deal with this problem.

Other approaches are feasible but not in widespread use. One is relative naming, `Alan's wife`, `Alan's wife's best friend`, etc. The Client Utility uses a different scheme, attribute based names, i.e., `SPOUSE=Alan`. This approach is very similar to that specified in, for example, the LDAP and OMG Trader interfaces. Rather than adopt either of these schemes, or any other for that matter, the Client Utility allows for an attribute description to be in any grammar.

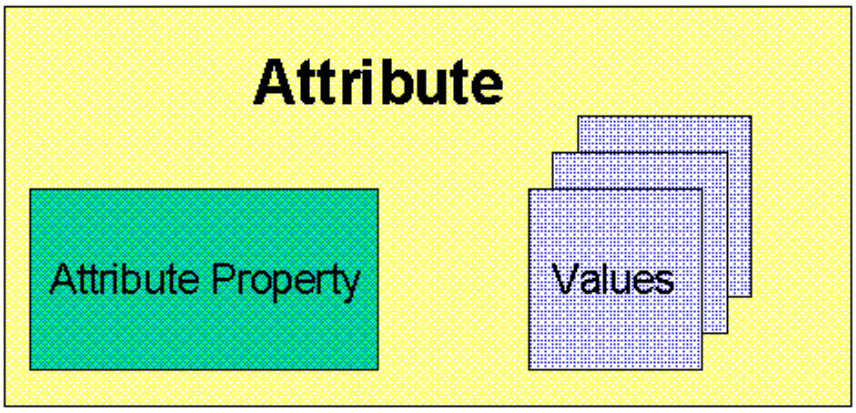
Each attribute specification, whether part of a resource description or a look-up request, specifies its grammar, also called its attribute vocabulary. An attribute vocabulary is a resource with a repository description just like any other resource. Hence, a task refers to an attribute vocabulary by name.

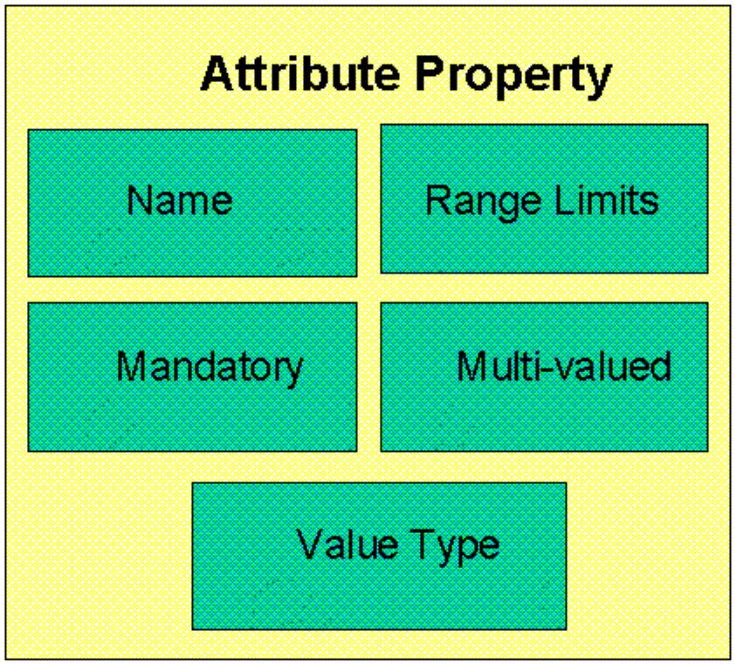



The attribute vocabulary contains all the pieces necessary to allow the Core to determine if two attribute specifications match. Of course, it has a specification of the attribute properties. The vocabulary also includes a specification of what constitutes a match with another specification in this vocabulary. The matching rules are specified in terms of components supplied by the Core. All standard data types and comparison rules are included in this set. These include equality testing on standard data types such as strings and integers, lexical order rules for strings, sorting rules for integers, etc..



The attribute vocabulary specification also has information to be used in comparing attributes from different vocabularies. There are rules for converting data types and translating various fields in the attribute properties. These are also built out of components supplied by the Core. For example, a floating point number in one vocabulary can be compared to an integer in another if we know whether or not to truncate the floating point number. Vocabulary translators allow us to do such things as translate the words in an attribute from French to German or from Unix to MVS. Each translator specifies the vocabulary it can translate to this one.




An attribute consists of an attribute property and one or more values. The attribute property is something with fields understood by the matching rules component. For example, in an OMG Trader vocabulary, the attribute property would include a name for the attribute. It can also include range limits on the

value, the data type of the values, whether more than one value can be specified, and if this field must be included in the specification to be matched. This last feature is useful if the attribute is a password that must be specified to find this resource description during a look-up.

---

## 4.6. Users and Machine Environments

On most systems the concept of a "user" is special, but not in the Client Utility. When a new user is given an account on a Client Utility system, the system administrator builds a name space that contains the user's initial environment and a protection domain specifying the user's mandatory key ring and default name space and registers them as persistent resources. After checking in, the user can do a look-up of the protection domain by specifying its attributes. This resource inherits the user's name space which establishes the desired environment. Next, the user tells the Core to change to this protection domain. The user environment is now ready, including the specification of the user's mandatory key ring and default name space.

Authentication can be handled in a number of ways. The simplest is to have one of the attributes be a password that must be presented. Of course, we don't store passwords in the clear; as with many systems a hash of the password is stored. If stronger authentication is needed, the name space can specify a grant authorizer.

The Client Utility also has an interesting way of looking at other machines. The system administrator has established the trust level of the foreign machine and the set of resources to be made available to users on that machine. These policies are implemented as if the task acting as a proxy for that machine were logging in as a user with the permissions of the foreign machine. In particular, the system administrator will have set up a protection domain and a name space with an environment containing names for the resources to be made available to the other machine. The proxy will execute commands on behalf of the machine in this protection domain. In particular, the proxy will present a specific export key which will enable it to see only resources that were made available to the other machine.

Some proxies will act on behalf of more than one machine. In this case, the proxy will get a name association for a different protection domain for each machine. There are two ways for the proxy to choose which protection domain

to use for requests on behalf of different machines. It can check in separately for each machine and get the correct protection domain. It can then act on behalf of a machine by presenting the appropriate token on behalf of the machine. Alternatively, it can explicitly switch from one protection domain to another as different machines make requests.

---

## 4.7. Advertising Services

Sometimes a look-up fails to find a match. The implication is that no machine has exported the resource to the machine on which the request was made or any machine among its extended look-up handlers. In this case, the task can ask an advertising service to find a supplier of the requested service. Unlike a look-up, which returns a name association when successful, a request to the advertising service returns a contact point. The analogy is to the telephone Yellow Pages. If you look up a plumber, you don't get back an unclogged pipe. You get a phone number you can use to contact the plumber.

The contact point is used by the task to pull from the supplier the description of the resource it wants to use. The task will send this information to the connection handler. If no connection has yet been made, the connection handler will use the connection information to find the machine expected to supply the resource. The service provider might have specified an IP address, a URL, a telephone number for a dial-in connection, or some other means of making contact. If the connection handler supports the designated connection mode, it contacts the provider, negotiates a Client Utility protocol, and does any mutual authentication needed. Next, the connection handler starts a proxy for the service provider and creates a repository view naming this proxy as its extended look-up handler. The proxy will complete the Client Utility connection by importing any resources and registering them in the Core repository and optionally adding them to the specified repository view.

If a connection has already been established to the supplier, the connection handler returns a name association for the repository view associated with the supplier. Notice that all knowledge of what connections have been made can be handled entirely by the connection handler. The connection handler is the only task that needs to understand communications protocols and the trust levels of other machines.

---

# 5. SECURITY MODEL

Controlling access to resources.



---

## 5.1. Introduction to Security Model

Security has many aspects. For our purposes, we'll define security in terms of

- physical security of the hardware,
- authentication of machines,
- authentication of users,
- privacy,
- access control.

While each of these components has important implications in any system, stand-alone or distributed, the Client Utility takes a different view of some of them.

Concerns over the physical security of the hardware are most often limited to questions of theft or destruction. The Client Utility says nothing about how the hardware itself is protected. However, the Client Utility must be able to deal with machines that are mistreated and are subject to unpredictable and frequent failure. The Client Utility architecture allows the construction of policies that tolerate such machines with minimal impact on other systems. The Client Utility also protects itself against a stolen machine by having a means of revoking privileges given to that machine once the theft has been reported. Even before the theft is taken into account, the damage done is limited to exactly the resources on the stolen machine and those resources exported to it.

Another security issue is knowing to whom you are speaking. Different levels of authentication between machines are used depending on the location of the machines. If all machines are in a secure environment and communicate over a

secure link, the degree of trust can be determined statically and no further authentication is needed. Attacks from insecure networks or machines outside a controlled environment are dealt with in a consistent manner.

Once the Client Utility is running, we need to authenticate people wishing to use its resources. Here, the Client Utility architecture presents some special problems. Recall that we want a user to be able to sit down at any machine in the Utility and gain access to his or her own environment. There are two cases to deal with, but both share a common problem. If the user is logging in to a new session, we need to find a machine that can identify the user. If the user is reconnecting to an existing session, we need to find the machine or machines that currently hold the session. Presently, we ask the user to provide this information as part of the login process, but we are investigating automatic means as well.

The Client Utility takes a different view of privacy than many other systems. While it is important to be able to protect the information being seen, a problem solved by encryption, it is often even more important to protect the fact that it is being seen. The basic mechanism used by the Client Utility provides this latter form of privacy. When a person accesses a resource through the Utility, the requester only knows that the Utility is satisfying the request while the provider only knows that it is giving the resource to the Utility. Of course, the Utility must keep information about the resource use for billing, dispute resolution, and any legal actions that may result, but neither party involved in the transaction has access to this data. The Client Utility allows a provider to require the requester provide identification information, but the Client Utility does not require this authentication.

The final aspect of security is access control, limiting who can do what to whom when. This control is defined by the access control matrix. There is a row in this matrix for every user and a column for every resource. The value at the intersection of a row and column tells the system what the specified user can do with the specified resource. For example, does this user have permission to delete this file? Even on a system with a modest number of users and resources, this matrix is sparse; most users have no access to most resources.

Since the access control matrix is quite large and very sparse, it is impractical to store all of it. Instead, cuts through the matrix are used. The most common data structure used in today's systems is an access control list, ACL. Associated with each resource is a list of users and their permissions. An equivalent structure, often used in the 1960's but rarely seen today, is a capability list, CL.

Associated with each user, or even user process, is a list. Each element of this list is a resource and the permissions granted to that user.

Access control lists are common today because they were more efficient than capability lists in the machines in existence when modern operating systems were being developed. Hence, they won out. However, their advantage is lost as an individual user sees less and less of the total environment and as the number of users of the system increases. Since the Client Utility is designed to handle millions of users on millions of systems, we decided to base our security model on an enhancement of the classical capability list, an ECL.

We need a secure base on which to build a secure system. There is no way to provide secure access if the security modules can be bypassed or modified by unauthorized users. In order to reason about security, we assume that the operating system enforces separation of address spaces. In order to support legacy applications running on legacy operating systems, we assume that all resources are accessed by making calls to the operating system kernel and that it is possible to intercept all these calls. Unlike some other systems, we don't require that the system be able to authenticate individual users. The Client Utility takes care of that aspect when it is necessary.

In the remainder of this section, we'll describe the Client Utility security model in more detail. We'll start with a discussion of the model applied to a single machine. Then, we'll describe how the model works when there is more than one machine available.

---

## 5.2. Control of Naming

On most systems people use today, the physical resources, such as files, have names known to all users. Any user can ask the operating system kernel to do something to the named resource. An important function of the kernel is to do the operation only if the user is authorized to make the request. This approach puts a lot of burden on the individual operating system components. Each must associate a set of permissions with each user of the system and know how to interpret the permissions for a diverse set of resources.

The Client Utility takes a two-stage approach to controlling resource access, dividing the problem into naming and permissions. In general, there are a large number of resources on a given machine that most users have no need to see. On a Unix system, for example, only someone with root privileges can use the

program that modifies user accounts. Allowing general users to see that this file exists violates a key security principle, that of least privilege. In other words, if you don't need it and can't use it, you shouldn't know that it exists. Doing otherwise opens up the possibility of certain attacks against the system.

The Client Utility Core assigns a protection domain to each task which contains the list of names that the task can use. Should the task name a resource not in its protection domain, it is told that the resource does not exist. The Core goes one step further; all the names in the protection domain are virtual. The names in the protection domain are specific to the task, and each name can be associated with a resource. When the task names a resource, the Core looks up the name in the task's protection domain to identify the resource being referenced. Now, the task can't even try to guess the name of an existing file by attempting to create one with that name; the Core creates a file with the designated name and simply maps it to a new name in the underlying file system.

The way the Client Utility controls the names available to a task is reminiscent of capability based systems first proposed in the 1960s. However, a classical capability contains both a reference to the resource and the access rights to be honored when the capability is presented to the kernel. This approach works reasonably well but has certain drawbacks. For example, the system needs to manage a number of capabilities for each resource. A file will need a capability for read access, one for write access, another for execute permission, perhaps one for append rights, *etc*. Another problem is that there is no association between the capability and the task holding it. In many capability based systems, the capability is a first class object that can be passed from one task to another. Revocation becomes a problem in such systems.

---

## 5.3. Access Rights

Once a task has a name for a resource, we need a way to control what it can do with that object. Can it read the file? Write it? Execute it? In all previous systems, the naming and access rights are combined into the access control matrix.

We have found that treating access rights independently of individual resources simplifies matters dramatically. After all, most users have read access to a large number of files; they don't need a separate capability for each one. The Client

Utility approach is to give each task a key that is associated with read permission to all these files. This key is just another resource that, when presented as part of a resource access, tells the file system to grant read privileges.

The mechanism is quite simple. Each task has a name space containing mappings between names and resources. The repository entry for each resource has a field containing permission fields. Each permission field is made up of a lock and a permission. (We intend that the permission field contain something to be interpreted by the resource proxy as a permission, but it is opaque to the Core. Hence, the field can contain anything the registering task desires.) When a task presents a key that opens the lock, the corresponding permission is forwarded to the resource proxy along with the request. Keys are also resources, but they don't have a permission that controls whether or not they can be used to extract a permission. Mere possession of a name for a key grants the right to use the key.

Each request to the Core for a resource access is accompanied by a list of key rings, each key ring containing zero or more keys. In addition, each task has a mandatory unconditional key ring that is always presented. The Core matches each key against each lock in the permission field of the named resource. The permissions associated with open locks get forwarded to the resource proxy. We now get conventional semantics with just a few keys. For example, Unix semantics needs 3 keys for everyone (read, write, execute), 3 keys for each group, and 3 keys for each user. These 9 keys can be put on each task's mandatory key ring when the task is started.

Keys are also used to control what resources may be added to a task's protection domain as well as what names are visible to a given request. Whenever a task wants to add resources, it asks the Core to associate resources with certain attributes to names in the task's name space. We control which tasks can get which resources by using the allow and deny fields in the visibility part of the security field of the repository entry for the resource. The request to add a resource requires that key rings be presented. The task can add a resource only if it presents keys that open at least one lock that appears in the resource's allow field. However, if the task presents any keys that open a lock in the deny field the system will act as if there is no resource that matches the request. Keys, being resources themselves, also have allow and deny fields, so we can control access to keys, as well. The visibility field is also checked against resources in the task's name space when a resource request is made. If the test fails, the Core acts as if the name were not associated with that resource.

We now see how to prevent general users from knowing about resources they have no business seeing. At task start-up, the system administrator specifies a particular key be put on the task's mandatory key ring. No name for this key is put into the task's name space which means that the task can not ask for this key to be removed from the key ring. This key can be put into the deny field of all system resources that the administrator wants to hide. Users authorized to see these resources won't get this key put on their mandatory key rings.

We can now quite easily implement some advanced features that are not so easily provided by other systems. One such feature is roles. Sometimes a user is a manager; sometimes the user is an employee. When acting as an employee, this person has no need to see certain management resources; when acting as a manager certain employee resources should be hidden. The user simply maintains separate key rings for each role. The mandatory key ring can have keys for role independent resources used by the task. Separate manager and employee key rings can grant certain permissions.

Another advanced security feature is compartmentalization. Compartments are structured to prevent mixing of resources that should not be mixed. For example, a consulting company may work for competing businesses. They would like to assure their clients that they can't see the resources from ABC, Inc. while doing work for XYZ, Ltd. Simply making the allow key for ABC's resources be the deny key for all of XYZ's resources, and *vice versa*, implements this policy.

We can also enforce military style security in which someone with Secret clearance can't read a Top Secret document. This policy is actually a bit more complicated. The so-called *-property says that you can read documents at the same or lower security level and write documents at the same or higher level. We enforce this property by giving someone with a Secret clearance the "Secret" key. This key opens locks associated with read permission for unclassified documents, the read and write permissions for secret documents, and the write permission for Top Secret documents. That's all there is to it. The only overhead is in setting up the permissions in the first place.

---

## 5.4. Intermachine Protocol

Thus far we've talked about security within a machine. The Client Utility would be quite limited unless it permitted access to resources on other machines. The

Distributed Resource Interchange Protocol (DRIP) specifies how machines in the Client Utility interact. Key to this protocol is the fact that all connections are pairwise. The fact that machine A makes some of its resources available to users on machine B is independent of whether or not it is also making resources available to users on machine C. Not only does this approach make the system more scalable, it also simplifies the security model.

Another important principle is that each machine is responsible for the security of its resources. All accesses to these resources, such as disk accesses, must necessarily come to the owning machine. At this time, it can verify the validity of the request. It can also revoke a previously granted privilege. This strong revocation is one of the great advantages Client Utility has over some of the capability based systems.

When two machines want to share resources with each other, they start the Client Utility communications. First, they identify each other, either with a shared secret exchanged by some out of band means or with some form of public key system. Once they have securely identified the system on the other side of the wire, they can decide on how much security they need between them. If the communication links are secure, they can begin exchanging information immediately. If the communications links might be tapped, they will exchange a session key first.

Once the machine on the other side of the wire is identified, each machine starts a task to act as a proxy for the remote system, its Remote Resource Proxy (RRP). This proxy has a protection domain containing only the resources it needs to get its job done and the resources being made available to users on the other system. These proxies then transfer over the wire a description of the resources being made available to users on the remote system. The receiving RRP asks its Core to create repository entries for these resources listing itself as the resource proxy.

When a task accesses a resource on another machine, the Core forwards the request to the resource proxy, in this case the RRP acting on behalf the machine that owns the resource. The RRP forwards the request across the wire to its counterpart. That task simply makes a request of its local Core. The task is quite accommodating; it will attempt to do anything it is asked to. However, since it is running in a protection domain that sees only the resources exported to the other side, the Core will not let it do anything to other resources. The RRP simply doesn't have names for resources not exported to the other machine. If the RRP is successful in running the command, it forwards the

returned data across the wire to its counterpart which does what any other resource proxy would do with return values.

Notice how simple this all is. The Core never sees a remote request; all requests come from local tasks running in protection domains created by the Core. The Core never forwards requests across the wire; they simply go to a resource proxy. In this case, the resource proxy doesn't actually do the work, but the Core doesn't care. A RRP can impersonate tasks running on the other machine by presenting keys that represent the permission of that task. Again, nothing special is needed; its just a different form of roles. Perhaps the greatest advantage is that the Core doesn't have to worry about what the other machine calls a user; that's handled by the machine running the task.

There are times when the Core does care about the identity of the user running the task on the other machine. Say that an authorized user of machine A sits down at machine B which happens to be in a hotel room. In general, machine B would be allowed to see very little of machine A's resources, certainly not enough to let the user do anything useful. In this case, the user authenticates with machine A which then raises its level of trust in machine B and exports additional resources.

---

## 5.5. Trust Model

The Client Utility supports a trust model that reflects reality. If I tell you a secret, and you blab it to everyone, I've misplaced my trust. No software could have prevented the disclosure of the secret. If my machine lets tasks on your machine read a certain file, I am trusting you not to pass that file on to those who shouldn't see it. If you do, I've misplaced my trust. This form of transitive trust makes the entire system manageable. Imagine the complexity if I tried to control what you did with bits I made available to you.

It is important to understand that the degree of trust put in the other machine is critical. There is no guarantee that it will enforce any of the security policies specified in the resource metadata. The requesting machine can forward all permissions, even if no keys were presented. It can give all tasks names for all imported resources. It can post all passwords in a publically accessible place. The point is that one machine should only export a resource that needs protection to another machine that will enforce the security policies. If a

mistake is made, if the importing machine violates the trust, the Client Utility can do nothing to correct the situation.

---

## 5.6. Attack Scenarios

We've made a lot of claims about security. How well will the system stand up to real attacks? We don't know for sure, but we can look at a number of attack scenarios to see if we can anticipate trouble. We'll look both at attempts to perform unauthorized actions and denial of service attacks. We won't worry about social problems, such as poorly chosen passwords or people who write their PINs on their ATM cards.

We assume that either the Client Utility Core is the native operating system, or there is a trampoline that gives the Core control on any attempt to access the native operating system. Some Unix systems support such a trampoline. Others can be modified to implement one as was done by Locus Computing Corp. for its Transparent Computing Facility distribution. If the operating system can't be modified to implement a trampoline, we can get some protection by providing a set of dynamic libraries that forward kernel calls to the Core. A determined hacker can get around these libraries. As we'll see, this hacker may be able to penetrate this one machine but will gain no hooks to penetrate others.

First consider a malicious user on a single machine who would like to do some unauthorized operation. Since any user has access to a process with enough resources to complete a valid login attempt, the attack could start there. This task will have a protection domain with some named resources. The attacker has no names for anything else, so the initial attack will be against those resources. We can make sure that the attack does no harm by not giving this process any keys that would allow it to modify any resources not needed by the login procedure.

Our attacker now attempts to get additional resources into the protection domain of this login process. This attack can be thwarted by putting an unnamed key on the task's mandatory key ring that is in the deny field of any resource not needed by the login procedure. The other resources can be password protected so that the attacker is limited to guessing account names and passwords. We hope that this attack will fail, but poorly chosen passwords will make the attacker's job easier. Of course, we can have an authorizer log the guesses. Should the authorizer decide that an attack is in progress, it can put an

unnamed key on the unconditional key ring of the attacking process that appears in the deny field of all password protected resources. At this point, the attacker can add no resources to the protection domain of the login task. After a suitable timeout, the key can be removed so that legitimate users can log in.

Say that the attacker has logged in, either by guessing a password or because the attacker is an authorized user. The attacker can now modify any resources available to the active account. The attacker can also add to its task's protection domain any resources that either don't have some form of password protection or for which the attacker knows or can guess the password. We note that certain critical resources can be protected by an audit trail or by a grant authorizer that can initiate a challenge/response identification of the requester.

Now that the attacker is running a legitimate process, messages can be constructed that attempt to confuse the Core or a resource proxy into performing an unauthorized action. The Core is structured so that each task, except for certain trusted, well-tested applications, talk to the Core only through a thread running in the Core, a client proxy. (If it is possible to do something to this thread that would crash the Core, then we use a separate process as the client proxy.) The application sends requests to the thread which marshals them into messages to the Core.

Common attacks, such as sending random messages, will be filtered by the marshaling thread; it simply won't know what to forward to the Core and will reject the request. The attacker might try sending a very large message in hopes of overflowing the thread's buffers. We can defeat this attack by taking the memory for the message out of the sender's allocation. Should the attack succeed in spite of our best efforts, the thread assigned to talk to the attacker would fail, cutting the attacker off from the Core.

The attacker could attempt to flood the Core with messages in hopes of denying service to other users. However, the Core has control over which thread it schedules next and can simply degrade the priority of a thread that is sending too many messages. This defense has the effect of eventually blocking the attacker because its marshalling thread's receive buffers will get full. The attacker could also try to make the Core fail by registering a resource with a very large specification. Again, the defense is to take the memory allocation from that of the attacker's task. The Core can also refuse to register resources that take up more space than some threshold.

Another denial of service attack would be to register a large number of resource in an attempt to fill up the repository. Two defenses are possible. First, the Core

can set a quota on the number of repository entries it allows any process to have. When this quota is exceeded, the Core can simply remove resources when additional ones are submitted. Legitimate tasks will know that the Repository is only a cache of the resources they want to provide and can establish procedures to handle overflows. A second defense is to let these resources overflow into the persistent repository. By structuring the search along language lines, only tasks searching for resources provided by the attacker will suffer any slowdown. Furthermore, the Core can make sure that all resources on its machine can only be registered by tasks it knows about. An attacker will be able to register a new language and any resources it likes in that language, but users won't be affected if they only use resources residing on the machine.

If we're dealing with more than one machine, the attacker can try to get the other machine to do something unauthorized on its behalf. However, all requests are sent to a remote resource proxy on the machine that owns the resource. This proxy has a protection domain that contains only the resources exported to the machine the attacker is using. Any resource that wasn't exported can't be accessed by the proxy. Hence, the RRP can't do anything unauthorized for the attacker even if it tried to.

The attacker could be running on a machine with a corrupted Core, modified operating system, or customized hardware. It doesn't matter. Even if the attacker refuses to honor the permissions in the exported resources, the attack fails because these permissions are checked when the RRP on the machine that owns the resources attempts the access. If some exported resources are password protected, the malicious Core could post the passwords to all users. This attack fails because we don't put the actual password in the attributes field; we put a one-way hash of the password. Only by negotiating with the machine that owns the resource can the attacker get the protected resource added to the RRP's protection domain.

The attacker could try to crash the other machine by overflowing message buffers. Depending on the structure of the messaging layer, this attack might succeed. However, if the basic messaging is implemented properly, the attacker will succeed only in causing the RRP to which it is talking to crash. The affect will be to isolate the attacker from the machine being attacked, but no other users will suffer.

---

# 6. MANAGING AND MONITORING A UTILITY

Managing a Client Utility machine is not necessarily harder than managing a machine running only a conventional operating system; it is just different. The Client Utility Core has a component, the Monitor, that collects data needed to manage the machine. The contents of all messages that pass through the Core are handed to the Monitor as well as any additional information the Router and Name and Resource Specific Information Managers find useful to save. The Monitor records this data in whatever format it finds useful in a database. This decoupling lets us separate the design of the message envelopes from the structure of the monitor database. Small machines may choose to filter the information put into the database to save space or may forward the data to a large machine for archiving.

The Client Utility also provides an interface to control the resources. However, the management functions that use this interface are simply tasks like any other. They just have a different set of keys than less privileged tasks. In particular, the System Administrator can produce a key ring with keys that allow full access to the monitor database. Other tasks will have limited access to the data held by the Monitor.

It is relatively easy to partition the monitoring functions. For example, we might want to allow an individual user to find out how much memory is being used in a particular session, but not find out how much is being used by another user. We might want the system administrator to be able to see system wide information and even modify the monitor database. Implementing these functions simply requires the Monitor to register resources with the Core that have permissions associated with certain keys. When a task makes a request, the resource proxy for the database can interpret these permissions to decide what requests to accept.

The Monitor can also act as an event distributor. Tasks, given the proper permissions, can subscribe to events related to Core activities. For example, a remote resource proxy might want to be notified if the metadata for one of the resources it exported has been changed. More generally, a task running some management software, such as SNMP or OpenView, can subscribe to events of interest.

# 7. INTERMACHINE PROTOCOL

How machines tell each other what they want done.



---

## 7.1. Introduction to the Intermachine Protocol

The protocol used by Client Utility machines has several stages - connection, authentication, exchange of resource descriptions, and use of remote resources. Each of these components was designed to be consistent with the scalability, heterogeneity, and security requirements of a large, distributed system.

The first step is finding a machine to talk to. Next, there is a protocol negotiation stage. If the machines speak a common dialect of the protocol, they can identify each other. Once they know who they're talking to, each can decide which of its resources it will let tasks on the other machine use. Finally, tasks can run using resources from other machines.

We now see how we can use resources without the Core being aware of their locations. Each machine contains a configuration identical to the single machine configuration described earlier. When a task accesses a resource, the Core looks up the name and identifies its resource proxy. If the resource is local, the resource proxy provides the response. If the resource comes from another machine, the resource proxy is really a remote resource proxy. The RRP, say on Machine A, forwards the request to the owning side, say Machine B. The RRP on machine B acts on behalf of the remote user and makes a request to its Core. The Core forwards the request to the resource proxy which returns the result to the requester, the RRP in this case. The RRP on Machine B sends the result to its counterpart on Machine A which sends the reply to the application as would any other resource proxy.

---

## 7.2. Connection

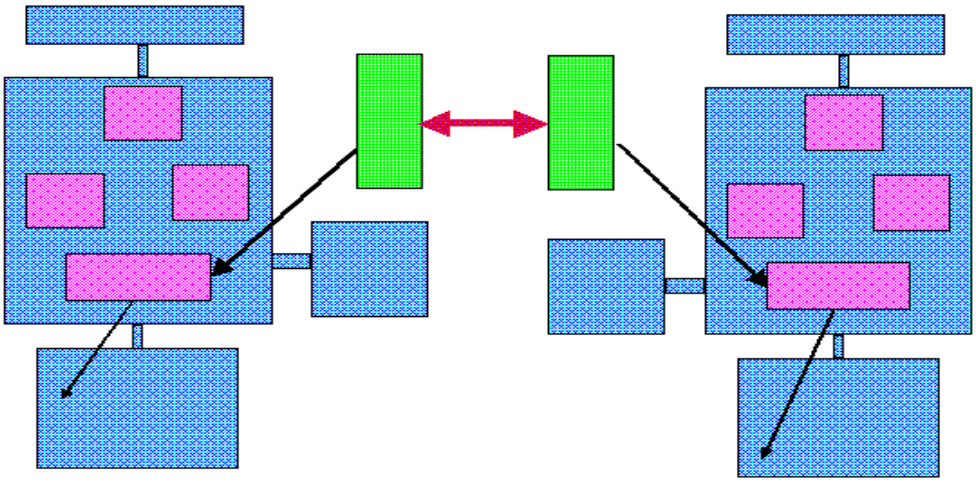
Any machine that wishes to allow other machines connect to it will distribute a connection object and tell its connection handler to act as a listener. This connection object tells everything needed to communicate,

- machine specific information, such as endianness and character set,
- connection modes, such as TCP or RMI,
- authentication and encryption protocols supported,
- DRIP and Client Utility protocols supported.

How the listening is done depends on a number of factors. Network connected devices that support TCP/IP can listen on a defined port for connections. A machine with a dial up connection can use a modem at the end of a phone number. Devices such as beepers and cellular phones may need to designate a computer to act as an intermediary.

There are a number of ways a machine can find other machines to connect to. In an enterprise, an administrator can assign an initial set of connection objects to each machine. Similarly, a home user could be assigned a contact within an ISP. In a less structured environment, the machine could go to a look-up service that would provide one or more initial contacts. Of course, it is always possible for the machine to broadcast a "Who will talk to me?" message.

Once a connection object is identified, the machine's connection handler will be instructed to act as an initiator for the connection. The initiator will examine the connection object to determine if a connection is possible. For example, both machines must share a common dialect of the DRIP. If a connection is possible, the initiator will contact the listener by the communications channel specified in the connection object. The machines next agree a common dialect of the DRIP protocol. (The connection object may specify more than one.) Once this step is completed, the machines can continue the connection phase of the DRIP.



## 7.3. Authentication

Once machines have established a connection, they need a means of identifying the party on the other side of the wire. The Client Utility allows machines to negotiate an authentication protocol in much the same way that they negotiate a connection protocol. It is assumed that every machine participating in the Client Utility supports four basic authentication schemes. They are

1. Name,
2. IP address,
3. Shared secret,
4. Public key.

Machines are allowed to define other authentications, but must decide how to deal with machines that don't support these extensions.

The first authentication scheme, namely Name, has three purposes. Clearly, no authentication is needed if the machine will only export freely available resources, such as advertising. Another use is for trusted machines in a protected environment communicating over a secure network. In this situation, the machines can assume no spoofing or snooping will occur and can trust the machine has sent its agreed upon name. The third use is when a user on a machine will assume responsibility for its security. The initial connection is made without authentication, but resources will be exported only after authentication of a known user. The user authentication will use any of the protocols allowed for machine authentication.

The second authentication scheme is for a less trusted environment that is safe from IP spoofing. For example, a set of machines behind a corporate firewall might decide to allow IP authentication to simplify the connection process. In this case, the machines identify each other by querying the socket connection for the IP address at the other end of the wire.

Shared secret authentication is most useful when a pre-existing agreement is in place, such as within an enterprise or between an ISP and one of its customers. Each machine uses the shared secret to generate an encryption key that is used to encode a message. These messages are exchanged, decrypted, and appended to each other. The combined message is encrypted, exchanged, and decrypted. Each party can now be sure that the other machine shared the secret. Notice that the contents of the messages are not part of the protocol since randomly selected messages will make a cryptographic attack more difficult even if the shared secret is changed only infrequently.

Shared secrets are only useful if there is some secure, out of band communication to exchange the secret. If there is no such channel, or if there are a large number of potential customers, public key systems can be used. This system can be used in a number of ways. An individual can be identified via a digital signature certified by a Certificate Authority. More useful for commerce is to have the individual submit a guarantee of payment signed by a bank or credit service. Merchants can be their own Certificate Authorities in this case because there are many fewer guarantors than customers.

If the communication line is subject to snooping, the machines will define a session key. If a shared secret is being used for authentication, using the combined authentication message instead of the shared secret to generate a session key makes a cryptographic attack harder. If we want a session key but did not require any authentication, we use a modified form of Diffie-Hellman key exchange that uses a shared secret in a secure way. We don't want to use the shared secret itself to generate a session key because its use makes a cryptographic attack simpler. If there is no shared secret, a form of Diffie-Hellman key exchange that is not subject to a man in the middle attack is used. If a public key system is being used, the session key can be exchanged using this method.

## 7.4. Exchanging Resource Descriptions

Once a machine has identified the other one, it can decide what resources tasks on the other machine can use. We'll call these two machines Owner, the owner of the resource, and Importer, the user of the resource. Remote resource proxies will be started on both Owner and Importer. The proxy on Owner is given a protection domain that contains only the set of resources it needs to do its job and a name space containing associations for resources it is to make available to Importer. The RRP on Owner asks its Core to provide an export description for its export name space. The Core traverses this name space and all other composite resources and returns a payload containing an export description for each exportable resource.

The export form includes the name that Owner wants Importer to use when referring to this resource. In this way, the two machines agree upon a set of names for the resources they are sharing. There is no need to export a resource that has already been assigned a name, so there is no problem if the set of resources being exported has cycles in the references. It also means that the export description of a resource needs to be generated only once.

Core managed resources are not usually exported. The reason is simple - they have internal state that only the Core on Owner can be expected to understand. If we exported such a resource, say a frame, Importer would have to ask Owner to look up name associations in the frame, and those associations would not correspond to resources on Importer. The RRP on Importer is told to make its own version of such resources and what components go with each. In other words, the resource is exported by value, rather than by reference.

An export description of a resource contains a modified version of the data held in the repository. Several fields are changed. For example, it makes no sense to point to an authorizer on Owner. Instead, the RRP on Importer is told that Owner would like to have Importer keep an audit trail or allow Owner to approve any transfer of name associations. The RRP on Importer then knows to name itself as a notify or grant authorizer for the designated resources. Also, keys are never exported because their internal state may be such that they accidentally open a lock on Importer. Instead, permissions are replaced with the name of the key the RRP on Owner should use when a request is forwarded from Importer.

The resource proxy field is not included in the export description because the RRP on Importer will name itself as resource proxy for all resources it registers. Owner's RRP also adds its name for the resource to the private resource specific data field. The RRP on Importer will make some changes of its own. For example, it will add resource specific data so it can identify the source of the resource. This information will be used when a task refers to an imported resource. The importing RRP may also add its name for the other machine to the public resource specific data field.

We use keys associated with locks in the visibility fields of the resource descriptions to control the export process. Every RRP is given the `export` key. This key opens the lock associated with the `export` permission for the name space. However, that key also opens a lock in the deny field of every local resource that should never be exported to another machine. Thus, the export descriptions returned by the Core are guaranteed to include no resource descriptions that correspond to non-exportable resources.

A resource that is to be exported to a specific machine has a lock in its allow field that is opened by the `export_xyz` key given to the RRP for machine `xyz`. When that RRP wants to get export descriptions for this set of resources, it presents this key. The normal visibility mechanisms limit the export to the set of resources intended for that machine. A resource that is exportable to any machine will have a lock in its allow field opened by the `export` key.

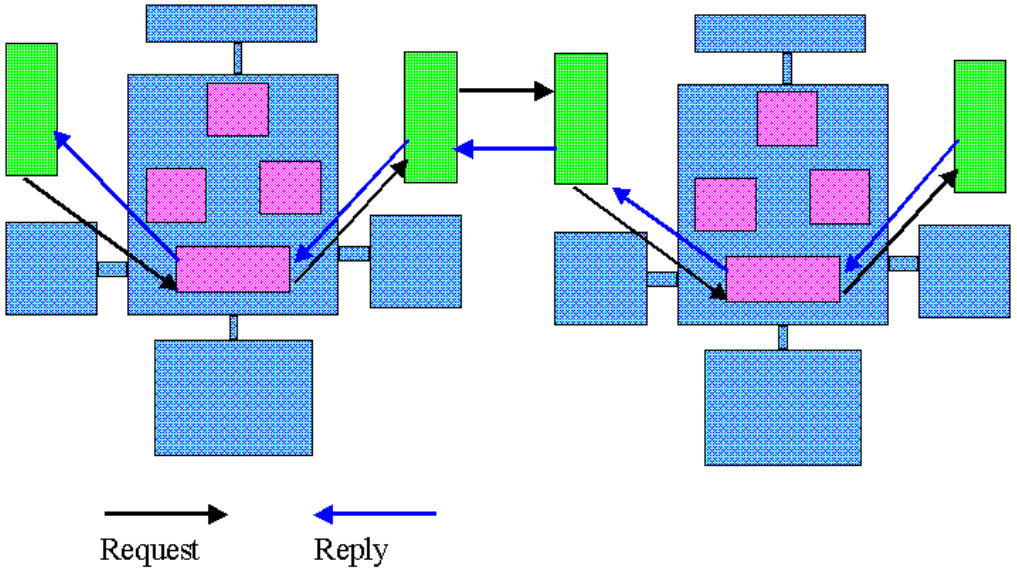
In order to make exportable resources visible to local tasks, all tasks other than RRPs will be given the `local` key. This key will open the corresponding lock in the allow field of the description of every resource that should be visible to local tasks. In general, any imported resource will have the locks corresponding to the `local` and `export` keys in its allow field. In special circumstances, we can limit further export of these resources by including a lock other than the one opened by the `export` key in their allow fields.

---

## 7.5. Resource Requests

We now have an internal repository with resources from a number of machines. Tasks running on the machine can add these resources to their protection domains in the same way that they add local resources. Tasks use a remote resource in the same way as they use a local one; they simply name it as the primary resource. However, instead of being handled by a local task, the

request goes to a RRP proxy for the machine that exported the resource. The RRP forwards the request across the wire.




The complication comes when the request requires the handler to have access to additional resources in the requesting task's protection domain. For a local request these additional resources have their name associations copied to the frame associated with the handler's inbox. For a remote request, these name associations appear in the inbox of the RRP.

The RRP needs to make sure that its counterpart on the other side of the wire has access to the resources needed to complete the job, so it needs to forward the resource descriptions. To do this, the RRP presents the appropiate export keys, `export` and `export_xyz` if it's talking to the machine it calls `xyz`. The visiblity keys ensure that export descriptions of only those resources that should be exported to `xyz` are delivered from the Core. It is the responsibility of the RRP on the side making the request to tell the RRP on the side that will service the request which resources being exported originally came from the serving side. All other resources are assumed to come from the requesting side.

The RRP on the side serving the request creates a name frame with the resources exported by the requesting side. Any of these resources that originated on the serving side will be put into the name frame using a local lookup. Resources that were previously imported and registered in the server's Core can be added the same way. In general, all other resources, such as those owned by a third machine, will be registered in the local internal repository and added to the name frame. Of course, if the local machine knows that it can get

the resource directly from the third machine, it can associate a name for its description directly.

Once the RRP on the serving side has constructed the name frame, it can issue the request in this name frame. In order to see the resources it needs to do the job, the Owner's RRP includes the `export` and `export_abc` keys on one of its key rings. This RRP also includes any keys that were identified via the permissions extracted on the requesting side.

---

# 8. SYSTEMS

Describes building systems from Client Utility components.

- 8.1 Introduction to Systems: Defines the systems of interest
- 8.2. Single Machine System: Building a usable infrastructure on a machine
- 8.3. Multiple Machine System: Combining individual machines to make a system

---

## 8.1. Introduction to Systems

Most of this document has focused on the interaction between tasks and the Core. A system is more. It must provide a resource proxy for every item to be managed by the Client Utility; it must include initial environments for all legitimate users; it must provide for interactions with other machines; it must provide standard services such as mail, file transfer, and the like.

---

## 8.2. Single Machine System

Each machine will have a number of resources managed by the Client Utility. It is important that the languages for these resource and their resource proxies be started as part of the system initialization. Otherwise, a malicious user could hijack the file system by registering its language and registering all the files with a fraudulent resource proxy.

A Client Utility system can be configured in many ways. A small appliance like a beeper would run only the over-the-wire protocol. A larger machine that would only use resources but not export any could implement only the repository so its users could find the resources they wanted to use. A common environment would be one that would provide only files to machines running services started by users on the machine. Of course, many machines will run the full infrastructure.

If the machine is to run untrusted code from other machines, it will need to protect its critical resources. These include files and memory, of course, but other types of resource also need protection. For example, the Java 1.2 security manager has 22 entry points. Eight of them refer to resources of no particular interest to the Client Utility, `AwtEventQueueAccess`, for example. Four have to do with network connections. Another 3 protect files while one guards printer operations. It may also be necessary to restrict the amount of virtual memory available to the task, but this limit might be easier to enforce with the native operating system than with the Client Utility.

---

## 8.3. Multiple Machine System

By implication, a Client Utility involves more than one machine although there is nothing to prevent someone from running the Client Utility on a machine not connected to the network. This machine will talk to other machines using the Distributed Resource Interchange Protocal (DRIP). However, to be truly useful in a multiple machine environment, a Client Utility machine will have to provide a number of services. These include mail, ftp, and nearly 200 others on a standard Unix system.

Since most of the security holes exposed in existing systems arise in these various services, a Client Utility system must treat them carefully. The main problem is that the services assume the messages they receive are from a well behaved program. Hackers use this trust to expose weaknesses and bugs in the implementation. Making matters worse is the fact that some of these task run with root privileges. The Client Utility limits the damage that can be done by running these services in the smallest possible protection domain. In addition, these services to do not have access to the network. Instead, they speak to a RRP that can filter the messages before they reach the service.

---

# 9. FAILURE LIMITATION

How the Client Utility manages and limits the effects of failures



---

## 9.1. Introduction

We know that hardware can fail, e.g., memory chips develop unrecoverable errors, and we design our machines to report and isolate such failures. However, even though we know software isn't perfect, we call such flaws "bugs" and fix them in the next release. The marketplace would not accept a machine that had not been designed to deal with hardware failures, yet software systems built this way are the norm.

When a software failure occurs on a single system, the application stops working. If the failure is in some critical system component, the whole machine locks up and must be rebooted. As annoying as this is, it's not too serious a problem. However, an identical failure in a large scale, distributed system can have wide ranging repercussions. In fact, the definition

> A distributed system is one in which a machine I never heard of can render my machine unusable.
>
> --Leslie Lamport

captures the problems that arise when no attempt is made to limit the damage done by a fault.

In addition, for an information infrastructure to really become as pervasive as electricity or water today, it must provide a high level of reliability. This is not just a matter of stacking up robust protocols and high-availability software components. By accepting the inevitability of software failures up-front and providing an architecture to limit their effects and manage their occurrences over time, we can help build a true computing utility.

## 9.2. FLOODS Architecture

Like the forward pass in football, three things can happen when a task uses a resource, and two of them are bad. The norm, we hope, is that the request is satisfied. If it is not, the failure can appear in one of two ways, when looking at the most abstract level. Either the supplier gives an erroneous response, or it doesn't respond at all in the time interval the requester is willing to wait. If we call the requester R, the supplier S, and the [optional] return value V, we can write these three cases as

- OK(R,S,[V]) - correct response provided
- BURP(R,S,[V]) - bad or unexpected response provided
- NAP(R,S) - not answering promptly

BURP is often called fail loud, and NAP, fail silent. The latter is a misnomer because there may be no failure, per se; it may just be that the supplier is slow, or temporarily disconnected. It is just the fact that the supplier may not be dead, only napping, that makes dealing with this failure mode so difficult. There is no guarantee that the response won't appear after the requester has decided the request failed.

The FLOODS architecture defines a "self-aware" object as one that reports its failures and/or failures from its suppliers, and accepts control commands from a designated task for the purpose of failure limitation and recovery. Each failure is reported as an event called a DROP (deducible record of object problem) to a task acting as the reporting task's FLOODS monitor. A FLOODS monitor collects DROPs from a number of tasks/objects, reasons about the set of failures and what to do about them, and issues FLICs (failure limitation intervention commands) to tasks/objects that will first limit and then repair the propagation of the effects.

There is an analogy to the way the electric utility deals with a failed tranformer. The lights go out in a neighborhood, BURP. Someone calls the hotline, issues a DROP to the FLOODS monitor. A repair team is sent to the site to repair the problem. Occasionally, the failure causes instabilities in the power grid. When this happens, the control center often turns off the electricity to other neighborhoods in an attempt to prevent the instability from spreading, FLIC.

The FLOODS monitor will do the same thing. When it receives a DROP, it will analyze the error. It may tell another task or the CU Core what actions need to

be taken to limit and repair the problem, swap in a hot spare, for example. However, if the FLOODS monitor starts receiving a large number of DROPs related to the same problem, it may become more active and tell some tasks to pause or even halt.

---

## 9.3. Analysis

The general analysis of failures, their causes, and the appropriate interventions to make, is a very complex process, often done by human experts. For that case, the FLOODS module can assist the human operator by gathering, visualizing, and archiving failure information. For many specific cases though, a set of possible failures, and appropriate interventions can be described in a formal language, amenable to logical deduction or algebraic manipulation. Such a mathematical engine can be built into the FLOODS monitor to automatically respond to failures, or assist human operators making decisions on how best to deal with the problem.

Mathematically, the challenge and opportunity is to represent failures and interventions in different semantic domains. Each "self-aware" object type can dynamically register its semantics, either as a set of axioms in a logical engine, or a set of rules in an algebraic solver. One of the key FLOODS contribution is to provide general purpose machinery to reason about failures across different semantic domains, which is the reality of layered software systems.

---

# 10. SCENARIOS

Some examples that show how the Client Utility System works.

- 10.8. Copying a Core Managed Resource: Making a clone

---

# 10.1. Introduction

In this section we'll walk through various tasks that the Core will have to perform. We'll assume that we have some resources in the repository. For example, the repository entry for a file might be

```
 CRH:            859
Language:       Unix_files
Proxy:          file_system_inbox
Attributes:     FS_grammar
                DESC="Bill's CU data"
                PW="30x9,Rq"
Private Data:   /users/bill/data
Public Data:    local,HP-UX/10.20
Permissions:    28CFA3 read
                3F323B write
                AD9732 execute
 Inheritance:    ---
Authorizer:     ---
Allow:          000000
Deny:           FFFFFF
```

In the implementation we use repository handles for all fields; for clarity, we'll simply include the data directly. (A convenient optimization is to use object references when the resource description is in memory and repository handles when in persistent storage.) The language, proxy, and authorizer fields would normally contain object references, but we'll use strings to make it easier to keep track of what we're talking about. In this example, the API is that of a Unix file system, the resource proxy is the inbox associated with the file system handler, no resources are inherited when this one is added to a frame, and there are no authorizers.

The attributes allow a task to look-up this resource in the repository. The first field specifies the grammar. Next come specifications in this grammar, first a description and second a password. The rules for deciding what constitutes a match are encapsulated in the grammar.

The private resource specific data is always passed to the designated resource proxy. In this example, this field gives the internal name for the file, `/users/bill/data`. Also in this example, the resource proxy handles more than one file system. The payload is in the `Unix_files` API. The second entry in the

public resource specific information field tells what API is to be used when making requests of the underlying file system. The first entry in the public data lets tasks know the source of the file, the local file system in this case.

The permissions corresponding to locks opened by keys on the requester's key rings are passed to the designated resource proxy. For example, if the user's designated key rings hold keys with values compatible with `28CFA3,3F323B`, requests to read and write the file will be honored, but requests to execute the file will be rejected.

The allow and deny fields allow many authorizer functions to be implemented without any more messages being sent. We assume that all tasks implicitly hold the Null Key, `000000`, and that no task ever holds the Negate Key, `FFFFFF`. If a task has any of the keys in the allow list, it can add the resource to its protection domain. If it has any keys on the deny list, it will be denied access and told the resource does not exist. Denial takes precedence.

---

## 10.2. Explicitly Add a Resource

Name associations for resources can be placed in a task's name space by the Core when the task is started, but there will always be times when we need to add a name association explicitly. In this Section we'll walk through adding the specific resource described in the previous Section.

The requester sends a message to the Core consisting of the following outbox envelope and payload. (The API in the payload is for illustration only; the actual API is more powerful.)

```
Key Ring Name Fields--------------
  Name:          (,)
Primary Name Field---------------
  Name:          ((,CUproject),my_user_frame)
  Label:         frame
Name Field-----------------------
  Name:          (,FileAPI)
  Label:         File
Error Mailbox Name Field---------
  Name:          (,my_error_mailbox)
  Label:         ---
Payload:  add_resource frame /myhome/data
           File DESC="Bill's CU data" PW="30x9,Rq"
```

The Core knows the identity of the task from the communications channel used. (On some operating systems, the sending task will have to explicitly identify itself using a token supplied by the Core.) The Core can now associate a protection domain with the request. Since no key rings were specified, the Core will look at only the keys on the task's mandatory key ring.

The primary resource specifies a Core managed resource, so the task knows that the Core will interpret the payload. This resource tells the Core where to put the name association, in the frame the sender calls `my_user_frame` in the name space it calls `CUproject`. The payload tells it what logical name to use, `/home/data`, what language the resource is in, `File`, and a list of the attributes the Repository entry must have to qualify as a match.

Note that in a name field the name space is specified only by name. The notation ((,name1),name2) means that name2 is resolved in the name space named (,name1), and name1 is resolved in the default name space. These names are resolved just like any other name. If the name space field is left blank, the name is resolved in the default name space.

---

## 10.3. Open and Use a File

One of the most common things we want to do is open a file and read its contents. Here, we'll walk through this case giving special attention to the overhead involved in the Client Utility Core.

Say that I have a task running on a native OS that wants to read a file having the name `/users/bill/data` in the underlying file system. In a conventional operating system, the software would first open the file for reading using this name. Eventually, the OS kernel would see `open r /users/bill/data` and return a file handle `FH`. Then, the task will start reading records, which the OS will see as `read FH`.

On a Client Utility system things work differently. The task has a personal name for the file, say `/home/data`. Let's assume that the task's name space has an explicit entry for this file, `/home/data 859`, which specifies the task's logical name and the repository handle for this resource. Whether done by a Client Utility aware application or an emulation library for a legacy application, the system constructs a message consisting of an Outbox Envelope and a payload. For brevity, we'll omit specifying the key ring and error mailbox name fields. The abbreviation for the message is

```
Key Ring Name Fields-------------
Primary Name Field---------------
  Name:        ((,CUproject),/home/data)
  Label:       file
Name Field-----------------------
  Name:        ((,CUproject),my_reply)
  Label:       reply
Error Mailbox Name Field---------
Payload: open r file
```

The `Key Ring Name Fields` names a collection of key rings to be used to extract permissions. The first name field specifies the resource being accessed by giving the task's logical name for the resource. The second name field names a resource for which the sender is the resource proxy. The file system resource proxy can use this resource to send replies. Name associations for both of these named resources get added to the resource proxy's inbox frame.

If the Core can't handle the message, the reason will be put into the designated error mailbox. Errors include an inconsistency in the arbitration for one of the named resources, or an invalid name space specification, for example. Note that the error mailbox need not be attached to the sending task; a separate task can be the error handler.

The payload tells the resource proxy what the request is. The resource proxy must know how the fields in the payload are related to the name fields specified in the envelope. Here, we'll assume that the resource proxy knows that the return mailbox is the second name field.

The router in the Core now constructs the inbox envelope,

```
Primary Name Field---------------------
  Name:          X8332
  Label:          file
  Public Data:    local
                 HP-UX/10.20
  Private Data:   /users/bill/data
  Permissions:    read
                  write
Name Field-----------------------------
  Name:          X2932
  Label:          inbox
Payload: open r file
```

and delivers the message to the inbox of the designated resource proxy. It puts the name associations for names `X8332` and `X2932` into the frame associated with the task's inbox. Hence, when the resource proxy sees `open r file`, it can use

the label `file` to find the logical name `((,inboxFrame),X8332)` should it need to refer to the resource designated `file` in the payload.

The resource proxy knows that its internal name for this resource is `/users/bill/data` and that the task has both read and write permission. It now performs the operation without further access to the Core. Of course, nothing prevents the resource proxy from forwarding the message to another task.

In this example, the returned value is a handle to the file. There are a number of ways to return the handle. The key is to make sure that the correct resource proxy gets the request and can associate the handle with the correct file.

- Return the handle in the payload of the reply. When using the handle, the requester names the file (to get the message to the proper resource proxy) and includes the handle in the payload. The resource proxy has no control over which tasks use the handle; any task can put the handle into a payload. However, the permissions get extracted on each request, so the resource proxy can enforce access rights.
- The resource proxy modifies the repository entry for the file, adding a new permission indicating the file is open. It would then give the requester a name association for the key that unlocks this permission. The requester would make requests in the file language making sure to specify a key ring that holds the file handle key. The name association for this key could be forwarded to another task unless a grant authorizer is named. The original permissions will be presented along with the permission representing the file handle.
- Create a new resource representing the file handle and naming the file system as its resource proxy. The requester can name this resource and include just the parameters for the desired operation in the payload. The resource proxy can control the use of this new resource only if it names itself as a grant authorizer. Otherwise, the task may forward the resource to another task. Again, the permissions are extracted on each request.

Let's look at the last option. The resource proxy returns this information to the requester by associating a specific resource with the file handle. (This resource can be created on the fly by the resource proxy, but it will be more efficient if the proxy keeps a pool of resources for this purpose.) The resource proxy sends a message to the requester designating the requester's logical resource with a payload containing the return code. The resource proxy allows future use of the file handle by giving the requester a name association for the resource representing the file handle.

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        ((,inboxFrame),X2932)
  Label:       reply
Name Field-----------------------
  Name:        (,FH_/users/bill/data)
  Label:       fileHandle
Error Mailbox Name Field---------
Payload:  1
```

The Core knows who gets the message because the resource with the resource proxy's logical name `X2932` specifies the requester as the resource proxy for this resource. The Core now constructs an inbox message for the requester.

```
Name Field-----------------------------
  Name:          Q83
  Label:         inbox
Name Field-----------------------------
  Name:          L448
  Label:         fileHandle
Payload:  1
```

The requester sees that the return code indicates success. The task now associates the resource named `L448` with the file handle. It knows that the primary resource, denoted with the label `inbox`, refers to the mailbox it allowed the resource proxy to use.

Now that the file has been opened, the application can read the contents by sending a message

```
Key Ring Name Fields-------------
Name Field-----------------------
    Name:       (,L448)
    Label:
Error Mailbox Name Field---------
  Payload:  read 1024
```

which gets delivered to the resource proxy. This proxy accesses the requested data, here 1024 bytes, constructs a reply envelope and puts the data from the file into the payload. Note that we've assumed that the resource proxy has kept the return resource. If we can't make this assumption, or if we want the data returned to a different mailbox, we need to send the resources explicitly. However, transferring these resources only once will be more efficient and controls which task gets the file data. In this scheme, the resource proxy can revoke the access to the file by deleting the file handle resource.

---

## 10.4. Creating a File

We've seen how to use a file, but how did it get created in the first place? In a conventional system, the client would say something like `open w /users/bill/new`, and the file system would create a file with this name. In the Client Utility we do things differently because the process requires creating the file and its repository entry.

The requester sends a message to the Core

```
Key Ring Name Fields-------------
Name Field-----------------------
    Name:       (,/myhome)
    Label:      /myhome
Name Field-----------------------
    Name:       (,replyResource)
    Label:      reply
Name Field-----------------------
    Name:       (,my_user_frame)
    Label:      inbox
Name Field-----------------------
    Name:       (,file_grammar)
    Label:      grammar
 Error Mailbox Name Field---------
   Payload:  open w /myhome/new
             attributes{grammar,DESC="Bill's new data" PW=""}
```

Since the primary resource is a in the `FileAPI`, the resource proxy for the file system will get the message

```
Primary Name Field---------------------
    Name:           B8857
    Label:          /myhome
    Public Data:     local
                    HP-UX/10.20
    Private Data:    /users/bill
    Permissions:     create
Name Field-----------------------------
    Name:           T918
    Label:          reply
Name Field-----------------------------
    Name:           Y1162
    Label:          inbox
Name Field-----------------------------
    Name:           C8
    Label:          grammar
Payload:  open w /myhome/new
            attributes DESC="Bill's new data" PW=""
```

At this point the file system handler knows that someone wants to create a file `new` in directory `/users/bill` and the requester has `create` permission. The file system handler can now create the file `/users/bill/new` in the underlying file system. It also registers this resource with the Core by sending the message

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        (,Repository)
  Label:
Name Field-----------------------
  Name:        (,Unix_files)
  Label:       Unix_files
Name Field-----------------------
  Name:        (,core_reply_mailbox)
  Label:       reply
Error Mailbox Name Field---------
Payload:  register Unix_files atts(DESC="Bill's new data" PW="")
            Resource((/users/bill/new),(local,HP-UX/10.20))
            Permissions(read_key,read),(write_key,write)
```

Because no name fields were included for the keys specified in the permissions, the Core will create new keys as part of the request. The file system resource proxy will get back a reply

```
 Name Field-----------------------------
   Name:          G4756
   Label:
 Name Field-----------------------------
   Name:          G8237
   Label:
 Name Field-----------------------------
   Name:          W2295
   Label:
Payload:  1
```

Then, the proxy sends a message

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        ((,inboxFrame),T918)
  Label:       inbox
Name Field-----------------------
  Name:        ((,coreInbox),G4756)
  Label:       /myhome/new
Name Field-----------------------
  Name:        (,FH_/users/bill/new)
  Label:       file_handle
Name Field-----------------------
  Name:        ((,coreInbox),G8237)
  Label:       readKey
Name Field-----------------------
  Name:        ((,coreInbox),W2295)
  Label:       writeKey
Error Mailbox Name Field---------
```

```
Payload:  1
```

to the requester who sees

```
Name Field----------------------------
  Name:          N3991
  Label:         inbox
Name Field----------------------------
  Name:          K27
  Label:         /myhome/new
Name Field----------------------------
  Name:          U9221
  Label:         file_handle
Name Field----------------------------
  Name:          Q22
  Label:         readKey
Name Field----------------------------
  Name:          V3957
  Label:         writeKey
Payload:  1
```

In this example, the requester includes its own name for the file in the payload. If it had wanted to hide this name from the file system resource proxy, it could have specified a different name and asked the Core to rename it in a separate message.

---

## 10.5. Coordinating Names

We've seen how a task can ask a resource proxy to act on a file without using a common name. In this case, the resource proxy never used the name of the resource. In this Section we'll look at how two tasks can coordinate their actions on a file when they each have a different name. We'll assume that they have previously exchanged reply resources so they can talk to each other.

Chuck has just created a file which he'd like to have Sally read but not modify. He can construct the message

```
Key Ring Name Fields-------------
Name Field-----------------------
    Name:      (,my_hearts_desire)
    Label:     sally
Name Field-----------------------
    Name:      (,/home/new)
    Label:     /draft
Name Field-----------------------
    Name:      (,read)
    Label:     read_key
```

```
Error Mailbox Name Field---------
  Payload:  Please review /draft
```

Sally will get

```
Name Field-----------------------------
  Name:          W3822
  Label:          sally
Name Field-----------------------------
  Name:          D5623
  Label:          /draft
Name Field-----------------------------
  Name:          K3
  Label:          read_key
Payload:  Please review /draft
```

Sally can construct her comments and send the reply

```
Key Ring Name Fields-------------
Name Field-----------------------
    Name:      (,I_wish_hed_ask_me_out)
    Label:     chuck
 Error Mailbox Name Field---------
  Payload:  Typo in line 5, should be "you and I"
```

which Chuck receives as

```
Name Field-----------------------------
  Name:          S3991
  Label:          chuck
Payload:  Typo in line 5, should be "you and I"
```

---

## 10.6. Declaring a Language

A language (API plus Client Utility specific information) is a resource just like any other. However, the resource proxy for languages imposes a very important rule. **There can not be two languages with the exact same attributes.** This rule makes it impossible for another machine to hijack requests for local resources.

Consider what might happen without this rule. The local file system proxy creates a language resource representing the API for files on the system and gives it the attributes `TYPE=Unix_files LOC=here`. Now, a malicious machine attaches and exports some resources, one of them being a language with these attributes. The malicious machine then exports resources in this language which get registered in the importing Core repository.

Applications running on this system need to get a name for the file system language before they can start dealing with files, so they do a look-up specifying these attributes. A task that subsequently does a look-up will match both languages, but it might get the malicious version of the language. From that point on, all its file requests will use the attacker's resources. Of course, the task could protect itself by setting its arbitration policy to report an error on multiple hits during look-up of the language, but then it has no way of knowing which of the languages is legitimate.

Except for this detail, declaring a language is exactly like asking a resource proxy to create any other resource.

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        (,language)
  Label:
Name Field-----------------------
  Name:        (,core_reply)
  Label:
Name Field-----------------------------
  Name:        (,owner_key)
  Label:
Name Field-----------------------------
  Name:        (,register_key)
  Label:
Name Field-----------------------------
  Name:        (,unregister_key)
  Label:
Error Mailbox Name Field---------
Payload:  create UnixFS atts(TYPE="Unix_files",LOC=here)
```

If the requester has the key that unlocks the permission to create a new language, and there is not already a language with the specified attributes, the Core will create the language and repository entry. The requester will get the reply

```
Name Field-----------------------------
  Name:          Y824
  Label:        UnixFS
Payload:  1
```

In general, only the owner can register or unregister a language, but these permissions can be delegated by sharing a name association to the specific keys.

## 10.7. Copying or Moving a Name Association

There are times when we want to move a name association from one place to another. For example, a task that has just received a message would like to receive a second message before it completes processing the first. If that task needs to use any of the name associations from the first message, it will have to move them to another frame because the Core clears the inbox frame of any name associations before delivering the message. Similarly, a task may wish to copy an association into another frame, perhaps to set up a child process's environment or for delegation. In either case, the task sends the message

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        (,inbox)
  Label:       outframe
Name Field-----------------------
  Name:        (,tempFrame)
  Label:       inframe
Name Field-----------------------
  Name:        (,inputArg)
  Label:       name
Error Mailbox Name Field---------
Payload:  move name from outframe to inframe
```

## 10.8. Copying a Core Managed Resource

There are times when a task wants to copy a Core managed resource. For example, the simplest way to delegate use of a key yet reserve the right to revoke the delegation is to create a copy. A name association for the copy can be given to the delegate without a key giving copy permission. This key will open the same locks as the original. When the granting task wants to revoke the delegation, it simply deletes the copy.

Tasks can also use copying to set up a child task. By default, the child will start with the same environment as the parent except for the bootstrap frame. At child set-up the parent can specify that the child get name associations for the parent's Core managed resources or associations for copies of them. However, the task may want to take control, sharing some name association for some frames and making copies of other.

In either of these cases the Core is told to make the copy with a message

```
Key Ring Name Fields-------------
Name Field-----------------------
  Name:        (,readKey)
  Label:       read
Name Field-----------------------
  Name:        (,myUserFrame)
  Label:       frame
Error Mailbox Name Field---------
Payload:  copy read to frame as readDelegate
```

---

# 11. RELATED WORK

Efforts that bear some degree of similarity with Client Utility

- 11.1. Introduction to Related Work: Client Utility architectural principles
- 11.2. Object Brokers: Relationship to object brokers
- 11.3. Distributed Operating Systems: Commercial and research distributed systems
- 11.4. Metacomputing: Wide area computing environments
- 11.5. Network Computer (NC): Thin client approach
- 11.6. Jini: Sun Microsystems services environment
- 11.7. Other Systems: Non-computing related systems

---

## 11.1. Introduction to Related Work

The Client Utility is based on the following architectural principles:

- Uniform representation for resources/services
- Utility manages/manipulates representation of resources
- Resource identity based on attributes, not global names
- Resource addressability through only through names
- Resource access protection through capabilities
- Integrated security architecture allows delegation with revocation
- Allow reasoning about APIs and attribute models
- Controllable dynamic-binding, registration, name lookups and invocations
- Seamless Distribution

These principles facilitate the creation of a distributed middleware OS that provides a consistent set of abstractions and services for applications and

resource providers in a wide area distributed system. This middleware operating system ensures secure access to resources, dynamic resource discovery/lookup, seamless integration of new technologies/APIs, availability, replication, fault-tolerance, location-independence, scalability, manageability and customizability. In fact, it acts as the substrate on which the vision is realized. Several research and commercial efforts have similar goals and share some of the architectural principles mentioned above. This note discusses these efforts.

---

## 11.2. Object Brokers

CORBA/COSS provides a disparate set of mechanisms that may be used to implement many of the abstractions of Client Utility. CORBA/COSS provides mechanisms such as Trader APIs, Property or attribute description APIs, naming APIs and so on. The current Client Utility implementation uses extended versions of many of these APIs in its implementation. In an abstraction stack, the Client Utility abstractions lie above those presented by CORBA/COSS. CORBA/COSS present a set of implementation tools, Client Utility is a new end-user computing paradigm that may use some of these tools as part of its implementation.

The Java platform APIs provide a similar set of APIs (JavaSpaces, management, electronic wallet *etc*). These do not add up to a platform for accessing computing resources in a large-scale distributed system, let alone a new end-user computing paradigm.

BusinessWare from Vitria shares some of the characteristics of the services architecture proposed by CU. It is an application integration product that can integrate disparate applications across many existing application infrastructures. The architecture is based on a publish-subscribe information bus that provides reliable event management, as well as authentication, authorization and encryption. Connectors atop this communication infrastructure provide integration with legacy applications by mapping application-specific invocations to those of the information-bus. A business process automator graphical tool allows for the creation/manipulation of workflows representing business processes.

On the positive side, the architecture provides good solutions for integrating business processes, composing event channels and generic resource discovery.

On the negative side, issues of openness (of the architecture), scalability and extensibility are handled through adhoc means within the implementation, as opposed to architecting for it. Although, CU has not provided for the same support for process-worflows, it architecturally represents component services and the composition of component services. Thus, the CU architecture is geared towards reasoning about services and can represent unanticipated use, manipulation and composition of services. Also, distribution, scalability and garbage collection are dealt with at an architectural level as opposed to simply expecting the implementation to handle it. Finally, the CU architecture's representation of protection domains and manipulation of names is completely unique and not represented in BusinessWare or any other system that we know of.

---

## 11.3. Distributed Operating Systems

Inferno (Lucent) is a small operating system that can run directly on hardware platforms or hosted on standard operating systems such as NT and Linux. Its intention is to be a distributed architecture independent Network OS. A few salient features include, all resources modeled as files, a VM to hide hardware architectural differences, personalizable name-spaces, a language to aid programming and miscellaneous cryptographic security mechanisms. Technologically, there are significant differences between the Client Utilty and Inferno. Although, both efforts have an uniform abstraction for all resources, Inferno believes that all resources can be treated like file, whereas the Client Utility effort does not believe in such overloaded semantics.

The Client Utility environment is geared towards interaction in a large-scale distributed environment and hence introduces features such as identity through attribute descriptions, scalable lookup and brokerage services, dynamic extensibility and an inter-machine trust and interaction model. Inferno, simply lacks these features. Finally, security was added on to Inferno as an afterthought, however the Client Utility started with the presumption of malicious applications and consequently implements a sophisticated capability-based resource-protection scheme. Inferno is a good native operating system for small devices and appliances, whereas the Client Utility is a platform for creating/managing/deploying/finding services in a large-scale distributed environment.

WebOS provides a common set of services that acts as an OS for large-scale distributed system. These services include persistent storage, resource discovery/management, security and remote process execution. Unfortunately, again these are provided as a disparate set of services - the abstractions that would tie these disparate services together do not exist in this system. In this sense, the use of the term Web"OS" is slightly misleading. A realization of Client Utility does include a middleware OS that provides similar services subsumed by an uniform resource abstraction.

In terms of goals and architectural principles, Millenium is very close to Client Utility. Millenium's goals include seamless distribution, worldwide scalability, transparent fault tolerance, security, resource management, resource discovery *etc*. Very little is known about the actual technology, since the only real source of information is a small write-up that is very short on details. The only differences seem to be the open-ness of the proposed system - Millenium does not seem to consider issues of allowing competitive access to services/resources to be very important. In terms of core technology, Millenium does talk about attribute based identity, aggressive abstraction, location/storage irrelevance, dynamic binding and introspection - all features of Client Utility technology. However, at this point it is not clear, how far the implementation of Millenium has progressed, but it is certainly a very visible effort at Microsoft Research.

---

## 11.4. Metacomputing

Legion from University of Virginia is focussed on creating a global virtual computer that presents an illusion of a powerful computer to any user. The focus is clearly on facilitating high-performance parallel applications. There are however enough similarities in the goals of Client Utility and Legion - location autonomy, secure access to resources, extensibility, resource management, interoperability, seamless distribution and so on. There is also a lot of focus on application-level parallelism and the creation of a single global name-space. On the other hand, Legion does not take into consideration legacy applications or integration with industry standards. Legion is what Client Utility would be limited to, if we had primarily focussed on high-performance applications - thus, issues like competitive access to services, hooks for accounting/payment would not receive much attention.

Globus (Globus Consortium) is an environment for developing and deploying large-scale distributed applications. A low-level toolkit provide support for communication, authentication and data access, while higher-level services provide parallel programming tools and functionality such as schedulers. This infrastructure provides an interesting set of mechanisms (much like CORBA/COSS) for building distributed applications. Apart from the obvious differences in the overall goal, Globus as compared to CU lacks the abstractions that help reason about distributed services/resources and execution entities. Ideas and mechanisms from Globus could be used as part of the underlying implementation for CU, but the goals, vision and architecture are different.

There are also a few other lesser known efforts such as (Hadas) and Globe that focus on infrastructures for large-scale distributed systems.

---

## 11.5. Network Computing

As a means of solving the problem of Total Cost of Ownership, the NC model of remote computing has been proposed, where relatively simple client hardware act as front-ends to large-backend servers that provide most of the computing infrastructure. The philosophy is that by simplifying the client hardware, a significant portion of the management costs can be moved from a myriad of clients to a set of tightly controlled servers.

Centralized servers are well understood and quite a few hardware solutions for the client-end hardware have been proposed (**NCs, NetPCs, JavaStations**). The comparison with CU lies mainly in the software infrastructure that is provided to support the NC environment. In many cases, such infrastructure consists of remote display protocols, such as those from **Hydra** (Microsoft) and **ICA** (Citrix) that allow remoting of Win95/NT desktops. These protocols can be realized as a combination of CU resource handlers and emulation layers. Some infrastructures, such as the Business Computing Utility (IBM) are a little more extensive and provide remote access to a services maintained/managed by IBM at their data centers. Such, end-user functionality is critical for reducing the management costs at IS shops, however unlike **BCU** which is a conglomeration of several point-solutions, CU provides an extensible, yet uniform infrastructure that can support such end-user paradigms and more. Other software infrastructures, such as **Metis** mentioned above and the distributed services of the Java platform offer other alternatives.

Perhaps, one of the more advanced NC-computing infrastructures is the **Tarantella** environment (SCO). Tarantella is a server-based application-broker that runs on various UNIX platforms. In essence, server-based applications can be accessed and customized by users from any java-based client (including many industry standard browsers). There are three important components to this technology - interposition of the X-protocol at the application servers, adaptive display protocol, called AIP (Adaptive Internet Protocol) and some degree of user session management through a database maintained at the server.

Currently, the implementation supports centralized application servers and there is no support for heterogeneity in applications (that is, NT applications are not supported). In short, Tarantella supports the notion of a virtual personal computer, one of the applications that can be built atop CU, without the heterogeneity or distribution promised by CU. Perhaps the fundamental difference is the fact that Tarantella simply interposes the X-protocol on application servers, while CU can potentially interpose any resource access. Also, unlike CU which has an uniform resource access protocol, Tarantella supports a single resource-specific protocol. Tarantella is a good point-solution for redirecting X-based display protocols, while CU is a generic infrastructure for manipulating resources of any kind. Finally, Tarantella does not claim or provide any support for management, deployment or composition of services, the main strengths of the CU infrastructure.

---

## 11.6. Jini

Sun Microsystem has a product for providing services in a distributed environment. Its stated goals are much like those of the Client Utility, a unified environment for providing services to clients, but its architecture is more limiting. According to the Sun materials, Jini is "intented for a trusted work group of limited size running Java applications". When we examine the architecture, we see the reason for each of these limitations.

Key to Jini is a JavaSpace, a distributed database containing name value pairs. The names are globally agreed upon names for services; the values are RMI stubs. When a task finds a resource in the JavaSpace, it gets back one of these RMI stubs. The task then uses this stub to communicate directly with the service provider.

We can now examine each of Jini's limitations.

- Trusted work group: There is no security model imposed other than the usual Java sandbox. In particular, tasks must trust the RMI stubs. Also, Jini provides no mechanisms for service providers to determine the access rights of requesters.
- Limited size: A JavaSpace, being a distributed database, doesn't scale. Sun puts a limit of 1,000 machines on a Jini environment, but experience with Linda tuple-spaces, essentially the same thing, shows that performance starts to degrade with more than a couple of hundred machines.
- Running Java applications: Since the value returned from the look-up is an RMI stub, either the application must be in Java, or a part of it must know how to invoke the stub.

Client Utility makes none of these assumptions. It is designed to scale to millions of machines; strong security is provided between machines; and applications can be written in any language.

---

## 11.7. Other Systems

Stanford's Digital Libraries Effort (partially funded by HP-Labs) provides a set of tools and mechanisms for discovering, managing, paying for and negotiating access to vast amounts of information available in a large-scale distributed system such as the internet. The tools and services work very well for sources of information, but abstractions that support the requirements of generic services/resources are not so well defined. Specifically, abstractions to deal with naming, security and binding critical for supporting seamless but competitive access to services do not exist. However, particular facets of the Digital library implementation are well thought out, such as the meta-data architecture, which has influenced the Client Utility's meta-data architecture. The Client Utility technology, on the other hand, provides mechanisms that aid the deployment, management, paying for and negotiating for access to services on the internet. Fundamentally, however the utility provides an end-user computing paradigm and Stanford's effort in Digital Libraries provides an interesting set of implementation tools.

Metis Thin Client Application Framework (IBM Research) is a development platform for building thin-client applications. They expect that applications will be componentized in different ways to be deployed in an NC dominated world. Hence, Metis provides secure, seamless access to resources. In terms of technology, there are strong similarities between the Metis application

framework's resource access/management services and those provided by Client Utility. Clearly, there are strong differences between the goals and visions of the two efforts.

Another IBM project, Flexible Control of Downloaded Content, addresses the problem of sandboxing downloaded executables. Naturally, they address many of the same issues in Client Utility, but their more limited goal led them to a less comprehensive solution.

GeoPlex (AT&T Labs) shares many of the same technological goals as CU, in that it intends to create a secure, scalable, standards-based, programmable and heterogeneous middleware for supporting internet services. However, the two efforts differ significantly in their architecture. The Geoplex abstractions include

- Hop consisting of sundry hardware that allows Geoplex networks to talk to non-Geoplex hardware
- Stores that implement a list of services including billing and customer-care
- Core supports databases that maintain usage-information, billing records
- Gates form a secure end-point that encapsulates firewalls, access-control and protocol-mediation

These abstractions seem to encapsulate the three-tier approach to building applications as well as network interactions, as opposed to the services that are to be actually deployed. Hence, the ability to reason about services, extend and manipulate them would be rather limited, while creating intelligent networking infrastructures such as VPNs should be facilitated by this effort.

BroadWay (X Consortium) integrates the web model of access to data with the X-protocol. This extends the X-protocol with mobility across firewalls and uniform displays. A CU resource handler and emulation code could capture the implementation of the Broadway architecture, but similarities end there. The goals, vision and technology of CU are geared towards general purpose services, while Broadway tends to add value to the X-protocol.

WABI (Sun Microsystem) is an emulation layer for supporting Windows applications on a Solaris platform. A deployed CU system may well include an emulation layer that transforms applications calls.

---

# 12. SUMMARY AND CONCLUSIONS

The Client Utility as described in these pages has some important implications for how people think about computing. Whereas today they think about machines they have purchased and software they have installed, the Client Utility leads them to think of services they can use. This Architecture for Services leads to a new way of providing computing. Services are built out of existing services by composition. These services are enhanced by interposition of new services that provide customization and additional functionality without rewriting entire applications.

This new view of computing as services is obtained by adhering to a few principles. The Client Utility deals only with resource metadata who's form is independent of the type of resource. The Core sees only local tasks requesting resources and local resource proxies; the Core is unaware that there are other machines. A distinction between "identity" and "identifier" is maintained in the sense that a task is its protection domain as far as the Core is concerned. Finally, the architecture provides mechanisms but does not specify policies.

---