e"speak

Architecture Specification

Version Beta 2.0
September 1999

HEWLETT® PACKARD

License Agreement

This license is reproduced from the license that you must agree to when the software is installed:

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

This document specifies the e-speak architecture. It defines the abstractions presented by the system and the components that implement those abstractions, and shows how the components interact to create useful services. The following companion documents are also available:

- *E-speak Programmers Guide* defines the interface for e-speak programmers and system developers building e-speak-enabled applications.
- *E-speak Installation Guide* shows how to install e-speak and how to run some simple applications.
- *E-speak Contributed Services* describes several sample applications included with the distributed software.
- *E-speak Tools Documentation* shows how to use tools provided for analyizing the system.

Vision

Computing with e-speak is a paradigm switch, aiming to bring a "just plug in, use the services you need, and pay per usage" model to computation, as opposed to the "install on your machine and pay per installation" model of computation prevalent today. E-speak is the infrastructure that realizes the vision of such a model. Instead of thinking of computing as some hardware you buy and the software you install on it, e-speak encourages you to think of computing as a set of services you access as needed.

The reality of computing today is that it is much more complex than a utility like the electric or water system. An immense variety of computing resources exists, both in type and in power, and a newer, faster, cheaper, or better resource will probably be invented by the time you finish reading this sentence. This dynamism is a formidable challenge to interoperability.

At the same time, most of these resources are being connected to each other on a range of scales, from homes to companies to the entire globe. The hardware necessary to support such a computational utility is already available and getting better by the day. On the software front, though the Web has essentially achieved the status of a *data* utility, actual computation remains mainly confined to individual machines and operating systems.

E-speak enables a *computation* utility by interposing on and mediating every resource access in a process called *virtualization.* This broad abstraction yields a model where machines, ranging from a supercomputer to a beeper, can be looked at uniformly and can cooperate to provide and use services.

Goals

E-speak aims at enabling ubiquitous services over the network—making existing resources (e.g., files, printers, Java objects, or legacy applications) available as services, as well as lowering the barriers to providers of new services. The infrastructure's goal is to provide the basic building blocks for service creation, including:

- Secure access to resources and service
- Usage monitoring, billing, and access control
- Advertising and discovery of new services
- Mechanisms for negotiation to find the "best" service
- Independence of operating system, language, and device
- Ability to support large enterprise-wide, intra-enterprise, and global deployments

Architectural Philosophy

This document specifies the e-speak architecture. There are four key concepts:

- **Resource**: Any computational service, such as a file or a banking service, that is virtualized by e-speak.
- **Client**: An active entity that requests access to Resources or responds to such requests.
- **Protection domain**: The part of the e-speak environment visible to a Client. This is analogous to a Unix shell.
- **Logical machine**: An active entity that performs the operations needed to implement e-speak.

E-speak is based on the following:

- All Resource access is mediated; e-speak sees all Resource requests.
- All Resource access is virtualized; e-speak maps between virtual and actual references.
- Names for Resources are shared by convention only; e-speak keeps a separate name space for each Client.

This document does not specify anything outside of the e-speak architecture. However, some implementation details are included to illuminate some points. These sections are marked “informational.”

Environment

E-speak is designed to work in a hostile, networked environment such as the Internet. It isolates service providers and their clients from an inherently insecure medium while allowing them to negotiate safely, form contracts, and exchange confidential information and services without fear of attack.

Intended Audience

This *E-speak Architecture Specification* describes the lower-level interfaces of e-speak for:

- Implementors of Client libraries to provide a higher level of abstraction for e-speak
- Implementors of utilities and tools to manage and manipulate e-speak
- Implementors of e-speak emulation routines that will be used in the run-time environment for legacy applications
- Implementors of extensions to existing services and resources used by Clients
- System administrators who will implement policies for security and resource lookup
- Those designing and building their own implementations of e-speak

Structure

This specification consists of the following major sections, in the order listed:

- An overview of the e-speak architecture
- A description of the data structures used by e-speak to describe Resources–Resource metadata
- The interfaces to the e-speak platform that are exposed as “Core-managed Resources”
- A description of Vocabularies, the mechanism for processing Resource descriptions to discover and match Resources to the Client’s description of Resources needed
- The e-speak Application Binary Interface (ABI), which is used by Clients to communicate with the e-speak platform

- The Inter-Core Communication Architecture and the E-speak Service Interchange Protocol (ESIP), which is used by e-speak plaforms to communicate
- The exceptions that can be generated by the e-speak platform
- The e-speak Event Service
- The e-speak management infrastructure (informational)
- The e-speak Respository used for storing Resource metadata (informational)
- A description of how the current e-speak security mechanisms are used (informational)
- A glossary of terms
- A brief description of probable future extensions to e-speak (informational)

Conventions

There are several document conventions worth noting:

- New terms are introduced in the document flow with italics.
- Programmatically visible architectural abstractions are written with the first letter of each word capitalized, such as Protection Domain.
- Logical names, method names, and other programmatic labels are written in `Courier` font.
- Even though e-speak is independent of the programming language, the specification uses Java syntax.
- Sections describing material outside of the architecture are denoted with the word “Informational” in the chapter or section title.

Chapter 2 Architecture Overview

All system functionality and e-speak abstractions build on top of one single first-class entity in the e-speak architecture—a Resource. A Resource is a uniform description of active entities such as a service or passive entities such as hardware devices. Unlike most platforms, e-speak deals only with data about Resources, *metadata*, and not Resource-specific semantics. Thus, a file Resource within the e-speak environment is simply a description of the attributes of the file and how it may be accessed. The e-speak platform does not access the file directly. A Resource-specific handler that is attached to the e-speak platform receives messages from e-speak and directly accesses the Resource.

Access to e-speak is provided by the e-speak Service Interface (ESI). Client applications and Resource Handlers are linked with a library that provides this interface. The library communicates with the e-speak platform using the e-speak Application Binary Interface (ABI). The e-speak platform mediates all Resource access. Every access to a Resource through e-speak involves two different sets of manipulations:

1. The e-speak platform uses its Resource descriptions for dynamic discovery of the most appropriate Resource, fine and coarse-grained access control, transparent access to remote Resources, and sending Events to management tools.

2. The Resource-specific handler directly accesses the Resource such as reading the disk blocks for a file.

E-speak treats all Resource accesses in exactly the same manner. This mediated yet uniform access is the design principle that allows the e-speak environment to accommodate any kind of Resource type flexibly, even Resources dynamically defined after the e-speak system has started.

The e-speak platform maintains an environment for each of its Clients, called a Protection Domain. A Protection Domain fixes the set of Resources available to the Client at any given time. The e-speak platform will not allow a Client to access a Resource that is not in the Client's Protection Domain.

A single instance of the e-speak platform is called a Logical Machine. Figure 1 shows a single Logical Machine. There may be multiple Logical Machines on a single physical machine, or the components of a Logical Machine may be distributed across multiple machines. Logical Machines are independent entities that communicate using the E-speak Service Interchange Protocol (ESIP). The relationship between ESIP and the e-speak ABI is shown in Figure 2.

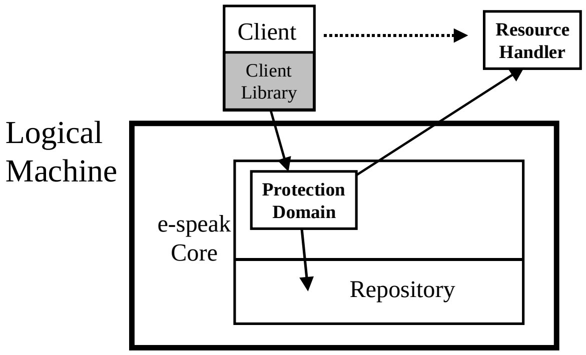

Figure 1 Resource access in e-speak

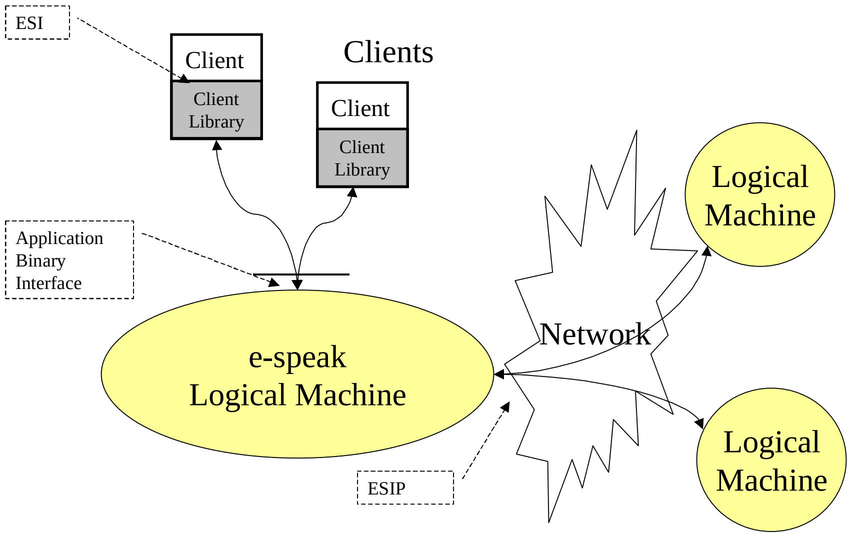

Figure 2 ESIP and the e-speak ABI

Each e-speak Logical Machine has a single instance of the e-speak Core. All Resource access is through the Core that uses the Resource metadata to mediate and control each access. To access a Resource, a Client sends a message to the e-speak Core naming the Resource. The e-speak Core uses the Resource metadata to determine if access is allowed and to locate the Resource Handler. Assuming access is granted, the e-speak Core forwards the message to the Resource Handler that, in turn, physically accesses the Resource.

Although Figure 1 shows the Resource Handler being outside the Core (i.e., in a separate process), the handler for some Resources is the Core itself; these Resources are inside the Core and are called *Core-managed Resources*. E-speak Clients can manage and interact with the Core by sending messages to these Resources. For example, one kind of Core-managed Resource is a Resource Factory. When a Client wants to create a new Resource instance, it sends a message to the Resource Factory to register the Resource metadata with e-speak.

Logically, there are three categories of Resource access:

- A service provider may choose to register the metadata of its service. The e-speak Resource model describes the contents of this metadata.
- A Client or service provider may look up a service and bind to it, prior to accessing it. The search rules and the information model for descriptions are defined by the registered Resource metadata.
- A Client or service provider may invoke an entry point on a service. The previous two are special cases of this last one, because they may be considered as invocations on entry points of Core-managed services.

In all cases, the mediating Cores perform access-control checks and name resolutions and generate monitoring information as part of this access.

Clients who wish to access a service do so through the e-speak Core, which uses the appropriate Resource metadata to route the Client request to the correct handler, after having performed all desired access control, name translation, and other e-speak functionality.

The following sections outline the various components of the e-speak architecture and describe the steps in a service access.

Mediation Architecture

Following are the main components of the mediation architecture:

- A set of Core-managed Resources inside the e-speak Core. The Core-managed services present the system functionality for managing the e-speak platform, including creating Protection Domains and their contents and managing Resource metadata.
- A Repository containing Resource metadata available to Clients of the Logical Machine. These are the metadata that the Core evaluates during any service access.

- A routing engine that mediates all service access messages based on the contents of the Protection Domain of the sender and the contents of the metadata of Resources referred to in the parameters of the message. The implications of this for naming and security are discussed below.

Resource Model

The Resource is a representation of an e-service within e-speak. Service providers register the metadata of their services (e-speak Resources) with the Core. This includes the information depicted in Table 1.

Table 1 Resource Model

Description	**An attribute-based specification of the Resource**
Vocabulary	The definition of the attribute grammar used in the specification
Resource Handler Mailbox	The process/thread/task that handles the Resource
Contract	Denotes the Application Programming Interface (API) supported by the provider, including version and similar information
Visibility and permissions	Access control information
Private Resource-specific data	Not interpreted by e-speak; meaningful to the provider; delivered to the Resource Handler as part of incoming messages
Public Resource-specific data	Not interpreted by e-speak; meaningful to the provider and the user of the Resource

The Resource metadata is maintained in the mediating Core's Repository. All functionality presented through the Core must have metadata within the Core. This is true even for the functionality provided by the Core itself.

Metadata System

The e-speak metadata system is based on the following architectural and semantic entities:

- Vocabularies are created as first-class Core-managed services through a VocabularyBuilder service. Thus, the model includes a metalanguage for creating a whole range of vocabularies with which to describe services, much like that of XML. XML document type definitions (DTDs) can be handled through the e-speak VocabularyBuilder.

  The representation of vocabularies as Resources ensures that they can be dynamically discovered and protected from illegal access, and that access to them is mediated as required, like any other service in e-speak. In the e-speak architecture, a created Vocabulary decides the validity of an attribute description provided by a registering service provider.

  The Core-managed Vocabulary service also includes a matching engine that is used to match Resource descriptions available in the Repository with search requirements of Clients of e-speak.

- Attribute-based service descriptions are used by service providers as part of service registration. These attribute descriptions are sets of name-value pairs in a specific vocabulary. The Vocabulary is either one that the service provider previously created (using the Vocabulary Builder) or discovered through the discovery facilities provided by the e-speak Core services.

- Search Recipes are objects that hold a Client's recipe for discovering a Resource. The Core uses this to process the Resource discovery request. A Search Recipe specifies what Resources the Client is looking for, how the lookup should be done, and what should be done if multiple matches occur. The Search Recipe contains the predicates and a Repository view mechanism with which to constrain the search. A search predicate is constructed with a Vocabulary and a constraint string expressed in that Vocabulary based on the Object Management Group trader services constraint grammar.

- The operational realization of the metadata system includes support for including arbitrary advertising services as part of extended searches, arbiters to optimize matches found through the Core Repositories, and integration of Vocabulary translation services with the lookup/discovery process. Advertising services provide scalability to service lookup in e-speak by supplying connections to machines that may possess the required service. Arbiters are used to effect special purpose optimizations such as handling multiple hits in lookups. Translation services can be integrated with Core-managed Vocabulary services or created as external services, thus allowing for translation between schemas and Vocabularies.

Naming Model

The e-speak naming system is based on the following principles:

- A naming system based on a Client-visible per-Protection Domain name space. Each Client has its own name space maintained in the Core (in the Client's Protection Domain) on behalf of the Client. Name spaces are maintained in container Core-managed Resources called Name Frames. The Name Frame in the Client's Protection Domain is called its default Name Frame. When the Client specifies the name of a service, the Core, starting with the default Name Frame, finds a mapping between the name and a name unique to this Core.
- The e-speak Core provides the only valid reference to a service as a name in the Client-specific name space. This is like a virtual address of a service. The physical address of a service, the Core's name is not a valid Client reference for a service.
- There are two ways for a Client to get a name for a service. First, another Client, application, or service provider can pass it the name. The receiver's reference for the passed service is not necessarily the same as the sender's, but both will map to the same Core name. Second, a Client may obtain a per-Client name through a bind call that requires a Search Recipe as a parameter. The e-speak system (Core and Client libraries) looks up the name in local Repositories,

known remote Repositories, and if necessary a global advertising service to locate the appropriate service and create a binding for the Client in its name space.

- Bindings in e-speak are objects that capture an algorithm. At their simplest, bindings may capture a Search Recipe. These bindings may be resolved and frozen to a specific Core name or names, resolved and cached, or simply resolved on each access. This gives the e-speak system an active naming model. Even when resolved, a Client reference may be bound to multiple Core names, which may be arbitrated prior to service access. This may be done by using a Client-specified arbitration service that picks one particular service from a list of services represented in a binding.

Security Model

E-speak security is built upon something like capabilities called *Keys*. (These are not cryptographic keys; they are a metaphor for something that opens a lock.) Clients possess sets of Keys that they send with each message. The Core tests these Keys against the metadata of the named Resources to see what privileges the Client has to access these Resources.

The metadata of a service contains two types of fields that specify how the e-speak security mechanisms will handle requests sent to the Resource:

- Visibility fields
- Permission fields

Each Resource's metadata has an Allow field and a Deny field (the Visibility fields), each containing a list of locks. When a Client sends a message to a Resource, the Core tests the Keys against these fields:

- Unless at least one of the Keys in the message opens a Lock in the Allow field, the Core acts as if the Resource does not exist. The test is bypassed if the Allow field is empty.
- If any of the Keys in the message opens a Lock in the Deny field, the Core acts as if the Resource does not exist.

The Visibility fields allow the Core to decide if a Client should be able to send a message to a Resource—should it be visible to the Client or not.

Once the Core has finished checking the Visibility fields, it checks the Keys against the Permission fields. The metadata describing a Resource can contain zero or more permissions. Each Permission field contains a string that is protected by one or more Locks. If any of the Keys sent by the Client in the message opens one of the Locks in the Permission field, the Permission string is delivered to the Resource Handler along with the message. The Resource Handler can interpret the string in any way it chooses; no semantics are associated with the string by the Core. For example, the handler for a file Resource might interpret the string "read" as meaning that the sender of the request is allowed to read the file. The Core checks each Permission field in the way described above, so the Resource Handler will receive a list of Permission strings such as "read", "write", "execute", and so on.

Keys are Core-managed Resources, so they can also be passed as parameters in messages (secondary Resources). This gives the receiver of the message possession of the Key and the privileges associated with that possession. A container Resource, called a *Key Ring*, is also defined for holding sets of Keys.

Communication

E-speak uses a mailbox metaphor to describe the interactions between Clients and the Core. This metaphor does not imply that any actual messaging is required, only that the interfaces are defined in terms of mailboxes. Mailboxes consist of two forms: an Outbox and a Core-managed Resource called an Inbox.

When a Client wants to use a Resource, it constructs a message consisting of a message header and a payload and inserts the message in the Client's Outbox. The Outbox is connected to the Core, which processes the information in the message header. If there is no error, the Core will construct a new header for the Inbox, add the payload from the sender, and forward the message to the designated Inbox. The Resource Handler reads the message header and the inserted payload to determine how to deal with the request.

E-speak uses peer-to-peer communication. The Core has no concept of a reply message. If the Resource Handler needs to return a value to the Client, it must specify a Resource listing an Inbox connected to the Client in the handler field of its metadata. Hence, in replying to a message, the Resource Handler changes roles with the Client.

Service Access

The basic mechanism Clients use to operate the e-speak Core is *message passing*. The Client libraries implement ESI, which is a more abstract interface wrapped around the message-passing layer.

All Clients have a communication point with the Core, the Outbox, from which they send Outbox messages composed of the elements shown in Table 2.

Table 2 Outbox message format

Field	Processed by Core	Contents
Message header	Yes	Message ID, primary Resource, other (secondary) Resources, and Key Rings
Payload	No	Request in handler API or straight data if this is a reply from a handler

The primary Resource is the Client's name for the Resource that it is accessing. The secondary Resources are the names of other Resources that will be delivered to the Resource Handler as parameters that it may need to handle the incoming message.

The e-speak Core then proceeds as follows:

1 Identifies the Client's name space from its Protection Domain.

2 Resolves all Resources specified in the message headers in that name space.

3 Finds the corresponding metadata in its Repository, checking visibility rights.

4 Extracts private Resource-specific data (RSD) and appropriate permissions of the primary Resource and all secondary Resources that have the same handler Inbox.

5 Monitors the operation for appropriate logging/filtering.

6 Creates a new message header appropriate for the handler.

7 Deposits new message header with the unchanged payload in the handler's communication point with the Core, called the Inbox.

The Inbox messages are formatted as shown in Table 3:

Table 3 Inbox message format

Field	Processed by Core	Contents
Message header	Yes	Primary Resource RSD and permissions, secondary Resources, RSD and permissions if same handler
Payload	No	Request in handler language, or straight data

Names for all Resources specified in the message header have to be created in the handler's name space.

ESIP Architecture

E-speak is designed to run in a distributed environment such as the Internet. To support distribution, e-speak introduces a new component: the Remote Resource Handler. The Remote Resource Handler is provided as part of the e-speak infrastructure and handles all requests for remote Resources; application programmers do not have to build their own remote Resource Handlers.

To obtain services from another Core, a Core must first obtain a Connection Object for that Core. A Client may receive a Connection Object in the payload of a message; for example, it might receive one in response to a request sent to an advertising service.

Using the e-speak Application Binary Interface (ABI), the Client's library passes the Connection Object to its Core, asking it to establish a connection to the remote Core. The Client's connection contacts the remote Factory and engages in a negotiation to establish various communication parameters such as the security of the communication channel. This communication channel is established between two Remote Resource Handlers (one at each Core) that are created during the initial phase of setting up the connection. The protocol used by the Remote Resource Handlers to communicate is ESIP—E-speak Service Interchange Protocol.

Once the communication channel is set up, the two Remote Resource Handlers export the metadata of a set of Resources to each other. To export a Resource's metadata to a remote Logical Machine, a Remote Resource Handler sends that metadata to its corresponding Remote Resource Handler on the remote Logical Machine—the importing Remote Resource Handler. The importing Remote Resource Handler registers the metadata with its local Core. Once the Resource is registered with the local Core, Clients on that Core can discover it by using attribute-based lookup, as described above. Some minor changes are made to an imported Resource's metadata. The most important is that the importing Remote Resource Handler registers itself as the Resource Handler for that Resource.

Now when a Client sends a message to a remote Resource, the local Core will route that message to the Remote Resource Handler. This forwards the message to the corresponding Remote Resource Handler on the remote Logical Machine. On the remote Logical Machine, the Remote Resource Handler forwards the message to its Core using the e-speak ABI, naming the destination Resource. The remote Core's metadata for the Resource contains the real Resource handler to which the message is sent. It is important to realize that the e-speak Core communicates with Remote Resource handlers using the e-speak ABI. Only Remote Resource Handlers use ESIP to communicate; this is intended to make it easy to replace the default implementation of a Remote Resource Handler and ESIP.

Both Cores test the Visibility fields: the Client's and the Resource's. There are two reasons for the tests by the Client's Core:

- The Client will get an early rejection if it cannot access the remote Resource, and no message will be sent on the network.
- It is possible to place extra controls on the Client's machine. If the machine is shared, not all users may be allowed to access the remote Resource. One way of doing this is to restrict which Clients get Keys that open Locks in the Resource's Allow field.

However, the Resource's Core takes ultimate responsibility for performing the visibility tests; it does not trust the Client's Core to do the checks correctly.

Once initial import and export have occurred, Clients on the two Cores can use Resources on the Core to which a connection has been made. When a Client sends a message to a Resource, it will likely include other Resources as parameters. When this happens, these Resources are exported by the Remote Resource Handler on the Client's Core as a side effect of sending the message. This makes the Resources available to the receiver of the message.

The basic mechanisms described above for importing and exporting metadata implement a pass-by reference model; the Resource's state is not migrated or copied. A pass-by value model is also supported: The Resource Handlers are responsible for generating and rebuilding a Resource state.

Remote Resource Handlers do not make all Resources available to other Cores. Two mechanisms restrict what will be exported:

- The Core will not accept a message from a Remote Resource Handler for a Resource whose name is not already in that Remote Resource Handler's Protection Domain. The set of Resources in the Protection Domain will grow as Resources are added to it that are implicitly exported as a side effect of sending them as parameters.
- The Remote Resource Handler will be given Keys that will be attached to messages before they are processed by the Core. Some of these Keys will open Locks in Allow and Deny fields. These Keys will ultimately determine what Resource will be exported to remote Cores.

An End-to-End Example

When the Client on Logical Machine A sends a message to its Core for a Resource on Logical Machine B, the following steps take place (see Figure 3):

1. The Client attaches any Keys it chooses and sends a message (using the e-speak ABI) to Core A naming the Resource.
2. Core A retrieves the Resource's metadata and tests the Visibility fields; this may cause the message to be rejected.
3. The metadata tells Core A to send the message to Remote Resource Handler A, using the e-speak ABI.
4. Remote Resource Handler A sends the message to Remote Resource Handler B using ESIP.
5. Remote Resource Handler B sends a message for the named Resource to Core B using the e-speak ABI.
6. Core B retrieves the Resource's metadata. The Visibility fields are tested. If this does not cause the message to be rejected, the Core attempts to match all Keys attached to the message to any locks in the Resource Permission fields.
7. Using the e-speak ABI, Core B sends the message to the Resource Handler, together with any Permission strings that have been unlocked. The Resource Handler interprets the Permissions to see if the requested action is allowed.
8. The Resource Handler sends the appropriate message to the Resource (if the Resource is an active entity such as a process) or executes the appropriate action

(in the case of a passive Resource such as a file). (This step is not shown in Figure 3, because this interaction is not part of the e-speak architecture.)

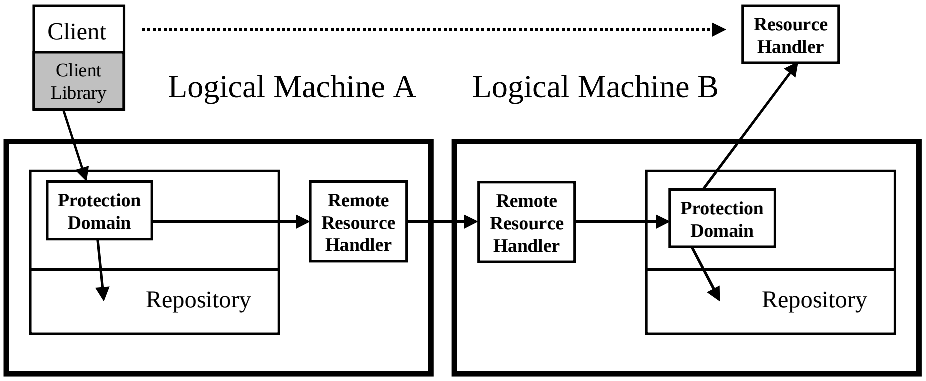

Figure 3 Distributed e-speak

The E-speak Service Interface (Informational)

An e-speak Client is an application running in its own address space written using an e-speak Service Interface (ESI). There is one ESI for each programming language supported. An example ESI is given in Chapter 1 of the *E-speak Programmer's Guide*. This provides a rich environment offering rapid, secure-service development, deployment, and management in a heterogeneous networked environment.

E-speak Services

E-speak has Event, management, authentication, and advertising services:

- The Event Service allows applications to collaborate by publishing Events and subscribing to Event distributors. The e-speak Core uses the Event Service to publish Events to the management service.

- The Management Services manage interconnecting sets of e-speak Cores, managing the distribution of Keys, metadata, and e-services registered as e-speak Resources.
- The Authentication Service provides a basic authentication mechanism for Clients based on password and user name. Other authentication mechanisms can be plugged into this service.
- The Advertising Service is used for distributed Resource discovery in large-scale environments.

Standards

The e-speak platform builds upon and uses existing industry standards wherever possible. In some cases, integration with industry standards is under way or planned. The specific areas of integration include:

- Database access—The persistent back-end for the Repository uses Java Database Connectivity, thus making it possible to send Repository queries to almost any relational database.
- Advertising services—The Advertising Service back-end is provided by Lightweight Directory Access Protocol.
- Transport protocols: The ESIP messaging stack supports pluggable transports. TCP/IP, IrDA, WAP, and HTTP are all candidate transports.
- Service description: The Vocabulary Builders support multiple different Vocabularies, including forthcoming support for XML and X.500 schemas.
- Component models: These models integrate the e-speak service abstraction with standard component models such as Java Beans and COM+.
- Management protocols and standards: Support for SNMP, ARM, and DEN is planned.
- Languages: An E-speak library exists for Java. Others such as C++, Python, and Perl are planned.

Summary

E-speak presents a uniform service abstraction, mediated access, and manipulation of Resource metadata. This creates an open service model, allowing all kinds of digital functionality to be reasoned about through a common set of APIs. New service types and semantics can be dynamically modeled using the common service representation of an e-speak Resource. Mediated access allows transformer services to be interposed seamlessly, allowing services such as text-to-speech translations, depending on Client appliance characteristics. Similarly, mediation allows third-party monitoring entities to throttle service access based, for example, on nonpayment.

The naming system provides active bindings and personal name spaces. The connection between Clients and Resources can be reasoned about and formed at run-time (upon each access if necessary) based on arbitrary search characteristics. Personalization of views and environments and hot-plug replacement of Resources all become possible.

The security model is based on a novel Key-Lock mechanism that can be used as electronic rights in e-commerce transactions. Keys allow fine-grained protected access to services. The Core processes Keys to control access, relieving the application programmer from much of the burden of determining whether to grant access.

The metadata system defines Vocabulary models as first-class entities in the system that can be reasoned about in the same manner as all other services. Translation and lookup through scalable advertising services are integrated into the model. Service location and discovery can thus seamlessly deal with a situation where the Client describes its requirement in an X.500 schema, while the service provider describes its service using an XML DTD.

The distribution model supports a flexible set of access methods. Thus, downloading printer drivers and the remote access of a file are equally well supported by the model. The separation of the infrastructure into interacting Logical Machines builds on the autonomous machine model provided by the Web.

These are the defining features of an open services platform. The collection of the capabilities discussed above will create an environment where services on the Internet can interact in a secure, dynamic, manageable way. The next chapter of the Internet (e-services) is being written, and e-speak will help us understand it.

Future Developments

Chapter 15, "Future Developments" describes an authenticated capability mechanism that is under development.

Chapter 3 Resource Descriptions and Specifications

E-speak makes a distinction between the data representing the state of a Resource and the data describing the management of the Resource. The Core mediates access to any registered Resource. However, e-speak is concerned only with the Resource state of Core-managed Resources, not with the Resource state of non-Core-managed Resources.

A Resource is described to e-speak by its metadata. The metadata is composed of a *Resource Specification* and a *Resource Description.* The Resource Description consists of information that provides the means of discovery for Clients. The Resource Specification includes:

- An Inbox that can be connected to the Resource Handler responsible for managing the Resource
- A specification of the security restrictions
- A variety of control fields

A Client registers a Resource by sending a message to a Contract Resource containing a Resource Description and a Resource Specification.

Together, Resource Descriptions and Resource Specifications include all information the Core needs to enforce the policies specified by the Client registering the Resource. If the registration succeeds, the Core returns a name bound to this Resource to the Inbox specified by the Callback Resource in the Outbox envelope.

ResourceSpecification

The ResourceSpecification class is defined below.

```
public class ResourceSpecification
{
   boolean byValue;
   ESName contract;
   ESName[] DenyKeys;
   ESName[]AllowKeys;
   ESMap metadataLocks;
   ESMap resourceSpecificLocks;
   ESMap publicRSD;
   ESMap privateRSD;
   ESName owner;
   ESName ResourceHandler;
   int eventControl;
   ESUID uid;
}
```

The type ESMap is serialized as ESArray (see Chapter 6, "Application Binary Interface" for the e-speak serialization format for ESArray). The e-speak convention for ESArray is that it consists of a sequence of pairs. Thus, the first and second element are a pair, the third and fourth element are a pair, and so on.

The current implementation of ResourceSpecification uses the type ResourceReference where ESName is given. ResourceSpecification is the abstract base class for ESName. ESName and not ResourceReference is passed by the e-speak Application Binary Interface (ABI).

boolean byValue;

If the byValue flag is True, the internal state of this Resource will be included with the Resource Specification and Resource Description sent to another Logical Machine. The Core will provide the value for Core-managed Resources. Currently, no mechanism is defined for providing the value of non-Core-managed Resources.

ESName contract;

The `contract` field is the name of the Contract Resource associated with the Resource. A Contract embodies the contract between the user and the provider of a Resource. It denotes such things as the Application Programmer Interface (API) passed through the payload of a message, as well as the name bindings that must be transferred and the access rights that must be presented. Every Resource must be registered in some Contract.

ESName[] DenyKeys;

The `DenyKeys` field is an array of Key names. When a Client sends a message to the Resource, if any of these Keys are included in the message, the Core acts as if this Resource does not exist. If the test is done as part of a lookup request, the Core will not check the attributes for a match. It is an error if any of the names in this field are not bound to a Key.

ESName[] AllowKeys;

The `AllowKeys` field is an array of Key names. When a Client sends a message to the Resource, if none of the Keys in this field are included in the message, the Core acts as if this Resource does not exist. If the test is done as part of a lookup request, the Core will not check the attributes for a match. If this field is empty, the test is bypassed. It is an error if any of the names in this field are not bound to a Key.

ESMap metadataLocks;

The first element in each pair of `ESMap` is an `ESName` that is the name of a Key. The second element is an array of strings: `String[]`. These strings are the Permissions that will be delivered to the Meta-Resource when access to the metadata for this Resource is requested by a Client. It is an error if `ESName` is not bound to a Key or if any of the Permission strings are not among those recognized by the Meta-Resource. These metadata Permissions are used in place of the Resource Permissions of the Meta-Resource.

ESMap resourceSpecificLocks;

The first element in each pair of `ESMap` is an `ESName` that is the name of a Key. The second element is an array of strings: `String[]`. These strings are the Permissions that will be delivered to the Resource Handler if the named Key is included in the message. The Resource Handler interprets these strings to determine what privileges the Client that sent the message has. For example, the string "read" might be interpreted by the Resource Handler that the Client has permission to read that file. It is an error if `ESName` is not bound to a Key.

ESMap publicRSD;

The first element in each pair of `ESMap` is a `string` used to tag the second element. The second element is of type `byte[]`. The `PublicRSD` field (public Resource-specific data) may be of interest to users of the Resource. Therefore, the Client registering the Resource may include information in this field. It is an error if either the tag or byte array is null or if the tags are not unique. This field can be used to carry code the Client can use for network object computing.

ESMap privateRSD;

The first element in each pair of `ESMap` is a `string` used to tag the second element. The second element is of type `byte[]`. The `privateRSD` field (private Resource-specific data) is used by the Resource Handler when a Client sends a message to this Resource. Therefore, the Client registering the Resource includes information in this field. This data is delivered to the Resource Handler. The intent is that only the Resource Handler have access to this data, but permission can be granted to any task using the e-speak security mechanisms. It is an error if either the tag or byte array is null or if the tags are not unique. This field is most often used to carry the Resource Handler's designation for the Resource.

ESName owner;

The `owner` field is the `ESName` of the active Protection Domain of the Client that registered the Resource. This field can be changed to another Protection Domain by any Client that unlocks the proper permission. It is an error if the `ESName` is not bound to a Protection Domain.

ESName ResourceHandler;

Messages sent to this Resource will be delivered to this Inbox. This field is `NULL` for Core-managed Resources. The Client that has connected to this Inbox will receive messages for this Resource. (The format of these messages is defined in Chapter 6, "Application Binary Interface.") This field may be `NULL` only if the Resource being registered is Core managed. It is an error if the `ESName` specified by the Client is not bound to an Inbox.

int eventControl;

If `eventControl` is non-zero, then whenever the Meta-Resource changes the Resource metadata (the Resource Description or the Resource Specification), it will publish an Event to the Core's Event distributor.

ESUID

```
public class ESUID
{
   private static final short COREID_MIN_BYTE_SIZE= 20;
   private static final short RESOURCEID_MIN_BYTE_SIZE= 12;
   private static final short ID_MAX_BYTE_SIZE= 64;
   byte[] coreId;
   byte[] resourceId;
   boolean local;
}
```

An ESUID contains a Core identity component and Resource identity component as well as an indication if the associated Resource is local or remote (imported). The Core identity is unique (to a high probability) while the Resource identity is unique within a given Core. Both identities have a minimum and maximum length as specified above.

ResourceDescription

`ResourceDescription` contains an array of Vocabularies and the attributes associated with each. Clients can specify a search request and ask the Lookup Service to find Resources with attributes that match the lookup request. An attribute specification includes a Vocabulary in which to interpret the attributes that describe the Resource.

`ResourceDescription` is an array of `AttributeSet` as shown below.

```
public class ResourceDescription
{
   AttributeSet[] attribSets;
}
```

`AttributeSet` consists of the `ESName` of a Vocabulary Resource and an `ESMap` of name-attribute pairs. The first element of an `ESMap` pair is a string, the second element is an `Attribute`. The string is the name of the `Attribute`. It is an error if `ESName` is not bound to a Vocabulary or if `Attributes` or their values are not valid in the named Vocabulary.

```
public class AttributeSet
{
   ESName attrVocab;
   ESMap attributes;
}

public class Attribute
{
   String name;
   Value value;
   ESList valueSet;
   Boolean essential;
}
```

The `name` field is the name associated with `Attribute`. The `Value` field contains any primitive type defined in the e-speak serialization format definition of `ValueAlt`.

The `valueSet` field is a set of `Value` types. This field is used if the attribute is multivalue; otherwise, the `value` field is used.

If `essential` is true, then this attribute must be included in any search request to discover a Resource with this attribute in its Resource Description.

Chapter 4 Core-Managed Resources

Clients interact with the e-speak Core by sending messages to Core-managed Resources. For example, the Contract Resource is used to register new Resource metadata. This section specifies the methods of each Core-managed Resource. It also describes the internal state that is passed if the Core-managed Resource is exported by value to another Logical Machine.

Conventions

In the following descriptions, before each method is a comment of the following form:

```
//Permissions: "String1", "String2", ..., "Stringn"
```

These comments should be interpreted as Permission strings, at least one of which must be delivered to the Resource Handler for it to invoke the method. If the Client does not include a Key that causes one of these Permission strings to be delivered, it does not have the privilege to execute the named method. A comment field of the following form should be interpreted as no Permission strings being required (no Keys are needed to execute the method):

```
//Permissions: NULL
```

All the methods described in this Chapter throw `ESInvocationException` (see Chapter 8, "Exceptions"), the base class for exceptions thrown by the Core to the Client during message processing. The Core throws the specific exception, allowing the programmer to deal with individual exceptions where appropriate and throw others up the call chain. Some methods also throw `ESServiceException`. The same rules apply. Programmers can catch or declare the parent class or deal with the specific exceptions thrown. Any of these method can throw any of the `ESRuntimeExceptions`, which need not be declared by the programmer.

For example, a programmer not wishing to deal with naming problems need only include `throws ESInvocationException` in the method declaration. That same programmer can catch a specific exception, say `QuotaExhaustedException`, and still deal with all other exceptions with this same `throws` declaration.

Each class definition starts with a list of static declarations. Each represents the code in the payload of the request that tells the Core which method to invoke.

Key

E-speak Keys have nothing to do with encryption. They are unauthenticated capability Resources: having a name for a Key Resource gives a Client the ability to use the Key to access Resources.

The import and export of Keys is explained in Chapter 7, “Inter-Core Communication”; Keys are not exported by value or by reference.

A Key Resource supports a single method: `cloneKey`. This creates a clone of the Key and returns a name for that clone. The clone confers exactly the same privileges as the original Key: It will match all the `Allow`, `Deny`, and `Permission` fields that the original matched. However, the clone and the original are separate Resources: They can be passed to Clients separately, and destroying one has no effect on the other.

Having a name bound to a Key is all a Client needs to use that Key, gaining the privileges associated with it. Hence, guarding access to name bindings to Keys is critical. Although it is possible to use care when giving out the bindings, there is nothing to prevent any Client from looking up a Key based on its attributes. **It is highly**

recommended that Keys be registered with an empty Resource Description. If attributes must be provided, one of them should be an essential attribute that can be thought of as a password:

```
public class Key
{
   public static final int CLONEKEY = 20001;

   //Permissions: "A", "C"
   public ESName cloneKey()
      throws ESInvocationException;
}
```

Key Ring

The `KeyRing` class is defined below:

```
public class KeyRing
{
   public static final int ADDKEY = 3001;
   public static final int REMOVEKEY = 3002;
   public static final int CONTAINS = 3003;

   ESName[] Keys; //State for pass-by-value

   //Permissions:"A", "C"
   public void addKey(ESName key)
      throws ESInvocationException;

//Permissions: "A", "C"
   public void removeKey(ESName key)
      throws ESInvocationException;
//Permissions: NULL
   public boolean contains(ESName key)
      throws ESInvocationException;
}
```

The method `addKey` adds a Key to the Key Ring. It will succeed only if `ESName` refers to a Key.

The method `removeKey` removes the given Key from the Key Ring Resource if it exists on the Key Ring.

The method `contains` checks for the existence of the given Key on the Key Ring Resource. The method returns false if the Key is not on the Key Ring, true otherwise.

Mailbox

E-speak has both Outboxes and Inboxes, but only Inboxes are exposed to Clients as Core-managed Resources. An Inbox is where a Client gets messages from the Core. A Client can have more than one Inbox, but each Inbox must be explicitly connected by the Client before it can be used to receive messages.

An Inbox cannot be exported.

The `Inbox` class is defined below:

```
class Mailbox
{
   public static final int ISCONNECTED = 4001;
   public static final int CONNECT = 4002;
   public static final int DISCONNECT = 4003;
   public static final int RECONNECT = 4004;

   //Permissions: NULL
   public boolean isConnected()
      throws ESInvocationException;
   //Permissions: "A", "C"
   public void connect(int slot)
      throws ESInvocationException;

   //Permissions: "A", "C"
   public void disconnect()
      throws ESInvocationException;

   //Permissions: "A", "C"
   public void reconnect(int slot)
      throws ESInvocationException;
}
```

An Inbox is a Core-managed Resource that provides a unidirectional communication channel from the e-speak Core to a Client. When a Client registers a Resource with the e-speak Core, it must assign an Inbox Resource as the “handler” for the Resource. Any service requests directed to the Resource are delivered to the Client on the I/O channel associated with the Inbox that was named the Resource Handler.

An Inbox can be in one of the two states: connected or disconnected. Upon creation, the Inbox starts in the connected state. The creator of the Inbox becomes the owner of the Inbox, and the Inbox is set up to use the I/O channel information passed with the request to create the Inbox. The Inbox remains in the connected state until the Client requests an explicit disconnect, or until the I/O channel associated with the Inbox is closed, at which time it is put in the disconnected state. If a Client sends a message to a Resource whose handler is an Inbox in the disconnected state, an exception is thrown by the e-speak Core.

One may argue that Inboxes are unnecessary and that the e-speak Core could store the I/O channel information in the Resource Handler field directly. There are two main reasons for having the Inbox store the I/O channel information and not the Resource—one has to do with Client restart, and the other with delegation. These are explained in the following subsections

Inbox and Client Restart

In the e-speak environment, a Client can recover from some types of failures, one of which is the failure of a Client process. In case a Client process dies and restarts, it can reconnect to the Core, discover and activate its previous Protection Domain, and discover and connect to the Inboxes owned by it. That way it can continue to serve the Resources that were registered by it during its previous incarnation.

Connecting to an Inbox involves updating the I/O channel information maintained by the Inbox. Keeping the I/O channel information in the Inbox helps simplify the Client's job at restart, because it has to discover and connect to only a few of them. If, instead, the I/O channel information is stored in all the Resources registered by the Client, it will somehow need to be updated all over the place upon reconnection by the Client.

Inbox and Delegation of Resource Handling

Under certain circumstances, a Client may want to delegate the handling of one or more Resources served by it to another Client. Inboxes make the delegation easy. Let's say Client A has registered 100 Resources, and named Inbox IB as its handler. After a while, Client A wants Client B to take over the handling of all these Resources. This can be achieved as follows:

1. Client A passes the name of the Inbox IB to the other Client, along with a Key/capability to perform a reconnect operation on the Inbox.
2. Client B requests the e-speak Core to reconnect it to the Inbox IB. The Core replaces Client A's I/O channel information with the information passed by Client B with a request to reconnect.
3. Any further service requests directed to any of the 100 Resources are diverted to the I/O channel specified by Client B. The process of reconnection is performed atomically; though logically the reconnect operation involves a disconnect operation on behalf of Client A and connect operation by Client B, no one really sees the transient disconnected state.

Name Frame

A Name Frame manages the bindings of ESNames to Resources. A Client's default Name Frame is part of its Protection Domain. This section first describes the structure of an ESName and a binding and then describes Name Frames and data structures used by Name Frames.

ESNames

The only way a Client can refer to a Resource when communicating with the Core is to specify an `ESName` for the Resource. An `ESName` consists of a string and a Name Frame for resolving that string. If the Name Frame is not specified, the string is resolved in the Client's root Name Frame. Hence, an `ESName` is defined as follows:

```
class ESName{
     ESName frame;
     String name;
}
```

The e-speak serialization format defines the serialization of `ESName` as an array of strings; the last element of this is the name of the Resource. All previous elements are the names of Name Frames.

Bindings

In e-speak, a name is bound to a *Mapping Object*, which consists of an array of accessors. An accessor can be one of two types, as represented in Table 4.

Table 4 Mapping Object accessor types and descriptions

Accessor Type	Descriptions
Search request	A set of attributes, their corresponding values, and a Vocabulary to use in interpreting them
Explicit binding	Binding to a single instance of a Resource

Thus, a name can be bound to:

- Zero or more Resources
- Zero or more Search Recipes
- Some combination of explicit bindings and search request bindings

The term *simple binding* is applied to a name bound to a Mapping Object that has a single explicit binding. The term *complex binding* is used otherwise.

Search Predicates, Search Recipes, and Name Search Policies

When a Client wants to find a Resource in e-speak, its query is translated to a Search Recipe. A Search Recipe specifies three search criteria and a view on the set of Resources registered in the Core. Each search criterion is expressed by a Search Predicate. The first criterion is used to reduce the set of Resources to a subset matching the Client's requirements. The second criterion expresses the Client's preferences for Resources in this subset. It is used to reduce the subset to return a singleton. Finally, the third criterion is used for arbitration when the subset cannot be reduced to one element.

A Client may use the same name for Resources of different types (file name, user name, and machine name, all called `nancy`, for example). Because the Core doesn't know the intent of the Client when doing the name resolution, it might match an ESName to the user `nancy` when the Client is trying to find the file `nancy`. Therefore, the Client should provide the Core with additional information to define the query. This information is defined in a Name Search Policy.

The class `Search Predicate` is described below:

```
class SearchPredicate
{
  AttributePredicate[] attrPredicates;
}
class AttributePredicate
{
     ESName Vocabulary;
     byte[] predicate;
}
```

`SearchPredicate` is an array of `AttributePredicates`. `AttributePredicate` consists of the name of a Vocabulary Resource and a predicate that is a constraint expression contained in a byte array. The constraint expression must be valid in the given Vocabulary.

Class `SearchRecipe` is defined below:

```
class SearchRecipe
{
  SearchPredicate constraint;
  SearchPredicate preference;
  SearchPredicate arbitrationPolicy;
  ESName repositoryView;
}
```

The `constraint` field specifies the first criterion.

The `preference` field specifies the second criterion. If the evaluation fails, the resulting set as computed previously is simply returned. Otherwise, a new set is returned.

If the result of the evaluation of `preference` has more than one Resource, and if a Client needs to restrict the set of Resources returned, it can specify an arbitration policy using a list of constraints defined in an Arbitration Vocabulary. Complex policies require the use of external arbitrators, and the tasks are responsible for implementing the requester's Arbitration Policy. These tasks may perform complex actions outside of the Core.

`ArbitrationPolicy` specifies the third criterion and defines what action is to be taken if there are multiple matches for a particular lookup.

The Repository View is a Core-managed Resource that constrains which set of Resources will be available to `SearchRecipe`. This will be a subset of all the Resources registered with the Core.

Class `NameSearchPolicy` is defined below:

```
public class NameSearchPolicy
{
   public static final int NSP_ANY = 0;
   public static final int NSP_SIMPLE = 1;
   public static final int NSP_EXPLICIT = 2;
   public static final int NSP_PARTIAL = 3;
   ESName contract;
   int bindingType;
   boolean matchSense;
}
```

`NSP_ANY` means match any binding types. `NSP_SIMPLE` means match simple binding types. `NSP_EXPLICIT` means match explicit binding types. `NSP_PARTIAL` means match partial binding types (this is not implemented in the current release, and will cause undefined behavior if used).

If `matchSense` is false, the meaning of the Name Search Policy is negated, so `listBindings` will return the names of bindings that do not satisfy the Name Search Policy.

Name Frame Methods

Some NameFrame methods throw `ESServiceException`. Chapter 8, “Exceptions” lists the exception hierarchy for NameFrame methods.

The `NameFrame` class is defined below:

```
public class NameFrame
{
   public static final int LOOKUP = 5001;
   public static final int ISBOUND = 5002;
   public static final int BIND = 5003;
   public static final int REBIND = 5004;
   public static final int UNBIND = 5005;
   public static final int RENAME = 5006;
   public static final int COPY = 5007;
   public static final int ADD = 5008;
   public static final int SUBTRACT = 5009;
   public static final int LISTNAMES = 5010;
   public static final int LISTBINDINGS = 5011;

   ESMap bindings;

   //Permissions: "A", "U"
   public void lookup(String baseName,
         SearchPredicate arbPolicy,
         ESName targetFrameHandle,
         String toBaseName)
      throws ESInvocationException, ESServiceException;

//Permissions: "A", "L"
public boolean isBound(String baseName)
   throws ESInvocationException;
```

```
//Permissions: "A", "+"
public void bind(String baseName, SearchRecipe recipe)
   throws ESInvocationException, ESServiceException;

//Permissions: "A", "M"
public void rebind(String baseName, SearchRecipe recipe)
   throws ESInvocationException;

//Permissions: "A", "-"
public void unbind(String name)
   throws ESInvocationException;

//Permission: "A", "+"
public ESName rename(String oldName,String newName)
   throws ESInvocationException, ESServiceException;

//Permissions: "A", "C"
public ESName copy(String toName, ESName from)
   throws ESInvocationException, ESServiceException;

//Permissions: "A", "M"
public ESName add(String name, ESName from)
   throws ESInvocationException;

//Permissions: "A", "M"
public ESName subtract(String name, ESName from)
   throws ESInvocationException;

//Permissions: "A", "L"
public String[] listNames(NameSearchPolicy nsp)
   throws ESInvocationException;
```

```
//Permissions: "A", "L"
public String[] listBindings(String aBaseName,
        NameSearchPolicy nsp,
        ESName targetFrame
)
    throws ESInvocationException;
}
```

A Name Frame can be exported by value or by reference. In the case of export by value, the Name Frame state is the bindings ESMap. The serialization for ESMap is defined by the e-speak serialization format. ESMap is an ESArray in which the convention is that consecutive elements are treated as pairs. In the case of bindings, the first element of a pair is the string component of ESName; the second is a MappingObject to which ESName is bound. The serialization format for a Mapping Object is also specified by the e-speak serialization format. The process by which Mapping Objects are serialized is described in the definition of the e-speak Serialization Protocol (ESIP).

All methods that create a new entry in a Name Frame return a Name Collision Exception if the name already appears in the target Name Frame. An explicit rebind or unbind is required before the name can be reused.

The lookup method is used to convert search requests to Resources' bindings. The name within this NameFrame of the binding to a SearchRecipe to be looked up is baseName. The policy used for arbitration is arbPolicy, in case a lookup results in a binding to multiple Resources. The name of the target NameFrame where the resultant name binding will be made is targetFrame. The name to bind in the target frame with the the results of the lookup is toBaseName. A new ESName is returned containing all the bindings obtained as a result of resolution.

The isBound method checks to see if the specified name (baseName) is bound in this Name Frame. It returns true if the name is bound.

The method bind binds SearchRecipe to a specified name (baseName) in this Name Frame.

The method rebind changes the binding of the specified name (baseName) in this Name Frame to the new SearchRecipe.

The method unbind removes the binding from NameFrame.

The method `rename` renames the binding associated with `oldname` to `newname.`

The method `copy` copies the binding of `from` to `toName`.

The method `add` adds the binding of `from` to the binding of `name` to give a new binding for `name`. It returns the `ESName` of the new binding of `name`.

The method `subtract` subtracts the bindings of `from` from the bindings associated with `name` to give a new binding for `name`. It returns an ESName for the new binding of `name`.

The method `listNames` returns an array of strings corresponding to all bindings that match `NameSearchPolicy nsp`. The Name Search Policy allows the Client to specify the type of binding and/or Contract in which the Resource is registered.

The method `listBindings` lists all the bindings of the argument `aBaseName` that match `NameSearchPolicy nsp`. These bindings are placed in the `NameFrame` named by `targetFrame`. The return value is an array of `String`, each element being the name of a new binding in `targetFrame`.

Table 5 lists the meanings of the Name Frame Resource Permission strings.

Table 5 Name Frame Resource permissions

String	Interpretation
A	All—All Resource Permissions
+	Add—Add bindings
-	Remove—Remove bindings
M	Modify—Modify bindings
C	Copy—Copy bindings from this Name Frame
D	Duplicate—Make a copy of the Name Frame
L	List—Find names
U	Use—Use name bindings

Importer Exporter

The Importer Exporter Resource is used by the Remote Resource Handler to construct the Export Form of Resources that it is exporting and to register Resource Descriptions and Resource Specifications of any Resources it is importing. Clients other than Remote Resource Handlers are unlikely to use this Core-managed Resource.

The Importer Exporter cannot be exported.

The Importer Exporter interface is described below:

```
public interface ImporterExporterInterface
{
   public static final int IMPORTRESOURCE = 11001;
   public static final int IMPORTMESSAGE  = 11002;
   public static final int EXPORTRESOURCE = 11003;
   public static final int REEXPORTRESOURCE  = 11004;

//Permissions: "A", "IR"
public ImportContext importResource(
        ImportContext importContext)
   throws ESInvocationException;

//Permissions: "A", "IM"
public void importMessage(
        ImportContext importContext,
        byte[] targetMsg)
   throws ESInvocationException;;

//Permissions: "A", "ER"
public ExportContext exportResource(
        ExportContext exportContext)
   throws ESInvocationException;
//Permissions: "A", "RR"
public ExportContext reExportResource(
        ExportContext exportContext)
   throws ESInvocationException;
}
```

The `ImporterExporter` Core-managed Resource is used by the Remote Resource Handler to create the Export Form for Resources being exported and to register Resource Descriptions and Resource Specifications of Resources being imported with the local Core.

It uses `ImportContext` and `ExportContext` types as parameters. These are defined below.

The `importResource` method is used to import the Resources contained in the `importContext` argument. These Resources will have their `ResourceDescriptions` and `ResourceSpecifications` registered in the local Core. The return value should be ignored.

The `importMessage` method imports the Resources contained in `importContext` and then forwards the `Message` contained in `targetMessage`. Because `Message` can contain many Outbox Message Atoms, this can result in messages being delivered to several Resource Handlers.

The `ExportResource` method is used to construct the Export Form for the Resources contained in `exportContext`. The returned `exportContext` contains these `ExportForms` and data required to construct the ESIP message as defined in Chapter 7, "Inter-Core Communication." The standard export policy is to export each Resource to a remote Logical Machine only once. Hence, if the Importer Exporter is asked to export a Resource more than once by the same Remote Resource Handler, the Importer Exporter will not create the Export Form for it.

The method `reExportResource` is used to force `ImporterExporter` to build `ExportForms` for the Resources in `exportContext`. Normally, `ImporterExporter` remembers if it has already built the Export Form of a Resource for a given Remote Resource Handler and will not build the Export Form for that Resource if it appears in subsequent calls of `exportResource` by the same Remote Resource Handler.

exportContext Class

The class `exportContext` is defined below:

```
class exportContext
{
   public static final byte BYVALUE= 1;
   public static final byte BYREFERENCE= 2;
   public static final byte BYNAME= 3;
   public static final byte RETURN= 4;
   private static final byte EXPORTFE= 32;

//Note private FORCE is commented out

   byte typeAndFlags;

public static final byte EXPORT_ONCE_ONLY = 1;
public static final byte EXPORT_REPEAT = 2;
public static final byte EXPORT_TOPLEVEL_ONLY = 4;
public static final byte EXPORT_RECURRSIVE = 8;

   byte exportPolicy;
   ESArray resources__;
   ESArray bytesArray__;
   ESArray tableArray__;
   ESArray mappings;
   ESArray mappingBytes;
   ESArray mappingTables;
}
```

The `resources__`, `bytesArray__`, and `tableArray__` are those used in `ESIPExportMessage`. These fields are described in "ESIP" on page 117. The `ImporterExporter` uses the contents of `resources__` to construct `bytesArray__` and `tableArray__` as described in "ESIP" on page 117. The `bytesArray__` field will contain the Export Form for the Resources in `resources__`.

The value of `typeAndFlags` controls what data is placed in the remaining fields of `exportContext` by the Importer Exporter. It can take one of the following values:

- `BYREFERENCE` : The Importer Exporter will construct the `resources__`, `bytesArray__`, and `tableArray` fields as described in "ESIP" on page 117
- `BYVALUE` : In the current implementation, this has the same effect as `BYREFERENCE` for ordinary Resources; in the future, this will be used to support export by value. Export by value is supported for some Core-managed Resources (see "Export by Value" on page 124).
- `BYNAME`: The Importer Exporter will not construct the `bytesArray` and `tableArray` fields. This is intended to support an export by name model and is currently not used.
- `RETURN`: This indicates that this export context is used as a return value (from `importerExporter` to the Remote Resource Handler).
- `EXPORTFE`: The Importer Exporter will interpret the names in `resources__` as `ExportFE` (export name) and will construct `ExportForm` for all Resources in `resources__`. This is used with the `importResources` method of the Remote Resource Proxy when the Remote Resource Proxy needs to send an `ESIPImportMessage`.

`BYVALUE`, `BYREFERENCE`, `BYNAME`, and `RETURN` are mutually exclusive. The Remote Resource Handler uses `BYVALUE` as a default. If any Client calling the Importer Exporter fails to set one of `BYVALUE`, `BYREFERENCE`, `BYNAME`, or `RETURN,` the Importer Exporter will throw an `InvalidValueException`.

The value of `exportPolicy` controls whether the Importer Exporter will export the Resource. It can take any one of the following values:

- `EXPORT_ONCE_ONLY`: Don't build `ExportForm` for Resources that have already been exported (the default behavior).
- `EXPORT_REPEAT`: Build `ExportForm` for all Resources even if they have already been exported.

- EXPORT_TOPLEVEL_ONLY: Don't build ExportForm for contained Resources.
- EXPORT_RECURSIVE: Build ExportForm for contained Resources (the default behavior).

EXPORT_ONCE_ONLY and EXPORT_REPEAT are mutually exclusive. EXPORT_TOPLEVEL_ONLY and EXPORT_RECURSIVE are mutually exclusive. If an incorrect combination of values is set, the Importer Exporter will throw an InvalidValueException.

The fields mappings, mappingBytes, and mappingTables are currently unused by Clients. They are used internally by the Importer Exporter.

importContext Class

The importContext class is defined below:

```
class importContext
{
   ESName exportFrame;
   ESName importFrame;
   ESArray bytesArray__;
   ESArray tableArray__;
   boolean retry;
   boolean exportByName;
   boolean updateFlag;
   ESArray resourceTables;
   ESArray retryTable;
   ESName inbox;
}
```

The fields and use are very similar to that of exportContext.

The exportFrame field is the name of a Name Frame used to contain the names of Resources that have been exported.

The importFrame field is the name of a Name Frame used to contain the names of Resources that have already been imported.

The `resources__`, `tableArray__`, and `bytesArray__` fields are as defined in "ESIP" on page 117.

Export by name is not supported, so if `exportByName` is set to true, it will cause undefined behavior.

If `updateFlag` is true, `importerExporter` will register all the Resources in `bytesArray__`, even if they have been imported previously.

The following fields are not used: `retry`, `update`, and `retryTable`.

Protection Domain

A Protection Domain encapsulates the Client's view of the system. A Client can have names for more than one Protection Domain, but messages to the Core will be interpreted only in the currently active Protection Domain. A Client can change its active Protection Domain to any Protection Domain for which it has a name.

When the Core receives a new message from a Client, it uses the default Name Frame in the Client's active Protection Domain for name resolution and extracts the Keys on the Mandatory Key Ring.

This facility can be used to support a notion of logon. Initially, a Client will have a very limited Protection Domain. Once the Client has been authenticated (i.e., logged on), it gets a new, richer Protection Domain.

The only Resources visible to the Client are those reachable by names rooted in the default Name Frame. The Keys on the Mandatory Key Ring control name visibility and access rights. The metadata for a new Protection Domain, its Mandatory Key Ring, and the root Name Frame are identical to that of the corresponding Resources of the parent.

Each Protection Domain is associated with a quota. The goal of this is to track and manage use of space in the Repository. To support this, each Protection Domain has three fields associated with it: used, soft limit, and hard limit. A Protection Domain is guaranteed to be able to allocate Resources up to its soft limit. A Protection

Domain may be able to allocate Resources up to its hard limit, depending on the memory usage of the Core. The default hard limit is 10,000,000 bytes, and the default soft limit is 30,000 bytes.

A Protection Domain cannot be exported.

The `ProtectionDomain` class is defined below:

```
class ProtectionDomain
{
   public static final int SWITCHPD = 6001;
   public static final int GETQUOTAINFO = 6006;
   public static final int SETQUOTA = 6007;


   //Permissions: NULL
   public BootstrapReply switchPD()
      throws ESInvocationException;

   //Permissions: A", "G"
   public Object[] getQuotaInfo()
      throws ESInvocationException;

   //Permissions: A", "S"
   public Object[] setQuota(long softQuota, long hardQuota)
      throws ESInvocationException;
}
```

The method `switchPD` switches the Client’s active Protection Domain to this Protection Domain (i.e., the Protection Domain receiving the method invocation). It returns a `BootstrapReply` object.

The `BootstrapReply` class is defined below:

```
public class BootstrapReply
{
   ESName Inbox;
   ESName CallbackResource;
   ESName ExceptionInbox
   ESName switchedFromPD;
}
```

The `BootstrapReply` object contains names for the Inbox, Callback Resource, and Exception Inbox (or Exception Handler) associated with the Protection Domain (see Chapter 6, "Application Binary Interface" for an explanation of how these are used.) It also contains a name for the old Protection Domain.

The `Object[]` array returned by `getQuotaInfo` and `setQuota` contains at least three values. The first is `Long` containing the total number of bytes currently consumed in the Core by this Protection Domain. The second is `Long` containing the soft limit in bytes. The third is `Long` containing the hard limit in bytes for this Protection Domain.

The following defines the set of initial names in the default Name Frame of a new Protection Domain:

```
public class CoreNames
{
   public static final String CURRENT_PD = "CurrentPD";
   public static final String MANDATORY_KEY_RING =
   "MandatoryKeyRing";
   public static final String INBOX_FRAME = "InboxFrame";
   public static final String SWITCHPD_FRAME = "SwitchPDFrame";
   public static final String CORE_FRAME = "Core";
/* The following names appear in the name frame "core"*/

   public static final String META_RESOURCE = "MetaResource";
   public static final String RESOURCE_FACTORY = "ResourceFactory";
   public static final String RESOURCE_CONTRACT_CONTRACT =
   "ContractContract";
   public static final String NAME_FRAME_CONTRACT =
   "NameFrameContract";
   public static final String KEY_CONTRACT = "KeyContract";
   public static final String KEY_RING_CONTRACT = "KeyRingContract";
   public static final String REPVIEW_CONTRACT =
   "RepositoryViewContract";
   public static final String PD_CONTRACT =
   "ProtectionDomainContract";
   public static final String VOCABULARY_CONTRACT =
   "VocabularyContract";
   public static final String INBOX_CONTRACT = "InboxContract";
   public static final String METARESOURCE_CONTRACT =
   "RepositoryViewContract";
   public static final String DEFAULT_VOCABULARY =
   "DefaultVocabulary";
   public static final String DEFAULT_EXTCONTRACT =
   "ExternalResourceContract";
   public static final String IMPORTER_EXPORTER =
   "ImporterExporter";
   public static final String CORE_DISTRIBUTOR = "CoreDistributor";
   public static final String EXPORT_EVERYWHERE_KEY =
   "ExportEverywhereKey";
   public static final String EXPORT_NOWHERE_KEY =
   "ExportNowhereKey";
   public static final String SYSTEM_MONITOR = "SystemMonitor";
}
```

"`InboxFrame`" is the special Inbox frame in which all the names of the Resources in the inbound message header get mapped.

"`Core`" is the name of the immutable frame that is shared by all the Clients where all the names of standard Core-defined and -managed Resources get put.

"`/Core/ResourceFactory`" creates all the Core-managed Resources (see"Resource Factory" on page 59). It contacts the MetaResource Factory automatically to create the corresponding MetaResource (see "MetaResource" on page 60).

Many of the Resources in "`Core`" are the names of Contract Resources for Core-managed Resources (see "Resource Contract" on page 58). Also included are the names of a number of other Resources such as the base Vocabulary (see "Vocabulary" on page 69), the Importer Exporter, the Core Distributor (for Core Events) (see Chapter 9, "Events"), and the System Monitor (see "System Monitor" on page 68).

Repository View

A Repository View contains references to a set of Resources.

When a Client does a lookup in a Repository View, the Core will attempt to match only those Resources included in the view. If no match is found, no accessor is added to the Mapping Object.

A Repository View can be exported by reference or by value.

The `RepositoryView` class is defined below:

```
class RepositoryView
{
   ESName[] Resources;

   public static final int ADD = 7001;
   public static final int REMOVE = 7002;
   public static final int CONTAINS = 7003;
   public static final int CLEAR = 7004;
   public static final int ADD_ELOOKUP = 7005;
   public static final int REMOVE_ELOOKUP = 7006;
//Permissions: "A", "+"
public void add (ESName res)
throws ESInvocationException;

//Permissions: "A", "-"
public void remove (ESName res)
   throws ESInvocationException;

//Permissions: NULL
public boolean contains (ESName res)
   throws ESInvocationException;

   //Permissions: "A", "-"
   public void clear ();
throws ESInvocationException;

   //Permissions: "A", "L"
   public void addExternalLookupHandler(ESName res);
throws ESInvocationException;

   //Permissions: "A", "L"
   public void removeExternalLookupHandler()
throws ESInvocationException;
}
```

An `externalLookupHandler` is not used in this release. Any attempt to use `addExternalLookupHandler` or `removeExternalLookupHandler` will cause undefined behavior.

Clients can `add` Resources to and `remove` Resources from a Repository View. The method `clear` removes all Resources from the Repository View. The method `contains` returns true if the Resource, `res`, is contained in the Repository View.

Resource Contract

A Resource Contract is an agreement between a Client of a Resource and the Resource Handler. This agreement includes the format of the `payload` in the `OutboxMessageAtom` and `InboxMessageAtoms` of `message`. The agreement also includes the secondary Resources required by the Resource, the Permissions that are needed, and so on. Hence, a Resource Contract denotes the Application Programming Interface (API) that is understood by the Resource Handler. The current release provides no means for enforcing this agreement; it is a convention.

Two Resource Contracts are available at system start-up in addition to those for Core-managed Resources. The default Resource Contract allows any Client to register a Resource. It is useful for Clients wishing to define Resources that don't specify a particular interface, such as Callback Resources. The second Resource Contract is for creating new Resource Contracts.

A Resource Contract contains a type string. This denotes the Resource type that is registered in this Resource Contract. A Resource Contract also contains a set of Vocabularies that can be used to describe and discover Resources of this type.

A Contract can be exported by value or by reference.

The `ResourceContract` class is defined below:

```
class ResourceContract
{
   ESName[] Vocabularies;
   string type;

   public static final int GETVOCABULARIES = 1002;

   //Permissions: "A", "G"
   public void getVocabularies(ESName targetFrame);
      throws ESInvocationException;
}
```

The method `getVocabularies` populates the Name Frame, `targetFrame`, with the names of the Vocabularies supported by the Resource Contract. The Name Frame `targetFrame` is cleared before the operation.

Resource Factory

A Client wishing to register a Resource with an e-speak Core uses the Resource Factory. This is also used for creating Core-managed Resources.

The `ResourceFactoryInterface` class is defined below:

```
public class ResourceFactoryInterface
{
   public static final int REGISTERRESOURCE = 10001

   //Permissions: NULL
   public void registerResource (
      ResourceDescription descr,
      ResourceSpecification spec,
      Boolean persistence,
      Object param,
      ESName targetFrame,
      String  toBaseName
)
   throws ESInvocationException;
}
```

The `registerResource` method takes `ResourceDescription` and `ResourceSpecification` as parameters. If persistence is true, the Core will preserve the metadata after the connection is closed and, in the case of Core-managed Resources only, also the state; otherwise metadata and state are not preserved after the connection is closed. The `Object` parameter is Resource-specific information used for creating Core-managed Resources. `Object` can be of any type supported in the e-speak Application Binary Interface (ABI). The `target-Frame` parameter is the `ESName` of a Name Frame in which the name for the new Resource will be put. The `toBaseName` parameter is the string component of the new Resource's ESName.

MetaResource

Every instance of e-speak provides a MetaResource that provides access to meta-data (Resource Descriptions and Resource Specifications). Once a Resource has been registered using a Resource Factory, the only way to access its metadata is through a message sent to the MetaResource.

MetaResources are not exported.

The Resource Manipulation Interface is defined below:

```
public interface ResourceManipulationInterface
{
public static final int UNREGISTER = 9001;
public static final int SETRESOURCEOWNER = 9003;
public static final int GETRESOURCEPROXY = 9004;
public static final int SETRESOURCEPROXY = 9005;
public static final int GETRESOURCECONTRACT = 9006;
public static final int EVALUATECONSTRAINT = 9007;
public static final int GETPUBLICRSD = 9008;
public static final int SETPUBLICRSD = 9009;
public static final int GETPRIVATERSD = 9010;
public static final int SETPRIVATERSD = 9011;
public static final int GETRESOURCEDESCRIPTION = 9012;
public static final int SETRESOURCEDESCRIPTION = 9013;
public static final int INSTALLMETADATALOCKS = 9014;
public static final int UNINSTALLMETADATALOCKS = 9015;
public static final int INSTALLRESSPECIFICLOCKS = 9016;
public static final int UNINSTALLRESSPECIFICLOCKS = 9017;
public static final int INSTALLALLOWLOCKS = 9018;
public static final int UNINSTALLALLOWLOCKS = 9019;
public static final int INSTALLDENYLOCKS = 9020;
public static final int UNINSTALLDENYLOCKS = 9021;
public static final int ISEXPORTEDBYVALUE = 9024;
public static final int SETEXPORTTYPE = 9025;
public static final int SETMETADATALOCKS = 9027;
public static final int SETRESOURCESPECIFICLOCKS = 9029;
public static final int SETALLOWLOCKS = 9031;
public static final int SETDENYLOCKS = 9033;
public static final int GETEVENTCONTROL = 9034;
public static final int SETEVENTCONTROL = 9035;
public static final int ISPERSISTENT = 9036;
public static final int ISTRANSIENT = 9037;
public static final int SETTRANSIENT = 9038;
public static final int SETPERSISTENT = 9039;
public static final int GETESUID = 9040;
public static final int GETQUOTA = 9041;

//Permissions: "A", "U"
public void unregister (ESName resource)
   throws ESInvocationException;

//Permissions: "A", "SO"
public void setResourceOwner (ESName resource)
   throws ESInvocationException;

//Permissions: "A"
public ESName getResourceProxy (ESName resource)
   throws ESInvocationException;
//Permissions: "A", "SH"
public void setResourceProxy (ESName resource,
      ESName resourceHandler)
   throws ESInvocationException;

//Permissions: "A", "GC"
public ESName getResourceContract (ESName resource)
   throws ESInvocationException;
```

```
//Permissions: "A", "GU"
public ESMap getPublicRSD(ESName resource)
   throws ESInvocationException;

//Permissions: "A", "SU"
public void setPublicRSD(ESName resource,ESMap rsds)
   throws ESInvocationException;

//Permissions: "A", "GI"
public ESMap getPrivateRSD(ESName resource)
   throws ESInvocationException;

//Permissions: "A", "SI"
public void setPrivateRSD(ESName resource, ESMap rsds)
   throws ESInvocationException;

//Permissions: "A", "GA"
public ResourceDescription getResourceDescription(ESName target)
   throws ESInvocationException;

//Permissions: "A", "SA"
public void setResourceDescription(ESName resource,
   ResourceDescription desc)
   throws ESInvocationException;

//Permissions: "A"
public void installMetadataLock (ESName resource,
      LockedPermissions lperms)
   throws ESInvocationException;

//Permissions: "A"
public void uninstallMetadataLock (ESName target, ESName key)
   throws ESInvocationException;;
```

```
// Permissions: "A"
public void installResourceSpecificLock(ESName resource,
     LockedPermissions lperms)
  throws ESInvocationException;

//Permissions: "A"
public void uninstallResourceSpecificLock(ESName resource,
  ESName key)
  throws ESInvocationException;;

// Permissions: "A", "SV"
public void installAllowLock (ESName resource,
  ESName key)
  throws ESInvocationException;

// Permissions: "A", "SV"
public void uninstallAllowLock (ESName resource, ESName key)
  throws ESInvocationException;

// Permissions: "A", "SV"
public void installDenyLock (ESName resource, ESName key)
  throws ESInvocationException;

// Permissions: "A", "SV"
public void uninstallDenyLock (ESName resource, ESName key)
  throws ESInvocationException;

//Permissions: "A"
public void setMetadataLocks(ESName resource, ESMap permKeyPairs)
  throws ESInvocationException;
```

```
//Permissions: "A"
public void setResourceSpecificLocks(ESName resource,
     ESMap PermKeyPairs)
  throws ESInvocationException;

//Permissions: "A", "SV"
public void setAllowLocks(EsName resource, ESSet Keys)
  throws ESInvocationException;

//Permissions: "A", "SV"
public void setDenyLocks(EsName resource, ESSet Keys)
  throws ESInvocationException;

//Permissions: "A", "GE"
public int getEventControl (ESName resource)
  throws ESInvocationException;

//Permissions: "A", "SE"
public void setEventControl (int setting)
  throws ESInvocationException;

//Permissions: NULL
public  boolean isPersistent (ESName target)
  throws ESInvocationException;

//Permissions: NULL
public  boolean isTransient (ESName target)
  throws ESInvocationException;

//Permissions: "A", "U"
public void setPersistent (ESName target)
  throws ESInvocationException;
```

```
//Permissions: "A", "U"
public void setTransient (ESName target)
   throws ESInvocationException;

//Permissions: NULL
public ESUID getESUID(ESName target)
   throws ESInvocationException;

//Permissions: NULL
public long getQuota(ESName target)
   throws ESInvocationException;
}
```

The convention for a Resource-specific data (RSD) array is that is consists of a sequence of pairs—the first element of each pair is a string used to tag the second element. (This is how it is used here— see e.g., `getPublicRSD`).

Most of the methods in a MetaResource are for setting or getting the fields of its Resource metadata. Some aspects of these methods warrant explanation and are discussed below.

The `unregister` method removes (unregisters) the Resource, `resource`, from the Repository. This removes `ResourceDescription` and `ResourceSpecification`; no more messages can be sent to the Resource after this operation.

The `setResourceOwner` method sets the `owner` of the Resource, `resource`, to the `ESName` of the calling Client's Protection Domain. There is no method for getting the `owner` of a Resource because of the potential security risk.

The `setResourceProxy` and `getResourceProxy` methods set and get the Resource Handler.

There is no method for setting the Resource Contract, because this cannot be changed once the Resource has been registered.

The LockedPermissions construct is used for manipulating Keys and Permissions:

```
class LockedPermissions
{
   ESSet permLabels;
   ESName Key;
}
```

The serialization format of ESSet is defined by the e-speak ABI serialization format (it is serialized as an array). Each element of ESSet is a string corresponding to the Permission that will be sent to the Resource Handler if the given Key is included in a message. For example, installMetadataLock installs a single Key and the set of Permissions that will be released by that Key if it is used to access the metadata by the MetaResource.

The methods for installing and uninstalling Locks should refer to installing and uninstalling Keys. A Lock is an internal data structure in the current implementation of e-speak. Clients never refer to Locks, they refer only to Keys. Clients give Keys as the parameters to these methods, not Locks.

ESMap in setResourceSpecificLocks and setMetdataLocks is an array in which the first and second element are a pair, the second and third element are a pair, and so on. Within each pair, the first element is the ESName of a Key and the second element is of type LockedPermissions.

The method getQuota() returns the total charge in bytes to the owner's quota due to that Resource.

Table 6 shows the interpretation of the Permission strings for a MetaResource.

Table 6 Metadata Permissions for all Resources

String	Interpretation
A	All—All metadata Permissions
C	Create—Create a Core-managed Resource
U	Unregister—Remove metadata from Repository
SH	Set handler—Change the Inbox that designates the Resource Handler
SO	Set owner—Change the Protection Domain that denotes the owner
GC	Get contract—Can be given a name bound to the Contract in which the Resource is registered
GA	Get attributes—Allowed to see all the attributes and be given a name bound to the Vocabulary for the attributes
SA	Set attributes—Change the attributes
SE	Set export—Change pass-by-value and export-by-name settings
GE	Get export—Query the pass-by-value and export-by-name settings
SP	Set persistent—Make Resource persistent
GP	Get persistent—Query persistence setting
SU	Set public RSD—Change public Resource-specific data
GU	Get public RSD—Allowed to see the public Resource-specific data
SI	Set private RSD—Change private Resource-specific data
GI	Get private RSD—Allowed to see the private Resource-specific data

Table 6 Metadata Permissions for all Resources *(Continued)*

String	Interpretation
MM	Merge metadata Permissions—Merge metadata Permissions from the metadata of one Resource with the metadata of another (Need “A” permission on target)
MR	Merge Resource Permissions—Merge Resource Permissions from the metadata of one Resource with the metadata of another (Need “A” permission on target)
SV	Set visibility—Change the Deny and Allow Locks

System Monitor

The System Monitor gives information about the running Core: the number of messages sent and received, the number of bytes sent and received.

The System Monitor cannot be exported.

`SystemMonitorInterface` is defined below:

```
public interface SystemMonitorInterface
{

   public static final int SM_ENABLECOUNTER = 15001;
   public static final int SM_DISABLECOUNTER = 15002;
   public static final int SM_READCOUNTER = 15003;

   public static final int SM_MESSAGE_RECV_COUNT = 1;
   public static final int SM_MESSAGE_SEND_COUNT = 2;
   public static final int SM_MESSAGE_SEND_BYTES = 3;
   public static final int SM_MESSAGE_RECV_BYTES = 4;
   public static final int SM_MAX_COUNTER_OPCODES = 32;
//Permissions: "W"
public void enable(int counter)
     throws ESInvocationException;

//Permissions: "W"
public void disable(int counter)
     throws ESInvocationException;

//Permissions: "R"
public long read(int counter)
     throws ESInvocationException;
}
```

Vocabulary

A Vocabulary is used to describe Resources and to specify lookup requests. Vocabularies are also used to define the state of Events.

The `Vocabulary` class is defined below:

```
public class Vocabulary
{
   String description;
   AttributePropertySet props;

public static final int GETDESCRIPTION = 8001;
public static final int GETPROPERTIES = 8002;
public static final int MUTATEPROPERTIES = 8003;
//Permissions: NULL
public String getDescription()
   throws ESInvocationException;
//Permissions: "A"
public AttributePropertySet getProperties()
   throws ESInvocationException;

//Permissions:"A", "C"
public void mutateProperties(AttributePropertySet props)
   throws ESInvocationException;
}
```

The method `getDescription` returns a human-readable string describing the Vocabulary.

The methods `getProperties` and `mutateProperties` are for getting and setting the `AttributePropertySet` of a Vocabulary. The definition of `AttributePropertySet` is given below:

```
public class AttributePropertySet
{
   ESMap AttributeProperty
}
public class AttributeProperty
{
   String attrName;
   ValueType valuetype;
   Value defaultValue;
   boolean multiValued;
   int rangeKind;
   double minRange;
   double maxRange;
   String description;

   public static final int NO_RANGE = 0;
   public static final int LEFT_RANGE = 1;
   public static final int FULL_RANGE = 2;
   public static final int RIGHT_RANGE = 3;
}
```

The first element of each pair in AttributePropertySet ESMap is the attrName of AttributeProperty; the second element is AttributeProperty.

In AttributeProperty, attrName is the name of the attribute.

If multiValued is true, defaultValue is assumed to be an ESSet of Values (see the e-speak serialization format for the definition of ESSet).

The fields rangeKind, minRange, and maxRange specify the range of defaultValue. NO_RANGE means that minRange and maxRange do not specify any restrictions. LEFT_RANGE means a value below minRange; RIGHT_RANGE means a value above maxRange; FULL_RANGE means a value between maxRange and minRange.

The description field is a human-readable description of the attribute property.

ValueType is defined below. The defaultValue field holds the value defined by valuetype. The list of possible types is given in the definition of ValueType. The serialization of Value is defined by the e-speak serialization format of the nonterminal ValueAlt.

The ValueType class is defined below:

```
public class ValueType
{
public static final String [] baseTypeNames =
        {STRING_TYPE,
        LONG_TYPE,
        DOUBLE_TYPE,
        BOOLEAN_TYPE,
        BIG_DECIMAL_TYPE,
        TIMESTAMP_TYPE,
        DATE_TYPE,
        TIME_TYPE,
        INTEGER_TYPE,
        FLOAT_TYPE,
        CHAR_TYPE
        BYTE_ARRAY_TYPE,
            BYTE_TYPE,
            SHORT_TYPE,
            NAMEDOBJECT_TYPE};

    string typeName;
    string description;
    string matcher;
}
```

The `typeName` field is a string name of the value type object; valid values are those in the `baseTypeNames` array.

The `description` field is the human-readable description of the value type, for example, `int`.

The `matcher` field is the string name of a matching function, for example, `isLessThan`.

Chapter 5 Vocabularies

This section specifies the construction and use of Vocabularies. It describes:

- Attributes as name-value pairs
- The use of common matching rules for standard data types
- Creating a new Vocabulary with supported value types
- Interoperability between different Vocabularies

Vocabulary Overview

A Resource Description is expressed using a Resource Description language called a *Vocabulary*. A Vocabulary is a Resource; to end the recursion, the Core bootstraps the description process by implementing a *Base Vocabulary*. This Base Vocabulary may be used to describe Resources in the absence of any other Vocabulary.

A Vocabulary is defined by `AttributePropertySet,` which is an array of `AttributeProperty.` Every e-speak system comes with an architected Base Vocabulary.

These rules are followed regarding Vocabularies:

- Attributes expressed in different Vocabularies cannot be matched.
- Attributes in one Vocabulary can be converted into attributes in another Vocabulary if a *Translator* Resource exists that is capable of the desired conversion.
- Vocabularies can be extended dynamically by adding new attributes.
- Any process can create a new Vocabulary dynamically.

- All attributes in a Resource Description must belong to the same Vocabulary. If a Resource has capabilities that can be described in multiple Vocabularies, it can use multiple Resource Description entries to represent it in the e-speak Core, as long as each entry uses only attributes belonging to one Vocabulary.

The e-speak Core will ship with one Basic Vocabulary preloaded. It is expected that the Basic Vocabulary will be always be in the Core and is accessible to all Clients. Clients are free to define their own Vocabularies. The creator of a new Vocabulary is responsible for the dissemination of information about the new Vocabulary to potential users.

Vocabulary Builder

E-speak will initialize a *Lookup Service* during start-up. This Lookup Service registers a *Vocabulary Builder* Resource. This builder is used by all authorized Clients to create new attributes and new Vocabularies. It is a relatively simple process to create new Vocabularies with new attributes using existing value types.

The Vocabulary Builder takes `AttributePropertySet` as the definition of the Vocabulary. The Attribute Property Set is an array of `Attribute Property` components. Each component has a number of fields, as shown in Table 7.

Table 7 Components of an attribute property

Type	Field	Meaning
String	name	Attribute name
String	description	Human-readable description
Value type	valueType	See Table 8 for encoding
Value	default	Default value
Boolean	multiValued	True if multiple values
Boolean	mandatory	Must be specified if True
Int	rangeType	0 no range 1 lower limit 2 both 3 upper limit
Double	minValue	Smallest allowed value
Double	maxValue	Largest allowed value

The e-speak Vocabulary Builder supports the value types shown in Table 8.

Table 8 Supported value types

Data type	Designator	Matching rules
Big decimal	"BigDecimal"	eq, ne, lt, le, gt, ge
Boolean	"Boolean"	eq, ne
Byte	"Byte"	eq, ne
Byte array	"ByteArray"	eq, ne
Char	"Char"	eq, ne
Date	"Date"	eq, ne, lt, le, gt, ge
Double	"Double"	eq, ne, lt, le, gt, ge
Float	"Float"	eq, ne, lt, le, gt, ge
Int	"Integer"	eq, ne, lt, le, gt, ge
Long	"Long"	eq, ne, lt, le, gt, ge
Object	"NamedObject"	
Short	"Short"	eq, ne, lt, le, gt, ge
String	"String"	eq, ne
Time	"Time"	eq, ne, lt, le, gt, ge
Time stamp	"Timestamp"	eq, ne, lt, le, gt, ge

All arithmetic and/or logical operations defined for each value type are supported. For example, a constraint can specify "a+b<c". Remember, equality testing on floating point numbers may give unexpected results.

A value type can be specified using a designator, such as:

```
ValueType intType = new ValueType(“Integer”);
```

Building a New Vocabulary

Any Client with `“create”` Permission in the Vocabulary Builder can create a new Vocabulary:

```
ESName createResource(
   ResourceDescription d,
   byte[] p);
```

The byte array is interpreted as an `AttributePropertySet` that includes a definition of the attribute properties used by the new Vocabulary. The Resource Description defines the part of the metadata used for discovery of this Vocabulary Resource.

The following example shows the specification of a Car Vocabulary that has only two attributes: `Model` and `Price`:

```
AttributeProperty p1 = new AttributeProperty(
   “Model”,new ValueType(“String”));
AttributeProperty p2 = new AttributeProperty(
   “Price”,new ValueType(“Double”));
```

and is added to a property set:

```
AttributePropertySet p = new
   AttributePropertySet();
p.add(p1);
p.add(p2);
```

Base Vocabulary

Each Vocabulary consists of a set of attribute properties; a string representing the name, something that carries the type of the value, and attribute properties. The Vocabulary also includes a set of matching rules. The Base Vocabulary is available at system start-up. It includes the attributes and value types shown in Table 9.

Table 9 Base Vocabulary description

Attribute name	**Value type**	**Comments**
Name	String	
Type	String	
Description	String	
KeyWords	String[]	Multivalued
Version	String	
Date	Date	"YYYY-MM-DD"
Time	Time	"HH:MM:SS"
TimeStamp	TimeStamp	"YYYY-MM-DD HH:MM:SS.FFFFFFFFF"
HashAlgorithm	String	
HashCode	BigDecimal	To authenticate contents

The hash algorithm is specified using well-known names, for example, `MD5`.

Translators (Informational)

The interoperation of different Vocabularies is supported through V*ocabulary Translators*. The translator can map attributes from one Vocabulary into another, but there is no direct linkage between a Translator Resource and any Vocabulary Resource. A translator service is not part of the e-speak architecture.

The translator implements:

```
ESName[][2] getVocabularyPairs();
```

which queries the translator about Vocabularies known to it. The translator returns an array listing all Vocabularies that it can translate in an ordered set. Each element in this array is a pair of Vocabulary names.

```
boolean isCompatible(Vocabulary vocabulary1,
   Vocabulary vocabulary2)
```

checks if the translator can translate from the first given Vocabulary into the second given Vocabulary. If the translator can perform the translation operation on the given pair of Vocabularies, it will return true. If the translator cannot perform the translation, or if it does not understand either of the Vocabularies, it will return false. The translation is done by:

```
SearchRecipe translate(SearchRecipe s,
   Vocabulary v2;
```

which returns a Search Recipe in the specified Vocabulary.

Chapter 6 Application Binary Interface

This section describes the e-speak Application Binary Interface, or e-speak ABI. This is the protocol that Clients use to send and receive messages from the e-speak Core. Most of the details of the messaging interface are hidden in the Client library. The discussion in this section is intended for those writing directly to the Core Application Programming Interfaces (APIs) or those writing the messaging component of such a library.

The section begins with an overview of how the Core processes a message. Next, the format of a message to the Core is defined, followed by the format of a message received from the Core. Finally, the e-speak serialization format is defined, which defines the on-the-wire format for message components.

Message Flow Through the Core

The only way for a Client to request access to a Resource from a Resource Handler is to send a message through the e-speak ABI to the Core. The only way for a Resource Handler to return a reply to a Client is to send a message through the e-speak ABI to the Core. Thus, the Core mediates all access between Clients and Resource Handlers; when a Client sends a message to a Resource, the Core meditation is transparent. Figure 4 illustrates the flow of information through the Core.

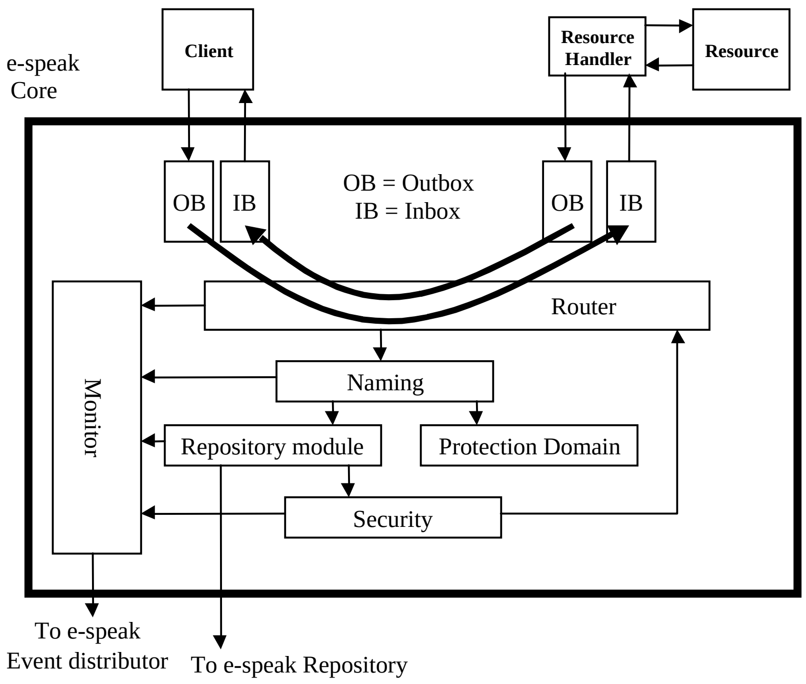

Figure 4 Information flow through the e-speak Core

The Core does not keep any information about replies to messages. As far as the Core is concerned, a reply is another message. If the Client needs a reply, it may wait or send another message; all messaging is asynchronous. Each asynchronous message has an identifier set by the sender. A reply can refer to this identifier so the Client knows which message the reply is for.

The Core will never keep any state about a message beyond the time needed to complete processing. Once the Core retrieves a message from a Client's Outbox, it guarantees to do one of three things:

1. Normally, the Core forwards the message to a Resource Handler. However, the Core cannot deliver the message if:
   - The Client's name for the target Resource is not valid
   - The Inbox specified in the Resource metadata is not connected to a Resource Handler
   - The Resource Handler's Inbox is full
2. When the Core cannot deliver the message, it will send an error message to the Client's designated *Exception Resource*.
3. If the Inbox associated with the Exception Resource can't take the error message for any reason, the Core will discard the message.

Note that the Exception Resource need not be connected to the Client sending the message; another Client can be designated to handle such error messages.

The Core processes messages from a given Client one at a time. It assures in-order processing and delivering of messages sent from a Client to the specified Resource's Resource Handler.

E-speak ABI Message Format

A message is an ordered sequence of Message Atoms, where each Message Atom is either an Inbox Message Atom or an Outbox Message Atom. All Message Atoms in a message must be the same type. A message having more than one Message Atom is called a ganged message, and can be used to submit a batch of requests. The following is a definition of the `Message` format:

```
public class Message
{
   int Length; //The number of message atoms
   short abiversion; //Currently 0x1010
   messageAtom[] atoms;
}
```

The serialization format for a message and its components is defined in "E-speak Serialization Format" on page 91. Once it has been serialized, each message is sent as an integer length, followed by an array of bytes (note that in the current implementation, the length will not be correct and is not used).

Sending a Message to the Core

To send a message to the Core, a Client serializes an instance of the `Message` class containing Outbox Message Atoms and sends it over its channel to the Core (e.g., a TCP connection). The format of an Outbox Message Atom is described below:

```
public class OutboxMessageAtom implements messageAtom
{
   short abiversion; //Currently 0x1010
   short secondaryAbiVersion;
   string msgID;
   string replyID;
   ESName primaryResource;
   ESName callbackResource;
   ESName exceptionHandlerResource;
   ESName[] keyRingResources;
   ESName[] secondaryResources;
   byte[] payload;
}
```

The `msgID` and `replyID` fields can be used by the Client and the Resource Handler to identify messages (e.g., matching a reply to a request). The Core does not use these fields.

The `payload` field of a Message Atom is a byte array that is interpreted by the Resource Handler. An empty array is acceptable. The payload contents are determined by the Application Programming Interface (API) that is part of the Contract registered with the Resource.

When the Core gets a message, it will add to `keyRingResources` the mandatory Key Ring in the Client's active Protection Domain. Next, its *Router* component extracts the names specified by the Client, including:

- The names of the Key Rings presented with this request.
- The name of the Exception Resource (typically, but not necessarily, connected to the Client) that is used to report problems.
- The name of the primary Resource that is the target of the message.

- The name of the Callback Resource, which is the Resource target for any reply messages (typically, but not necessarily connected to the Client). It may be set to null if, for example, no reply is required.
- The names for the secondary Resources, which are Resources that may be needed by the Resource Handler.

The `primaryResource` field is used to determine the destination of the message. The `secondaryResources` field provides the Resource Handler with names of other Resources it may need to do its job. The Router forwards these names to the *Naming* component of the Core.

The Core attempts to resolve these names through the Client's active Protection Domain by finding the Mapping Object associated with each name. A Client can have only one active Protection Domain at any time, but it can switch to a different Protection Domain any time it chooses. The Mapping Object for the primary Resource is used to identify the target of the message. The Mapping Object is used by the Core to refer to a Resource, a Search Recipe, or any combination.

The Repository Handle (or handles) from each Mapping Object is used by the Repository component of the Core to retrieve the metadata for that Resource. This metadata is forwarded to the *Security* component. The Security component applies two checks to validate the use of those Resources. The first check concerns filtering the Mapping Object. The Mapping Object is filtered to remove any references to Resources that fail the visibility test. Next, the Security component extracts the Permissions associated with each Resource in the filtered Mapping Object that has the same Resource Handler as the primary Resource. In the current implementation, these operations are only done for the primary Resource. Throughout this process, the *Monitor* component publishes Events related to this activity.

The metadata for the primary Resource will contain the Inbox for the Resource Handler, so having completed the security checks, the message comes back to the Router. If the Core cannot unambiguously identify the Resource Handler, it will send an error message to the Client's designated Exception Resource. The possible exceptions are described in Table 10.

Table 10 Exceptions for unresolved identification of a Resource Handler

Exception	Description
NameNotFoundException	The lookup procedure failed to find a Mapping Object.
UnresolvedBindingException	The only accessors in the Mapping Object are Search Recipes.
MultipleResolvedBindingException	The explicit bindings in the accessors refer to Resources with different Resource Handlers.
UndeliverableRequestException	The Resource Handler does not have the Resources needed to receive this message, or the Handler Inbox is not currently connected.

Receiving a Message from the Core

To forward a message to a Client, the Core serializes an instance of the `message` class containing Inbox Message Atoms. The definition of an Inbox Message Atom and its associated components is given below:

```
public class ResourceInfo
{
   ESName resource;
   byte[][] RSD
   string[][] permission
}
```

```
InboxMessageAtom
{
   short abiversion; //Currently 0x1010
   short secondaryAbiVersion;
   int slot;
   string msgID;
   string replyID;
   ResourceInfo primaryResource;
   ResourceInfo callbackResource;
   ResourceInfo exceptionHandlerResource;
   ResourceInfo[] secondaryResources;
   byte[] payload;
}
```

The e-speak serialization for `ResourceInfo` and `InboxMessageAtom` is defined in "E-speak Serialization Format" on page 91.

The `Slot` field is used to enable many Inboxes to share a single channel (TCP connection in the current implementation). The slot identifies which Inbox the message is from.

`ResourceInfo` contains the `ESName` of the Resource. If the referenced Resource has the same handler as the primary Resource, `ResourceInfo` also contains the private Resource-specific Data (RSD) of the Resource and any Permission strings that had matching Keys presented by the Client message. In other cases, the Permissions and RSD arrays in `ResourceInfo` will be empty. Because the Mapping Object for `ESName` may refer to several Resources, the `RSD` and `permission` fields are two-dimensional arrays. There is one outer element for each separate Resource referred to by the Mapping Object. In the present implementation, these fields are filled only for the primary Resource.

Note that Search Recipes in a Mapping Object will not have their Permissions or their RSD made available, because these bindings have not been completed, that is, they have not been explicitly bound to a Resource or Resources.

The `payload` contains the data defined by the API associated with the Contract of the Resource. If the message is being forwarded to a Resource Handler (a non-Core-managed Resource), the Core forwards the payload from the sending Client without examining its contents.

Messages from the Resource Handler to the Client

E-Speak implements a peer-to-peer communications model for messaging.

The Core does not distinguish between a message sent from a Client to a Resource Handler and a reply from the Resource Handler back to the Client. The Resource Handler sending a reply to a Callback Resource is treated as the Client, and the Client receiving the reply is treated as the Resource Handler for the Callback Resource.

Clients may have more than one Inbox. The only way for a Client to receive a message from any other Client is to register a Resource listing one of its Inboxes in the Resource Handler field of the metadata. Clients can manage different classes of messages by registering different Resources designating different Inboxes. Clients can also deal with different message classes by associating certain classes with Events.

Format of Payload for Core-Managed Resources

E-speak specifies the payload format for messages sent to and received from Core-managed Resources (`PayloadForCore`, `PayloadFromCore`). It does not specify the payload format for non-Core-managed Resources. The class `PayloadForCore` defines the payload format of messages sent to Core-managed Resources:

```
public class PayloadForCore
{
   int command;
   Ob[]arguments;
}
```

The command field is an integer specifying the method to be invoked. This is followed by an array of arguments. The type Ob is any type defined in the e-speak serialization format.

If any part of any of the arguments is ESName, the ESName instance is replaced with an instance of PayloadReference as defined below. ESName can be part of one of the arguments if it is a component of a class or an element of a container. For instance, the first argument might be an array of 10 ESNames; each ESName will be replaced by its own PayloadReference in the serialized payload. All such ESNames will be inserted into the secondaryResources array of OutboxMessageAtom. The value of resourceID in PayloadReference is the index of the ESName's entry in secondaryResources:

```
class PayloadReference
{
   public static final int SECONDARY_RESOURCE = 4;
   int resourceType;
   int resourceID;
}
```

The Resource Type will always be the value SECONDARY_RESOURCE.

The payloadFromCore class specifies the payload format of messages received from Core-managed Resources:

```
public class PayloadFromCore
{
      public static final int REPLY_PAYLOAD = 1;
      public static final int EXCEPTION_PAYLOAD = 2;
      public static final int EVENT_PAYLOAD = 3;
      int type;
   Ob[]arguments;
}
```

The type field indicates whether the message is a reply, an exception, or an Event. The elements of the arguments array of type Ob can be of any type defined in the e-speak serialization format. Any ESName will be replaced by an instance of PayloadReference and the ESName inserted into secondaryResources, as described above.

The on-the-wire format of `PayloadForCore` and `PayloadFromCore` is specified in the e-speak serialization format.

Initial Connection to the Core

The e-speak Core listens on a TCP port for Client connections (the default port is 12345). When it receives a connection request a TCP channel is created between the Client and the Core. The Core creates a default protection domain for the Client and sends message back to the Client (see PayloadFromCore) containing a `BootstrapReply` object.

E-speak Serialization Format

The basic types recognized are `byte`, `short`, `int`, `long`, `float`, `double`, and `string`. The four integral types `byte`, `short`, `int`, and `long` are 1, 2, 4, and 8 bytes long respectively, and are always sent most significant byte first. The `float` and `double` types are sent just as in Java. The string type is syntactically synonymous with byte[], but is intended to contain text rather than arbitrary binary data, and the text must be a valid UTF-8 encoded string as per RFC 2279.

We also recognize arrays of types. The type foo[] is sent as a length followed by that many instances of type foo. If the length is -1, then a NULL is returned. If the length is 0, an empty array is returned. Otherwise an array with that many elements is returned.

A map is sent in the same syntactic way as an array, but there is an implicit Key/value association between pairs of elements; all the evenly indexed elements (0, 2, etc.) are Keys, and all the odd indexed elements are values. Some maps may allow multiple occurrences of the same Key.

The `length` field is encoded in a single byte if the value of the length is -1..62 inclusive; the encoding is 129 more than the length. Thus, -1 is sent as the byte value 128, and a length of 3 is sent as the byte value 132. Lengths from 63..2^31-1 are sent as a 4 byte integer. Lengths below -1 or greater than 2^31-1 are illegal at the present time.

Most elements in the following Backus–Naur Form (BNF) are sent as `Ob`. An `Ob` consists of a signal byte that indicates the type of the object that follows, followed by the data for that object.

Signal bytes are (in this release) entirely single bytes. They are encoded by literal ASCII characters (e.g., A), literal ASCII characters but with the high byte set (e.g., *A), and small byte values (e.g., #3).

The following productions are the basic but nonterminal types.

The four container classes `ESMap`, `ESArray`, `ESSet`, and `ESList` will be removed; all containers should be sent as `Ob[]`.

In the following BNF, the meta-symbol => means "is sent as." The convention is as follows:

`type` => [*interpreted as field* :] (**type sent on the wire** | `type`)

```
String        => string
Integer       => int
Long          => long
Boolean       => byte
Null          =>
ByteArray     => byte[]
ObjectArray   => Ob[]
ESMap         => map
ESArray       => Ob[]
ESSet         => Ob[]
ESList        => Ob[]
```

These are the primary e-speak objects in communications with the Core:

```
ESName        => string[]
RSD           => byte[]

ResourceSpecification =>
   byValue:byte
   contract:Ob   // ESName
   denyLocks:Ob[]// ESName[]
   allowLocks:Ob[] // ESName[]
   metadataLocks:map // (ESName, String[])
   resourceSpecificLocks:map // (ESName, String[])
   publicRSD:map // (ESName, byte[])
   privateRSD:map // (ESName, byte[])
   owner:Ob      // ESName
   proxy:Ob      // ESName
   eventControl:int
   esuid:Ob      // byte[], byte[], boolean

ResourceDescription => desc:Ob[]// AttributeSet[]

AttributePropertySet => Ob[]// ESmap AttributeProperty

ValueType =>
   typeName:string
   description:string
   matcher:string
```

```
AttributeProperty =>
   name:string
   valueType:Ob  // ValueType
   default:Ob   // Value
   multiValued:byte
   rangeKind:int
   minRange:double
   maxRange:double
   description:string

Attribute =>
   name:string
   value:Ob     // Value
   essential:byte

AttributeSet =>
   vocab:Ob     //ESName
   data:Ob[]    //ESmap (string, Attribute)

Value =>
   val:ValueAlt
   nextVal:Ob   // Value (in lists of values)
```

Multivalued attributes are list of values, these are supported by `nextVal`.

```
SearchRecipe =>
   constraint:Ob  //SearchPredicate
   preference:Ob  //SearchPredicate
   arbitrationPolicy:Ob    //SearchPredicate
   repositoryView:Ob//ESName

SearchPredicate => Ob[]//AttributePredicate[]
```

```
LockedPermissions =>
    permLabels:Ob[]  //String[]
    key:Ob       //ESname

LiteralName => Ob[]//String[]

AttributePredicate =>
    attrVocab:Ob  //ESname
    predicate:byte[]

NameSearchPolicy =>
    contract:Ob   //ESname
    bindingType:int
    matchSense:byte

BootstrapReply =>
    inbox:Ob     //ESname
    callbackResource:Ob//ESname
    exceptionHandlerResource:Ob//ESname
    switchBackResource:Ob//ESname

ESException =>
    errno:int
    info:Ob[]    //String[]

ESRuntimeException =>
    errno:int
    info:Ob[]    //String[]

ESUID =>
    bytes1:Ob    //byte[]
    bytes2:Ob    //byte[]
    islocal:byte
```

These are the primary e-speak objects used for messaging.

```
Message =>
   length:int
   abiversion:short
   messageAtoms:Ob[]// InboxMessageAtom | OutboxMessageAtom

InboxMessageAtom =>
   abiVersion:short
   secondaryAbiVersion:short
   slot:int
   msgID:string
   replyID:string
   primaryResource:Ob  //ESname
   callbackResource:Ob //ESname
   exceptionHandlerResource:Ob//ESname
   secondaryResources:Ob[]  //ESname[]
   payload:showbytes

OutboxMessageAtom =>
   abiVersion:short
   secondaryAbiVersion:short
   msgID:string
   replyID:string
   primaryResource:Ob  //ESname
   callbackResource:Ob  //ESname
   exceptionHandlerResource:Ob  //ESname
   keyRingResources:Ob[]  //ESname
   secondaryResources:Ob[]  //ESname[]
   payload:showbytes

ResourceInfo =>
   fsn:Ob       //ESname
   privateRSD:Ob[][]  //byte[][]
   permissions:Ob[][]//string[][]
```

```
PayloadReference =>
   type:int
   id:int
```

These e-speak objects are used to build payloads to or from the Core.

```
PayloadFromCore =>
   type:int
   args:Ob[]  //Any type defined in the serialization format

PayloadForCore =>
   type:int
   args:Ob[]  //Any type defined in the serialization format
```

These e-speak objects are used for Events.

```
SubProfile =>
   pred:Ob      // EventPredicate
   callback:Ob  // CallBack
   subid:Ob     // Id
   subcode:string

Id => string

PubProfile => Ob// EventPredicate

EventPredicate =>
   eventTypes:Ob[]  // String[]
   payload:Ob[] // Ob is an e-speak serliazable object

CoreEventPredicate =>
   eventTypes:Ob[]  // String[]
   payload:Ob[] // Ob is an e-speak serializable object
CallBack => Ob// ESname
```

```
CoreEvent =>
   eventType:string
   eventAttrs:Ob  // EventAttributeSet
   controlAttrs:Ob  // EventAttributeSet
   payload:Ob   // Ob is an e-speak serializable object

EventAttributeSet =>
   vocab:Ob     // ESname
   data:Ob[]    // ESmap (string, Attribute)
   format:string
```

This e-speak object is used by the Client library.

```
ParamUnit =>
   object:Ob  // Any e-speak or Java serializable object
   type:string
```

These classes are used when exporting Core-managed Resources.

```
RepositoryHandle =>
   handleType:string
   idValStr:string

Vocabulary =>
   description:string
   properties:Ob// AttributePropertySet

ResourceContract =>
   vocabs:Ob[]  // ESname
   type:string

MappingObject =>
   resolved:map  // ESname[]
   unresolved:Ob[]// SearchRecipe[]

NameFrame => map// (String, MappingObject)[]
```

```
Key =>

KeyRing => Ob[]// ESname

RepositoryView => Ob[]// ESname
```

These e-speak objects are used by the intercorecom.

```
ESIPMessage =>
   header:Ob     // ESIPHeader
   payload:Ob    // ESIPExportMessage

ESIPHeader => int

DripExportMessage =>
   resources:Ob[][]  // ExportFE
   bytes:Ob[]    // Serialized ExportForm (bytes[])
   table:Ob[][]  // ExportFE
   updateflag:byte
   msg:byte[]
ExportFE => string

ImportContext =>
   exportFrame:Ob  //ESname
   importFrame:Ob  //ESname
   bytesArray:Ob[]      // Serialized ExportForm (bytes[])
   tableArray:Ob[][]      // ExportFE
   retry:byte
   exportByName:byte
   updateFlag:byte
   resourceTables:Ob[][]      // ExportFE
   retryTable:Ob[]  // unused
   inbox:Ob     //ESname
```

```
ExportContext =>
   typeFlags:byte
   exportPolicy:byte
   resourceTables:Ob[][]    // ExportFE
   bytesArray:Ob[]    // Serialized ExportForm (bytes[])
   tableArray:Ob[][]    // ExportFE
   mappings:Ob[][]  // unused
   mappingBytes:Ob[]  // unused
   mappingTables:Ob[][]// unused
```

If an object is encountered that is not well known, we'll fall back on Java serialization to encode it. This will, of course, only work if both sides are using Java serialization.

```
JavaSerializedObject => **rest**

Ob =>
'*S', String | '*I', Integer | '*J', Long | '*B', Boolean |
'*N', Null | '*b', ByteArray | '*v', ObjectArray | '*H', ESMap |
'*z', ESArray | '*s', ESSet | '*l', ESList | 'F', ESName |
'E', RSD | 'S', ResourceSpecification | 'D', ResourceDescription |
'A', AttributePropertySet | 'Y', ValueType |
'T', AttributeProperty | 'a', Attribute | 'B', AttributeSet |
'V', Value | 'C', SearchRecipe | 'c', SearchPredicate |
'g', LockedPermissions | 'n', LiteralName |
'q', AttributePredicate | 'N', NameSearchPolicy |
'b', BootstrapReply | 'Z', ESException | 'z', ESRuntimeException |
'[', ESUID | 'M', Message | 'I', InboxMessageAtom |
'O', OutboxMessageAtom | 'R', ResourceInfo |
'L', PayloadReference | 'f', PayloadFromCore |
'P', PayloadForCore | 'U', SubProfile | 'e', ExtEvent | 'w', Id |
'p', PubProfile | 'J', EventPredicate | 'j', CoreEventPredicate |
'K', CallBack | 'Q', CoreEvent | 't', EventAttributeSet |
'u', ParamUnit | 'h', RepositoryHandle | 'v', Vocabulary |
'r', ResourceContract | 'm', MappingObject | 's', NameFrame |
'k', Key | 'y', KeyRing | 'W', RepositoryView | 'l', Lock |
'd', DripMessage | 'H', DripHeader | 'G', DripExportMessage |
'x', ExportFE | 'i', ImportContext | 'X', ExportContext |
'*O', JavaSerializedObject
```

These are the alternatives for the Value class.

```
StringValue => string
LongValue => long
DoubleValue => double
BooleanValue => byte
BigDecimalValue => string
TimestampValue => string
DateValue => string
TimeValue => string
IntegerValue => int
FloatValue => float
CharValue => short
ByteArrayValue => byte[]
ByteValue => byte
ShortValue => short
InvalidValue =>
ValueAlt => '#0', StringValue | '#1', LongValue |
'#2', DoubleValue | '#3', BooleanValue | '#4', BigDecimalValue |
'#5', TimestampValue | '#6', DateValue | '#7', TimeValue |
'#8', IntegerValue | '#9', FloatValue | '#10', CharValue |
'#11', ByteArrayValue | '#12', ByteValue | '#13', ShortValue |
'#-1', InvalidValue
```

Security Implications (Informational)

A malicious Client might try to trick a Resource Handler into accessing a Resource for which it does not have permission or that is not visible to it. The Client can do this by sending a Search Recipe constructed so that when the Resource Handler completes the binding, it would be to a Resource not visible to the original Client, or if it is, the Client might not have permission for the requested operation.

The Core has no way of checking the Permissions or Visibility fields of Search Recipes before they are resolved. This means that when a Resource Handler receives a Search Recipe as a Secondary Resource, it needs to take special care that this is not an attack. Many Resource Handlers might refuse to complete incomplete bindings, treating them as an invalid parameter instead. Others might require the Client to authenticate itself.

Chapter 7 Inter-Core Communication

This chapter describes the Inter-Core Communication Architecture. The two main components of the Inter-Core Communication Architecture are the Remote Resource Handler and the Connection Factory. The Connection Factory is responsible for the establishment of communication, the negotiation of the stack parameters, and the instantiation of communication. The Remote Resource Handler uses the generated stack to communicate with its peer using the E-speak Service Interchange Protocol (ESIP). There is one Connection Factory per Logical Machine.

The Inter-Core Communication Architecture uses the concept of a network stack. Each layer provides specific services to the next layer above and below.

The Inter-Core Communication Architecture uses the e-speak serialization format defined in the e-speak Application Binary Interface (ABI) described in Chapter 6, "Application Binary Interface." The e-speak ABI defines the on-the-wire format for the objects described in this chapter. Specifically, the e-speak ABI serialization format is used for sending requests from Clients to the Connection Factory and Remote Resource Handlers.

Design Objectives

E-speak is designed to work in heterogeneous environments where there will be all kinds of platforms such as PCs and personal mobile devices that use various communication media such as the Internet, infrared, and radio. Therefore, when two Logical Machines contact each other, both parties have to agree on a common way of communication. This implies negotiation and dynamic creation of protocol stacks.

Thus, inter-Core communication must be able to do the following:

- Handle different mediums of communication: Ethernet, infrared, radio, and the like
- Provide easy integration in different environments: PC, palmtop, pagers, and the like
- Provide a negotiation framework for protocols and their parameters
- Enable dynamic creation of network stacks based on negotiation results
- Satisfy different security requirements (including a version that does not use cryptography)
- Provide a manageable way to deal with versions
- Support interoperability between secure and insecure stacks

Communication Architecture Overview

Figure 5 shows the principal components of the Inter-Core Communication Architecture and how they interact to establish a communication channel between two Logical Machines. The process is as follows:

1. The Client invokes the Connection Factory, CFa, asking it to connect to the Core represented by the Connection Object, CO. (To do this, it sends a message to COREa with CFa as the primary Resource.)
2. CFa receives the message from COREa.
3. The Connection Factories, CFa and CFb, negotiate the parameters and then generate the Remote Resource Handler stack. The layers of the stack are negotiated from bottom to top. When a layer is negotiated, the corresponding implementation is generated and used to negotiate the parameters of the next layer above.

4 CFa and CFb create their Remote Resource Handlers, RRHa and RRHb, with the generated stack as a parameter.

5 RRHa and RRHb connect to each other.

Once the Remote Resource Handlers are connected, they go through the initial process of Resource import and export (Resource exchange). Once this is done, the two Cores can use each other's Resources. The Remote Resource Handler interaction is described in "Session Layer Protocols" on page 117.

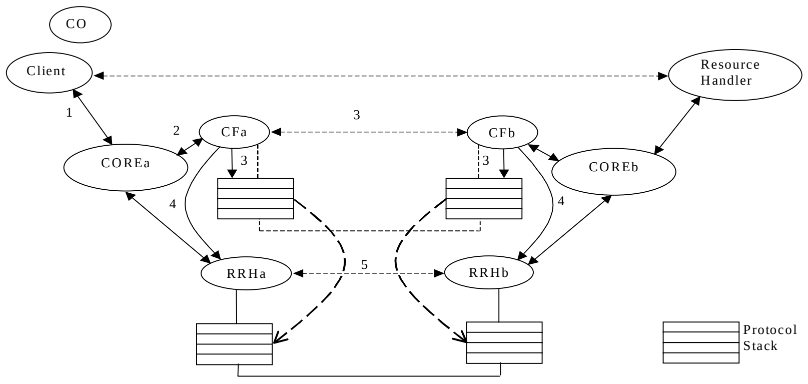

Figure 5 Connection and stack instantiation

The Inter-Core Communication Protocol stack used by the Remote Resource Handlers is shown in Figure 6. From the bottom of the stack upward, the function of these layers is as follows:

- The Transport Layer deals with connection establishment, connection release, flow control, error detection, and error recovery. It can be seen as a pipe that reliably transmits a stream of bits to and from the peer. If the stack is instantiated over TCP, this layer can be null.
- The Security Layer is responsible for securing the stream. It provides authentication, encryption, and traffic padding, if applicable.

- The Session Layer enables two Remote Resource Handlers to manage a session. It handles session recovery in case of a crash and can also resume a session previously suspended. The heartbeat monitors that a session is still alive.
- No functionality is currently defined for the Core Interconnection Layer. It can be used by developers to add functions to the stack.
- The E-speak Service Interaction Protocol Layer is responsible for Resource import, Resource export, and forwarding messages. The e-speak Service Interchange Protocol (ESIP) is the protocol that Remote Resource Handlers use to communicate with each other.

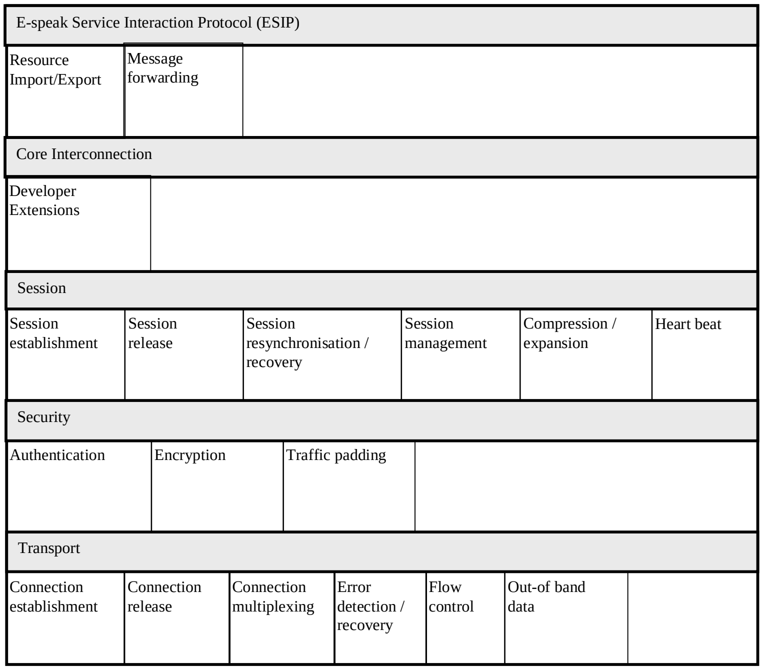

Figure 6 The Inter-Core communication stack

The Connection Object

A Connection Object describes how to connect to a Logical Machine. Connection Objects can be written down and transmitted out of band (e.g., in e-mail). Connection Objects can be constructed from strings.

The format of this string depends on the protocol. The general form is:

```
<PROTOCOL IDENTIFIER>: <PROTOCOL SPECIFIC INFORMATION>
```

The `protocol identifier` field is delimited from the `protocol specific information` field by one or more space characters. Currently only one protocol identifier is defined: TCP. The format for TCP is:

```
TCP: <fully qualified domain name>  <port number>
```

The Connection Factory Interface

The Connection Factory Interface is available to Clients via the e-speak ABI. The format of messages sent to the Connection Factory is defined by the e-speak ABI serialization format. The Connection Factory defines no new exceptions. It uses the standard e-speak exceptions:

```
public interface ConnectionFactoryInterface
{
   public String openConnection( byte[] ConnectionObject);
   public void closeConnection( byte[] ConnectionObject);
   public void closeConnection( String localName);
   public byte[] getMyCO();
   public ESMap getConnections();
   public void exportOnConnecting(ESname resource)
      throws MultipleResolvedBindingException,
         UnresolvedBindingException;
   public byte[] readConnectionObjectFile(String fileName);
}
```

The `openConnection()` method returns a string that can be used to identify the connection later. A null is returned if the connection cannot be established.

Two methods are provided for closing a connection. One takes a connection object; the other takes a name that should have been returned by a previous call, `openConnection()`.

The method `getMyCO()` returns the Connection Object for this Connection Factory.

The method `getConnections()` returns an ESMap of strings representing the currently open connections.

The method `exportOnConnecting()`is used to ensure that the Resource passed as a parameter will be exported automatically whenever a new connection is established. The given Resource referenced by the given name must be a simple binding; otherwise, an exception will be thrown.

The Remote Resource Handler Interface

The Remote Resource Handler Interface is described below:

```
public interface CoreProxyInterface

{
  public void exportResource(ESName resource,
                                       Boolean forceExport)
    throws ExportFailedException

  public void exportResource(ResourceReference resource,
    Boolean forceExport,
    Boolean toplevelExport)
      throws ExportFailedException;

  public void importResources( ESArray exportnames);
}
```

The first `exportResource()` method exports the Resource to the remote Core to which the Remote Resource Handler is connected.

If the boolean `forceExport` is set to true, the export will occur even if the Resource has already been exported. Normally, a Resource is exported only once, but the `forceExport` flag can be used to update the exported Resource, for example, in case of a change in the metadata.

The default export policy is to export recursively all dependent Resources (Resources found in the metadata or state of Core-managed Resources). This is not necessary if the Client knows that no dependent Resources have been modified. The second `exportResource()` method will only export the top level Resource if the `toplevelExport` flag is true. However, if the `importerExporter` finds that a dependent Resource has not yet been exported, an `ExportException` will be thrown.

Both versions of `exportResource()` assume that the given ESname is bound to a single Resource. If this is not the case, the export will fail.

The `importResources()` method is used as part of an error recovery mechanism. The Remote Resource Handler takes the same approach as the Core; thus, there is no handshaking between the two Remote Resource Handlers. If the importer fails, the importer and exporter will be out of sync. Resources may have been exported by the exporter that have not been imported by the importer. This can cause problems the second time the same Resource is exported because the exporter will send only the export name without re-creating the Export Form, because its state indicates that the Resource has already been exported. The importer, on the other hand, will not be able to process the import because it will not find the export name in the list of names that it has already imported.

When this out-of-synchronization error is detected, the Importer can ask its Remote Resource Handler to import the Resource using `importResources` (the export names are the ones that have been out of synchronization). This request will eventually cause the exporter to re-export the same Resources.

Payload Format

The Connection Factory and Remote Resource Handlers are non-Core-managed Resources. The `payload` field in Outbox Message Atoms of messages sent to these services and the `payload` field in Inbox Message Atoms of messages received from these services are defined by the class `payloadForICCA`.

```
class payloadForICCA
{
     Ob[] arguments;
}
```

Each field in `payloadForICCA` is serialized according to the e-speak ABI serialization format.

For an Outbox Message Atom, `interfaceName` will either be `CoreProxyInterface` or `ConnectionFactoryInterface` and `methodName` will be the string corresponding to one of the methods in these interfaces. Each element of the arguments array can be any type defined in the e-speak ABI serialization format. For `InboxMessageAtom`, the `interfaceName` and `methodName` fields are not used.

Layer Factory Negotiation Protocol

A negotiation protocol is required to generate protocol stacks dynamically. A new stack is built each time a connection is made to a new Logical Machine, so that the requirements of different devices and communication media can be accommodated at run-time. The Layer Factory Negotiation Protocol (LFNP) defines the procedures and packet formats to negotiate parameters for a stack layer.

LFNP Message Definition

Message Header

An LFNP message has a fixed header format, shown in Figure 7, followed by a protocol proposal.

Level	Version	Type	Length
1 byte	1 byte	1 byte	2 byte

Figure 7 LFNP header

Level (1 byte) indicates which layer the message is negotiating. Valid values are shown in Table 11.

Version (1 byte) indicates the version of the LFNP in use. The current version is 0x01.

Table 11 Negotiation levels and corresponding values

Level	**Value**
Transport proposal	0
Authentication proposal	1
Security proposal	2
Session proposal	3
Core interconnection proposal	4
ESIP version proposal	5
Reserved	6-255

Type (1 byte) indicates if the payload is an offer (0x00) or counteroffer (0x01).

Length (2 bytes) indicates the length of the total message (header + payloads) in bytes.

Protocol Proposal

A protocol proposal is a list of *protocol identifiers*.

Protocol Identifier

A protocol identifier (PID) is a 1-byte field. It specifies a protocol.

The semantics of the PID are related to a layer. The values defined currently are shown below:

- Transport: 0x00 for TCP; 0x01 for HTTP (HTTP is currently not supported in the implementation.)
- Security: 0x00 for null; 0x01 for DES (Not supported in the current implementation)
- Session: 0x00 for SP (The session protocol defined in this document)
- Core interconnection: 0x00 for ESIP:CI (The protocol defined in this document)
- ESIP version: 0x00 (ESIP version 1.0)
- Authentication: 0x00 (authentication by name)

Negotiation Example

In Figure 8, we present the set of messages exchanged during the negotiation of the Transport Layer. The initiator proposes three protocols, identified by PID-1, PID-2, and PID-3.

Initiator to Listener: Message 1

Level	Version	Type	Length	PID	PID	PID
0x00	0x01	0x00	0x10	PID-1	PID-2	PID-3

PID=protocol identifier

Listener to Initiator: Message 2

Level	Version	Type	Length	PID
0x00	0x01	0x00	0x06	PID-1

PID=protocol identifier

Figure 8 Messages exchanged during the negotiation of the Transport Layer

After these two messages, all the parameters that have been negotiated for the layer and PID-1 will be used.

Note that a counteroffer message (of type 0x01) is used only if none of the protocols in the original message is acceptable.

Building the Stack

The stack is built from the bottom up, starting with the Transport Layer. Once a layer is instantiated, LFNP is run over that layer to negotiate the next layer. For example, the negotiation of the Session Layer uses the Security Layer, which in turn uses the Transport Layer. This ensures that the protocol stack is tested as it is built.

Transport Layer Protocols

Two protocols are defined for the Transport Layer: TCP and HTTP. Currently, only TCP is supported.

Authentication Layer Protocols

The Authentication Layer is negotiated immediately after the Transport Layer has been negotiated and instantiated. It is run over the Transport Layer. Once the Authentication Layer is negotiated, it is run to authenticate the remote machine and then removed. The authentication information can be used to determine negotiation policy for the remaining layers, and it can be used by the Remote Resource Handler to determine what Protection Domain it will use.

The format for the Name Authentication Protocol message is shown in Figure 9.

Message type	Length	Name
1 byte	1 bytes	Length - 2 bytes

Figure 9 Format of Name Authentication Protocol

The following values are valid message types:

- 0x00—Give me your name
- 0x01—This is my name

The `name` field can be any arbitrary string.

Security Layer Protocols

The only protocol defined for this layer is the null protocol; no messages are exchanged between Security components in the protocol stack. In the current implementation, the Security component simply relays messages up and down the stack.

Session Layer Protocols

The Session Protocol (SP = 0x00) is the null protocol.

Core Interconnection Layer Protocols

The Core Interconnection Layer Protocol (CI:ESIP= 0x00) is the null protocol.

ESIP

ESIP relies on the e-speak Serialization Protocol defined as part of the e-speak ABI for encoding its protocol messages. Now we specify ESIP protocol messages:

```
public class ESIPMessage implements ESSerializable
{
   ESIPHeader hdr__;
   ESIPPayload pyld__;
}

public class ESIPHeader implements ESSerializable
{
   public final static int ESIP_UNDEFINED = 0;
   public final static int ESIP_CONTROL_MESSAGE = 1;
   public final static int ESIP_IMPORTMESSAGE= 2;
   public final static int ESIP_EXPORTMESSAGE= 3;

   int msgType__;
}

public class ESIPExportMessage extends ESIPPayload implements
ESSerializable
```

```
{
  private ESArray resources__;
  private ESArray bytesArray__;
  private ESArray tableArray__;
  private boolean updateFlag__;
  private byte[] msg__;
}
```

`ESIPExportMessage` is used for explicitly exporting Resources to the remote Logical Machine (an explicit export message) and also for forwarding messages to Resource Handlers on the remote Logical Machine (a forwarded message). In the latter case, any Resources sent as secondary Resources will be implicitly exported as a side effect of sending the message.

The `resources__` field is a two-level array of Resources being exported. Each outer element corresponds to a Message Atom. (Recall that multiple messages can be sent simultaneously, each as a separate atom.) Each inner element of the array is the export name (`ExportFE`) of the Resource.

If the `ESIPExportMessage` is a message to be forwarded to a Resource Handler, then the following ordering is assumed for the elements of *`resources__`*:

1. The Callback Resource
2. The Exception Handler
3. The secondary Resources
4. Key Export Names

   Key Export Names are the export names (`ExportFE`) of all Keys previously imported from the remote machine and that have been attached to the message.

The `msg__` field is the message sent by the Client to the Core with all string fields of `ESName` replaced by the corresponding export name (`ExportFE`). The format of this message is defined by the e-speak ABI. This array is empty if the message is an explicit export. Note that the `msg__` field is the only place that the primary Resource is included in a forwarded message: It will be in the Outbox Message Atoms contained in the message. Primary Resources do not appear in resources__, bytesArray__, or tableArray__. It is assumed that the primary Resource is already known to the machine to which the message is being forwarded.

The structure of bytesArray__ reflects that of resources__: It is a two-dimensional array in which each outer element corresponds to a Message Atom. Each of the inner elements of bytesArray__ is the Export Form (defined below) for a Resource. The elements of the array will be the Export Form for the elements of the resources__ array that have not already been exported to the remote machine. In general, the Export Form for a Resource will contain the names of other Resources. The Export Form for all Resources whose names it contains follows this element as a separate entry in bytesArray__. So there is one entry in bytesArray__ for each Resource. Within entries in bytesArray__, names of contained Resources are replaced with a marker, as described below:

```
public class Marker
{
   byte SignalByte; //Defined in the e-speak ABI
   byte label;      //always 'L'
   int n;           //4 bytes, always the number 4
   int index;       //index of entry in tableArray__
}
```

The process is recursive, terminating only when a Resource that is encountered has already been imported from this machine or exported to this machine, or whose Export Form is already in this message. Once the Export Form for the first element of resources__ is completed, the next element of bytesArray__ will be the Export Form for the second element of resources__. This continues until the Export Form for all elements of resources__ (that have not already been exported) have been included in bytesArray__.

Note that the ordering of Resources in bytesArray__ is not significant, because all entries are self-describing.

The array tableArray__ contains the export name (ExportFE) of all Resources named in the Export Form of the elements of resources__. So the first time a Resource is encountered contained in the Export Form of another Resource, its export name (ExportFE) is appended to tableArray__. If the Resource is encountered again, no addition is made to tableArray__.

If the `updateFlag__` field is set to true, the receiver of this message should not ignore any Resources contained in this message that it has already imported. Instead, it should update the Resource metadata (and state in the case of export by value):

```
public class ESIPImportMessage extends ESIPPayload implements
ESSerializable
{
   ExportFE [] exnames__;
   byte[] esmsg__;
}
```

`ESIPImportMessage` consists of a list of `ExportFE` (export names) to be imported and optionally a byte array containing the message that caused the error condition to be detected.

```
public class ESIPControlMessage extends ESIPPayload implements
ESSerializable
{
   public static final int TYPE_UNKNOWN= 0;
   public static final int TYPE_CLOSE_CONNECTION_1= 1;// phase 1
   public static final int TYPE_CLOSE_CONNECTION_2= 2;// phase 2
   public static final int TYPE_OPEN_CONNECTION_1= 21;//
   public static final int TYPE_OPEN_CONNECTION_2= 22;//
   public static final int TYPE_OPEN_CONNECTION_3= 23;//
   public static final int TYPE_OPEN_CONNECTION_4= 24;//

   int type__;
}
```

`ESIPControlMessages` are used to synchronized Remote Resource Handlers when a connection is established. The side that originally initiated the connection sends a message of type OPEN_CONNECTION_1, waits until it gets a message of type OPEN_CONNECTION_2 , sends a message of type OPEN_CONNECTION_3 and then waits to get a message of type OPEN_CONNECTION_4. This is shown in Figure 10.

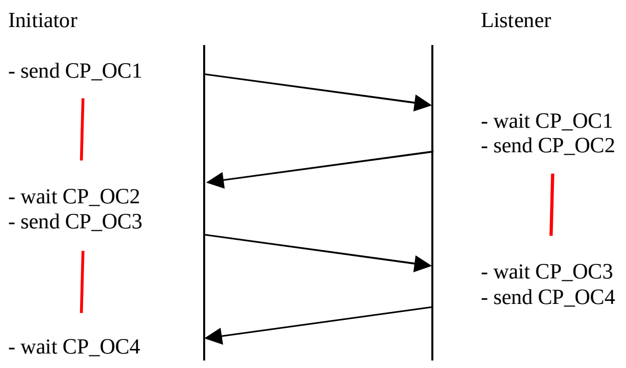

Figure 10 ESIP control messages opening a connection

The protocol for closing a connection is similar, but either side may initiate it. This is shown in Figure 11.

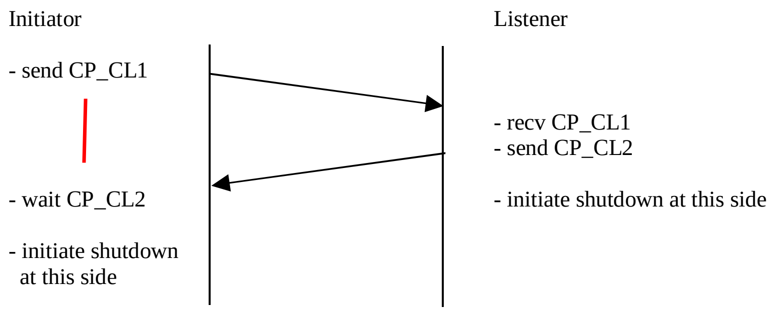

Figure 11 ESIP control messages closing a connection

If a problem is encountered with the messaging, an `UndeliverableRequestException` will be returned.

Export Form for Resources

Class `ExportForm` is defined below:

```
public class ExportForm
{
   ExportFE ExportName;
   String ResourceType;
   ResourceDescription desc;
   ResourceSpecification spec;
   byte[] ResourceState;
}
```

The components of `ExportForm` are serialized in the format specified by the e-speak ABI.

The Resource type string is `Key`, `KeyRing`, `NameFrame`, `Contract`, `RepositoryView`, `Vocabulary`, or `ExternalResource`. `ExternalResource` denotes a non-Core-managed Resource.

The `desc` and `spec` components are the metadata for the Resource. There are two main differences from what was provided by the Client that originally registered the Resource—the handling of the Resource-specific data (RSD) fields and Permissions. The public RSD is always included in the Export Form, but the private RSD is included only if the Resource is being exported by value. Similarly, the Resource Permissions and metadata Permissions are exported as Key-label pairs if the export is by value. If the export is by reference, only the names of the Keys are transmitted. The differences are summarized in Table 12.

Table 12 Metadata export

Element	Included
Repository Handle	Not included
Resource Handler	Not included
Resource Owner	Not included
Contract	Export name of the contract Resource
Attributes	Export name of the Vocabulary Resource Attributes unmodified
Pass by value flag	Boolean
Export by name flag	Boolean
Persistence flag	Boolean
Public RSD	Unmodified
Private RSD	Unmodified if passed by value Not included if passed by reference
Resource Permissions	Export name of Key that opens Lock
Metadata Permissions	Export name of Key that opens Lock
Deny Locks	Array of export names of Keys that open Locks
Allow Locks	Array of export names of Keys that open Locks

The class `SecondaryExportForm` is defined below:

```
public class SecondaryExportForm
{
   ExportFE ExportName;
   String ResourceType; // always "MappingObject"
   byte[] MappingObjectSerialization;
}
```

The Export Form for a secondary Resource is more complicated, because the Mapping Object for `ESName` can be complex. Hence, the entry in `bytesArray__` for a secondary Resource is the ESSerialization of the Mapping Object as defined in the e-speak ABI. This Mapping Object will contain references to Resources. The export name for each of these Resources will be entered into `tableArray__`, and the Export Form for these Resources will be added to `bytesArray__`, as described previously.

Export by Value

Export by value is supported only for Core-managed Resources (extensions will be provided in a later release). Export by value is indicated by the `Pass by Value` flag in the exported Resource Description.

Imported Resource Metadata (Informational)

Most of the data fields in `ExportForm` can be translated directly into the form needed for the Resource Specification and Resource Description, with the exception of Permissions. The metadata Permissions are unchanged when the Resource is exported by value. The Resource Permissions are registered by associating the ESName of a Key with the export name (`ExportFE`) of that Key. This way, when a Permission is extracted by the importer's Core, the Remote Resource Handler will know what Keys were presented and can tell the exporter which Keys to present.

For example, suppose a Resource Permission is protected by a Key called `AddKey` and the export name for this Key is `ExportAddKey`. The importing Core will create a Permission containing the name `ExportAddKey`. This Permission will be protected with some other (local) Key, say `ImportAddKey`. Now a Client on the importing Core must present the Key `ImportAddkey` to unlock the Permission `ExportAddKey` that gets delivered to the importing Remote Resource Handler. This Remote Resource Handler can then tell the exporting Remote Resource Handler that a Key corresponding to `ExportAddKey` was presented.

When a Resource is imported, the importer may see export names (`ExportFE`) in the `ExportForm` metadata for locks and Vocabularies (recall that Vocabularies are part of a Contract Resource) that have not yet been exported to it. These might be exported later, or they may never be exported. For example, suppose we don't want to export the `AddKey` referred to in the above example. We need a way to prevent the importing Core (which may be compromised) from presenting the `ExportAd-dKey` Permission to trick the exporting Remote Resource Handler into wrongly presenting `AddKey` to its Core.

The solution is to use the Visibility fields. The metadata for `AddKey` on the exporting Core contains a single Key, called *local*, in the Allow field. *Local* is never exported and it is not on any Key Ring used by the exporting Remote Resource Handler, nor is it on any Name Frame available to the exporting Remote Resource Handler. Hence, `AddKey` is not visible to the exporting Remote Resource Handler, so it can never be tricked into wrongly presenting it to its Core.

The Visibility fields of a Resource exported by reference are registered with the export name of the Keys. However, each Key must also appear as a Permission so the importer will know that it was presented. Thus, if there is a Key K1 in an Allow field on the exporting Core that has an export name EK1, then on the importing Core, the Allow field will have a local Key EK1-L. A Permission is also unlocked by EK1-L that delivers the name EK1 to the importing Remote Resource Handler. If EK1-L is the only Key in the Allow field, then a Client must present EK1-L. This will cause a Permission EK1 to be delivered to the importing Remote Resource Handler. This Remote Resource Handler is then able to tell the exporting Remote Resource Handler that the (Visibility) Key corresponding to EK1 was presented. Unlike the situation in the Repository of the Logical Machine that owns the Resource, a Key need appear in the Permissions field only once because this is sufficient to demonstrate to the exporting Remote Resource Handler that it was presented.

Restriction on Export of Core-Managed Resources

Resources can be exported by reference or by value. If a non-Core-managed Resource is passed by value, the Remote Resource Handler treats the value as a byte stream. If a Core-managed Resource is passed by value, the builder in the Core needs to interpret the value specification.

Some Core-managed Resources can't be exported at all. For example, a Protection Domain holds a task-specific state, such as the handles to the communications ports. This state is meaningless to another machine. The same applies to an Inbox. A Key could be exported by reference, but there is no practical way to use the value to test it against metadata. A Key could be exported by value, but its state could very well conflict with that of a Key on the importing Logical Machine. Instead, the importer creates a new Key and puts it in the correct places in the metadata.

Other Core-managed Resources can be exported by reference, but only a subset of the interface is available. In particular, such a Resource cannot be used as part of message processing. Restrictions are specified in Table 13.

Table 13 Core-managed Resource export restrictions

Resource	Pass by value allowed	Pass by reference restrictions
Key	No, but a new instance is created to be used in its place	N/A
Key Ring	Yes	Cannot be used in the Key Rings field of a message header
Resource Contract	Yes	Cannot register a Resource in this Contract
Inbox	No	N/A
Name Frame	Yes	Cannot be used as a component of an ESName sent to the Core for name resolution
Protection Domain	No	N/A
Repository View	Yes	Cannot be used in a Search Recipe
Resource Factory	No	N/A
MetaResource	No	N/A
Importer Exporter	No	N/A
Vocabulary	Yes	Cannot be used in a Search Recipe

As Table 13 shows, Keys and Protection Domains are handled differently than other types of Core-managed Resources. The internal state of a Key is used to open Locks, but this internal state is only meaningful on the Logical Machine that created the Key. However, the Resource Handler will still need to know what access rights were unlocked when a request from a remote Client reaches it. Having the importer create a Key that appears in the imported metadata and in the Resource's state avoids any conflicts without requiring global uniqueness of Key values.

The state of those Core-managed Resources exported by value may need to be kept synchronized. Interested Clients can subscribe to an Event that the Core generates each time it changes the metadata or state of a Core-managed Resource. These updates are passed between the Logical Machines.

Chapter 8 Exceptions

Overview

E-speak defines a set of *exceptions* to inform Clients when an error occurs in the system. Two classes of exceptions are defined: run-time exceptions and recoverable exceptions.

Exceptions are described as subclasses as they are in object-oriented languages. E-speak supports other languages, such as C, that don't include the concept of exceptions or of subclasses. The Application Binary Interface (ABI) describes how this hierarchy is encoded for such Clients.

Run-Time Exceptions

Run-time exceptions are thrown when programming errors occur. A program catching such exceptions may terminate. `ESRuntimeException` has the following subclassed exceptions:

- `CorePanicException` is thrown when the Core is unable to process the request. Although the Core will attempt to notify all Clients of its inability to continue operating, it will also reply with this exception for as long as it can. The Core may continue to accept new messages as the problem may be limited to the execution of a single message.
- `ServicePanicException` is thrown when a service is unable to process the request. This may be a terminal error for the service, in which case the service will exit. Or it may simply mean that the request being processed caused an internal error that was not recoverable, and the service will accept new requests.

- `RepositoryFullException` is thrown when the request attempted to add additional information to the Core's Repository, but the Repository was full. This exception can be recovered from if the Client is able to delete one or more Resources from the Repository. It is a run-time exception because almost every message can possibly throw this exception, and the Client has no guaranteed recourse (because some other application can consume the Repository space freed up by this Client).
- `OutofOrderRequestException` is thrown when the state of the system is inconsistent with the request.
- `InvalidParameterException` is thrown by any other programming errors. This exception has three subclasses:
  - `NullParameterException` is thrown where a null parameter was supplied but is not allowed. This error is often caused by passing an uninitialized object.
  - `InvalidValueException` is thrown when a parameter is outside the allowed range.
  - `InvalidTypeException` tells the programmer that the name specified is bound to the wrong type of Resource.

Recoverable Exceptions

Recoverable exceptions occur due to a problem with the state of the system. For example, when the Client sends a message to request access to a Resource, the message may be undeliverable, perhaps because the Client's Inbox is full. Recovery for this case may be as simple as resending the message.

The base exception is `ESException`. This exception is subclassed into two major categories: `ESInvocationException` and `ESServiceException`.

`ESInvocationException` is a base class for all the exceptions that may be thrown by the Core back to the Client occurring during the processing of the request. Exceptions thrown by most handlers are included here to reduce the number of explicit classes of exceptions that must be caught. This exception is further subclassed into:

`NamingException` results from a wide variety of problems. Regardless of the cause, this exception, or any of its subclasses, is thrown only for the primary Resource of the message header. Five subclasses are defined:

- `NameNotFoundException` is thrown when the name resolution process failed to find a given name. The Client can recover by changing `ESName`.
- `EmptyMappingException` is thrown when a Mapping Object is associated with the name, but that Mapping Object has no usable accessors. This condition arises when the accessor has no elements, the elements refer to unregistered Resources, or the Resources did not pass the visibility tests. The Client can recover by changing `ESName` or trying again with a different set of Keys.
- `UnresolvedBindingException` is thrown when all the accessors of the Mapping Object are search requests. The Client can recover by requesting a lookup using the search request.
- `MultipleResolvedBindingException` is thrown when the Mapping Object has more than one explicit binding.

`StaleEntryException` is thrown if the Resource no longer exists. The Core will remove any stale handles from the Mapping Object before returning the exception. A retry will not result in this exception unless another referenced Resource has been unregistered.

`PermissionDeniedException` is thrown by any Resource Handler when the appropriate Resource Permission can't be unlocked. The Client can recover by retrying with a different set of Keys.

`QuotaExhaustedException` is thrown when the Client attempts to define more Resources than it is allowed as defined by the quota assigned. The Client may delete other Resources (thus freeing up quota) and reattempt the request.

`RecoverableCoreException` is thrown when there is a problem while processing the request. There are two associated subclass exceptions:

- `RequestNotDeliveredException` is thrown when the Core never started processing the message. This exception can be thrown by the Client library if it implements time-outs or in by the Core if the corresponding queue is full. It may be possible to recover from this exception by resending the message.
- `PartialStateUpdateException` is thrown when the Core cannot finish processing the message. The Client may need to find out what state was changed before attempting recovery, for example, by examining the state of the meta-data.

`UndeliverableRequestException` is thrown when the message cannot be delivered to the Resource Handler. There are two subclass exceptions:

- `RecoverableDeliveryException` is due to temporary conditions such as a full Mailbox. Recovery can be as simple as retrying.
- `UnrecoverableDeliveryException` is due to a condition that is unlikely to change quickly. The Client can recover by selecting a binding that points to a different Resource Handler.

`ESServiceException` is a base class exception for all service-defined exceptions. The only service defined by the Core is the Name Frame Service, which has the following exceptions defined:

- `NameCollisionException` is thrown when the name specified in an add, copy, or similar operation is already defined in the Name Frame.
- `LookupFailedException` is thrown when no Resources are found that match a Search Recipe.
- `InvalidNameException` is thrown when a string designating a name is not found in the Name Frame.

Exception State

Each exception returns some data. The first element is an integer that identifies the exception; the rest is a string containing exception-specific information. Table 14 shows the ranges used for the various classes of exceptions.

Table 14 Ranges for different exception classes

Class	Range
Run-time	Between 0 and 10
Recoverable	Between 11 and 99
Client defined	100 and above

The specific integers used for each exception are given in Table 15.

Table 15 Exception state

Exception	Code	Other State
Runtime Exceptions		
Invalid parameter	1	
Null parameter	2	
Invalid value	3	
Invalid type	4	
Out of order request	5	

Table 15 Exception state *(Continued)*

Exception	Code	Other State
Core panic	6	
Recoverable Exceptions		
Recoverable Core	11	
Repository full	12	
Request interrupted	13	
Request not delivered	14	
Permission denied	15	
Undeliverable request	16	
Unrecoverable undeliverable request	17	
Recoverable delivery	18	
Naming	20	
Empty mapping	21	
Multiple resolved binding	23	
Name not found	24	

Table 15 Exception state *(Continued)*

Exception	Code	Other State
Name collision	30	
Lookup failed	31	
Warning Exceptions		
Stale entry	29	
Visibility failure	32	

Chapter 9 Events

This section describes the Event Service, a lightweight, extensible service targeted at loosely coupled, distributed applications. Events provide a publish-subscribe mechanism for communication built on top of e-speak messaging.

The Event specification defines two interfaces.

- The ListenerIntf defines the format of Event notifications.
- The DistributorIntf defines the format of publish and subscribe requests.

Event Model

E-speak supports an extended form of the familiar publish-subscribe Event Model. There are four logical entities in the e-speak Event Model, whose interactions are illustrated in Figure 12.

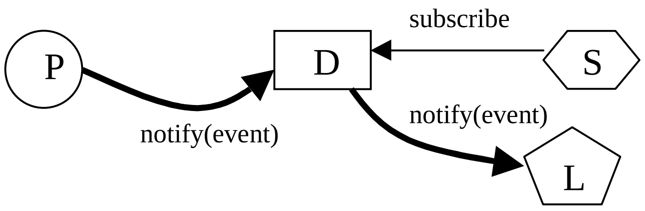

Figure 12 Interactions in the Event Model

A *Publisher* (P in the figure) is an entity that generates an Event notification message. The recipient of an Event notification is called a *Listener* (L). A *Distributor* (D) is an extension of a Listener. It receives Events and forwards them to other Listeners. A *Subscriber* (S) is an entity that registers interest in a particular Event with a Distributor and designates the Listener to whom Events should be sent. The Subscriber and the Listener are typically the same physical entity. Similarly, it is fairly typical for a Publisher to act as a Distributor of its own Events.

The Core itself is an example of an Event Publisher. It sends Events to a trusted Client called the *Core Distributor* to signal state changes such as a change in a Resource's attributes. The Core Distributor may then distribute these Events to interested Clients that have appropriate authority.

Interaction Sequence

To interact with a Distributor, Publishers and Subscribers need to obtain a name binding for the Distributor's Resource. Hence, the first step in a typical interaction sequence is for a Distributor to register a Resource in the Repository. The attributes of this Resource indicate that it supports the Distributor interface. An Event Publisher can then obtain a reference to this Resource by an attribute-based lookup and declare its intent to send Events by sending a publish request to the Distributor.

Independently, a Subscriber also finds the Distributor Resource using attribute-based lookup and subscribes to a set of Events by sending a subscription request. The subscription specifies a Resource, called a CallBack Resource, for which a Listener is the Resource Handler.

Upon receiving an Event from the Publisher, the Distributor looks up its list of active subscriptions. If the incoming Event matches the filter in the subscription, the Distributor forwards the Event to the designated Listener.

Figure 13 illustrates a typical Event notification process where the Subscriber and the Listener have been folded into a single Client.

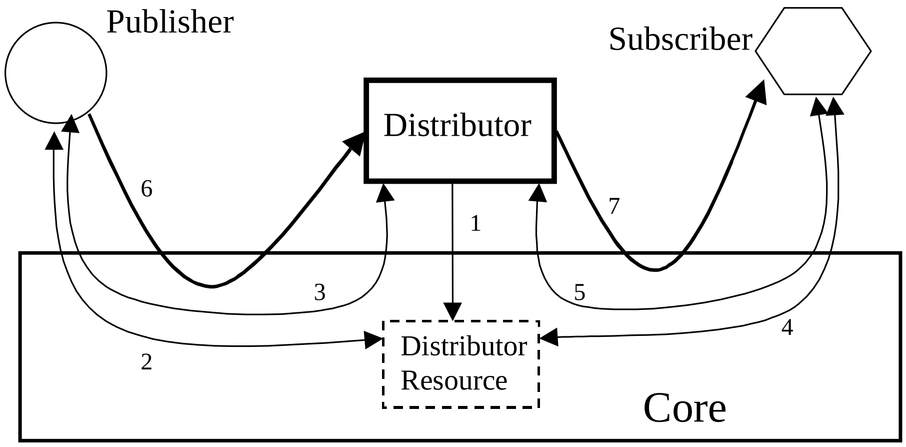

Figure 13 Distributor Event Notification Process

The numbers in the figure represent the steps in the process:

1 The Distributor registers with the Core.

2 The Publisher discovers the Distributor.

3 The Publisher sends a `publish` request to the Distributor describing the Events it will be generating.

4 The Subscriber discovers the Distributor.

5 The Subscriber sends a `subscribe` request to a Distributor describing the Events it is interested in.

6 The Publisher sends the Event to the Distributor using a `notify` message.

7 The Distributor forwards the Event to the Subscriber (also using a `notify` request).

Note that the Subscriber and the Publisher are not the target for any calls. Thus, only interfaces for Distributors and Listeners are needed. The Listener interface includes only the methods required to receive an Event. This interface is listed below:

```
interface ListenerIntf
{
   public  String   notify      (Event event);
   public  void     notifyAsync (Event event);
}
```

Distributors are just specialized Listeners; they extend `ListenerIntf` and add methods that allow publish/unpublish and subscribe/unsubscribe:

```
interface DistributorIntf extends ListenerIntf
{
   public  Id publish   (PubProfile publication);
   public  Id unpublish (Id pubId);
   public  Id subscribe (SubProfile subscription);
   public  Id unsubscribe (Id subId);
}
```

Of course, Event-system builders can extend these interfaces and define other interactions between Distributors and Publishers/Subscribers.

Distributor Vocabulary

A vocabulary is defined in which Distributors can be registered.

Table 16 Distributor vocabulary

Attribute Name	Value Type	Comment	Meaning
ServiceName	String		Name assigned to Distributor
ServiceType	String		Type of distributor
EventTypes	String	Multivalued	Event types handled
Persistent	Boolean	always false	True if Distributor state survives Core restart
Buffered	Boolean	always false	True if Distributor is able to accept events faster than it can forward them
Secured	Boolean	always false	True if event state is tamper proof
QOSLevel	Integer	always 0	Quality of service level assigned by Distributor
Multiplexed	String	Multivalued	Type of aggregation and summarization

The Service Name, Service Type, and Event Types are strings that are assumed to have meaning to Publishers and Subscribers who have discovered the Distributor. For example, the Core Distributor could be described with a Service Name of "Core", a Service Type of "Core", and Event Types of "Repository" and "Metadata".

The Persistent, Buffered, Secure, and QOSLevel attributes must be set as shown because the current Distributor implementation does not support these features. The Multiplexed attribute can be set by Distributors to describe how they combine events. For example, a Distributor may aggregate billing events from a particular customer and publish an aggregate event to the subscribers. The values assigned are assumed to have meaning to the Publishers and Subscribers of the events.

Data Model

This section describes the data associated with each interface and the interfaces used to access Events.

Event

An e-speak Event is a set of named attributes, where each attribute is a name-value pair. An Event also contains a reference to an e-speak Vocabulary. The Vocabulary enumerates the names of allowed attributes and their types. Specifying a Vocabulary in an Event makes the Event content self-describing. A recipient of a self-describing Event does not need to know anything about the Event's content *a priori*; it can query the Vocabulary to determine the Event's attributes and their types and then extract the values of the attributes it is interested in. Event generators may choose to leave the `vocabulary` field `null`, in which case Event attributes must be agreed upon *a priori*, the default meaning being the e-speak Base Vocabulary.

An Event is defined as follows:

```
interface Event
{
   EventAttributeSet getEventAttributeSet();
   void              setAttributeSet();
   String            getEventType();
   void              setEventType();
}
```

where `EventAttributeSet` contains a reference to a Vocabulary and a hash table of attributes, each of which is a name-value pair keyed by attribute name:

```
class EventAttributeSet
{
  ESName    vocabulary;
  ESMap     attributes;
  void setVocabulary(ESName vocabulary);
  void setAttributes(ESMap attributes);
  ESName getVocabulary();
  ESMap getAttributes();
}
```

The `eventType` string is a special attribute that is so commonly used that it has its own accessors. Event Publishers typically use this field to identify the category that the Event belongs to. The Event Attribute Set Vocabulary can also be used to denote the Event type.

Subscription

A subscription consists of a subscribe request with the signature:

```
Id subscribe( SubProfile subscription)
```

`SubProfile` contains a `CallBack` field that specifies the target destination for Events that match the subscription. It also describes the Events of interest through `EventAttributeSet`. As described above, `EventAttributeSet` contains a reference to a Vocabulary and a set of name-value pairs. In this case, the values denote matchers for each attribute. A Distributor has complete control over the kinds of matchers it is willing to accept. Matchers execute in a Distributor's address space; hence, Distributors may restrict the kinds of matchers they accept for both security and performance. A filter is a tuple of the form (attributeName, matcher). Depending on the application, the matcher may be as simple as an integer selected from a list of predefined matchers that the Distributor supports or as complex as byte code that the Distributor loads and executes. The Distributor will only send an event to the listener if it satisfies the conditions of the filter.

```
class SubProfile
{
   CallBack           listener;
   EventAttributeSet  filter;
}
```

`CallBack` in `SubProfile` encapsulates a name binding for the Callback Resource to which all Events that match the filter should be sent.

```
class CallBack
{
   ESName  listener;
}
```

The Listener Resource in `CallBack` must be a simple binding. If the binding is not simple, the Distributor will return a Naming Exception. If the filter is not expressed in a legitimate Vocabulary, or if the specified matchers are not supported by the Distributor, an Invalid Parameter Exception is returned.

If the subscription is acceptable to the Distributor, it returns an Id that the Subscriber can use to unsubscribe. An Id contains an integer that identifies the subscription:

```
class Id
{
   int   subId;
}
```

The Subscriber can end the subscription by sending an unsubscribe request to the Distributor.

```
Id unsubscribe(Id id);
```

The Distributor will return a Permission Denied Exception if the subscription is not active. Otherwise, the Id is echoed back. The Distributor may terminate the subscription if the Listener's Callback Resource is unregistered.

Publication

A publication request consists of a publication profile, `PubProfile`. `PubProfile` is similar to `SubProfile` and describes the attributes of the Events that the Publisher intends to send. All other interactions mimic the interactions between a Subscriber and a Distributor.

Core-Generated Events

The Core is a Publisher of Events. All Events published by the Core go to a single service called the *Core Distributor Service*. This service is the Resource Handler for several Distributor Resources, each dealing with a Core-generated Event of a different type. These are:

- Changes to the state of the Repository
- Changes to the state of Core-managed Resources

These types are used to maintain the coherence of metadata and the Resource state shared by value. Both are described in the *Base Event Vocabulary*.

The Core Distributor is very much like the Remote Resource Handler—it is implemented as a Client, but it needs to be part of the trusted computing base because it gets information about every (monitored) Resource's state change and could gather sensitive information. Figure 14 illustrates the configuration.

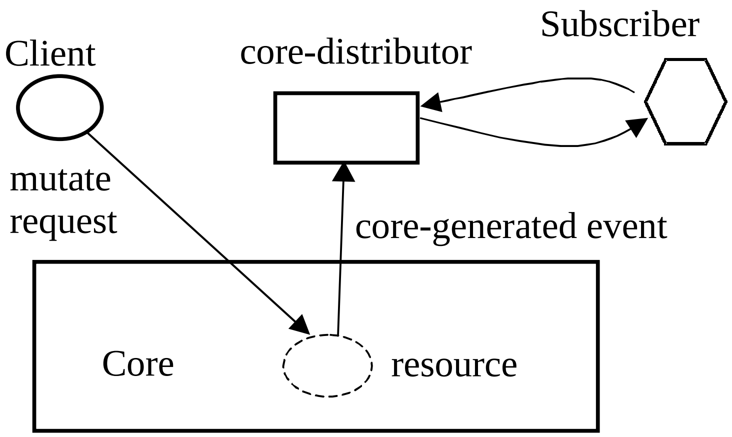

Figure 14 Core Distributor configuration

The Event state depends on the type of Event, but some data is the same for each type. The common fields are shown in Table 17.

Table 17 Event state common to all Events

Type—Event Type	"core.mutate.<metadata>.<op>" "core.mutate.<resource>.<op>"
Time—Distributor's time stamp	"YYYY-MM-DD HH:MM:SS.FFFFFFFFF"
SendId—Sender's message Id	String
Resources	ESName[]

Because e-speak messaging guarantees in-order delivery of messages between a specific sender and receiver, the Distributor's time stamp allows higher-level services to build a "happened before" relationship among Events from different Publishers. The message Ids are useful for debugging and intrusion detection.

Because some Core Events may be used for billing and intrusion detection, the Core will reduce the rate at which it processes requests if it can't publish these Events at a sufficient rate. Because most requests that pass through the Core result in one or more Events being processed, care is needed so that publishing them does not inordinately reduce the throughput of the system. The cost of publishing each Event is amortized by having the Core wait until a significant number of Events has occurred and publishing them all at once.

The load on the Publisher and Distributor is reduced further by having the Distributor provide a filter to the Publisher. The Core Distributor's filter specifies the Resources for which the Core must publish Events. Metadata and Resource Events might be used only for maintaining the consistency of exported Resources. The Core need not generate Events for messages involving Resources not meeting these criteria. In addition, the Distributor tells the Publisher which fields will be forwarded to Subscribers. The Publisher need not include fields that won't be forwarded, reducing the burden even for Events that have Subscribers. The Publisher may publish events that do not satisfy the Distributor's filter.

Networks of Distributors

A Distributor can summarize data from Events and publish Events to other Distributors. Cycles are possible in such a situation unless care is taken. The Core Distributor never takes Events from any place but the Core, so there can be no cycles involving the Core Distributor.

A Distributor can receive Events from another Distributor simply by issuing a subscription request. It can then accumulate Events, abstract their data, and publish new Events or merely forward the Events it receives. The same kind of control flow that is used between Publishers and Distributors can be used between Distributors to avoid sending Events that have no subscription.

Events in a Distributed Environment

Events are messages that trigger special actions by the recipients. In particular, when a Client receives an Event, the callback registered for this Event is invoked. It would be inappropriate for the Remote Resource Handler to invoke the callback. In fact, the Remote Resource Handler has no idea what to do with the Event. As currently implemented, no special action is needed. The result will be delivery of the Event to the Client with no special action on the part of the Remote Resource Handler.

The state of Resources exported by value and the metadata of all exported Resources is not synchronized by default. Clients wishing to synchronize exported or imported Resources register for the Core-generated metadata and Resource Events. They also subscribe to the Resource Event if the Core-managed Resource is being exported by value.

Care is needed to avoid cycles. Consider an exported Resource that has its metadata changed on the importing side. Assume that a Client on each Logical Machine has subscribed to metadata Events for this Resource with Core Distributors from both Logical Machines. When one Client makes a change, they both get the Event.

Even if the Client making the change doesn't respond to the change Event, the other Client must make the change on its Logical Machine. This change will generate an Event that will reach the first Client. Not having any knowledge of the source of the Event, the Client will make the change again. These two Clients would continue repeating the same change forever except for the fact that the Core generates a Resource or metadata Event only if the state is actually changed. Hence, the second change on each side does not generate an Event, and the cycle is broken.

Other cycles can occur. Two Clients that make changes to the metadata while the Events are propagating can generate a cycle that is not broken so simply. The problem is that they are both changing the same item without synchronizing. Such conditions are almost certainly programming errors. No action taken by any e-speak component can be guaranteed to break such cycles. Only the Clients have sufficient information to detect the problem.

Chapter 10 Management (Informational)

Architecture

The management of an e-speak-based system consists of three levels.

E-speak Elements Management Level

Complete management of an e-speak-based system requires management of the basic e-speak infrastructure itself. The e-speak element model describes the set of models and elements. The elements have a set of management interfaces that provide basic management access to the e-speak element such as Resources, Cores, Repositories, and so on. This aspect of e-speak system management is referred to as *e-speak element management-level management*.

E-speak Infrastructure

Agents, the basic management components in the management architecture, refer to software entities that collectively provide the management functionality for e-speak. The management system itself is implemented as collection of e-speak services that are e-speak Clients. This design choice enables the management of agents in the same way any other e-speak Client is managed.

Although the goal of system management can be conceptually classified into five categories (Fault, Configuration, Account, Performance, and Security), the actual implementation of the functionality often demands similar mechanisms, such as data acquisition and distribution, policy maintenance and consultation, data processing, and decision making. This implies that it is a good design choice to build a general management infrastructure that provides these mechanisms separately from the specific management goal. The individual management functionality can then be built on top of this management infrastructure with much less effort. This level is referred to as the *e-speak management infrastructure level*.

E-speak provides a design for some simple, automated, rule-based decision mechanisms. Also bear in mind that e-speak management agents will be implemented as e-speak Clients. Thus, some high-level abstractions for inter-Client communication above the current messaging level will be extremely helpful in creating a management environment for e-speak.

E-speak System Model Level

The e-speak infrastructure does not mandate a specific system model. However, to provide FCAPS management, in particular account, security, and configuration management, a system model beyond the current e-speak architecture is required. Therefore, the first step is to create an e-speak system model (ESys) and illustrate how the management infrastructure can be used to achieve FCAPS management over ESys. Developers are welcome to adopt and extend the ESys system model; however, ESys is *not* the only system model that can or should be built on top of e-speak. ESys merely illustrates how to create a complete and manageable system on top of e-speak that leverages the e-speak power.

A system model should define such things as the user model, the Client model, the security model, and a management model.

Core Management

E-speak Cores can be managed at two levels: at the level of Keys, Name Frames, and so on, and at the level of polices and models. Both of these levels may be exported to the management applications directly or through the Protocol Providers. Whichever level of management is exported into the management middleware, system administrators must be able to control the access to the agents who have been given Keys for that purpose.

The alternative to providing a low-level interface is to provide a high-level interface that gives a more abstract view of the Core and has the models and policies that the Core uses built in. Core Controllers using different models for managing the Core provide customized interfaces for those models.

When a Core is created, some base Resources need to be created and their metadata must be initialized to correct values. For example, the Base Vocabulary must have its metadata Permissions set to control who can make changes to it. Services must connect to the Core, and the management station must validate that these services are in fact authentic.

The management interface to the Core includes functions to create, delete, and modify local Core Resources and their metadata. The access to the low-level management interface must be carefully controlled by the use of Keys and Protection Domains.

System Structure

The management middleware is made up of a set of communicating agents. The agents run on the e-speak platform and use its messaging when communicating. Agents come in four basic classes, as shown in Table 18.

Table 18 Management agents

Class of Agent	**Agent Function**
Core Controllers	Receives commands from other agents and translate them into actions on the Core
General Agents	Provides fault, account, configuration, performance, and security management
Protocol Adapters	Mediates between other agents and a management system using a management protocol
Event Distributors	Passes Events from Publishers to Subscribers

The management middleware layer provides the architecture with:

- Support for many different types of management protocols, whatever is most suitable to the system manager
- Support for management systems that perform the majority of the system management tasks
- Support for management systems that perform only basic display functionality, and leave the rest to the middleware agents
- Support for systems managed by one type of management system at the local level and another at the domain level

- A guaranteed level of security concerning management information and access to control functions
- A means to split up the function of management into well-formed objects with clean interfaces

Agents communicate to provide the management functionality. Every agent uses a common Interface Definition Language (IDL) to define the interfaces that it exports and also implements a common agent management interface to enable management of the management system itself. Agents are implemented as e-speak services and as such will be registered in the Repository and communicate with each other using e-speak messaging.

Core Controllers

Core Controllers are the lowest level of the management middleware and operate locally on the Core. Core Controllers can be low level or high level or both. If they are low-level controllers, they do some basic initialization before contacting the designated management agent that will configure the Core. If they are high-level controllers, they are able to do all or most of the configuration unaided. Core Controllers have no particular status within the Core except that they may have unusually high privileges.

Event Distributors

Each Core has a Core Event Distributor that handles Events generated by the local Core. These local Event Distributors form the lowest level of the Event distribution graph that exists in the management middleware. The Event Distributors use polices (supplied by a high-level agent) to determine which (if any) of the Events are propagated to other distributors.

Protocol Providers

E-speak management agents, when implemented as e-speak Clients, will communicate with each other using the IDL syntax. To interface with existing management systems that use different Protocol Provider agents, Protocol Provider agents will translate IDL method calls from and into the corresponding management protocols. A Management Information Base (MIB) for the management agent or element being managed must be defined for each protocol that needs to be supported. Protocol Provider agents also must translate the security semantics between e-speak and the security supported by the protocol (if any exists).

Security Models

Rather than the management system creating Keys and Protections Domains and setting metadata fields directly, the Core Controller provides an interface that has high-level methods for creating new users or services. Because the Core Controller now understands the security model, it can enforce the model over all Resources that are registered with the Core.

Rather than storing the policies that decide how metadata is set in the management system and setting the metadata remotely, the management system loads the correct Core Controller onto the system and tweaks the polices through a high-level interface. The Core Controller can now operate locally according to central management policy.

A small number of high-level models cover the majority of use modes for the e-speak Logical Machine. The high-level interfaces to the Core either could be implemented locally in the Core Controller or could be implemented as management agents that use the low-level interface provided by the Core Controller.

Chapter 11 Distributed Management (Informational)

This chapter describes e-speak management for a collection of Logical Machines, including the system models and distributed agent infrastructure.

Management Middleware

The management middleware layer illustrated in Figure 15 uses the Core Controller and Event Distributor on each Logical Machine to feed information to various distributed agents that collect, summarize, and forward information to other agents. Eventually, the data reaches a *Protocol Provider* that does the translation to the form needed by the management system.

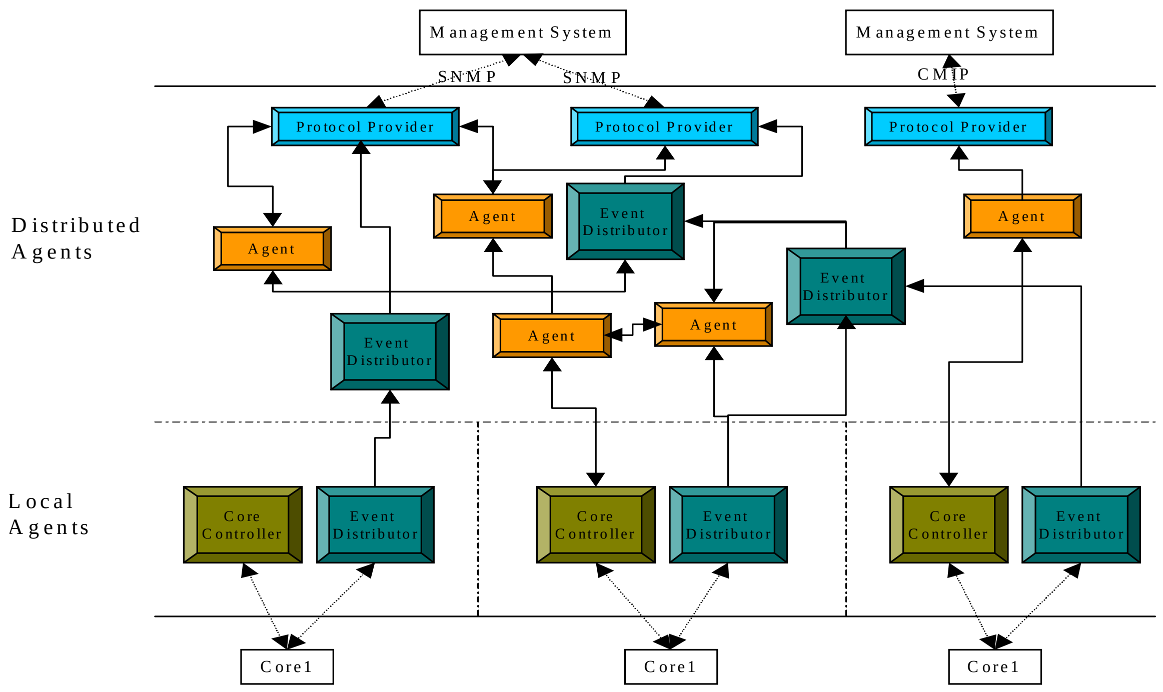

Figure 15 Connecting local management into a system

The management middleware layer provides the architecture with a number of benefits. In particular, it can support:

- Many different types of management protocol, whatever is most suitable to the system manager
- Management systems that perform the majority of the system management tasks
- Management systems that perform only basic display functionality and leave the rest to various middleware agents
- Systems that are managed by one type of management system at the local level and another at the domain level

This layer provides a guaranteed level of security for management information and access to control functions. It also splits up the function of management into well-formed objects with clean interfaces.

Architectural Model

Managing a collection of Logical Machines is based on the same agent structure used to manage a single machine. In a distributed environment, the e-speak management architecture has three dimensions:

1. Structural model
2. Domain model
3. Service model

Each dimension deals with a different aspect of management.

Structural Model

Management applications need not directly implement e-speak messaging but can use existing management protocols, such as CMIP or SNMP, to talk to the e-speak management middleware, as shown in Figure 16.

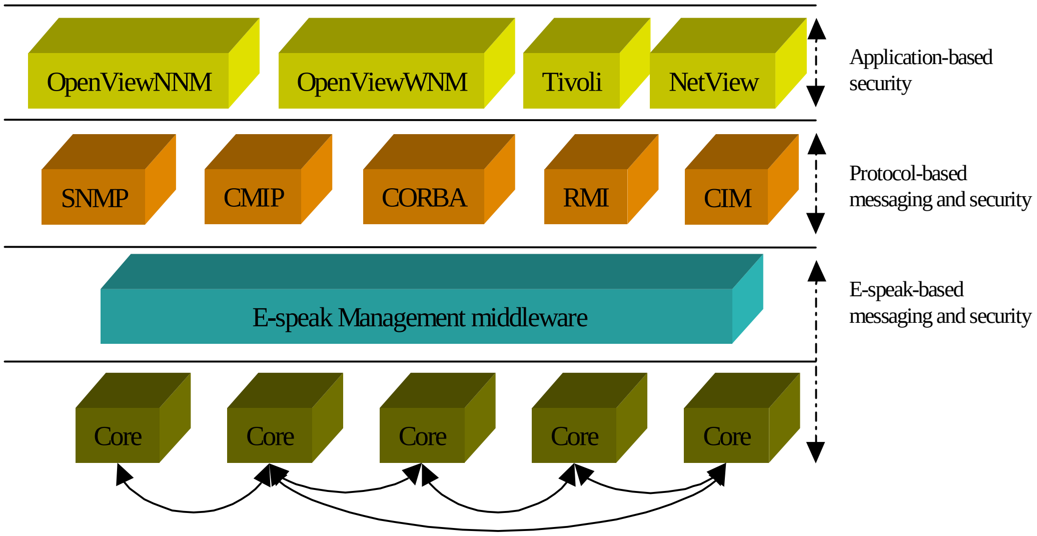
OpenViewNNM
OpenViewWNM
Tivoli
NetView
Application-based security
SNMP
CMIP
CORBA
RMI
CIM
Protocol-based messaging and security
E-speak Management middleware
E-speak-based messaging and security
Core
Core
Core
Core
Core

Figure 16 Structural model of management system

The management middleware converts from the standard protocols to e-speak messaging, providing a higher-level model than just basic Core internals.

Domain Model

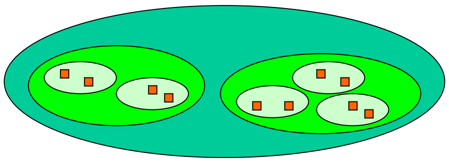

Figure 17 E-speak domain model

E-speak can be used in a variety of circumstances, from the home to a corporate office. Each Core can be seen to belong to a domain of control (Figure 17), which has the responsibility of managing that Core.

Domains of control can be hierarchically structured; for example, a workstation in a department may have its own local management application with responsibility over just that machine, while also being part of the department domain and being managed by a department management station. E-speak access control is used to grant management rights to the Clients exercising this control.

Service Model

Services inherently cross organizational boundaries, and therefore domains of control. A service may be exported by one company, resold by another, and finally used inside an enterprise's firewall.

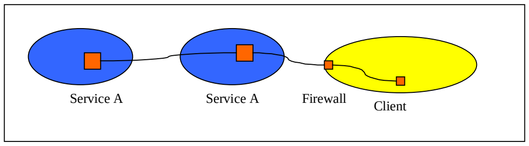

Figure 18 E-speak service model

To manage such a service requires a management system that can pass information between domains of control and therefore different management systems (Figure 18). If the user of the service encounters difficulties, then resolving the problem requires the coordination of several management systems. E-speak management uses ESIP, the e-speak Service Interchange Protocol, to pass control information around the system. The management middleware collects information from a variety of sources and presents it to the management system.

Chapter 12 Repository (Informational)

The Repository is not part of the e-speak architecture because Clients have no direct interaction with it. However, understanding the operation of the Repository helps in understanding other parts of the architecture. Also, the behavior of the system depends on how the Repository is configured. This chapter describes the reference implementation, the Core-Repository interfaces for including Repositories of different internal structures, and various scalability issues.

Repository Overview

The Repository holds the data needed by the Core. This data includes the Resource metadata as well as the internal state of Core-managed Resources. The Repository is also read by the Lookup Service when a Client requests a lookup. These two operations have different design points. Access to metadata and Core-managed Resources is done frequently and needs to be low latency. Lookup requests are akin to database queries; they are less latency sensitive but must be completed relatively quickly.

Repository Structure

To support the conflicting goals of flexible query lookup on a large persistent set and rapid access to a smaller, transient subset, the reference implementation of the Repository described here is divided into two components: the *Repository Database* and the *Repository Access Table.*

The Repository Database provides persistent storage and efficient lookup request processing. This component is left parameterized in the Core-Repository interface. All that is needed is an appropriate database interface. This design allows different implementations of the Repository to select the most appropriate database based on relevant business and technical considerations.

A very broad range of persistent repository implementation is allowed. This Repository Database interface gives another architectural degree of freedom. For instance, in the case of a battery-backed RAM device or in situations where persistence is simply not a requirement, a pure RAM-based Repository Database implementation is feasible. Thus, the Repository Database need not have a large footprint.

The second component, the Repository Access Table, is fully resident in memory in the reference implementation. This access table is rebuilt from data in the Repository Database as part of a system restart. The access table supports a fast associative lookup of information based on Repository Handles. It may be a cache of the Repository data, or it might be large enough to hold all the data needed for Resource access.

Information Flow

Every e-speak installation comes with an in-memory Repository that does not support persistence. To add the feature of scalability, a *glue* layer must be provided to convert Core requests to the Repository into meaningful requests to the selected implementation. This glue layer must implement the information flow methods described in this section. In addition, the glue layer can also include interfaces specific to the selected Repository implementation, such as setting controls.

The Repository Database has two interfaces used by the Core. The Core-Repository interfaces have methods to:

- Register and unregister Resources
- Access the metadata corresponding to a given Repository Handle

- Modify the metadata corresponding to a given Repository Handle
- Look up Resources that match a Search Recipe

The Client can access these methods only indirectly by invoking methods in the Contract, Name Frame, and MetaResource:

```
public RepositoryHandle registerDescription(
     String                name,
     ResourceDescription   d,
     ResourceSpecification s)
  throws InvalidSpecificationException;

public void unregisterDescription(
     RepositoryHandle handle)
  throws StaleHandleException;

public ResourceDescription accessDescription (
     RepositoryHandle handle)
  throws StaleHandleException;

public ResourceSpecification accessSpec (
     RepositoryHandle handle)
  throws StaleHandleException;

public RepositoryHandle mutateDescription (
     RepositoryHandle      handle,
     ResourceDescription   d,
     ResourceSpecification s)
  throws StaleHandleException;
```

The second interface is presented to the Core by the Repository to invoke the Lookup Service for a Repository lookup request. This interface is invoked when the Client does a lookup in a Name Frame:

```
public RepositoryHandle[] find (SearchRecipe recipe)
   throws InvalidSearchRequestException;
```

The Repository may access permanent storage, but the protocol used for such access is not part of the e-speak architecture.

Increasing Scalability

Because a Resource can be used only if it has been registered in the local Repository, it is important to consider the eventuality of a full Repository. Two kinds of e-speak Repositories are based on deployment needs: a *thin Repository* and a *fat Repository*.

A thin Repository does not have enough disk space to grow with the number of Repository entries. Its purpose is to support Repository Handle-based access, with latency on the order of microseconds. This support is provided on a smaller, transient, subset of Repository entries, which corresponds to "in-use" Resources. A thin Repository is very sensitive to stale data; it must enforce strong policies to:

- Dispose of stale entries, and
- Prevent marginally accessed entries from accumulating.

A thin Repository may have no persistent storage of its own. Thus, because the number of Repository entries that can be stored in a thin Repository is small, an *in-memory* Repository implementation is appropriate.

A fat Repository has a lot of disk space and may act as a server to a thin Repository. Clearly, such a Repository may be highly available. The primary purpose of such a Repository is to support Resource lookup requests with "reasonable" latency (on the order of milliseconds). A fat Repository is not very sensitive to stale data. Because the number of Repository entries that can be stored in a fat Repository is very large, Repository implementation based on a database is appropriate.

A thin Repository may use a fat Repository to fulfill its scalability needs, and a fat Repository may simultaneously serve many thin Repositories. However, many devices may not need such support because their transient state can hold all the information necessary.

The communication between a fat Repository that provides services to a thin Repository is not part of the e-speak architecture. However, because the security of the system depends on the integrity of this communication, the link must be protected. It is the security of the communication link that makes the Repository part of the Core, irrespective of the physical machine that holds the Repository.

Chapter 13 Intramachine Security (Informational)

This chapter describes the security mechanisms for intramachine communication. The e-speak security mechanisms are currently under review; it is expected that they will be replaced by a cryptographic security architecture.

Overview of Intramachine Security

E-speak security is provided by the Core-managed Resources and the metadata associated with each Resource. In particular, the security mechanisms are built using:

- Keys—The capabilities that provide for both coarse and fine-grain access control
- Key Rings—The containers for Keys
- Protection Domains—Encapsulate the Client's view of the system

Assumptions

The e-speak Core normally runs as a user application on top of a native operating system. The Core and several of its Clients use configuration data normally stored in files. The Repository is usually a database with its state kept on disks controlled by the operating system. In fact, the executable images of the Core and Clients most likely reside on these same disks.

The ultimate security of the system is no stronger than the security provided by the underlying operating system. If an unauthorized user can modify any of these files, the security is compromised. The operating system files must be protected by the most robust means available. This protection may include:

- Setting operating system Permissions to limit access
- Securely storing hash codes to prevent tampering
- Encrypting the files to prevent leaking information

It is assumed that there is separation of address spaces used by the Core and each of its Clients. These Clients include external Resource Handlers (for non-Core-managed Resources). No Client may directly access any of the Core's address spaces, and two different Clients will generally not be able to access each other, unless they have mutually agreed to do so (such as by using shared memory). Hence, the only means of interaction between a Client and a Core is through the Application Programming Interface (API) exposed by the Core-managed Resources or the abstraction of these APIs presented by the e-speak programming libraries.

E-speak provides no protection against attacks mounted against the native operating system. Buffer overflow attacks against the TCP stack (or any other underlying protocol stack), guessing administrator passwords, social engineering, and the like are all beyond the control of the e-speak security infrastructure.

Keys, Permissions, and Locks

E-speak Keys, as distinct from encryption keys, are capabilities: They confer the privilege to perform some action on a Resource or set of Resources. Keys are Core-managed Resources whose internal structure is not visible to Clients.

Locks are objects used internally by the Core to protect Resources. Each Key matches only one Lock, but many Keys may match a given Lock. A Key that matches a Lock is said to *unlock* or *open* that Lock. A Key may be cloned to produce a new Key opening the same Lock. A cloned Key can be removed from the system, revoking the privileges of every Client who has that Key, without affecting others who have a different clone.

Permissions are strings stored in the Repository. Permissions are protected by Locks. E-speak defines two kinds of Permissions: metadata Permissions and Resource Permissions. Metadata Permissions control access to metadata associated with a Resource. Resource Permissions control access to operations on the Resource. Resource Permissions are interpreted by the Resource Handler, which is the Core for Core-managed Resources. Metadata Permissions are always interpreted by the Core. Permissions provide fine-grain access control.

Whenever a Client sends a message to a Resource, zero or more Keys are attached to that message. The Core searches the Repository entry for the Resource to determine which Locks the Keys open. Any Permission strings protected by an opened Lock are sent to the Resource Handler. The Resource Handler is free to interpret these strings any way it chooses to provide access control for Resource operations.

The Resource Handler is relied on to interpret the Permission correctly and enforce the fine-grain access control. By controlling what Keys a Client can present, e-speak can control what actions that Client can perform on a Resource. By controlling the Keys associated with the metadata, e-speak can control what actions a Client can take on the Repository entry.

Resource metadata is managed by the Core, so e-speak provides a MetaResource that acts as the Resource Handler for metadata. Clients send a message to the MetaResource naming the Resource whose metadata is to be accessed. For example:

```
ResourceDescription desc = metaResource.getDescription(file);
```

When a message is sent to the MetaResource, the metadata Permissions of the referenced Resource are unlocked by the requester’s Keys. If the correct Permission is unlocked, the MetaResource will perform the requested operation.

Key Rings and the Mandatory Key Ring

Key Rings are containers for Keys. Clients can choose which Key Rings will be sent with a message. However, a special Key Ring, called the *Mandatory Key Ring*, is specified in the Client's Protection Domain and is always attached by the Core to each message sent by the Client. All other Key Rings are named by the Client as any other Resource would be. Note that, in general, the Client has a name bound to its Mandatory Key Ring, which means it can add and remove Keys from its Mandatory Key Ring.

Because Keys and Key Rings are Resources, Clients can pass names for them to other Clients just as they can for any other Resource. This provides the mechanism for distributing Keys and Key Rings.

Visibility Tests

The metadata for each Resource has both an Allow and a Deny field. Each field contains a list (which may be empty) of Locks. Often, the Allow and Deny fields together are referred to as the *Visibility fields*.

Each time the Core receives a message for a Resource, it performs tests on the Allow and Deny fields of every Repository Handle it encounters in processing the request. These tests are referred to as *visibility tests*.

First the Core tests if any Key sent with the message matches a Lock in the Deny field. If a match is found, the Core acts as if the Repository Handle does not exist.

Second, the Core tests if any Key sent with the message matches a Lock in the Allow field. If none of the Keys matches, the Core acts as if the Repository Handle does not exist. This test is bypassed if the Allow field is empty.

The Visibility fields enable implementation of access control enforced by the Core. Permissions enable implementation of access control enforced by Resource Handlers.

Protection Domains

A Protection Domain contains the Client's Mandatory Key Ring, its default Name Frame, and other information the Core needs in managing the Client. A Client is only able to access Resources reachable from the default Name Frame. Note that this is not the same as the set of Resources currently in that Name Frame because it can contain name bindings associated with other Name Frames, and a Client can use a lookup to discover and add more Resources to a Name Frame.

From a security viewpoint, the crucial function of the Protection Domain is that it contains the Client's Mandatory Key Ring, which is attached to all outgoing messages. This Key Ring contains Keys that match the Visibility fields of the target Resource or unlock certain Permissions. Thus, even if the Client is able to reach the Resource from its Name Frame, the Visibility fields will determine whether it can actually send a message to the Resource. In addition, other Keys sent will cause Permissions to be sent to the Resource Handler, which determines precisely what actions the Client can take on the Resource.

Example of an Intramachine Message

An example of the how the security mechanisms are used when a Client sends a message to a Resource on the same Core is provided in Figure 19.

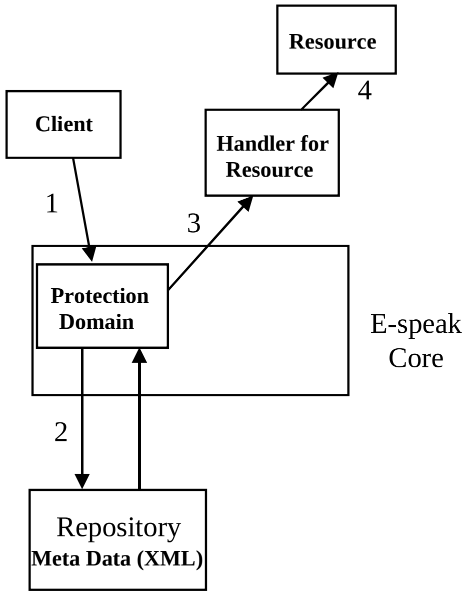

Figure 19 Sending a message to a Resource on the same machine

The processing occurs as follows:

1 A Client sends a message to its Core, naming the target Resource and attaching (i.e., naming) any optional Key Rings.

2 The Core attaches the Mandatory Key Ring. Then the Core retrieves the metadata for the Resource from the Repository. The visibility checks are made using the Allow and Deny fields. If these checks do not cause the message to be rejected, the Core attempts to match all the Keys on each of the Key Rings to any Locks for the Resource Permissions.

3 The Core sends the message to the Resource Handler, together with any Permissions that have been unlocked. The Resource Handler interprets the Permissions to see if the requested action is allowed.

4 The Resource Handler sends the appropriate message to the Resource (if the Resource is an active entity, such as a process) or executes the appropriate action (in the case of a passive Resource such as a file).

Key Management

Key management is crucial for e-speak. This requires that the Client not be able to discover the Key names by using attribute-based lookup. The best way to do this is to make sure that Keys do not have any attributes that could be used to discover them. Another is to use an *essential* attribute that acts as a password.

A Client can add and remove Keys from Key Rings, but only if it has a name bound to the Key available through its Protection Domain and if it presents a Key that opens the appropriate metadata Lock for permission to modify the Key Ring.

A Key is a Core-managed Resource, and it can be removed from the system just like any other Resource. If this happens, any Client in possession of a name for the Key (or with the Key attached to one of its Key Rings) will lose the privileges conferred by that Key. This mechanism is a convenient way of managing privileges. Many Keys that all match the same Lock can be cloned; removing a Key affects only those Clients that have that Key—it does not affect other Clients with a Key cloned for the same Lock.

Abstract Security Models

This chapter describes the basic mechanisms for security in e-speak. The Key management tools under development will manage and use these mechanisms to deliver a more abstract, though familiar, model to system administrators and users.

Chapter 14 Intermachine Security (Informational)

This chapter describes how security is provided for intermachine communication when the Client and the Resource are on different Logical Machines. The e-speak security mechanisms are currently under review.

Overview of Intermachine Security

Once two machines have been connected, the only entities aware of distribution are the Remote Resource Handlers. As far as the Client, Resource Handler, and Core are concerned, all communication appears to be taking place with an entity on the same Logical Machine, as shown in Figure 20.

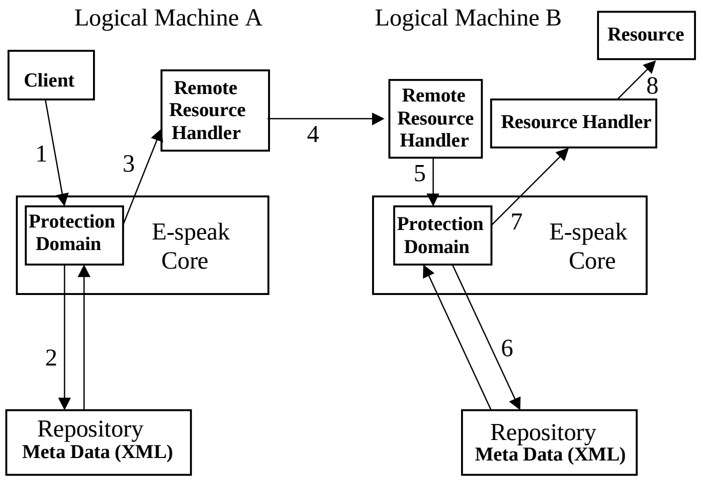

Figure 20 Communication between two Logical Machines

In Figure 20, both the Client on Logical Machine A and the Remote Resource Handler on Logical Machine B are Clients of their respective Cores. Each will have a Mandatory Key Ring (contained in its Protection Domain) that will be attached by the Core to any message sent. Each will hold other Key Rings, which optionally they may choose to attach to messages they send.

The Protection Domain of the Remote Resource Handler on Logical Machine B fixes the set of Resources visible to any Client on Logical Machine A. Its Mandatory Key Ring will have visibility Keys (Keys that match Locks in Allow and Deny fields of target Resource metadata). Logical Machine B's Core will allow its Remote Resource Handler to send messages to a Resource only if it has a Key that matches an Allow Lock and no Keys that match a Deny Lock. In addition, the Remote Resource Handler is unable to send a message to any Resource that is not reachable through the default Name Frame contained in its Protection Domain; it can send messages only to Resources for which it has a name.

Controlling What Can Be Exported

The Protection Domain for the Remote Resource Handler controls what Resources can be accessed on behalf of remote Clients. The problem is controlling what can be added to the Protection Domain. For example, a remote Client might do a lookup request that results in a binding for a Resource that should be available only to local Clients. Also, the Remote Resource Handler may receive a message from a local Client that transfers names for such Resources.

The solution in e-speak is to use the Visibility fields. Each Resource that is to be made available to local Clients has a Lock in its Allow field that is denoted *local*. The Mandatory Key Ring in the Protection Domain set up for all other Logical Machines will not have the corresponding Key, but all other Clients will. Hence, the Remote Resource Handlers cannot see such nonexportable Resources. A Resource that should be exportable to any Logical Machine will have an *exportable* Lock in its Allow field. Every Remote Resource Handler will be given its Key.

Selective export of Resources can be controlled by adding other Keys to the Allow field and putting them on the proper Mandatory Key Rings. For example, a Resource that is to be exported only to Logical Machine A would have a Lock called *export A* in its Allow field in addition to the *local* Lock. The Protection Domain that embodies the policy for Logical Machine A will have a Mandatory Key Ring containing the corresponding Key. Note that a Resource can be made available to remote Clients and hidden from local Clients by omitting the *local* Lock from the Allow field.

Example of an Intermachine Message

When the Client on Logical Machine A sends a message to its Core for the Resource on Logical Machine B, the following steps take place (see Figure 20):

1. The Client attaches any optional Key Rings it chooses and sends a message to the Resource, which is routed by Core A.
2. Core A attaches the Mandatory Key Ring and retrieves the metadata for the Resource from the Repository. The visibility checks are made (Allow and Deny fields). If these checks do not cause the message to be rejected, the Core at-

tempts to match all the Keys on each of the Key Rings to any Locks for the Resource Permissions. The metadata indicates that Remote Resource Handler A is the handler for this Resource.

3 Core A sends the message to Remote Resource Handler A, together with any Permission strings that have been unlocked. Remote Resource Handler A forwards these unlocked Permissions as the names of Keys.

4 Remote Resource Handler A sends the message and unlocked Permissions to Remote Resource Handler B.

5 Remote Resource Handler B attaches the designated Keys to one of its Key Rings and sends a message for the named Resource to Core B.

6 Core B attaches the Mandatory Key Ring contained in the Protection Domain of Remote Resource Handler B and retrieves the metadata for the Resource from the Repository. The visibility checks are made (Allow and Deny fields). If this does not cause the message to be rejected, the Core attempts to match all the Keys on each of the Key Rings to any Locks for the Resource Permissions.

7 Core B sends the message to the Resource Handler, together with any Permission strings that have been unlocked. The Resource Handler interprets the Permissions to see if the requested action is allowed.

8 The Resource Handler sends the appropriate message to the Resource (if the Resource is an active entity such as a process) or executes the appropriate action (in the case of a passive Resource such as a file).

The security mechanisms check that the message is allowed to be sent to Remote Resource Handler A and the message is allowed to be sent to the Resource Handler on Logical Machine B. The first of these checks enables the administrator for Core A to control which Clients are allowed to access which Resources on Logical Machine B. Typically, these checks will involve Keys that have been created and managed on Core A. For example, the administrator might place a Lock in the Allow field of the metadata for Remote Resource Handler A and give a Key for that Lock only to certain trusted Clients. Hence, only these trusted Clients will be able to send messages to, and access Resources on, Logical Machine B.

The second of these checks enables the administrator of Core B to control what is accessible to Core A (and hence all Clients on Core A). Typically, certain Allow and Deny Keys would be attached to the Mandatory Key Ring of Remote Resource Handler B to fix the set of accessible Resources. For example, suppose all the Resources on Core B have a single Lock in their Allow field and are divided into two mutually exclusive sets: either Key K1 or Key K2 unlocks the Allow Lock (but not both). If K1, but not K2, is placed on the Mandatory Key Ring of Remote Resource Handler B, Clients on Core A will be able to access only Resources on Core B that have their Allow Lock matching K1.

In addition, some Locks associated with Permissions might be exported from Core B, while others might be withheld. For example, a file resource might have read and write Permissions, but only the read Permission is exported. If the Key to unlock the read Permission is placed on the Mandatory Key Ring of Remote Resource Handler B by the administrator for Logical Machine B, then every message sent from Logical Machine A will have that Key attached to it. Alternatively, it might not be placed on the Mandatory Key Ring of Remote Resource Handler B or on any of its optional Key Rings. In this case, the Key must be on one of the Key Rings attached by the Client on Core A, if that Client wants to execute a read request. (The administrator of Logical Machine A will have to give the Key to the Client.)

Exporting a Key

Keys and Key Rings can be accessed like any other Resource because they are Core-managed Resources. If they have been exported, they can be accessed from Clients on different Logical Machines. Note, however, that the export process for Keys is slightly different than for other Resources. When a Key is imported, the Remote Resource Handler creates a new Key and binds the export name for the Key to it. This name is bound as a Permission in the Resource metadata that has a reference to this Key.

Secure Local and Remote Access (Informational)

This section presents a scenario to illustrate how the security mechanisms are used locally and remotely and what happens when a Resource is exported.

Secure Local Resource Access (Informational)

An example of secure local Resource access is provided in Figure 21. Logical Machine A has a Resource, a list. A Client CA on A has the name for this list as `MY-LIST`. `MY-LIST` is bound to a Repository Handle, RH, in a Mapping Object, `A-LIST-RH`.

CA has a Key that unlocks the `ADD-MEMBER` Permission for list `A-LIST-RH`. This Key, which CA calls `CA-CAN-ADD`, is also a Resource, which CA's Core identifies as `A-KEY-RH`. The internal state of the Key `A-KEY-RH` is some piece of data in CA's Core, for example, the number 634-5789.

CA can add a member to the list by sending a message to its Core, with the payload `<ADD,NEW-MEMBER>`, naming `MY-LIST` as the primary Resource and making sure that `CA-CAN-ADD` is on its Key Ring. CA's Core translates these names to `A-LIST-RH` and `A-KEY-RH`, respectively. CA's Core uses the Key state (634-5789) to verify that the Key `A-KEY-RH` unlocks the Permission `ADD-MEMBER` for the list Resource A-LIST-RH. A's Core then forwards the request to the list Resource Handler, which verifies that the `ADD-MEMBER` Permission was extracted and that `ADD` was requested. The new member is placed in the list.

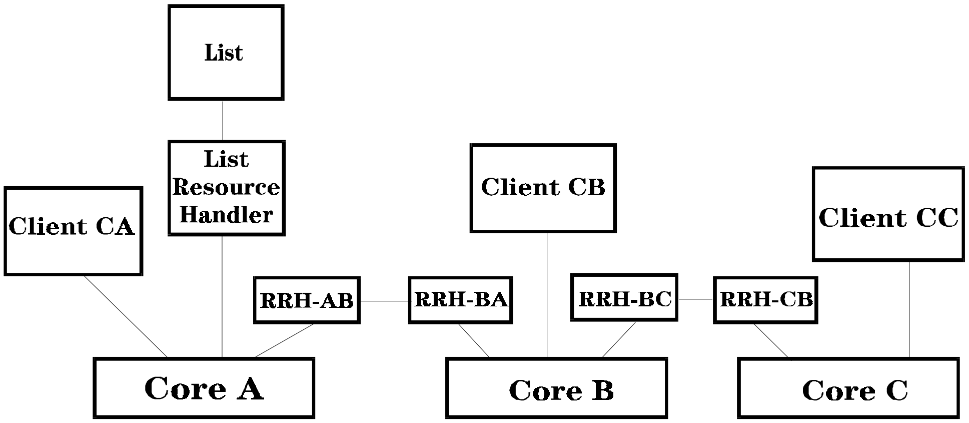

Figure 21 Accessing the list Resource

Exporting the Resource

Logical Machine A and Logical Machine B establish a connection using the e-speak intermachine protocol. The connection establishes RRH-AB running on A and RRH-BA running on B. A exports the list to B and gives B permission to add new members to it. To do this, A first exports the Key `A-KEY-RH` to B as follows.

A's Core creates an Export Form for A-KEY-RH. On A, the Core and RRH-AB agree on the shared name `ADD-KEY` for this Resource. RRH-AB now sends this Export Form over the wire to RRH-BA on B. The two Remote Resource Handlers use the shared name `ADD-KEY` for this Resource.

Now, on B, RRH-BA registers the Resource `ADD-KEY`. This creates a Key Resource on B that the Core identifies by a Repository Handle: B-KEY-RH. B-KEY-RH, being a Key, has some internal state: 857-1501. (This internal state has nothing to do with the internal state of A-KEY-RH in A's Core.)

NOTE: This example uses the same name ADD-KEY in three ways:

- As the shared name between A's Core and RRH-AB
- As the shared name between RRH-AB and RRH-BA
- As the shared name between RRH-BA and B's Core

One or both Remote Resource Handlers could keep name mapping tables; in this case, the names would all be different.

Once A-KEY-RH has been exported, A's Core exports the list A-LIST-RH with ADD-MEMBER Permission granted by Key A-KEY-RH to B as follows:

- A's Core creates an Export Form for A-LIST-RH. On A, A's Core and RRH-AB agree on the shared name CUSTOMER-LIST for this Resource. In the Resource metadata for A-LIST-RH is a field containing Permission (A-KEY-RH, "ADD-MEMBER"). The corresponding field in the Export Form for CUSTOMER-LIST appears as ADD-KEY.
- RRH-AB now sends the Export Form for CUSTOMER-LIST over the wire to RRH-BA on B's machine. The two proxies use the shared name CUSTOMER-LIST for this Resource. If we assume the Resource is being passed by reference, the Export Form consists of all the relevant fields in the Repository entry identified by A-LIST-RH.
- Now, on B's machine, RRH-BA registers the Resource CUSTOMER-LIST. This creates a Resource on B's machine that B's Core identifies by a Repository Handle: B-LIST-RH. When RRH-BA converts the Export Form to a register form, the ADD-KEY Permission field becomes ADD-KEY, ADD-KEY.
- When the Resource gets registered, Core B replaces the name ADD-KEY with the corresponding Repository Handle, so the Repository holds B-KEY-RH, "ADD-KEY".
- RRH-BA adds a field to its Resource-specific data for this Resource indicating that its name for this Resource is CUSTOMER-LIST.

Secure Remote Resource Access

This part of the example provides the description for remote Resource access in a secure environment. A Client CB on B, by some means, has obtained name bindings for the list and its Key; `PASSENGER-LIST` and `CB-CAN-ADD`, respectively, in CB's Name Frame. CB adds a new member to the list by sending a message to its Core with payload `<ADD,NEW-MEMBER>`, naming `PASSENGER-LST` as the primary `RESOURCE` and making sure that the Key CB-CAN-ADD is on his Key Ring.

B's Core translates these names to B-LIST-RH and B-KEY-RH. B's Core uses the Key state (857-1501) to verify that the Key B-KEY-RH unlocks the Permission "`ADD-KEY`" for list Resource B-LIST-RH. B's Core now sees that B-LIST-RH is managed by RRH-BA, so it sends a message to RRH-BA, naming `CUSTOMER-LIST` (translation for B-LIST-RH) as the primary Resource, with payload `<ADD, NEW-MEMBER>`, making sure that the `ADD-KEY` Permission (unlocked by B-KEY-RH) is in the message.

RRH-BA now sends this across the wire to RRH-AB on A. On Logical Machine A, RRH-AB translates this into a message sent to A's Core, naming `CUSTOMER-LIST` (translation for `CUSTOMER-LIST`) as the primary Resource, with payload <ADD, NEW-MEMBER>, making sure that ADD-KEY is on the Key Ring. A's Core translates these names into A-LIST-RH and A-KEY-RH.

Just as before, A's Core uses the Key state (634-5789) to verify that the Key A-KEY-RH unlocks the Permission "`ADD-MEMBER`" for the list Resource A-LIST-RH, and forwards this request to the list handler so the new member is placed in the list.

Note that the Permissions were checked twice: once on Logical Machine B to get permission for "`ADD-KEY`" and once on Logical Machine A to get permission for "`ADD-MEMBER`".

Exporting an Imported Resource

A Logical Machine that imports a Resource may need to export the same Resource to another Logical Machine. Suppose Logical Machine B needs to export the Resource to Logical Machine C. To do this, B's Core creates Export Forms for B-KEY-RH and B-LIST-RH. These are shared with RRH-CB running on B's machine under some agreed-upon names (e.g., `C-KEY` and `C-LIST`). These are transmitted to RRH-CB running on Logical Machine C and registered on C as C-KEY-RH and C-LIST-RH.

A Client CC on C adds a new member to the list just as before. A request to invoke `ADD-MEMBER` on C-LIST-RH with Key C-KEY-RH gets forwarded via the Remote Resource Handlers and reduces it to a request to add a member to B-LIST-RH with Key B-KEY-RH. The Permissions here get checked three times: once on C's machine to see that C-KEY-RH unlocks Permission "C-KEY" before transmitting the request to B's machine, once on B's machine, and once on A's machine.

Observe that the Client on C doesn't know anything about Logical Machine A. When this Client accesses the list, Logical Machine B is providing the service as far as it is concerned. Nor does Logical Machine A know anything about Logical Machine C. When the Client CC adds a new member to this list, Logical Machine B is making the request as far as A is concerned.

Chapter 15 Future Developments

This chapter describes changes that will be made to the e-speak architecture in future releases:

- A major revision to the security model
- Simplification of Resource import and export using ESUIDs

Security

Keys and locks as described in this version of the e-speak architecture specification will be removed and replaced by an attribute certificate based security model based on the Simple Public Key Infrastructure (SPKI) being developed within the Internet Engineering Task Force (see http://www.ietf.org/). The essential difference between this model and a more conventional use of certificates (e.g., X.509 as used in Secure Sockets Layer), is that the attribute certificates are used to make arbitrary statements about entities in an e-speak system. In an X.509 system certificates are conventionally used to authenticate identity.

For example an identity certificate may state that the public Key in the given certificate is to be associated with the identity “Joe Doe.” Anybody who can demonstrate knowledge of the private Key corresponding to the public Key is deemed to be Joe Doe. Attribute certificates are used to make statements about attributes associated with Joe Doe: “Joe Doe is an HP employee,” “Joe Doe’s office location is in Cupertino, California,” “Joe Doe is a network administrator.”

An attribute certificate is issued by an Attribute Certificate Issuer (ACI) and is issued to a particular entity associated with public Key (in the simplest case this would be an individual (e.g., John Doe), but it could be a group). Not all ACIs need to be trusted equally, for example one ACI might be trusted to issue attribute certificates asserting the named individual is an Hewlett-Packard employee, a different ACI might be trusted to issue attribute certificates asserting the named individual is a citizen of the United States.

Suppose John Doe needs to perform some action on a Resource that requires him to prove that he is a network administrator. To do this he would have to present his identity certificate and his network administrator certificate. The e-speak certificate processor processes the certificates to verify that he has the correct attributes. This involves verifying that John Doe knows the appropriate private Key (associated with his identity), checking the certificates for validity and checking that the ACIs are trusted to issue certificates asserting the individual is a network administrator.

Many aspects of e-speak security will not change: there will still be Resource permissions defined in the metadata (as now); there will still be an “allow” visibility test (no deny); the Core will still mediate access (check visibility and permissions).

An example that also illustrates some additional features are shown in Figure 22.

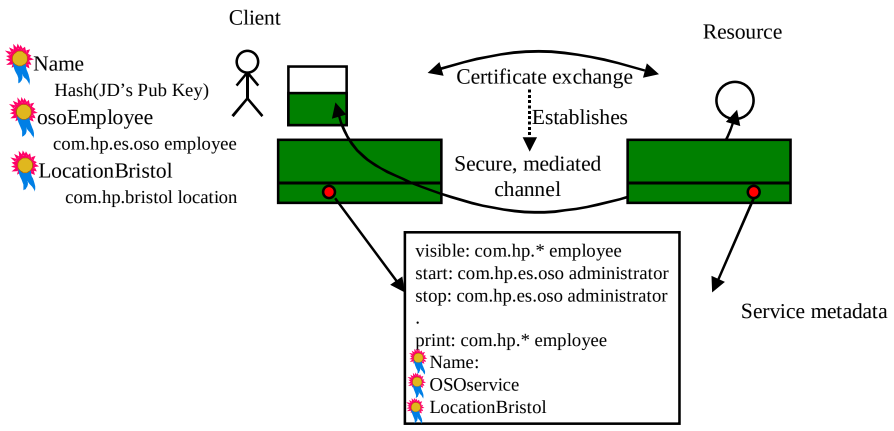

Figure 22 Revised e-speak security model

The Client in Figure 22 has three certificates: an identity certificate, an employee certificate and a location certificate. The employee and location certificate illustrate the structured name space defined for attributes: an employee of the HP, e-speak, Open Services Operation (com.hp.es.oso employee); located at HP's Bristol (UK) site (com.hp.bristol location). An HP employee is denoted in the printer Resource's metadata as "com.hp.* employee". (The "*" is a wildcard character.)

The Resource metadata has three attribute certificates (for brevity the attributes are not shown): a name certificate; an owning entity certificate (Open Service Operation) and a location certificate. This allows Clients to authenticate Resource attributes as well as allowing Resource to authenticate Client attributes. The model also allows Cores to have attribute certificates so that Clients, Resources, and Cores can authenticate each other's attributes.

The Resource metadata shows that visibility of the Resource and the ability to invoke the print operation will be granted to anybody who can authenticate the HP employee attribute. However, the ability to invoke the start or stop operations will only be granted to somebody who can authenticate the HP, e-speak, Open Services Operation, administrator attribute.

Note that there is a strong analogy between attributes and roles and that is how we anticipate this security model to be used: Clients and Resources will be issued certificates denoting their roles. Clients and Resources can change roles without changing their public Key (identity) by being issued new attribute certificates. This reflects the way in which we expect e-services to be built: roles changing frequently, with identities being relatively invariant.

Before communication between the Client and Resource takes place, they must agree a session Key to establish a secure session. To do this, they engage in a Key exchange protocol in which certificates are exchanged and authenticated. The session Key is disclosed to the Client's Core and the Resource's Core. The Client and Resource never disclose their private Keys to anybody.

Delegation is supported explicitly. ACIs can issue attribute certificates allowing the subject of the certificate to issue a delegate certificate. For example, if John Doe also has a certificate stating he is a manager which can be delegated, John Doe himself can issue a certificate delegating that attribute to another employee (e.g., Mary Smith). This might be useful when John Doe is on vacation.

Time is used for revocation, typically an attribute certificate would have a lifetime of days (rather than months or years that might be the lifetime of an X.509 identity certificate). Certificates also have a begin time and an end time, allowing John Doe to state the time for which Mary Smith has the role of manager.

Part of the configuration process at e-speak start up is configuring which ACIs are trusted for which attributes. We call this configuring the Trust Assumptions. The following is an example of the Trust Assumptions that might be appropriate for the Printer Resource shown in Figure 22.

- HPpubKey: com.hp.* employee
- OSOpubKey: com.hp.es.oso *
- ownerPubKey: ALL

These Trust Assumptions state:

- Anything given authority by the Hewlett-Packard public Key is trusted to issue attribute certificates designating the Hewlett-Packard employee attribute.
- Anything given authority by the Open Service Operation public Key is trusted to issue attribute certificates for Hewlett-Packard, e-speak, Open Services Operation attribute.
- Anything given authority by the Resource owner's public Key is trusted for all attributes.

By "given authority by the X public Key" we mean that X has issued the certificate for the attribute or has delegated the certification of that attribute. In the latter case the certificate processor will require the delegation certificate as well as the attribute certificate before it can determine if the attribute has been successfully authenticated.

We anticipate the e-speak will be deployed in environment where existing public Key infrastructures exist already (e.g., for X.509 and SSL). The e-speak attribute certificate base security machinery will interoperate with these environments by using X.509 attributes and using X.509 certificates for identity.

The new e-speak security model was developed to meet the requirements stated in "Capabilities" on page 189.

Capabilities

This is an outline of the requirements we have for supporting capabilities in e-speak. We are using the term capability abstractly, and no particular implementation should be assumed, whether it be public Key certificates, Kerberos tickets, or e-speak Keys.

A *capability* expresses permission to perform operations or obtain service. It may be used indirectly to express a role, such as a user's membership of a group. Capabilities will often be issued in response to authentication, but may be issued for any reason deemed sufficient by the issuer. Once issued, capabilities are presented to services to obtain facilities. In e-speak terms, capabilities are used to extract permissions on Resources and control Resource visibility. Capabilities generalize the functionality of e-speak Keys, while preserving other aspects of the e-speak architecture such as name virtualization and brokering.

1. Capabilities must be usable in a fully distributed context, with the capability issuer, the receiver of the capability, and the service it is presented to all being potentially on different machines. For example an authentication or logon service should be able to issue a Client capabilities that it can use elsewhere.
2. Capabilities should remain secure even when remote machines cannot be relied on to enforce security, and when communications security cannot be relied on.
3. It should be possible to create a service that requires a Client to possess a given capability without giving the service itself that capability, and without requiring that the service installer have the capability.
4. It should be possible for a Client to convince a service that it possesses a capability without giving the service the ability to convince anyone else that it has the capability.
5. It should be possible to present a capability to a remote service without giving the capability away to observers along the communication path.
6. It must not be possible to fake a capability.
7. It should be possible to tie a capability to a particular Client, so that only that Client can successfully use it.

8 It should be possible for a Client to delegate some of its capabilities to another service, so that the service can perform operations on the Client's behalf using the delegated capability. The service must not be able to abuse the delegated capability to obtain the Client's capability itself. Also it should be possible for the Client to define which other services the delegated capability may be used on, and for how long.

9 It should be possible to control propagation and use of capabilities in a distributed system, so that otherwise valid capabilities cannot be used where controls apply. This includes controlling the visibility of capabilities.

10 It should be possible to create restricted capabilities that can only be used for a given period of time, or can only be used a given number of times.

11 It should be possible to revoke a capability, so that although it was legitimately acquired it may no longer be used.

12 It should be possible to create anonymous capabilities that do not reveal the identity of the owner or user.

13 It should be possible for the issuer of a capability to communicate it to a Client by any means (e-mail, floppy disk, in writing).

Unique Core Names

ESUIDs will be used to simplify the process of import and export.

Resource Export

The copy of an exported Resource on another Core will have the same Core-name as the original Resource, although it may have different Metadata. The name for an exported Resource is the stringified form of its unique Core name. Cores need to remember which Resources have been exported, and a Name Frame can be used for that purpose. If the Core name generation algorithm is random enough, the probability of creating two Resources with the same Core name is very low (we can obviously prevent Core name collision in each individual Core). However, since the probability is not zero even for well-implemented Cores, and hostile Cores might deliberately create duplicate Core names, we need to deal with the eventuality. We propose using a short-circuiting protocol to discover if duplicate Core names actually refer to the same Resource, and resolve them if they do. Resource authentication will be used to decide if a Resource is genuine, and reject fakes.

Short-circuiting is the process of determining if two local Resources are different imports of the same Resource on another Core. It involves querying the home Core of both Resources and using the replies to determine if they were answered by the same Resource Handler.

If a duplicate authentic Resource is detected and error is reported, and the duplicate is not imported.

Chapter 16 Glossary

Term	Meaning
Advertising Service	A service for looking up resources not registered in the local Repository. It returns zero or more Connection Objects.
Allow field	A field in the metadata of a Resource consisting of a set of Locks. If any Lock in this field is opened by a Key presented by the requestor, the Resource will be accessible.
Arbitration policy	A specification within the search request accessor for naming that provides the logic to resolve multiple matches found for a name search.
Attribute Vocabulary	See **Vocabulary**.
Base Vocabulary	A Vocabulary provided at system start-up.
Builder	An entity identified by a Remote Resource Handler that is used to construct the internal state of a Resource imported by value.
Callback Resource	A Resource that the Client lists in its message header that can be used as a primary Resource when a reply is to be sent to the Client.

Term	Meaning
Client	Any active entity (e.g., a process, thread, service provider) that uses the e-speak infrastructure to process a request for a Resource.
Client library	The interface specification that defines the interface for e-speak programmers and system developers that will build e-speak-enabled applications.
Connection Factory	A Logical Machine's component that does the initial connection with another Logical Machine.
Connection Object	An object created by a Logical Machine to allow connections to it. The object is distributed widely to other services such as an Advertising Service.
Contract	See **Resource Contract.**
Core	The active entity of a Logical Machine that mediates access to Resources registered in the local Repository.
Core Event Distributor	A Core-managed Resource whose purpose is to collect information on e-speak Events and make such information available to management tools within the infrastructures.
Core-managed Resource	A Resource with an internal state managed by the Core.
Deny field	A field in the metadata of a resource consisting of a set of Locks. If any Lock in this field is opened by Keys presented by the requester, the Resource will not be accessed.

Term	**Meaning**
Distributor Service	A service that forwards published Events to subscribers.
Event	A message that results in the recipient invoking a registered callback.
Event filter	A subscription specification expressed as a set of attributes in a particular Vocabulary that must match those in the Event state in order for a Client to receive notification on publication of an Event.
Event state	A reference within an Event to its expressed set of attributes in a particular Vocabulary. These attributes must match the Event filter in order for the subscriber to receive notification of the Event.
Exception Resource	A Resource listed in a message header that can be named as the primary Resource in an error message.
Explicit Binding	An accessor that contains a Repository Handle.
Export context	An object created by the Remote Resource Handler for the exported Resource's metadata and state information (when its pass-by value is set). This object also defines the export and import.
Export Form	The form of a Resource metadata and, optionally, state that is sent to another Logical Machine.
Exporter	A reference to a Remote Resource Handler when it is in the process of exporting data to another Logical Machine through another Remote Resource Handler.

Term	Meaning
Export Name Frame	A Name Frame that holds a name for each exported Resource.
Importer	A reference to a Remote Resource Handler when it is in the process of importing data from another Logical Machine through another Remote Resource Handler.
Import Name Frame	A container that holds a name for each imported Resource.
Inbox	A Core-managed Resource used to hold request messages from the Core to a Client.
Key	An entity that opens Locks in Resource metadata as a means of expressing a Client’s access rights to Resources. Also used to test the Client’s right to use a name bound to the Resource.
Key Ring	An entity that holds a number of Keys.
Listener	A Logical Machine that has created a Connection Object in order to listen for connection requests to it from another Logical Machine acting as an initiator. Also the term used for the Client that receives an Event.
Lock	Stored in the metadata of a Resource. Can be opened by Keys presented by a Client.
Logical Machine	A Core and its Repository.

Term	Meaning
Lookup request	Resources with attributes matching the lookup request will be bound to a name in the Client's name space.
Lookup Service	The component that performs lookup requests used to find Resources that match attribute-value pairs in the Resource Description of Resources registered in the Repository.
Mailbox	Either an Outbox or an Inbox.
Mapping Object	An object binding an ESName to Resources or a Search Recipe.
Message	Means of Client-Core communication.
Metadata	Data that is not part of the Resource's implementation, but is used to describe and protect the Resource.
Name Frame	A Core-managed Resource that associates a string with a Mapping Object.
Name Search Policy	A name conflict resolution tool used by the Core to find the appropriate strings when looking up names in a Name Frame.
Outbox	The location where the Client places a message to request access to a Resource.
Pass-by value	A metadata field, which, when set to true, includes the state of the Resource in the Export Form.

Term	**Meaning**
Permissions	Access rights to a Resource or its metadata forwarded if a Key presented with the message opens the corresponding Lock.
Primary Resource	The target of a message sent by the Client when requesting access to a Resource.
Protection Domain	The environment associated with a particular Outbox from which Resources can be accessed.
Publish	A request sent to the Distributor Service to publish Events.
Remote Resource Handler	A Client that is established on both a local and a remote Logical Machines to handle intermachine communication.
Reply Resource	See **Callback Resource**.
Repository	A passive entity in the Core that stores Resource metadata and the internal state of Core-managed Resources.
Repository entry	The metadata of a Resource as stored in the Repository and made available to the Core when a Client's requests to access Resources are processed.
Repository Handle	An index into the Repository associated with the metadata of a Resource.
Repository View	A Resource that can be used to limit the search for particular Resources in a large Resource Repository, much as a database view restricts a search within a database.

Term	Meaning
Resource Contract	A Resource denoting an agreement between the Client and the Resource Handler for use of a particular Resource. The agreement includes a provision for the Client to use an API known to the Resource Handler when making the request for the Resource.
Resource	The fundamental abstraction in e-speak. Consists of state and metadata.
Resource Description	The data specified for the Attribute field of the metadata as represented by the Client to the Core. See also Resource Specification.
Resource Factory	An entity that can build the internal state of a Resource requested by a Client.
Resource Handler	A Client responsible for responding to requests for access to one or more Resources.
Resource Specific Data	A metadata field of a Resource. Carries information about the Resource. Can be public or private to the Resource Handler.
Resource Specification	Consists of all metadata fields, except the Attributes field, as represented by the Client to the Core.
Secondary Resource	Additional Resources included in a message header that may be needed by the Resource Handler.
State	Data a Resource needs to implement its abstraction.

Term	Meaning
Visibility fields	A reference to both the Allow and the Deny fields of the metadata for a Resource.
Visibility test	A reference to tests performed on the metadata Allow and Deny fields invoked by the Core when a Client requests access to a Resource.
Vocabulary	A Resource that contains the set of attributes and value types for describing Resources.
Vocabulary Builder	A Core-managed Resource registered by the Lookup Service that is used to create new value types, attributes, and Vocabularies.
Vocabulary Translator	A reference to a mechanism that is used to provide interoperation between different Vocabularies by mapping attributes from one Vocabulary into another through a Translator Resource.