e-speak

Architectural Specification

Developer Release 3.01
June 2000

HEWLETT® PACKARD

HEWLETT-PACKARD COMPANY 3000 Hanover St. Palo Alto, CA 94304

Contents

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 is probably 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.
- **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 show 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 are used in the run-time environment for legacy applications
- Implementors of extensions to existing services and resources used by Clients
- System administrators who 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 mechanisms used for access control
- The e-speak communication architecture
- The exceptions that can be generated by the e-speak platform
- The e-speak Event Service

- The e-speak management architecture
- The e-speak Respository used for storing Resource metadata (informational)
- A description of how localization is implemented (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 shown 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 can 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 *Session Layer Security Protocol* (SLS). 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, 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 is analogous to a “home directory” in an operating system. It contains bindings to Resources created by the Client and e-speak keeps track of memory usage due to these bindings.

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 Session Layer Security Protocol as shown in Figure 2.

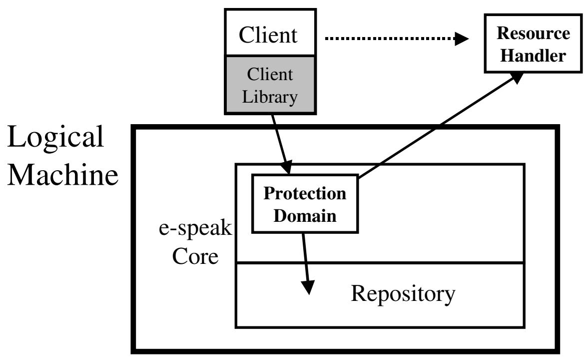


Figure 1 Resource access in e-speak

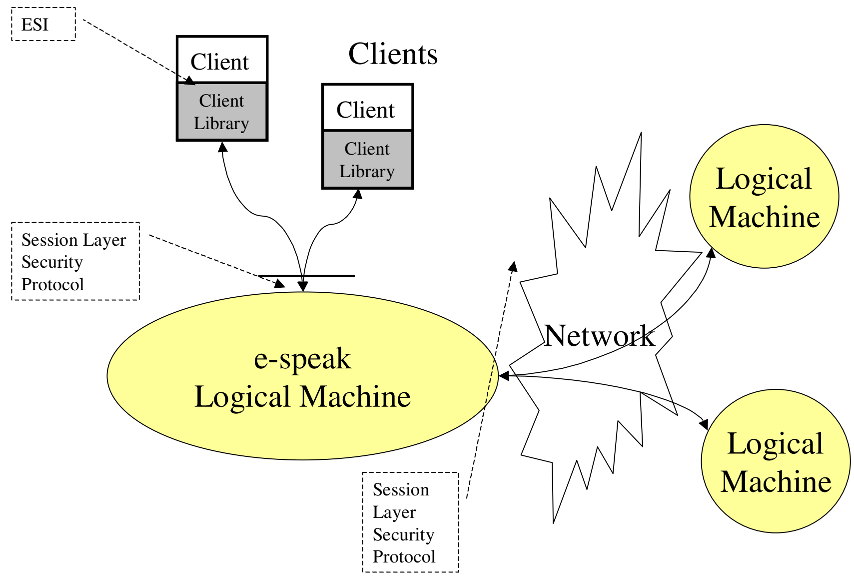


Figure 2 Communicating e-speak logical machines

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 locate the Resource Handler and 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 can 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 e-speak Cores perform name resolution 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 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 routes all service access messages based on the contents of the metadata of Resources referred to in the parameters of the message. The implications of this 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 attributes and their types used in descriptions and lookups
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
Resource mask, owner public key and service ID	Access control information
Private Resource-specific data	Data important to the provider of the Resource, such as the provider's internal name for the Resource. Not interpreted by e-speak.
Public Resource-specific data	Data important to the user of the Resource, such as a stub for invoking methods. Not interpreted by e-speak.

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. 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 Vocabu-

lary and a constraint string expressed in that Vocabulary. The predicate is expressed in a form 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 a means to find Resources not registered in the local Repository. 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 Vocabularies.

Naming Model

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

- E-speak Names are Universal Resource Locators (URLs).
- Name spaces are maintained in container Core-managed Resources called Name Frames. Name Frames can themselves contain other Name Frames, so each e-speak Core has a hierarchy of Name Frames beginning from its “root” Name Frame. By default when the Client specifies the name of a service, the e-speak Core, starting with its root Name Frame, finds a mapping between the name and a name unique to this Core. In addition a client can specify a name beginning in the root Name Frame of another e-speak Core, by specifying the host in the e-speak Name.
- 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. 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.

Access Control

E-speak security is based on a Public Key Infrastructure (PKI). Specifically it uses the Simple Public Key Infrastructure (SPKI) developed within the Internet Engineering Task Force.

All entities in e-speak (Resources and Cores) are identified by public keys. To authenticate an entity we verify it knows the private key corresponding to the given public key. No entity should ever intentionally share its private key or give anybody access to the private key.

Any entity can create a key-pair. Provided the private key is kept secret, the key-pair is unique to that entity. However, having a key-pair gives you no power in the system. It is necessary also to have certificates stating access rights issued to your public key.

In e-speak access rights are stated in *attribute certificates*. So as well as the conventional use of certificates to bind a name to a public key (e.g. X.509), we also use certificates to bind access rights to public keys. This avoid having to have online access control databases or access control lists.

To decide whether to honor an incoming request a Resource Handler must decide if it has a certificate (or certificates) granting access rights for the request. If it finds such a certificate, it must verify that the sender of the request knows the private key corresponding to the public key in the certificate to which the access rights have been given (formally this is the subject of the certificate). It does this by a cryptographic protocol that is described in Chapter 7, “Communication”.

Finally before honoring the request, the Resource Handler must verify that it trusts whoever issued the certificate. It does this by verifying that the certificate has been signed by an entity that it trusts. Resource Handlers do not trust all certificate issuer’s equally. A Resource Handler can choose whether to trust a given certificate issuer and may restrict what access rights a given certificate issuer can issue.

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 extracts Resource specific metadata and security information from its Repository and inserts this in the message before forwarding 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.

The Resource specific metadata and security information inserted by the Core into a message can be used by the Resource Handler to determine how to process the message.

As each message is routed the e-speak Core may generate events for logging and monitoring.

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.

Session Layer Security Protocol

All messages exchanged between e-speak Cores and between e-speak Cores and Clients use the Session Layer Security protocol. This provides secure message passing between entities as well as unprotected message exchanges. Applications can choose whether to use secure message passing or not.

Session Layer Security protocol is designed to support e-speak mediation. E-speak mediation requires e-speak to modify certain parts of the message so that the message can be routed between endpoints and means there are not a TCP connection between the endpoints. These requirements means that existing protocols such as SSL (Secure Sockets Layer) or TLS (Transport Layer Security) are not suitable for end to end security in e-speak.

Session Layer Security protocol allows multiple secure sessions to be multiplexed over a single TCP connection. This means that two e-speak Cores can be connected via a single TCP connection with many Clients and have many different secure sessions to different e-speak Resources.

Session Layer Security protocol also supports tunnelling. During firewall traversal we might want the firewall to control the client access rights to the internal LAN for every packet. However, we might not want the firewall to see all the traffic in clear (therefore, losing the end-to-end security property). With Session Layer Security protocol we can nest a secure session inside another one, possibly with different end points, allowing us to achieve both goals simultaneously.

Session Layer Security protocol is designed to support SPKI for access control. It performs the negotiation of access rights that need to be proven represented by multiple SPKI certificates.

Session Layer Security supports the following encoding types for messages.

- CLEAR_DATA: The message is not encrypted or protected against modification.
- PROTECTED_DATA: The message is not encrypted but it is protected against modification.
- SECURE_DATA: The message is encrypted and protected against modification.

Session Layer Security has been designed to be independent of transport. However, for interoperability between e-speak Cores and interoperability between e-speak Cores and Clients, direct implementation over TCP is assumed. Other implementations are possible, including passing SLS messages over HTTP, or through shared memory.

Core to Core Communication

Communication between e-speak Cores uses the Session Layer Security protocol. Two core-managed Resources are used for remote communication between e-speak cores: the Connection Manager (for connection management) and Remote Resource Manager (for management of remote resource metadata).

The Connection Manager sets up the initial connection, manages it and closes it down when it is no longer needed. It requires the host name (or IP address) and port of the remote e-speak Core to set up a remote connection. The Connection Manager has a well known name: es://<server>/CORE/ConnectionManager. So given the host and port number (i.e. the <serve> part) a Connection Manager can negotiate with the remote Connection Manager to establish a connection between the two e-speak Cores. Once two Connection Managers have established a connection between their respective cores exchange Resources with each other using their Remote Resource Managers for Resource export and import. Resources.

The Remote Resource Manager is responsible for managing metadata: importing and exporting resources from the remote e-speak core. The Remote Resource Manager on any given e-speak core is: es://<server>/CORE/RemoteResourceManager. So given that two Connection Managers have established a connection between two e-speak Cores. The Remote Resource Managers can communicate with each other to exchange Resources.

All resources can be exported by reference, in which case a copy of the metadata of the Resource is sent to the remote core. In addition certain Core-managed Resources can be exported by value, in which case a copy of the Resource is sent to the remote e-speak Core.

Resource import and export serves a number of purposes in e-speak.

- Resource discovery is made more efficient for local Clients, because a copy of the metadata is available locally.
- The lookup mechanism requires that a vocabulary is availably locally for both Resource registration and lookup in that vocabulary. Using Resource export, a vocabulary can be defined once and exported to wherever it is needed.
- Name Frames can be defined containing a set of useful bindings (akin to an environment for a Client). Using Resource export, these Name Frames can be made available locally wherever a Client needs them. This can be particularly useful for mobile clients.

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 constructs and Session Layer Security message setting the “to” address to the Name of the Remote Resource(e.g. es://<host_for_core_B>/resource/foo). The from address is set to the Name of the Client (e.g. `es://<host_for_core_A>/client/bar`). This message is sent using TCP to Core A, where it is placed in the Client’s outbox.
2. A message handling thread in Core A, picks up the message and sends it to Core A’s router. Core A’s router determines that the message is for Core B. It checks that it has a connection with Core Band forwards the message to the relevant inbox.

3 A message handling thread on Core A picks the message up from the inbox and transmits it to Core B (via a TCP connection) where it is placed in the outbox for incoming messages from Core A.

4 The router on Core B resolves the Name for the "to address" in its root Name Frame. The resolved name is a binding to the Resource metadata.

5 Core B's router retrieves the Resource's metadata. This tells it to which inbox to send the message. It also extracts the Private Resource Specific data and various security information that is used by the Resource Handler to process the message.

6 A message handling thread picks the message up from the inbox and sends it to the Resource Handler through a TCP connection.

7 Any communication between the Resource Handler and the physical Resource is outside of the control of the e-speak Core.

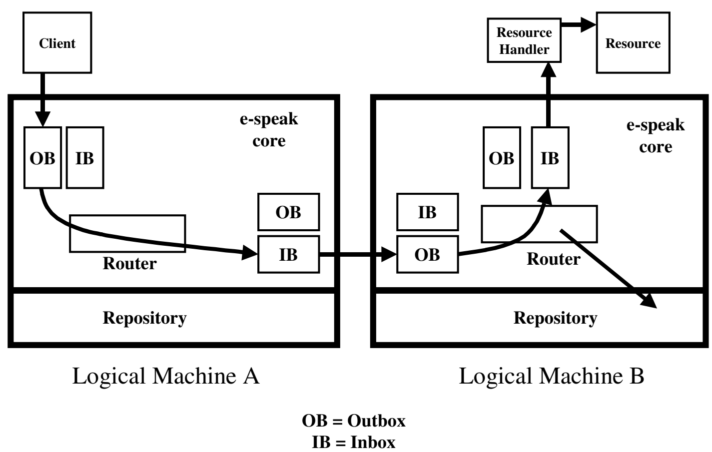


Figure 3 Distributed e-speak

The processing for any return message is similar, except the roles of the respective e-speak Cores are reversed (recall e-speak is asynchronous - the Core does not distinguish between request and reply messages when routing them).

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 shown 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 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 metadata, and e-services registered as e-speak Resources.
- 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: E-speak supports 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 (Enterprise) Java Beans, CORBA, and COM+.
- Management protocols and standards: Support for SNMP, ARM, and DEN is planned.
- Languages: An E-speak library exists for Java, but e-speak has been designed to be language independent. Any language can be used to construct an Session Layer Security message which is all that is required to use e-speak.

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.

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 access control is based on a Public Key Infrastructure using attribute certificates for scalable distributed security. This is supported by the Session Layer Security protocol which allows messages to be protected against tampering, eavesdropping or replay. In addition the Session Layer Security protocol allows unprotected messages to be sent, should security not be needed. Session Layer also supports authenticated tunneling for efficient and secure firewall traversal.

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 creates 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 helps us understand it.

Chapter 3 Resource Descriptions, Resource Specifications and Resource Types

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 *Resource Factory* 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;
   FilterSpec filter;
   ADR metadataMask;
   ADR resourceMask;
   ADR ownerPublicKey;
   ADR ServiceId;
   ESMap privateRSD; //Not exported if export by reference
   ESMap publicRSD;
   ESName owner; //Not exported
   ESName resourceHandler; //Not exported
   int eventControl;
   ESUID uid;
   String URL;
}
```

The type `ESMap` is serialized as `ESArray` (see Chapter 7, “Communication” 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. `ResourceReference` is the abstract base class for `ESName`. `ESName` and not `ResourceReference` is passed by the e-speak Application Binary Interface (ABI).

The owner and resourceHandler fields are not included when the ResourceSpecification is serialized for export (see Chapter 7, “Communication”) and the privateRSD field is only included in an export serialization if the export is export by value (Chapter 7, “Communication”).

boolean byValue;

If the `byValue` flag is True, the internal state of this Resource is included with the Resource Specification and Resource Description sent to another Logical Machine. The Core provides 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. Every Resource must be registered in some Contract.

FilterSpec filter;

```
class FilterSpec{
   ESSet Vocabularies;
   String constraint;
}
```

The filter field consists of a set of vocabularies and a constraint. This is intended for use by a service provider to register a constraint that can be evaluated to determine if the Resource should be returned in response to an evaluation of a SearchRecipe in a lookup request in a NameFrame (see "Name Frame" on page 38). The constraint uses the UserProfile associated with the client. In this way a Resource can only be discovered by certain Clients that satisfy the constraint.

ADR metadataMask;

The metadataMask controls which operations manipulating the Resource's meta-data has security disabled. The interface name in the metadataMask is always the ResourceManipulationInterface. The format of the Resource Masks is specified in Chapter 6, "Access Control".

ADR resourceMask;

The resourceMask determines which operations supported by the Resource has security disabled. The format of the Resource Masks is specified in Chapter 6, "Access Control".

ADR ownerPublicKey;

This field contains the owner public key. The format is specified in "SPKI BNF Formats" on page 106.

ADR ServiceId;

This field contains the serviceId of the Resource. ServiceIds are defined in "Service Identity" on page 87.

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) can be of interest to users of the Resource. Therefore, the Client registering the Resource can 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.

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 are delivered to this Inbox. This field is `NULL` for Core-managed Resources. The Client that is connected to this Inbox receives messages for this Resource. The format of these messages is defined in "Messages from the Resource Handler to the Client" on page 151. This field can 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 Resource metadata (the Resource Description or the Resource Specification) is changed, an Event is published to the Core's Event distributor.

ESUID

```
public class ESUID
{
   byte[] UniqueId;
}
```

An ESUID contains a byte array that is up to 64 bytes long. An ESUID is guaranteed by probabilistic means. In the current implementation is consists of 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) and is 20 bytes long. The Resource identity is unique within a given Core and is 12 bytes long.

String URL;

This field is the ESName (represented as a string) by which the registering entity refers to the Resource. It is an ESName (URL) which others can use to access the Resource.

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;
}
```

Each element in the `ESArray` is an `AttributeSet`.

An `AttributeSet` consists of the `ESName` of a Vocabulary Resource and an `ESMap` of name-attribute pairs.

The Vocabulary is one in which the attributes have meaning. See Chapter 5, “Vocabularies”.

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;
   Boolean essential;
}
```

The `name` field is the name associated with `Attribute`. The `Value` type is defined in "Vocabulary" on page 55. It can contain a single value or a set of values (sets of values are not supported in the current release).

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.

Resource type

The e-speak Core associates a type with every Resource registered. The following defines the currently recognised Resource types.

```
class resourceType{
   static int INBOX_CODE = 0;
   static int META_RESOURCE_CODE = 1;
   static int PROTECTION_DOMAIN_CODE = 2;
   static int RESOURCE_FACTORY_CODE = 3;
   static int CONTRACT_CODE = 100;
   static int CORE_DISTRIBUTOR_CODE = 110;
   static int IMPORTER_EXPORTER_CODE = 120;
   static int MAPPING_OBJECT_CODE = 140;
   static int NAME_FRAME_CODE = 150;
   static int REPOSITORY_VIEW_CODE = 160;
```

```
    static int SECURE_BOOT_CODE = 170;
    static int SYSTEM_MONITOR_CODE = 180;
    static int VOCABULARY_CODE = 190;
    static int CORE_MANAGEMENT_SERVICE_CODE = 200;
    static int DEFAULT_VOCABULARY_CODE = 210;
    static int DEFAULT_CONTRACT_CODE = 220;
    static int FINDER_SERVICE_CODE = 230;
    static int CONNECTION_MANAGER_CODE = 240;
    static int REMOTE_RESOURCE_MANAGER_CODE = 250;
    static int EXTERNAL_CODE = 1000;
    static int EXTERNAL_RESOURCE_CONTRACT_CODE = 1001;
    static int INBOX_CODE = 0;
    static int META_RESOURCE_CODE = 1;
    static int PROTECTION_DOMAIN_CODE = 2;
    static int RESOURCE_FACTORY_CODE = 3;
    static int CONTRACT_CODE = 100;
    static int CORE_DISTRIBUTOR_CODE = 110;
    static int IMPORTER_EXPORTER_CODE = 120;
    static int MAPPING_OBJECT_CODE = 140;
    static int NAME_FRAME_CODE = 150;
    static int REPOSITORY_VIEW_CODE = 160;
    static int SECURE_BOOT_CODE = 170;
    static int SYSTEM_MONITOR_CODE = 180;
    static int VOCABULARY_CODE = 190;
    static int CORE_MANAGEMENT_SERVICE_CODE = 200;
    static int DEFAULT_VOCABULARY_CODE = 210;
    static int DEFAULT_CONTRACT_CODE = 220;
    static int FINDER_SERVICE_CODE = 230;
    static int CONNECTION_MANAGER_CODE = 240;
    static int REMOTE_RESOURCE_MANAGER_CODE = 250;
    static int ACCOUNT_MANAGER_CODE = 260;
    static int USER_ACCOUNT_CODE = 270;
    static int EXTERNAL_CODE = 1000;
    static int EXTERNAL_RESOURCE_CONTRACT_CODE = 1001;
    static int DEFAULT_ACCOUNT_VOCABULARY_CODE = 1002;
}
```

Chapter 4 Core-Managed Resources

Clients interact with the e-speak Core by sending messages to Core-managed Resources. For example, the Resource Factory 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

All the methods described in this Chapter throw `ESInvocationException` (see Chapter 8, “Exceptions”), the base class for exceptions thrown by the e-speak Core to the Client during message processing. The e-speak 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.

Connection Manager

The connection manager is described in (xref to communications chapter).

Core Management Resource

The core management Resource is not supported in the current release.

```
interface CoreManagementInterface extends ManagedServiceIntf{
   int ping(int pingValue)
     throws ESInvocationException;
   ESName[] getClientConnections()
     throws ESInvocationException;
   boolean stopServingOutbox(ESName ProtectionDomain)
     throws ESInvocationException;
   boolean stopServingInbox(ESName Inbox)
     throws ESInvocationException;
   boolean startServingOutbox(ESName ProtectionDomain)
     throws ESInvocationException;
   boolean startServingInbox(ESName Inbox)
     throws ESInvocationException;
   boolean removeProtectionDomain(ESName ProtectionDomain)
     throws ESInvocationException;
   boolean denyNewClientSessions()
     throws ESInvocationException;
   boolean acceptNewClientSessions()
     throws ESInvocationException;
   long getTotalMemory()
     throws ESInvocationException;
   long getFreeMemory()
     throws ESInvocationException;
   void startJVMGC()
     throws ESInvocationException;
```

```
    void stopJVMGC()
      throws ESInvocationException;
    void setJVMGCInterval(int millis)
      throws ESInvocationException;
    int getJVMGCInterval()
      throws ESInvocationException;
    boolean isJVMGCRunning()
      throws ESInvocationException;
    void startScavenger()
      throws ESInvocationException;
    void stopScavenger()
      throws ESInvocationException;
    void setStatsNum(int num)
      throws ESInvocationException;
    ESArray getScavengerStats()
      throws ESInvocationException;
}
```

The Core Management Resource provides a way for a client to manage its own and other cores. The Core Management Resource is also a managed Core-managed resource: it implements the interface ManagedServiceIntf described in (xref to managment chapter).

The method ping checks that the core is up and returns the value specified

The method getClientConnections returns a list of protection domains that are currently being used.

The methods stopServingOutbox and startServingOutbox tell the e-speak core to stop or start serving the outbox associated with the protection domain specified.

The methods stopServingInbox and startServingInbox tell the e-speak Core to stop or start serving messages to the Inbox specified.

The method removeProtectionDomain removes the Protection Domain specified. Any client is using the Protection Domain is expected to be disconnected and any Resources contained in the Protection Domain are deregistered.

The methods denyNewClientSessions and acceptNewClientSessions tells the e-speak core to stop or start accepting new connections from clients.

JVM management methods

The following methods are specific to e-speak Cores implemented in Java. An e-speak Core cannot implement these methods in which case it returns a Method-NotImplemented exception.

The methods getFreeMemory and getTotalMemory get the free memory or total memory in the e-speak Core's Java Virtual Machine (JVM).

The methods startJVMGC() and stopJVMGC() start and stop the e-speak Core's JVM garbage collector.

The methods setJVMGCInterval and getJVMGCInterval() set and get in milliseconds the interval between runs of the JVM garbage collector.

The method isJVMGCRunning() returns true if the JVM garbage collector is running.

Scavenger management methods

The current implementation of the e-speak Core has a scavenger that looks for resources in the repository that are no longer valid and removes them. Examples of resource that may no longer be valid include the following.

- Resources registered in a Protection Domain that has been removed.
- Resources imported from another e-speak Core after the connection to that Core is closed.

An e-speak Core cannot implement these methods, in which case it returns a MethodNotImplemented exception.

The methods startScavenger and stopScavenger enable and disable the scavenger from running.

The scavenger also records statistics for each run as follows.

```
class ScavengerStats
{
```

```
    Int runNo;
    Long timeElapsed;
    Int numInspected;
    Int numCollected;
    Int totalNumInspected;
    Int totalNumCollected;
    String phase;
}
```

The runNo field indicates the current run (the first run is run number 1).

The timeElapsed field is the time taken for the run in milliseconds.

The field numInspected indicates the total number of Resources inspected in this run.

The field numCollected indicates the total number of Resources removed in this run.

The fields totalNumInspected and totalNumCollected are the running totals since the e-speak Core was started.

The phase field contains the string “Mark” or “Sweep”, this denotes whether the run is a “mark” or “sweep” run. Resources are only removed from the repository (and the numCollected count incremented) on a sweep run. There is no notion of “mark” or “sweep” phases on Resources in the cache.

The scavenger keeps statistics for a certain number of runs. This is set by method setStatsNum in the CoreManagementInterface. The method getScavengerStats returns an ESArray of containing an instances of ScavengerStats in each element. (The current implementation returns an instance of the Java Vector class.)

Remote Resource Manager

The Remote Resource Manager is described in (xref to communications chapter).

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 implements the MailboxInterface defined below:

```
interface MailboxInterface
{
    boolean isConnected()
      throws ESInvocationException;

    void connect(int slot)
      throws ESInvocationException;

    void disconnect()
      throws ESInvocationException;

    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 “Resource 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.

You can 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 somehow needs 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 certificate 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 Client B's I/O channel information.

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. ESNames are defined in fully in (xref to ESName section of communications chapter).

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 2.

Table 2 Mapping Object accessor types and descriptions

Accessor Type	Descriptions

Table 2 Mapping Object accessor types and 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 can 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 can 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 are available to `SearchRecipe`. This is a subset of all the Resources registered with the Core.

Class `NameSearchPolicy` is defined below:

```
 class NameSearchPolicy
{
    static final int NSP_ANY = 0;
    static final int NSP_SIMPLE = 1;
    static final int NSP_EXPLICIT = 2;
    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 causes undefined behavior if used).

If `matchSense` is false, the meaning of the Name Search Policy is negated, so `listBindings` returns 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:

```
 class NameFrame
{
   ESMap bindings;

    void lookup(String baseName,
         SearchPredicate arbPolicy,
```

```
        ESName targetFrameHandle,
        String toBaseName)
     throws LookupFailedException, InvalidNameException,
     StaleEntryAccessException;

boolean isBound(String baseName)
  throws ESInvocationException;

void bind(String baseName, SearchRecipe recipe)
  throws NameCollisionException, QuotaExhaustedException,
  ESInvocationException, ESServiceException;

void rebind(String baseName, SearchRecipe recipe)
  throws ESInvocationException, NameCollisionException;

void unbind(String name)
  throws ESInvocationException, InvalidNameException
  QuotaExhaustedException;

void rename(String oldName,String newName)
  throws ESInvocationException, ESServiceException
  InvalidNameException, NameCollisionException;

void copy(String toName, ESName from)
  throws ESInvocationException, ESServiceException
  NameCollisionException,InvalidNameException,
  StaleEntryAccessException, QuotaExhaustedException;

void add(String name, ESName from)
  throws ESInvocationException, InvalidNameException,
  StaleEntryAccessException;

void subtract(String name, ESName from)
  throws ESInvocationException InvalidNameException,
  StaleEntryAccessException;
```

```
 String[] listNames(NameSearchPolicy nsp)
   throws ESInvocationException, NameNotFoundException;

 String[] listBindings(String aBaseName,
      NameSearchPolicy nsp,
      ESName targetFrame
)
   throws ESInvocationExceptionInvalidNameException,
   StaleEntryAccessException, QuotaExhaustedException;
}
```

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. A MappingObject consists of a set of SearchRecipes and explicit bindings to resources. The explicit bindings are internal pointers (repository handles) to the resource metadata in the e-speak Core's repository. A MappingObject is serialized as an ESSet containing the SearchRecipes in the MappingObject (explicit bindings are not contained in the serialized form transmitted in the case of pass by value).

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 is 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`.

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

Finder resource

The finder resource is for finding services provided by the core.

```
interface FinderInterface {
  FinderResults find(SearchRecipe recipe, int maxToFind)
     throws ESInvocationException, LookupFailedException;
  FinderResults find(FinderContext context)
     throws ESInvocationException, LookupFailedException;
}
```

The method find(SearchRecipe recipe, int maxToFind) finds services based on the SearchRecipe; maxToFind is the maximum number of results to return. If maxToFind=0 then the request is to know if there are any search results.(But, no need to know the actual results). If maxToFind=-1 the method returns all results found.

```
class FinderResults{
   ESname[] esnames
   resourceType[] resType
   FinderContext context;
}
```

The esnames field is the set of ESnames that match the search recipe.

The field esnames is set to null if there are no search results.

In the case when maxToFind=0 if the search is successful, esnames has a single element and esnames[0] is null.

If there are no further elements to find (i.e. all matching services have been returned) context is set to null.

A LookupFailedException thrown when there are errors in the e-speak Core while performing the search. The class resourceType is defined in (xref to resource description and specification chapter).

The class FinderContext contains an opaque byte array used as the parameter to find(FinderContext) to get the next maxToFind set of matching results, if the search results in more than maxToFind elements.

```
class FinderContext{
   byte[] queryContext;
}
```

The method find(FinderContext context) is used to get more results when find(SearchRecipe recipe, int maxToFind) indicates there are more than maxToFind matching services (by returning a non null FinderContext object in FinderResults).

Protection Domain

A Client's Protection Domain is analogous to a user's home directory in an operating system. It contains a root Name Frame in which the Client can place bindings.

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 can 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:

```
interface ProtectionDomainInterface
{
    ESName[] switchPD()
      throws ESInvocationException, PermissionDeniedException,
      NameNotFoundException StaleEntryAccessException,
      QuotaExhaustedException;

    Object[] getQuotaInfo()
      throws ESInvocationException PermissionDeniedException,
      NameNotFoundException;

    Object[] setQuota(long softQuota, long hardQuota)
      throws ESInvocationException, PermissionDeniedException,
                        NameNotFoundException;

    ESName newProtectionDomain(String name,
      boolean persistent
   )
      throws PermissionDeniedException;
}
```

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 an array of ESName. There are two elements in the returned array. The first element is the ESName for the old Protection Domain. The second element is the ESName for the new 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 method newProtectionDomain, creates a new Protection Domain. The name parameter is the name given when registering the new Protection Domain in the default vocabulary. The parameter persistent is set to true, if the new Protection Domain is to be made persistent. The return value is the ESName of the new Protection Domain.

The following initial names are defined in the default Name Frame of a new Protection Domain:

"CurrentPD" is bound to the Protection Domain itself

"Core" is bound to the core name frame (es://host/core) (see xref to comms chapter section on ESNames).

Repository View

A Repository View contains references to a set of Resources.

When a Client does a lookup in a Repository View, the Core attempts 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;

 boolean add (ESName res)
  throws ESInvocationException PermissionDeniedException,
  StaleEntryAccessException, NameNotFoundException,
  QuotaExhaustedException;

 boolean remove (ESName res)
  throws ESInvocationException PermissionDeniedException,
  StaleEntryAccessException, NameNotFoundException,
  QuotaExhaustedException;

 boolean contains (ESName res)
  throws ESInvocationException PermissionDeniedException,
                StaleEntryAccessException, NameNotFoundException;

    boolean clear ();
     throws ESInvocationException QuotaExhaustedException
     PermissionDeniedException, NameNotFoundException;

    boolean addExternalLookupHandler(ESName res);
     throws ESInvocationException PermissionDeniedException,
     StaleEntryAccessException, NameNotFoundException;

    boolean removeExternalLookupHandler()
     throws ESInvocationException StaleEntryAccessException
     PermissionDeniedException, NameNotFoundException;
}
```

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

In general, all methods return true if they are successful, false if they fail. Clients can `add` Resources to and `remove` Resources from a Repository View. Attempts to add a Resource already in a Repository View fails, as does attempting to remove a non-existing Resource. 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;

    void getVocabularies(ESName targetFrame);
```

```
        throws ESInvocationException PermissionDeniedException,
        StaleEntryAccessException, NameNotFoundException;
}
```

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:

```
 class ResourceFactoryInterface
{
    void registerResource (
      ResourceDescription descr,
      ResourceSpecification spec,
      Boolean persistence,
      Object param,
      ESName targetFrame,
      String  toBaseName
)
   throws ESInvocationException PermissionDeniedException,
   StaleEntryAccessException, NameNotFoundException,
   NameCollisionException;
}
```

The `registerResource` method takes `ResourceDescription` and `ResourceSpecification` as parameters. If persistence is true, the Core preserves the metadata after the Client's connection is closed and, in the case of Core-managed Resources only, also the state; otherwise metadata and state are not preserved after the Client's 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 serialization format. The `targetFrame` parameter is the `ESName` of a Name Frame in which the name for the new Resource is put. The `toBaseName` parameter is the name of the new Resource in the Name Frame.

(TODO: Need to explain what the Object parameter contains for each type of Resource)

Resource Manipulation Resource

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. All methods can throw PermissionDeniedException, StaleEntryAccessException and NameNotFoundException

```
 interface ResourceManipulationInterface
{
 void unregister (ESName resource)
   throws ESInvocationException;

 void setResourceOwner (ESName resource)
   throws ESInvocationException;

ESName getResourceOwner(ESName resource)
   throw ESInvocationException;

ESName getResourceProxy (ESName resource)
   throws ESInvocationException;

 void setResourceProxy (ESName resource,
     ESName resourceHandler)
```

```
    throws ESInvocationException;

  ESName getResourceContract (ESName resource)
    throws ESInvocationException;

ADR getMetadataMask(ESName target)
    throws ESInvocationException;

void setMetadataMask(ESName target, ADR mask)
    throws ESInvocationException;

ADR getResourceMask(ESName target)
    throws ESInvocationException

void setResourceMask(ESName target, ADR mask)
    throws ESInvocationException

ADR getOwnerPublicKey(ESName target)
    throws ESInvocationException;

void setOwnerPublicKey(ESName target, ADR key)
    throws ESInvocationException

ESMap getRSD(PublicESName resource)
    throws ESInvocationException;

  void setPublicRSD(ESName resource,ESMap rsds)
    throws ESInvocationException;

  ESMap getPrivateRSD(ESName resource)
    throws ESInvocationException;

  void setPrivateRSD(ESName resource, ESMap rsds)
    throws ESInvocationException;

  ResourceDescription getResourceDescription(ESName target)
```

```
    throws ESInvocationException;

  void setResourceDescription(ESName resource,
    ResourceDescription desc)
    throws ESInvocationException;

 int getEventControl (ESName resource)
    throws ESInvocationException;

  void setEventControl (int setting)
    throws ESInvocationException;

   boolean isPersistent (ESName target)
    throws ESInvocationException;

   boolean isTransient (ESName target)
    throws ESInvocationException;

  void setPersistent (ESName target)
    throws ESInvocationException;

  void setTransient (ESName target)
    throws ESInvocationException;

 ESUID getESUID(ESName target)
    throws ESInvocationException;

 ESName getUrl(ESName target)
    throws ESInvocationException;

 long getQuota(ESName target)
    throws ESInvocationException;

 ResourceType getType(ESName target)
    throws ESInvocationException
```

```
ADR getServiceID(ESName target)
   throws ESInvocationException

void setServiceID(ESName target, ADR id)
   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.

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

The methods getMetadataMask and setMetadataMask are used for getting and setting those operations for which security is disabled for a particular Resource's metadata: anybody can invoke the methods listed in this mask to manipulate the particular Resource's metadata. The methods getResourceMask and setResource-Mask perform the analogous function for the operations supported by the Resource itself.

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:

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

String getDescription()
   throws ESInvocationException;

AttributePropertySet getProperties()
   throws ESInvocationException;

boolean mutateProperties(AttributePropertySet props)
   throws ESInvocationException QuotaExhaustedException,
   StaleEntryAccessException;
}
```

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 method `mutateProperties` returns true if the Vocabulary's AttributePropertySet is changed, false otherwise. The definition of `AttributePropertySet` is given below:

```
 class AttributePropertySet
{
   ESMap AttributeProperty
}
 class AttributeProperty
{
   String attrName;
```

```
    ValueType valuetype;
    Value defaultValue;
    String definition;
    boolean multiValued;
    int rangeKind;
    double minRange;
    double maxRange;
    String description;
    int keyIndexType;

    static final int NO_RANGE = 0;
    static final int LEFT_RANGE = 1;
    static final int FULL_RANGE = 2;
    static final int RIGHT_RANGE = 3;
    static final int NO_INDEX = 0;
    static final int HASH_INDEX = 1;
    static final int TREE_INDEX = 2;
}
```

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.

The definition field is a constraint to be evaluated to compute the value. This is for future enhancement, such constraints are not supported in the current release.

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.

The keyIndexType field is to support efficient lookup in a repository based on a Database Management System (DBMS). Valid values of keyIndexType are: NO_INDEX, HASH_INDEX and TREE_INDEX. If the value is HASH_INDEX or TREE_INDEX the attribute is used as an index by the DBMS. This is discussed further in the (xref to “Repository” (informational) chapter).

`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:

```
 class ValueType
{
        static final String STRING_TYPE = "String";
        static final String LONG_TYPE = "Long";
        static final String DOUBLE_TYPE = "Double";
        static final String BOOLEAN_TYPE = "Boolean";
        static final String BIG_DECIMAL_TYPE = "BigDecimal";
        static final String TIMESTAMP_TYPE = "Timestamp";
        static final String DATE_TYPE = "Date";
        static final String TIME_TYPE = "Time";
        static final String INTEGER_TYPE = "Integer";
        static final String FLOAT_TYPE = "Float";
        static final String CHAR_TYPE = "Char";
        static final String BYTE_ARRAY_TYPE = "ByteArray";
        static final String BYTE_TYPE = "Byte";
        static final String SHORT_TYPE = "Short";
        static final String SET_TYPE = "Set";
        static final String NAMEDOBJECT_TYPE = "NamedObject";
        static final String OTHER_TYPE = "Other";

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

The `typeName` field is a string name of the value type object; valid values are those in the `strings` defined in the class definition.

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`.

```
class Value{
   static final byte STRING_TYPE_CODE = 0x00;
   static final byte LONG_TYPE_CODE = 0x01;
   static final byte DOUBLE_TYPE_CODE = 0x02;
   static final byte BOOLEAN_TYPE_CODE = 0x03;
   static final byte BIG_DECIMAL_TYPE_CODE = 0x04;
   static final byte TIMESTAMP_TYPE_CODE = 0x05;
   static final byte DATE_TYPE_CODE = 0x06;
   static final byte TIME_TYPE_CODE = 0x07;
   static final byte INTEGER_TYPE_CODE = 0x08;
   static final byte FLOAT_TYPE_CODE = 0x09;
   static final byte CHAR_TYPE_CODE = 0x0A;
   static final byte BYTE_ARRAY_TYPE_CODE = 0x0B;
   static final byte BYTE_TYPE_CODE = 0x0C;
   static final byte SHORT_TYPE_CODE = 0x0D;
   static final byte SET_TYPE_CODE = 0x0E;
   static final byte NAMEDOBJECT_TYPE_CODE = 0x0F;
   static final byte OTHER_TYPE_CODE = 0xFF;
   static final byte INVALID_BASE_TYPE_CODE = 0xFF;

   byte tCode;
   object val;

}
```

Permissible values of tCode are shown in the class definition. The type code defines the object contained in the class instance. The e-speak serialization format defines the serialization format for all these object types with the following exceptions.

In the case of tCode being SET_TYPE_CODE, val contains a set of values. This is not supported in the current release.

BIG_DECIMAL, DATE, TIMESTAMP, TIME and NAMEDOBJECT are sent as strings. These types are taken from the following java packages.

- java.math.BigDecimal
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

The Account Manager Resource

The Account Manager is Resource is for managing a notion of user accounts on an e-speak Core. A user account contains various information about the user including its PSE (Private Security Environment). This enables a user to authenticate to the Account Manager (via userid, password) and to retrieve their PSE. In the current implementation they need a passphrase to unlock their PSE to access their key material (see Chapter 6, “Access Control”).

The class AuthInfo defines the basic information used by the Account manager to identify a user

```
class AuthInfo{
   String userName;
   String passPhrase;
   String homeAddress;
}
```

The home address indicates the “home e-speak Core of a user” in host:portNumber format. An example is:

```
myhost.myCo.com:1234
```

The class UserProfile defines the basic information stored by the Account Manager for each user.

```
class UserProfile{
   AuthInfo authInfo;
   String userESURL;
   String userInformation;
   String userType;
   ProfileAttributeSet preferences;
   byte[] pse;
}
```

The userESURL is the ESName of the representing the user's Account Resource this is a Protection Domain. This ESName is bound to the user's Protection Domain in a Name Frame created by the Account Manager (rather than the default Name assigned to the Protection Domain in the e-speak Core's root Name Frame). Note that this ESName also includes the host and portNumber of the user's "home e-speak Core". An example of a userESURL is:

```
es://myhost.myco.com:12345/Core/AccountManager/myhost.myco.com:1234/myName
```

When the Account is registered a Protection Domain is created and registered in the Base Account Vocabulary (see section"Base Account Vocabulary" in Chapter 5, "Vocabularies") with attributes from AuthInfo and UserProfile. This means the Account Resource (Protection Domain) can be discovered using attribute based lookup, just like any other e-speak Resource.

UserType and UserInformation are arbitrary strings that can be assigned by an application. These are defined in the BaseAccount Vocabulary, so they can be used to lookup Users.

The PSE is opaque, the byte array is not interpreted by the e-speak Core.

The <ProfileAttributeSet preferences> field is a set of name, value pairs defined as follows.

```
class ProfileAttributeSet{
   AttributeSet attrs;
   String format;
}
```

If the format string is set to "VOCAB", the AttributeSet attrs is defined in a vocabulary specifed in the attrVocab field of the AttributeSet (see Chapter 3, “Resource Descriptions, Resource Specifications and Resource Types”), otherwise the format string is set to "NONE" and the AttributeSet attrs contains an arbitrary set of name-value pairs that are not necessarily be valid in any vocabulary.

The ProfileAttributeSet contains secret information. The intent is that this information should not to be visible to any application other than the one that registered the account..

```
public interface AccountManagerInterface {

   public String registerUser(UserProfile up)
      throws PermissionDeniedException, StaleEntryAccessException,
      NameNotFoundException;

   public boolean unregisterUser(AuthInfo authInfo,
         String accountName)
      throws PermissionDeniedException, StaleEntryAccessException,
      NameNotFoundException;

   public boolean authenticateUser(AuthInfo authInfo)
      throws PermissionDeniedException, StaleEntryAccessException,
      NameNotFoundException;

   public UserProfile getUserProfile(AuthInfo authInfo,
         String accountName)
      throws PermissionDeniedException, StaleEntryAccessException,
      NameNotFoundException;

   public boolean setUserProfile(AuthInfo authInfo, UserProfile up)
      throws PermissionDeniedException, StaleEntryAccessException,
      NameNotFoundException;

   public String[] getAllUsers()
```

```
        throws ESInvocationException;

    public boolean addDescription(AuthInfo authInfo,
            String accountName, AttributeSet as)
        throws ESInvocationException;

    public String getUserESURL(String accountName)
        throws ESInvocationException;
}
```

The function getAllUsers returns a list of the ESNames (in stringified form) of the Account Resource (Protection Domains) of all registered users.

The function addDescription is used for adding a new AttributeSet to the user's Account Resource (Protection Domain). This can be in any vocabulary, not just the Base Account Vocabulary.

The accountName parameter in getUserProfile and getUserESURL is must match the userName in the AuthInfo of the intended account.

The function getUserESURL returns a String corresponding to the ESNames(URLs) of the user's Account Resource (Protection Domain).

The user Interface

This is not implemented in the current release

```
interface UserInterface {
    public String getDescription()
        throws PermissionDeniedException, NameNotFoundException;

    public AttributePropertySet getProperties()
        throws PermissionDeniedException, NameNotFoundException;

    public void mutateProperties (AttributePropertySet props)
        throws PermissionDeniedException, NameNotFoundException;
```

```
}
```

Appendix: Method Names

In messages sent to Core-managed Resources (xref to communications chapter). The method is identified by a string. The following strings are used.

```
AccountManagerInterface
PF_REGISTERUSER
PF_UNREGISTERUSER
PF_AUTHENTICATEUSER
PF_GETUSERPROFILE
PF_SETUSERPROFILE
PF_GETALLUSERS
PF_ADDDESCRIPTION

ConnectionManagerInterface
OPENCONNECTION
GETCONNECTIONS
CLOSECONNECTION
CLOSECONNECTIONFROMREMOTE

CoreManagementInterface
PING
GETCLIENTCONNECTIONS
STOPSERVINGOUTBOX
STOPSERVINGINBOX
STARTSERVINGOUTBOX
STARTSERVINGINBOX
REMOVEPROTECTIONDOMAIN
DENYNEWCLIENTSESSIONS
ACCEPTNEWCLIENTSESSIONS
GETTOTALMEMORY
GETFREEMEMORY
START_JVM_GC
STOP_JVM_GC
SET_JVM_GC_INTERVAL
GET_JVM_GC_INTERVAL
IS_JVM_GC_RUNNING
START_SCAVENGER
STOP_SCAVENGER
GET_SCAVENGER_STATS
SET_NUM_STATS


FinderInterface
FIND
FINDNEXT
```

```
MailboxInterface
ISCONNECTED
CONNECT
DISCONNECT
RECONNECT

ManagedServiceIntf (implemented by Core management resource)
GETNAME
GETDESCRIPTION
GETOWNER
GETUPTIME
GETVERSION
GETERRORCONDITION
GETSTATICINFO
COLDRESET
WARMRESET
START
STOP
SHUTDOWN
REMOVE
GETSTATE
GETVARIABLEENTRIES
GETVARIABLENAMES
GETVARIABLEENTRY
SETVARIABLE
GETRESOURCEENTRIES
GETRESOURCENAMES
GETRESOURCEENTRY
SETRESOURCE

NameFrameInterface
LOOKUP
ISBOUND
BIND
REBIND
UNBIND
RENAME
COPY
ADD
SUBTRACT
LISTNAMES
LISTBINDINGS
NEW_SUB_FRAME

ProtectionDomainInterface
SWITCHPD
GETDEFAULTFRAME
SETDEFAULTFRAME
```

```
GETQUOTAINFO
SETQUOTA
NEW_PROTECTION_DOMAIN

RemoteResourceManagerInterface
EXPORTRESOURCE
IMPORTRESOURCEFROMMSG
IMPORTRESOURCE
EXPORTRESOURCEFROMMSG
UNEXPORTRESOURCE
UNEXPORTRESOURCEFROMMSG
UPDATEEXPORTEDRESOURCE
UPDATEEXPORTEDRESOURCEFROMMSG
UPDATEIMPORTEDRESOURCE
UPDATEIMPORTEDRESOURCEFROMMSG
EXPORTONCONNECTING

RepositoryViewInterface
ADD
REMOVE
CONTAINS
CLEAR
ADD_ELOOKUP
REMOVE_ELOOKUP

ResourceContractInterface
REGISTERRESOURCE
GETVOCABULARIES

ResourceFactoryInterface
REGISTER_RESOURCE

ResourceManipulationInterface
UNREGISTER
GETESUID
SETRESOURCEOWNER
GETRESOURCEOWNER
GETRESOURCEPROXY
SETRESOURCEPROXY
GETRESOURCECONTRACT
GETPUBLICRSD
SETPUBLICRSD
GETPRIVATERSD
SETPRIVATERSD
GETRESOURCEDESCRIPTION
SETRESOURCEDESCRIPTION
GETEVENTCONTROL
SETEVENTCONTROL
ISEXPORTEDBYVALUE
SETEXPORTTYPE
GETQUOTA
GETMETADATAMASK
SETMETADATAMASK
```

```
GETRESOURCEMASK
SETRESOURCEMASK
GETOWNERPUBLICKEY
SETOWNERPUBLICKEY
GETSERVICEID
SETSERVICEID
ISPERSISTENT
ISTRANSIENT
SETTRANSIENT
SETPERSISTENT
GETURL
GETTYPE

UserInterface (not implemented in the current release)
GETDESCRIPTION
GETPROPERTIES
MUTATEPROPERTIES

VocabularyInterface
GETDESCRIPTION
GETPROPERTIES
MUTATEPROPERTIES
```

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

The e-speak Core ships with one Basic Vocabulary preloaded. It is expected that the Basic Vocabulary are always 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.

Building a Vocabulary

A Vocabulary is built the same way as any other resource, using the Resource Factory. The request specifies an `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 3.

Table 3 Components of an attribute property

Type	Field	Meaning
String	name	Attribute name
String	description	Human-readable description
Value type	valueType	See Table 4 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 4.

Table 4 Supported value types

Data type	Designator	Matching rules	Operations
Big decimal	"BigDecimal"	eq, ne, lt, le, gt, ge	+, -, *, /
Boolean	"Boolean"	eq, ne	AND, OR
Byte	"Byte"	eq, ne	
Byte array	"ByteArray"	eq, ne	

Table 4 Supported value types (Continued)

Data type	Designator	Matching rules	Operations
Char	"Char"	eq, ne	+ (concatenate, returns String)
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	+ (concatenate)
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 can 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 an appropriate certficate can create a new Vocabulary using the Resource Factory (xref to Resource Factory in Core Managed Resource Chapter). The Client must provide 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 5.

Table 5 Base Vocabulary definition

Attribute name	Value type	Comments
Name	String	
Type	String	
ResourceSubtype	String	
ESGroup	String	
ESCategory	String	
Description	String	
KeyWords	String	Multivalued
Version	String	
ESDate	Date	“YYYY-MM-DD”
ESTime	Time	“HH:MM:SS”
ESTimeStamp	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`.

The description string for the Base Vocabulary is: "E-speak base vocabulary".

Base Account Vocabulary

The Base Account Vocabulary is also available at startup. It is used for discovering user accounts.

Table 6 Base Account Vocabulary

Attribute name	Value type
UserName	String
UserInfo	String
UserType	String
UserLocation	String
UserESURL	String

The above attributes match the fields defined in AuthInfo and UserProfile. For more information, see "The Account Manager Resource" section in Chapter 4, "Core-Managed Resources."

The description string for the Base Account Vocabulary is: "E-speak base user vocabulary".

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 returns true. If the translator cannot perform the translation, or if it does not understand either of the Vocabularies, it returns false. The translation is done by:

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

which returns a Search Recipe in the specified Vocabulary.

Chapter 6 Access Control

The basis of e-speak access control is a Public Key Infrastructure (PKI). In the remainder of this chapter we assume the reader is familar with the principles of PKI, sometimes also known as Public Key Cryptography. There are many texts to which the reader can refer [see for example *Schneier*, *Pfleeger*, *Stallings*].

All entities in e-speak (users, services, cores etc) are identified by public keys. To authenticate an entity we verify it knows the private key corresponding to the given public key. No entity should ever intentionally share its private key or give anybody access to the private key.

The means by which a private key is protected is implementation dependent: not part of the architecture. It is very important that the private key is held securely, so it is not unintentionally made available to others. In the default implementation the private key is encapsulated inside a Private Security Environment (PSE) object, described below.

Any entity can create a key-pair. Provided the private key is kept secret, the key-pair will be unique to that entity. However, having a key-pair gives you no power in the system. It is necessary also to have Certificates stating the access rights issued to your public key.

To decide whether to honor an incoming request a service must decide if the accompanying certificate (or certificates) grant access rights for the request. Before that, it verifies that the sender of the request knows the private key corresponding to the public key in the certificate to which the access rights have been given (formally this is the Subject of the certificate). It does this by a cryptographic protocol described in Chapter 7, “Communication” .

Finally before honoring the request, the service must verify that it trusts whoever issued the certificate. It does this by verifying that the certificate has been signed by an entity that it trusts.

Comparison with X.509 Certificates

The most common use of certificates is in X.509 based infrastructures to link an entity's name to its public key (technically the X.509 Distinguished Name). This is how certificates are used in the web. A drawback is that, typically, having used the certificate to verify the name, a service needs to consult an authorization database to determine the access to be granted.

E-speak certificates are more general than this. They are signed (authenticated statements) linking a public key to a Name or a Tag. (Certificates linking a Name to another Name also exist, and are described below.) The word "tag" distinguishes the field concerned from an X.509 "attribute", whose function is broadly similar. A Tag typically states an access right. Thus to make an access control decision a service does the following:

- Examines the tag in the certificate to see if it grants access
- Checks the entity making the request knows the corresponding private key
- Verifies the certificate has been issued (is signed by) an entity it trusts

X.509 name certificates are issued by entities called Certificate Authorities. To avoid confusion with this, in e-speak we refer to entities issuing certificates as Issuers. E-speak Issuers can issue either Name or Attribute certificates.

Another feature in e-speak not found in X.509 is that it implements a split trust model. An entity does not have to trust all Issuers equally. It need not trust any given Issuer at all. Those it does trust, it can only trust to issue certificates granting access to a subset of its operations.

Conversely, issuing certificates in e-speak is not a reserved prerogative: anyone can do it. Whether or not the certificate will grant access to any Resource depends on whether the Resource Handler trusts the Issuer for the service in question. The list of which Issuers are trusted for what is called Trust Assumptions. This is discussed later in this chapter.

Derivation from SPKI

E-speak implements the Simple Public Key Infrastructure (or SPKI) [see *RFC 2692-2693*]. In addition to the properties already described, SPKI specifies a structure and set of operations on Tag and Name certificates. These are used to parse and process the certificates when making access control decisions. The processing and access control is discussed later in this chapter. Certain tags (e-speak tags) are defined that will be checked explicitly by the e-speak infrastructure before an access is authorized. However, applications can choose to use any syntactically valid SPKI tag. E-speak will check that certificates containing such tags are valid, but will not use them for an access control decision. The application will have to interpret these non e-speak tags when making access control decisions. Core managed Resources will ignore non e-speak tags.

Certificate Management

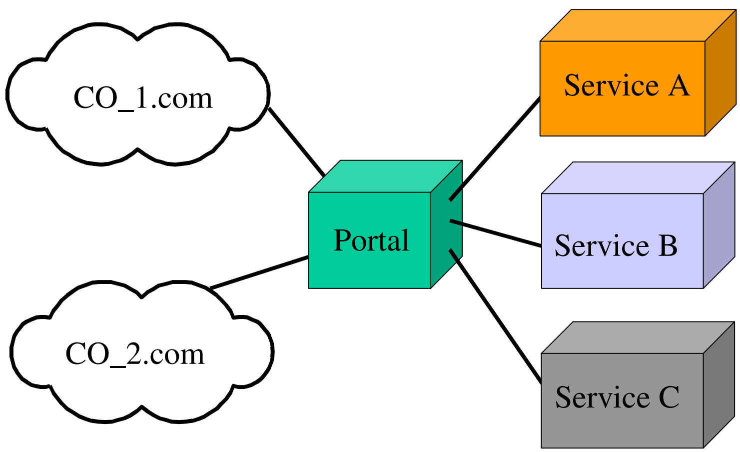
The process by which an Issuer decides to issue a certificate granting access rights to an entity is implementation dependent and therefore not part of the architecture. The general process would be for entities to register either with some Issuer or with a separate Registration Authority (RA). For registration, the entity may need to provide credentials such as credit card number, social security number, bank details, employee ID, user id., and full name. Once the registering body is satisfied it will issue a certificate, or give instructions for issue. The registering body can be fully automated, or can queue registrations for human inspection.

A given entity can have several certificates that have been issued to it. If no strategy is adopted to structure and manage certificate issuing then there can be very large numbers of certificates required. Administrators and operators would find it difficult to run e-speak systems, and operations such as access revocation would be extremely hard. Hence we discuss and recommend certain strategies for certificate management. These are based around familiar concepts such as user-groups (or roles), found in several common operating systems. These are not part of the architecture. The management strategy practised must reflect the business requirements of the deploying organization.

Anybody can create a key-pair in e-speak and then register to get an Issuer to issue certificates to the public key. There is no notion of a centralized, all powerful, trusted Certificate Authority. Instead entities choose which Issuers they trust for what. Authentication in e-speak relies on proof of knowledge of the private key: there is no centralized authentication service. Hence the e-speak security architecture is a global, fully distributed and single sign-on.

Example Of Certificate-based Security (Informational)

Consider the diagram below. Two large ".com" companies are accessing a portal to use services provided by the portal. For simplicity we have shown only 3 services.



The data held by the services can be sensitive, so both companies would like to be sure that their employees are accessing the correct portal and services. In addition, having made arrangements for access to the portal (and paid fees), both companies might prefer to be responsible for managing their own lists of employees and control who can access the portal's services.

From the portal's point of view, it probably only wants to deliver services to paying customers and only to deliver those services that each customer has paid for.

Suppose CO_1 has done a deal with the portal to access services A, B and C, and CO_2 has done a deal to access service A and C only. Lets further suppose that CO_1 and CO_2 are each running an Issuer , called Issuer1 and Issuer2 respectively. The portal configures A and C to trust both Issuer1 and Issuer2; it configures B to trust Issuer1 only. Then CO_1 and CO_2 can issue certificates to each of their employees. CO_1's certificates will be honored at A, B and C, but CO_2 certificates will only be honored at A and C.

Each time a service sees a certificate from either company that grants access, it increments the bill for that company. This leaves each company in control of who among its employees gets access to the services for which it has paid. Each company is in control of revocation (e.g. if the employee leaves). In addition the portal can immediately revoke access to an entire company, by removing the company's Issuer from the list of trusted Issuers.

Each company may want to make sure that their employees are accessing only genuine services. To do so CO_1's Issuer issues a Tag certificate binding each of service A, B and C's public keys to a tag such as : "CO_1 approved". It must then ensure that its employees configure their clients to check for this tag before accessing the service. Similarly CO_2's Issuer issues a Tag certificate to services A and C conferring an attribute that is meaningful to CO_2.

Note that this requires very little authorization data to be held and managed by the portal. It only needs to remember the public keys of CO_1 and CO_2's Issuer. If access control were based on authenticating a name and mapping accesses to that name, then the portal would have to keep a list of all employees in each company that can access any of the services, and which accesses are allowed for each name - much more data to manage and maintain.

Authorization Data

The informal structure of an authorization certificate is:

Certificate header: a constant field starting " (cert "

Issuer: the public key of the Issuer

Subject: the public key or the name of the entity granted the certificate

An optional "delegation" field

Tag: Details of what is authorized

Optional validity qualification and comment.

In this structure, it is the tag that requires most attention by client applications.

Tags

As e-speak implements SPKI, any valid SPKI tag can appear in a certificate. The BNF for SPKI is given in the *SPKI BNF Format s*ection. In this section we give some example SPKI tags that can appear in certificates and explain the BNF for a tag.

E-speak defines a set of standard tabs (see "E-speak Authorization Tags" on page 83), that is checked automatically by the infrastructure. The examples given in this section are not standard e-speak tags, so they would have to be checked explicitly by the application.

An SPKI tag is an S-expression, that is a list enclosed in matching "(" and ")".

The BNF for a tag is:

```
<tag> = "(" "tag" <tag-expr>* ")" ;
<tag-and> = "(" "*" "and" <tag-expr>+ ")" ;
<tag-expr> = <byte-string> | <tag-simple>
      | <tag-prefix> | <tag-range>
      | <tag-set> | <tag-and>
      | <tag-star> ;
<tag-simple> = "(" <byte-string> <tag-expr>* ")" ;
<tag-prefix> = "(" "*" "prefix" <byte-string> ")" ;
<tag-range> = "(" "*" "range" <range-ordering> <low-lim>? <up-
lim>? ")" ;
<tag-set> = "(" "*" "set" <tag-expr>* ")" ;
<tag-star> = "(" "*" ")" ;
```

```
<tag-and> = "(" "*" "and" <tag-expr>+ ")" ;1
<range-ordering>= "alpha" | "numeric" | "time" | "binary" | "date"
;
<up-lim> = <lte> <byte-string> ;
<low-lim> = <gte> <byte-string> ;
<lte> = "l" | "le" ;
<gte> = "g" | "ge" ;
```

A tag is a list of lists, with each list denoted by brackets. In its simplest form (tag-simple), a tag is simply composed of byte-strings. The access control machinery must interpret the meaning of the tag when making an access control decision. The following examples are adapted from SPKI examples previously published as Internet drafts. An example form for tags applying to a file system is:

```
(tag (files <pathname> <access> ))
```

An instance of such a tag is:

```
(tag (files //ftp.espeak.net/pub/EspeakArch.pdf read))
```

A client presenting a certificate containing the above tag is allowed read access to EspeakArch.pdf (assuming authentication was successful).

<tag-set> field

Groups of permissions can be granted using the "tag-set" form:

```
(tag (files //ftp.espeak.net/pub/EspeakArch.pdf (* set read
write))
```

This grants read and write acess to the file.

<tag-prefix> field

A set of permissions having a common prefix can be granted using the "tag-prefix" form:

```
(tag (files (* prefix //ftp.espeak.net/pub/ ) (* set read write))
```

This grants read and write access to any file under the pub directory.

---

1 The <tag-and> field is an e-speak specific extension to SPKI.

<tag-star> field

The "tag-star" form stands for the set of all valid s-expressions and byte strings.

```
(tag (files (* prefix //ftp.espeak.net/pub/ ) (*))
```

The above tag grants all permissions on all files under pub.

```
(tag (files (*) (*))
```

Note that trailing "(*)" can be omitted. So the above is equivalent to:

```
(tag (files))
```

The two last tags both grant all permissions on all files anywhere.

```
(tag (*))
```

The above grants all permissions on anything. This might look as though it is conferring a lot of power. However, e-speak has a split trust model: the issuer of the certificate containing this tag might only be trusted by a single Resource.

<tag-and> field

The "tag-and" form is not used in writing a certificate. It expresses the authorizations conferred by the set of tags in the following expression. This is analagous to a set-intersection operation: the authorization resulting from a "tag-and" form will be that satisfying each and every one of the following tags. So it is more restrictive than that of any of the tags on its own.This form is used internally when authorization depends on more than one certificate. The process is described under *Tag Intersection.*

<tag-range> field

The "tag-range" form stands for the set of all byte strings lexically (or numerically) between the two limits. The ordering parameter (alpha, numeric, time, binary, date) specifies the kind of strings allowed. For example, the following tag indicates the authorization to issue purchase orders whose value is less than $5000.

```
(tag (purchaseOrder (* range numeric le 5000 )))
```

The following indicates a salary between $50,000 and $100,000

```
(tag (salary (* range numeric  ge 50000 le 100000)))
```

E-speak Authorization Tags

E-speak tags are valid SPKI tags that will be checked by the infrastructure. For core-managed Resources the e-speak core will check that a valid certificate is presented containing a tag that authorizes the operation. For non-core-managed Resources, it is assumed that the resource handler will check there is a valid certificate containing a tag that authorizes the operation. However, the e-speak core cannot enforce this; the resource handler is responsible for Resource security.

E-speak tags that authorize access to services have the following form:

```
(tag (net.espeak.method <interface> <method> <serviceId>))
```

The following tag authorizes the "stop" operation in the serviceManagementInterface for the identified Resource.

```
(tag (net.espeak.method ServiceManagementInterface stop
xxxxyyyyzzzz))
```

The forms tag-star, tag-prefix, tag-set and tag-range can all be used within an e-speak tag. So the following tag authorizes operations on the ServiceManagement interface in two different Resources.

```
(tag (* set (net.espeak.method ServiceManagementInterface stop
xxxxyyyyzzzz)
          (net.espeak.method ServiceManagementInterface (* set
stop start) aaaabbbbbccccc)))
```

The long strings at the end represent the ServiceId, described below.

The following form authorizes every method on every ServiceManagementInterface on Resources that trust the issuer.

```
(tag (net.espeak.method ServiceManagementInterface (*)  (*) ))
```

Or equivalently:

```
(tag (net.espeak.method ServiceManagementInterface ))
```

The following authorizes any method within the given interfaces (core managed Resources) on any object:

```
(tag (
   (net.espeak.method
      (* set
         ResourceFactoryInterface
         ResourceManipulationInterface
         ManagedServiceInterface
         CoreManagementInterface
         NameFrameInterface
               )
   )
)
```

Lets assume we have an interface called "file" and the serviceID is set to a notional path name (a non default value). The following tag authorizes the read operation on all files below the pub directory.

```
(tag (net.espeak.method file read (* prefix es.espeak.net/pub/)))
```

If serviceId's are set to ordered numerical or alphabetical values, then the tag-range form can be useful in the <serviceId> portion of a tag.

Currently we have only defined e-speak tags for the Network Object Model. This assumes a set of services with one or more interfaces, each interface containing one or more methods. The programming of J-ESI and the interaction with core-managed Resources follow this model. However, e-speak can support other programming models: an XML document exchange model and a direct messaging model have both been implemented. The tags used by these models are part of the programming models. There are not part of the core architecture, since the core does not need to interpret them: the resource handlers do it.

Masks

The Mask controls which operations will have security disabled. The Mask for a Resource is part of the metadata for that Resource (see Chapter 3, “Resource Descriptions, Resource Specifications and Resource Types”, section.“ResourceSpecification”). If an operation appears in a Resource Mask, anybody can invoke that operation.

Masks are specified as tags. The basic method tag format is

```
(net.espeak.method <interface name> <method name>)
```

In the metadataMask the interface name is the core interface being specified, and the method name is the operation in that interface. For metadata this will be be the ResourceManipulationInterface, and the method name one of its methods. For the resourceMask the interface name will be one of the interfaces supported by the Resource.

In the resource mask for an external Resource the interface name is the fully-qualified name of the interface class. For a Core-managed Resource, the interface name is the not quallified, so we just have “NameFrameInterface” and “ProtectionDomainInterface” etc. The method name is the name of the method in the interface, plus the concatenated argument types. This allows overloaded methods to be distinguished.

The metadata mask is used by the in-core metaresource when performing metadata operations. The resource mask is passed to the service handler by the core for the service handler to use when performing operations on the service itself.

The masks are completely general tags, so the mask tag itself, or any of its fields, can use the tag matching features such as sets, prefixes and ranges. The interface and method names, for example, do not have to be string literals, they can be sets or prefixes.

This tag masks method foo in interface net.espeak.examples.ExampleIntf:

```
(net.espeak.method net.espeak.examples.ExampleIntf foo)
```

This tag masks method foo in interface net.espeak.examples.ExampleIntf and method bar in interface net.espeak.examples.Example2Intf

```
(net.espeak.method (*set
```

```
      (net.espeak.examples.ExampleIntf foo)
      (net.espeak.examples.Example2Intf bar)
      )
)
```

This tag masks all methods beginning with foo:

```
(net.espeak.method net.espeak.examples.ExampleIntf  (* prefix foo))
```

This tag masks methods foo and bar:

```
(net.espeak.method net.espeak.examples.ExampleIntf  (* set foo bar))
```

Methods with prefix foo or bar:

```
(net.espeak.method net.espeak.examples.ExampleIntf
  (* set (* prefix foo) (* prefix  bar)))
```

All methods in the interface:

```
(net.espeak.method net.espeak.examples.ExampleIntf )
```

This is equivalent to

```
(net.espeak.method net.espeak.examples.ExampleIntf  (*))
```

since missing trailing elements match anything.

All methods foo in InterfaceA and bar in InterfaceB:

```
(* set (net.espeak.method InterfaceA foo)
       (net.espeak.method InterfaceB bar))
```

All methods:

```
(net.espeak.method)
```

or simply

```
(*)
```

Service Identity

The serviceId field in the Resource specification (see Chapter 3, “Resource Descriptions, Resource Specifications and Resource Types”) can contain any valid SPKI tag-expression, defined as a "tag-expr" in the BNF (see "SPKI BNF Formats" on page 106). This tag-expression can be set by anybody with a certificate, from an Issuer trusted by the MetaResource, authorizing setServiceId in the MetaResource. The serviceId field is delivered to the resource handler with each message for the Resource.

The service identity is used by the resource handler when verifying standard e-speak tags (see "Verifying tags and tag intersection" on page 98).

Default ServiceId

A default assignment is made by the core when it encounters a standard e-speak authorization tag without an authorized service id. The format is:

<serviceId> = "(" "net.espeak.service" <service class> <service name> <unique id> ")"

<service class> is set to the first available value of:

1.) The name attribute in the Resource specification contract, if any.

2.) The contract type, if any.

3.) A 64-bit random no. if neither of the above exists.

<service name> is set to:

1.) The "name" in the Resource description

2.) A 64-bit random no., if 1) is not found.

<unique id> is a 64-bit random no.

A secure random number generator should be used, so that the probability of accidental authorization when the default has been used will be infinitesimal..

Advantages of ServiceIds

The serviceId is intended for use by applications to identify services without using the Resource name or access path (ESNames). This decouples authorization from resource naming and has several advantages:

- Service ESNames can be changed without affecting authorization
- Authorization can be revoked by changing a service's identity, without changing its ESName
- In a replicated service replicas can all have the same identity
- Tag patterns (the "tag-star" form) can be used effectively, limiting the number of certificates issued

None of this is possible using ESName for service identity.

Protection of ServiceIds

Service identity plays a crucial role in authorizing access to a service (see "Verifying tags and tag intersection" on page 98). It is essential that the setServiceID operation source is protected, so that a valid certificate is required to invoke it.

Names: Userids, Groups....

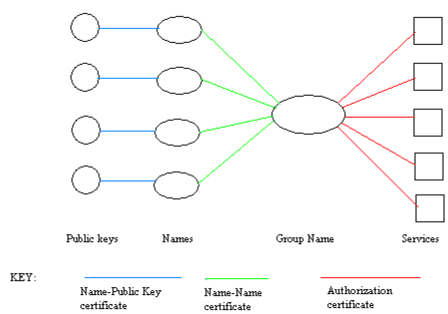
E-speak also supports SPKI name certificates, of two types. In the first place, an Issuer can issue a certificate that binds a public key to a name. This has similarities to X.509 certificates which bind a public key to an X.509 Distinguished Name. (A Distinguished Name is a name in a special format, distinguishing it globally from any other name.)

SPKI name certificates do not restrict the syntax of the name, other than requiring them to be a bytestring. Instead, names are scoped by the public key of the issuer. Referring back to our example (see "Example Of Certificate-based Security (Informational)" on page 78), both CO_1 and CO_2 could have an employee named John Doe. Assuming each company had an Issuer that issued name certificates binding these names to public keys, the fully qualified name for each John Doe is:

```
Public key of CO_1 issuer: John Doe
Public key of CO_2 issuer: John Doe
```

Hence the portal (and anybody else) would have no difficulty distinguishing between the two instances of John Doe.

The second type of name certificate binds a name to a name. For example we might want to bind John Doe to the name "users". This kind of certificate confers membership of the group "users" on the userid John Doe. It can be used to build a role- or group- based security model, such as represented below (see "Managing certificates (informational)" on page 103).



The algorithm which relates a public key to an authorization in this case is described below (see "Name Reduction" on page 96). [See also *RFC 2693*]

Certificate Structure

The two kinds of certificates in e-speak are Authorization Certificates that bind a tag to a public key or a name and Name Certificates that bind a name to a public key or a name. The following sections describe and explain the BNF which specify these types.

Some general features of the specification are:

Nearly every field begins with its name as a literal string.

* is used to mean "0 or more cases of the preceding field"

* is also used to mean "anything valid" in the tag-star field described above

+ means "one or more instances of the preceding field"

? means the preceding field is optional

"uris" means a field with one or more URI's.

The full SPKI BNF is given at the end of this Chapter (see "SPKI BNF Formats" on page 106).

Authorization Certificates

The format for an authorization certificate is:

```
<cert> = "(" "cert" <version>? <cert-display>?
<issuer> <issuer-info>?
<subject> <subject-info>?
<deleg>?
<tag>
<valid>?
<comment>? ")" ;
```

The optional <version> field defines the <version> of the certificate. The optional <cert-display> field is designed to provide hints for display. Neither of these fields is used in the current version of e-speak; the parser will ignore them.

Issuer field

The <issuer> field is the public key of the Issuer issuing the certificate; it is defined as follows.

```
<issuer> = "(" "issuer" <principal> ")" ;
<principal> = <public-key> | <hash-of-key> ;
<hash-of-key> = <hash> ;
```

```
<hash> = "(" "hash" <hash-alg-name> <hash-value><uris>? ")" ;
<hash-alg-name> = "md5" | "sha1" | <uri> ;
<hash-value> = <byte-string> ;
<public-key> = "(" "public-key" <pub-sig-alg-id> <s-expr>*
<uris>? ")" ;
<pub-sig-alg-id>= "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-
pkcs1" | "dsa-sha1" | <uri> ;
```

The <issuer-info> field is intended in SPKI to provide a list of one or more URIs for certificates from which the Issuer derives its authority to issue the certificate. This is to support delegation: one Issuer can issue a certificate to another Issuer with the delegation field present. (It is the literal "propagate".) Suppose a service trusts the first Issuer directly, and not the second Issuer. If a client presents a certificate issued from the second Issuer, the service will need to see the delegate certificate conferring the privilege on the second Issuer before it authorizes access. The URIs would specify the location of delegate certificates. This is not used in the current version of e-speak. Instead, the required supporting certificates are obtained during the Session Layer handshake (see Chapter 7, “Communication”). The parser will ignore this field.

The "hash-alg-name" and "pub-sig-alg-id" fields identify algorithms used for hashing and for signature verification - usually the literal abbreviated algorithm names given. The "uri" alternative in each case could be used to give a URI of some other algorithm.

Subject field

The <subject> field denotes the entity to which the certificate is issued.

```
<subject> = "(" "subject" <subj-obj> ")" ;
<subj-obj> = <principal> | <name> | <object-hash> ;2
<principal> = <public-key> | <hash-of-key> ;
<name> = <relative-name> | <full-name> ;
<relative-name> = "(" "name" <byte-string>* ")" ;
<full-name> = "(" "name" <principal> <byte-string>* ")" ;
```

The <subject> is either a public key, a name or the hash of an object. If the subject is a public key, then the entity presenting the certificate must prove possession of the corresponding private key before authorization is granted. This uses the cryptographic protocols described in Chapter 7, “Communication”.

If the <subject> is a name, then authorization is granted to the entity that has a certificate binding that name to its public key (see "Name Certificates" on page 94). Several certificates can be required to prove this. For example, the authorization certificate may be issued to a name such as "users". The name "users" can be conferred on another name "John Doe", a real world person. So to get authorization three certificates are needed:

- A certificate binding John Doe’s public key to his name
- A certificate binding John Doe to the name "users"
- A certificate granting the authorization to "users"

John Doe needs to prove possession of the private key corresponding to the public key in the first certificate using the protocols described in Chapter 7, “Communication”.The algorithm for name reduction to arrive at the certificate binding the name to a key and handling compound names such as <public key: "John Doe" "Favorite People"> is described below (see "Name Reduction" on page 96).

<relative-name> and <full-name>

A <relative-name> is assumed to have been issued by the Issuer whose public key is in the issuer field. In contrast, a <full-name> is a fully qualified name, explicitly scoped by the public key of the Issuer which conferred it. This principle is extended for names "issued by" names [see *Compound Names* below]. The use of qualified names allows any Issuer to issue certificates to names that have been issued by any other Issuer.

---

2 The definition here departs from SPKI slightly. SPKI defines <subj-obj> = <principal> | <name> | <object-hash> | <keyholder>; E-speak does not support <keyholder>. If the parser encounters a keyholder field, it will throw an exception. Which exception depends on the point from which it is invoked. One of the e-speak exceptions specified in Chapter 8, “Exceptions” will be thrown.

<object-hash> (Informational)

An <object-hash> is intended for the issue of authorization certificates to entities such as files and executables. The tag in such a certificate might describe a property of the file or the executable. This is not used in the current version of e-speak. The parser will ignore the field.

<subject-info> (Informational)

The optional <subject-info?> field is defined as follows.

```
<subject-info> = "(" "subject-info" <uris> ")" ;
```

The intent of this field is to provide a list of URIs that provide information about the subject. For example if the subject is a hash of a key, it might provide the location of the key being hashed. If the subject is a name, it might provide the location of the name certificates. This field is not used in the current version of e-speak. The parser will ignore it.

Delegation field

The optional <deleg> field is defined as follows.

```
<deleg> = "(" "propagate" ")" ;
```

If this field is included in a certificate, then the subject is authorized to delegate the authorization specified in the certificates tag. The subject does this by issuing certificates containing the tag, or a subset of the tag's privileges. This is discussed further under *Delegation*.

Validity field

The optional <valid> field is defined as follows.

```
<valid> = <valid-basic> <online-test>* <restrictions> ;
<valid-basic> = <not-before>? <not-after>? ;
<not-after> = "(" "not-after" <date> ")" ;
<not-before> = "(" "not-before" <date> ")" ;
<date> = <byte-string> ;
```

If the valid field is missing, the certificate is assumed to be valid without constraints. The fields <online-test> and <restrictions> defined in [*Working Draft*] are not supported in the current version of e-speak; the parser will ignore them. The <valid-basic> field is used to support time-based revocation, as described under *Certificate Revocation*.

A <date> field is an ASCII byte string of the form:

```
YYYY-MM-DD_HH:mm:SS
```

This is always UTC. For example, "1997-07-26_23:15:10" is a valid date. So is "2001-01-01_00:00:00". "MM" is a two digit integer in the range 1 to 12; "mm" and "SS" are two-digit integers in the range 0 to 59.

The optional comment field is defined as follows:

```
<comment> = "(" "comment" <byte-string> ")" ;
```

Anything is this field is intended to provide information to humans. It is ignored by e-speak.

Name Certificates

The format for name certificates is:

```
<name-cert> = "(" "cert" <version>? <cert-display>?
              <issuer-name> <issuer-info>?
              <subject> <subject-info>?
              <valid> <comment>? ")" ;
<issuer-name> = "(" "issuer" "(" "name" <principal> <byte-string>
")" ")" ;
<principal> = <public-key> | <hash-of-key> ;
```

The characteristic feature of a name certificate is the the <issuer-name> field. This defines the issuer of the certificate plus the name of the certificate holder. "Issuer-name" does <u>not</u> mean "name of issuer". The byte-string in this field is the certificate holder's name, and the <principal> is the issuer's public key, or a hash of it. In the latter case, there can be a following field containing a URI of the full key, but this is not currently used or supported in e-speak.

Public Keys

An example of a public key is:

```
(public-key
    (rsa-pkcs1-md5
     (e #03#)
     (n
      |ANHCG85jXFGmicr3MGPj53FYYSY1aWAue6PKnpFErHhKMJa4HrK4WSKTO
      YTTlapRznnELD2D7lWd3Q8PD0lyi1NJpNzMkxQVHrrAnIQoczeOZuiz/yY
     VDzJ1DdiImixyb/Jyme3D0UiUXhd6VGAz0x0cgrKefKnmjy410Kro3uW1|
)))
```

The long string between "|" 's is a number encoded in base64 notation for relative brevity. This is a feature of BNF advanced syntax [see *BNF Notation* below].

Such items can be written in certificates, but in the following text, we use "PK XXX" as an abbreviation for "XXX's public key".

Example

Taking the example (see "Names: Userids, Groups...." on page 88), the following certificate could be issued by CO_1's Issuer.

```
(cert         Certificate A
   (issuer (name (PK CO_1) "John Doe"))
   (subject (PK John Doe))
   (not-after "2001-01-01_00:00:00")
)
```

The underlining is referred to in the next paragraph.

Name Reduction

The objective of name reduction is to reduce the name that appears in a subject field to a single public key, a <principal>. Name reduction replaces the name in a subject field, by rewriting it with the subject field from the corresponding name certificate. It uses the fact that a fully qualified name in a subject field has the same format as <principal> <byte-string> in an issuer-name field. For example, given **Certificate A** above, suppose there is an authorization certificate:

```
(cert
Certificate B
   (issuer PK X)
   (subject (name (PK CO_1) "John Doe"))
   (tag (net.espeak.method CoreManagementInterface ))
)
```

The two underlined fields being the same, we can replace <subject> in B by <subject> in A, giving certificate C:

```
(cert
Certificate C
   (issuer PK X)
   (subject  (PK John Doe))
   (tag (net.espeak.method CoreManagementInterface ))
)
```

Compound Names

Suppose an Issuer called "Editor" issues a name certificate to "Foreign Desk"; and this entity in turn issues one to "Paris Correspondent". Each will have PK holder as its Subject. An authorization certificate could be made out as follows:

```
(cert
        (issuer PK Accounts)
   (subject (PK Editor) "Foreign Desk" "Paris Correpondent"
   (tag (Dinner_Expenses (*range le 200) (currency  FF))3
```

3 Not an e-speak tag.

```
)
```

The subject field is a Compound Name. Accessing the name certificates implied in the subject field from left to right, we replace this field successively by:

```
(subject  (PK Foreign Desk)  "Paris Correspondent)
(subject  (PK Paris Correspondent) )
```

- yielding a certificate which can be authenticated.

Name reduction is defined formally as part of the tuple reduction rules in [*SPKI theory*, *RFC 2693*]. This also includes an algorithm for combining validity fields. If the validity fields are dates (as in the current e-speak implementation), then informally we take the latest <not-before> date and the earliest <not-after> date. If the <not-after> date obtained in this way is before the <not-before> date, then the reduction has failed.

Wire format for certificates

The "on-the-wire" format for certificates is the BNF Canonical Syntax (see "SPKI BNF Formats" on page 106).

Delegation

E-speak supports SPKI delegation. If an Issuer is not trusted directly by the entity checking the authorization, its certificates cannot effectively authorize more than the delegate certificate authorizes. The SPKI certificate reduction rules [see *RFC2693 - AIntersect*] describe formally how this is enforced. Informally, it is done by intersecting the authorizations specified by all tags in the delegation chain, and taking the smallest validity period as decribed in the *Name Reduction* section.

Consider the following certificate.

```
(cert
   (issuer PK X)
   (subject PK Y)
   (propagate)
```

```
      (tag (net.espeak.method CoreManagementInterface ))
      (not-after "2000-10-01_00:00:00")
   )
```

Suppose Y now issues a certifcate to Z as follows.

```
   (cert
      (issuer PK Y)
      (subject PK Z)
      (tag (net.espeak.method ))
      (not-after "2001-01-01_00:00:00")
   )
```

Here Y is attempting to authorize more, for longer than was contained in the certificate issued to it by X.

Suppose an entity, checking that Z is authorized, trusts X directly, but not Y. The two certificates above form the delegate chain by which Z is obtaining its power. The entity intersects the two tags (as described below), combines the validity times (as described above) and rewrites the issuer field according to the reduction rules described in [*RFC2693*] to get the following certificate.

```
   (cert
      (issuer PK X)
      (subject PK Z)
      (tag (net.espeak.method CoreManagementInterface ))
      (not-after "2000-10-01_00:00:00")
   )
```

Hence it is not possible for Y's certificate to authorize more for longer than the original certificate granted to Y by X, from entities which don't trust Y directly.

Verifying tags and tag intersection

Tag verification is the process of determining whether the set of certificates presented contain the required authorization. SPKI tags define sets of authorizations. For example the following tag authorizes all methods on all instances of the CoreManagementInterface.

```
(tag (net.espeak.method CoreManagementInterface ))
```

So the above tag "contains" the following tag (xxxxyyyyzzzz is the serviceID).

```
(tag (net.espeak.method CoreManagementInterface  ping
xxxxyyyyzzzz))
```

Appending elements to the end of a tag reduces the set of authorizations specified. So:

```
(tag (net.espeak.method CoreManagementInterface ping))
```

specifies less than

```
(tag (net.espeak.method CoreManagementInterface ))
```

In the case of a delegation chain, where the successive certificates authorize:

1) services A, B, C

2) services B, C, D

3) services B, D, E -

the only service authorized will be B - the only member of the "intersection" of the three certificates.

Implementing Verification

In e-speak, each time an object receives a request to invoke a method, the security infrastructure will check that there is a certificate that contains the tag needed to invoke the operation. For Core-managed Resources the security infrastructure is contained in the core. For other Resources, it is part of the Resource. The security infrastructure is part of the current implementation of J-ESI, and clients can use the security infrastructure API's for their own resource handlers.

For example, if an attempt is made by a client to invoke the "ping" operation on a CoreManagementInterface, the infrastructure will check that there is a certificate that contains the tag (tag (net.espeak.method CoreManagementInterface ping xxxx-yyyyzzzz)), where xxxyyyzzz is the serviceID of the service being invoked.

For this to work the infrastructure must know the serviceID of the Resource. The serviceID is part of the Resource's metadata, and the core presents the serviceID with each request. It is trusted to present the correct serviceID.

For a certificate to authorize an operation we also have to check that the certificate is issued by somebody trusted to authorize the particular operation on the particular Resource [see *Trust Assumptions* section]. This means checking the public key of the issuer and the signature of the certificate. It is done automatically by the security infrastructure.

Authorization certificates can be issued to names as well as public keys. If a certificate issued to a name is presented that authorizes the operation, the name must be reduced to the public key of the invoker, as described in the *Name Reduction* section. The invoker's public key will be authenticated by the protocols described in Chapter 7, "Communication".

Authentication of Services (Informational)

In addition to the e-speak tags specified in the E-speak *Authorization Tags* section, a client or service can ask for application-specific tags to be checked, by invoking the security infrastructure APIs. Since no e-speak tags are specified for servers to present to clients, any authentication of the service by the client will be application-specific. For example a client might check for a tag identifying the Id. of a service, such as:

(tag (net.espeak.serviceID xxxxyyyyzzzz))

This means that the server will have to get a certificate issued to it containing this tag. See the *Certificate Issuers and Registration* section below.

The security APIs for checking application-specific tags are outside the architecture. They are application-specific, and no application-specific tags are supported for core-managed Resources.

Certificate Issuers and Registration (Informational)

There is no restriction in e-speak on who can issue a certificate. Anything that has a public key can do it. A certificate gets its power either from trust in the Issuer, or from a delegation chain down from a trusted Issuer. If the issuer is not trusted directly by a service and has no delegate certificate, its certificates will not authorize access to that service.

The processes of issuing a certificate and of deciding to issue one are application-specific: not part of the architecture. In some applications an entity can undergo a registration process whereby some real-world characteristics are verified (credit card numbers, social security numbers and the like). Registration can be fully automated, or it can involve human inspection.

Service Ids.

Problems can arise if services have the same service identity, either accidently or deliberately. For example, a service might use a fake serviceId and ask someone to issue privileges for that serviceId. The issuer would then think it was issuing privilege on the fake service, when in fact it was issuing privileges on the real service. To avoid these problems, anyone claiming ownership of a serviceId must be required to produce a certificate granting it to them. This prevents serviceId spoofing.

Unique service identies can be enforced by all Issuers knowing all previously issued service identites, or having the Issuer itself generate and issue a cryptographically secure and unique service identity in a certificate, or by relying on the service identities generated by e-speak, using a cryptographically secure random-number generator.

Note that sometimes we want to have the same identity for multiple services. For example, the services might be replicated. So, whether service identities are required to be unique and how this is enforced is not part of the architecture.

Trust Assumptions (Informational)

The basis for establishing trust assumptions is:

- Who you trust and for what.

- The importance of protecting this information from tampering.
- The need to conceal or to reveal who you trust.

All this is application specific, and trust assumptions are not part of the e-speak core's architecture.

Trust assumptions define whose certificates will be honored, and the acceptable set of tags in each case. Both clients and services can have trust assumptions. Trust assumptions do not appear in any of the e-speak protocols (core to core, or client to core APIs).

It can be important for a client or server not to reveal certain trust assumptions, containing information of potential use to an attacker. Conversely, a trust assumption might need to be broadcast, for example to let potential (paying) clients know the Issuer they need to get a certificate from, to access a service.

It is essential to prevent unauthorized tampering with trust assumptions, so that attackers cannot add themselves to the list of trusted entities.

The current implementation uses self-issued certificates to store trust assumptions. A certificate is only accepted as a trust assumption if it is self-issued. The format of trust assumption certificates in the current implementation of e-speak is exactly like that of an authorization certificate. The client or service must distinguish between authorization certificates and trust-assumption certificates. This should be easy: authorization certificates will be exchanged between two entities as part of the message protocols (see Chapter 7, "Communication"). Trust assumption certificates will probably be stored locally on disk. They should in any case be separate from authorization certificates.

The following certificate authorizes the entity CertificateIssuer to issue certificates authorizing any method in the CoreManagementInterface.

```
(cert
   (issuer PK self)
   (subject PK CertificateIssuer)
   (tag (net.espeak.method CoreManagementInterface ))
)
```

The following certificate means the entity trust itself to issue any certificate.

```
(cert
   (issuer PK self)
   (subject PK self)
   (tag (* ))
)
```

Note that trust assumptions can use names or public keys as subjects.

Certificate Revocation

In the current implementation the only supported means of expressing validity is time (the <valid-basic> element). Once a certificate is issued it is valid until it expires.

SPKI supports online tests for validity. Future releases of e-speak will probably do the same, and support the principle of a certificate revocation list (CRL).

Managing certificates (informational)

The current implementation of e-speak has a Certificate Issuing Service (CI) that can be used to issue certificates authorizing access to services that trust it. This CI might be used to manage access to a set of services on a set of e-speak cores. Here we outline the way in which the CI manages its certificates as a guideline to those who want to implement their own CI.

The CI implements a notion of users and groups. When a user registers with the CI, this service issues a name certificate binding the user's name (userid) to a public key. Thereafter all certificates are issued to the userid rather than the user's public key. This means that to revoke all access to a user we need only revoke the certificate binding the userid to the public key.

The CI also maintains a list of groups analagous to the groups you might find used in operating system security architectures (e.g. "users" and "administrator"). An operator of the CI can create new groups.

To add a user or users to a group, the operator selects the userid or userids and group. The CI then issues a name certificate to each user, binding the userid name to the group name.

To issue authorization certificates for a service, the CI needs to know what interfaces and methods are available on the service (the client stub is used for this). The CI presents a simple GUI listing the methods for each interface as well as listing groups and userids. The operator can select the group or individual user and what interfaces or methods they will be allowed to access. The CI issues an authorization certificate to the userid or the group.

Whenever the CI issues a certificate, it records this in its policy database. The policy database is used to drive access revocation and certificate renewal.

The CI provides a certificate directory interface from which stored certificates can be retrieved. This allows services to see what certificates have been issued to users and permits users to retrieve certificates that have been issued to them.

Revoking and Renewing certificates

A certificate is valid until it expires. To save having to renew all certificates frequently, an Issuer might choose a relatively short period of validity when issuing name certificates binding a user's name to a public key. Other certificates, particularly authorization certficates would have longer periods of validity.

The policy for certificate renewal, enforced by the CI, is to renew automatically all certificates in its policy database as they approach expiry. Renewed certificates can be retrieved from the CI's certificate directory.

If a CI operator revokes access or removes a user, the certificate(s) are removed from the policy database immediately. This means the certificates will not be renewed and can no longer be retrieved from the directory. Entities that have retrieved certificates from the certificate directory can continue to use them until they have expired.

Note that all a user's power is revoked as soon as the certificate binding their name to a public key expires.

Renewing keys

The CI supports key renewal by issuing a certificate binding the user's new key to the user's name. All other certificates issued to the user will remain valid as they are issued to the user's userid (name). The user may undergo a process similar to registration to convince the CI that the new key is valid.

Private Security Environments (Informational)

Private keys must never be shared and so need to be stored securely. How private keys are stored is a matter for the owner of the key and has no impact on the e-speak protocols or APIs used to interact with the core. It is therefore not part of the architecture. In the current implementation a PSE or Private Security Environment is used. This stores the keys in an encrypted file on a disk.

The private keys are never revealed to the the application. Instead data is sent to the PSE object when it requires signing. The PSE framework has been designed so that the underlying mechanisms can be changed to accomodate devices like smart cards.

Interoperability with X.509 (Informational)

X.509 certificate infrastructures [see *RFC 2459*] are becoming more and more common. An X.509 certificate binds an entity's distinguished name to its public key. This is very similar to the way in which a SPKI name certificate binds a name to a public key. One difficulty is that in SPKI the Issuer is denoted by a public key. In X.509 the Certificate Authority is denoted by a "distinguished name". Its public key is not required to be in the certificate. When it isn't there, the Certificate Authority is assumed to have a well-known public key.

An e-speak CI can take an X.509 certificate, verify it (check it is signed by a trusted Certificate Authority) and issue an e-speak Name Certificate binding an encoding of the subject's X.509 distinguished name to the subject's public key. (This is not supported in the current release.)

In addition the e-speak certificate verifier could be extended to handle X.509 name certificates natively, automatically converting them to SPKI name certificates as outlined above. (This is not supported in the current release.)

X.509 version 3 also supports attribute certificates and work is on going within the IETF on defining a profile for attributes to use within the Internet's PKIX infrastructure. It is not possible to define a useable mapping from X.509 attribute certificates into SPKI authorization certificates, as X.509 attributes can be arbitrary. In principle it should be possible to define a mapping from SPKI certificates into X.509 attribute certificates.

SPKI BNF Formats

E-speak uses two BNF syntaxes. The "advanced" syntax is used for manually-input data and human reading. It has been used throughout this document. Its advantages for this purpose are allowing white spaces (including line-feed), and base64 and hex codings of numbers. The base64 coding allows public keys to be written with relative brevity.

The parser accepts certificates in advanced syntax or canonical syntax, and outputs them in canonical syntax. This is used for all internal operations, such as protocol exchanges, and for serialized transmission. All hashes are computed on data in canonical syntax. This is necessary, because varying numbers of white spaces would produce invalid hashes.

Advanced Syntax

The advanced syntax follows. Its initial non-terminal is <s-part>.

```
<alpha> = [a-zA-Z];
<base64> = "##" (<base64-char> | <space>)* "#" ;
<base64-char> = <alpha> | <digit> | [+/=];
```

```
<bytes> = <token> | <string> | <raw-bytes> | <quoted-string> |
<base64> | <hex> ;
<byte-string> = <display-type>? <bytes> <decimal> = [0-9]+ ;
<digit> = [0-9];
<display-type> = "[" <bytes> "]" ;
<hex> = "#|" (<hex-digit> | <space>)* "|";
<hex-digit> = [0-9A-Fa-f];
<punctuation> = [\-./_:*+=] | [’‘,@] | [$%^&\[\]#~<>?:|] ;
<quoted-string> = "#<" {delimiter char c} {delimiter string s not
containing c} {c}
              {any character strng not containing s} {s} ;
<raw-bytes> = "#" <decimal> "*" {binary byte string of that
length} ;
<s-expr> = "(" (<s-part> | <space>)* ")" ;
<space> = [ \t\r\n]*;
<s-part> = <byte-string> | <s-expr> ;
<string> = "\"" {string chars} "\"";
<token> = (<alpha> | <punctuation>)
(<alpha> | <punctuation> | <digit> )* ;
```

We also allow end-of-line comments indicated by !. Comments are treated as white space.

Within a string C conventions can be used, including octal escape sequences. Specifically:

```
\b backspace (010)
\f formfeed (014)
\n newline (016)
\r return (015)
\t tab (011)
\nnn octal escape
```

Where nnn is a 3-digit octal numeric in the range 0_8 - 177_8 , which is 0_{10} - 127_{10}

Canonical Syntax

The canonical syntax defines the following.

```
<bytes> = <raw-bytes> ;
<decimal> = [1-9] [0-9]* | "0" ;
<s-expr> = "(" <s-part>* ")" ;
```

This disallows space in lists, all byte forms except counted string, and insists that decimal numbers have no redundant leading zeros. Hashes are always computed over canonical forms.

Within certificates, lists must start with a byte string and be non-empty:

```
<s-expr> = "(" <bytes> <s-part>* ")" ;
```

The following is the BNF currently recognized. The top-level non-terminals are:

```
_ <cert>: a certificate.
_ <name-cert>: a name certificate
_ <proof> : a certificate justification.
<proof> is used in the messaging protocol.  (See ]
```

In the messaging protocol (See Chapter 7, “Communication”) we use tag lists for queries and requirements.

```
<cert> = "(" "cert" <version>? <cert-display>?
        <issuer> <issuer-info>?
        <subject> <subject-info>?
        <deleg>? <tag> <valid>? <comment>? ")" ;
<cert-display> = "(" "display" <byte-string> ")" ;
<comment> = "(" "comment" <byte-string> ")" ;
<date> = <byte-string> ;
<date-expr> = <byte-string> ;
<deleg> = "(" "propagate" ")" ;
<full-name> = "(" "name" <principal> <byte-string>+ ")" ;
<gte> = "g" | "ge" ;
<hash> = "(" "hash" <hash-alg-name> <hash-value><uris>? ")" ;
<hash-alg-name> = "md5" | "sha1" | <uri> ;
<hash-of-key> = <hash> ;
<hash-value> = <byte-string> ;
<issuer> = "(" "issuer" <principal> ")" ;
```

```
<issuer-info> = "(" "issuer-info" <uris> ")" ;
<issuer-name> = "(" "issuer" "(" "name" <principal> <byte-string>
")" ")" ;
<low-lim> = <gte> <byte-string> ;
<lte> = "l" | "le" ;
<name> = <relative-name> | <full-name> ;
<name-cert> = "(" "cert" <version>? <cert-display>?
          <issuer-name> <issuer-info>?
          <subject> <subject-info>?
          <valid> <comment>? ")" ;
<not-after> = "(" "not-after" <date> ")" ;
<not-before> = "(" "not-before" <date> ")" ;
<n-val> = <byte-string> ;
<object-hash> = "(" "object-hash" <hash> ")" ;
<one-valid> = "(" "one-time" <byte-string> ")" ;
<online-test> = "(" "online" <online-type> <uris>? <principal>
<s-part>* ")" ;
<online-type> = "crl" | "reval" | "one-time" ;
<principal> = <public-key> | <hash-of-key> ;
<proof> = (<cert> | <name-cert>)*
<public-key> = "(" "public-key" <pub-sig-alg-id> <s-expr>*
<uris>? ")" ;
<pub-sig-alg-id>= "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-
pkcs1" | "dsa-sha1" | <uri> ;
<range-ordering>= "alpha" | "numeric" | "time" | "binary" | "date"
;
<relative-name> = "(" "name" <byte-string>+ ")" ;
<requires> = "(" "requires" <tag>* ")" ;
<restrict-date> = "(" "date" <date-expr> ")" ;
<restriction> = <restrict-date> | <target> | <requires> ;
<restrictions> = <restriction>* ;
<reval> = "(" "reval" <version>? <subj-hash> <reval-body> ")" ;
<reval-body> = <one-valid> | <valid-basic> ;
<signature> = "(" "signature" <hash> <principal> <sig-val> ")" ;
```

```
<sig-val> = <s-part> ;
<subject> = "(" "subject" <subj-obj> ")" ;
<subject-info> = "(" "subject-info" <uris> ")" ;
<subj-hash> = "(" "cert" <hash> ")" ;
<subj-obj> = <principal> | <name> | <object-hash>;
<tag> = "(" "tag" <tag-expr>* ")" ;
<tag-and> = "(" "*" "and" <tag-expr>+ ")" ;
<tag-expr> = <byte-string> | <tag-simple>
        | <tag-prefix> | <tag-range>
        | <tag-set> | <tag-and>
        | <tag-star> ;
<tag-simple> = "(" <byte-string> <tag-expr>* ")" ;
<tag-prefix> = "(" "*" "prefix" <byte-string> ")" ;
<tag-range> = "(" "*" "range" <range-ordering>
<low-lim>? <up-lim>? ")" ;
<tag-set> = "(" "*" "set" <tag-expr>* ")" ;
<tag-star> = "(" "*" ")" ;
<target> = "(" "target" <tag-expr>* ")" ;
<up-lim> = <lte> <byte-string> ;
<uri> = <byte-string> ;
<uris> = "(" "uri" <uri>* ")" ;
<valid> = <valid-basic> <online-test>* <restrictions> ;
<valid-basic> = <not-before>? <not-after>? ;
<version> = "(" "version" <byte-string> ")" ;
```

The elements <reval>, <online-test> (and related elements such as crl) and <restrictions> are parsed but silently ignored in the current implementation. Architectural extensions will be introduced to support these elements.

References

Pfleeger - C.P. Pfleeger: "Security in Computing", Englewood Cliffs, N.J., Prentice-Hall, 1989

RFC 2396 - T. Berners-Lee, R. Fielding, U.C. Irvine, T. Ylonen:

"Uniform Resource Identifiers (URI): Generic Syntax", Aug. 1998

RFC 2459 - R. Housley, W. Ford, W. Polk, D. Solo: "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", Jan. 1999

RFC 2692 - C. Ellison: "SPKI Requirements", Sept. 1999

RFC 2693 - C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ilonen:

"SPKI Certificate Theory", Sept. 1999

Note: For all RFC's, access www.ietf.org

Schneier - Bruce Schneier: "Applied Cryptography", 2nd. Edition, John Wiley & Sons, 1996

Stallings - William Stallings: "Cyptography and Network Security: Principles and Practice", 2nd. Ed., Prentice Hall, Upper Saddle River, N.J, 1999

Working Draft - C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ilonen: "Simple Public Key Certificate", Internet Draft <draft-ietf-spki-cert-structure-0.5.txt, (Expired 18 Sept. 1998)

Chapter 7 Communication

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 core. The only way for a Resource Handler to return a reply to a Client is to send a message through the e-speak core. Thus, the e-speak core mediates all access between Clients and Resource Handlers. It is the only entity to accept connections: Clients and Resource Handlers establish connections to an e-speak Core so that they can communicate with each other.

What is mediation in this context? Mediation includes some or all of the following functions.

- The e-speak core determines to which resource handler or handlers to route the message.
- The e-speak core determines how to route the message (routing path).
- The e-speak core can process and transform the message headers and contents. Only limited processing of the message is possible if security is enabled.

Mediation is transparent to the Client and Resource Handler. Figure 4 illustrates the flow of information through the Core.

All messages exchanged between Clients and e-speak cores are formated as ESPDU's (E-speak Protcol Data Units) ("E-speak Protocol Data Units (ESPDUs)" on page 147). The destination or "TO" field of the ESPDU is an ESName ("ESNames" on page 117). ESNames conform to the format and grammar defined for Universal Resource Identifiers in RFC 2396.

The e-speak core does not keep any information about replies to messages. As far as the e-speak core is concerned, a reply is another message. If the Client needs a reply, it can 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. Figure 4 shows the flow of messages through an e-speak core when a Client sends and receives messages froma Resource Handler.

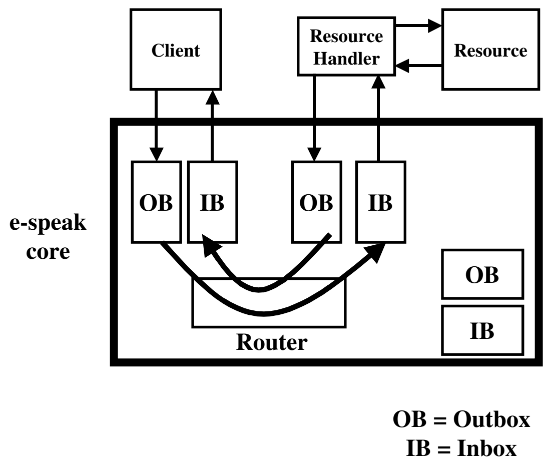


Figure 4 Message flow with an e-speak core

The the e-speak core never keeps any state about a message beyond the time needed to complete processing. Once the it retrieves a message from a Client's Outbox, the e-speak core guarantees to do one of three things:

1. Normally, the it forwards the message to a Resource Handler. However, it cannot deliver the message if:
    - The "To" field of the message is not valid Resource
    - The Inbox specified in the destination Resource metadata is not connected to a Resource Handler
    - The Resource Handler's Inbox is full
2. When it cannot deliver the message, the e-speak core returns an error message to the Client.

3 If the Client's Inbox can't take the error message for any reason, the e-speak core discards the message.

Figure 5 shows the message flow when a Client sends and receives messages from a Resource Handler on a remote e-speak core. The same protocol is used to exchange messages between e-speak cores as is used to exchange messages between Clients and e-speak Cores. All messages are formated as ESPDU

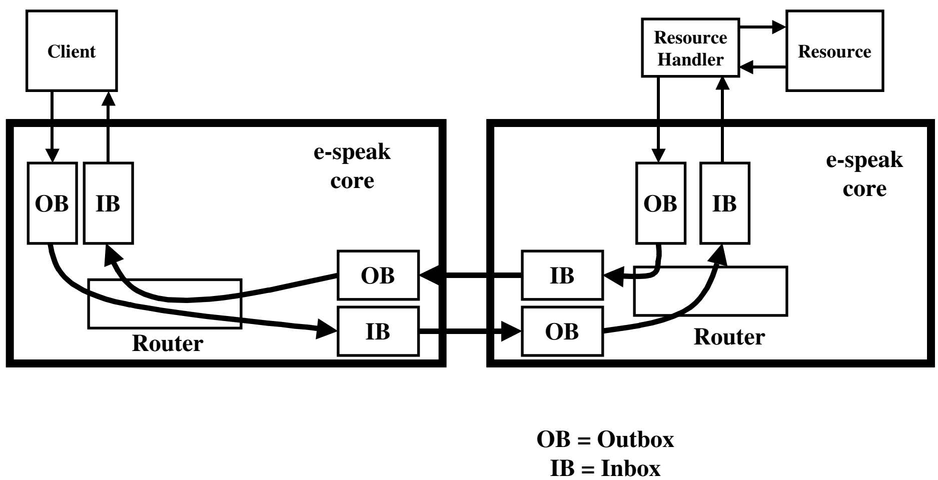


Figure 5 core to core message flow

The Session Layer Security (SLS) Protocol is run between a client and a resource handler to establish a secure session between the client and the resource handler. All SLS messages are carried as ESPDU messages.

A secure session has the following properties.

- All messages exchanged between the two end points are authenticated. This prevents messages being changed or messages being inserted into the TCP connection by a third party (e.g. an attacker).

- All message exchanged between the two end points are protected against replay. This prevents a third party capturing the messaging and replaying it at a later date to trigger a repeat of the action taken by the recepient.

Messages exchanged in a secure session can be encrypted for confidentiality. This prevents a third party from reading the contents of a message.

In addition to two modes of secure session, SLS supports non-secure sessions, in which messages are exchanged without encryption, authentication or protection against replay.

The Session Layer Security protocol is described in "Session Layer Security Protcol (SLS)" on page 122.

Two e-speak cores can exchange core manged resource and resource metadata. This is known as import and export of resources. This is needed for several reasons, including the following.

- A resource cannot be discovered in a vocabulary on an e-speak core unless the vocabulary has been registered on that core. If the vocabulary was created on another e-speak core, it is registered by importing it.
- A resource cannot be registered in a contract on an e-speak core, unless the contract has been registered on that core. If the contract was created on another e-speak core, it is registered by importing it.
- Resource metadata can be imported into an e-speak core, to cache it. This makes lookup of those resources faster.

Resource import and export is described in "Core to core communication" on page 152.

The format for data exchanged with e-speak resources is defined by the e-speak serialization format in "E-speak Serialization Format" on page 166. This data is carried within the payload of SLS messages.

ESNames

ESnames denote an access path to a resource. ESnames conform to the format and grammar defined for Universal Resource Identifiers (URIs) in RFC 2396. ESNames are therefore URIs and mores specifically, since they denote the access path for the resource, ESNames are also Universal Resource Locators (URLs). ESNames have the following format.

```
es://<host>/<relative path>
```

The host part of an ESName is either the host name or the IP address of the host on which the e-speak core is located together with an optional port number. If the port number is not specified, the current implementation throws an exception. Discussions are underway with IANA for a standard port number to be assigned.

The full form of an ESName (es://<host>/<relative path>) is known as an absolute ESName. Subsets of this syntax also denote ESNames (see “ESName BNF” on page 121). However, it may not be possible to resolve such ESNames if we do not have the necessary context.

The relative path component of an ESName must be unique on the given e-speak core. The path is relative in the sense that it is given a global context by the <host> element of the ESName. If the <host> element is missing (e.g. es://path or es:/path), then the resolver must decide the global context in which to begin resolution. Usually the global context is assumed to be the current host.

The path consists of a set of strings separated by “/”, for example “a/b/c”. The path is resolved by taking each string element in order and resolving that in the current name frame. If this returns a name frame the next element is resolved in that name frame. The process continues until there are no more elements in the path in which case we have resolved the ESName to the intended resource. The first element is resolved in the root name frame of the e-speak core denoted by the server part of the ESName. For example, taking “a/b/c”, “a” is resolved in the e-speak core’s root name frame to return a name frame which we denote NF(a). Next b is resolved in NF(a), to return a name frame which we denote NF(a b). Finally c is resolved in the name frame NF(a b).

If a string element of the path component other than the final component fails to resolve to a name frame, name resolution has failed. If the final element fails to resolve to a resource, name resolution fails.

When a client registers a resource with an e-speak core, the default name frame for that resource, is root name frame of the client's protection domain.

If two clients on the same host bind two different resources in their root frames under the same relative path (e.g /a/resource) they have different absolute ESnames even though the host part is the same:

```
es://host/client1PDRootFrame/a/resource and
es://host/client2PDRootFrame/a/resource
```

If a client terminates and its protection domain is not persistent, there is a recursive deletion of all names within the protection domain's root name frame. This means any URLs handed out rooted in this name frame are invalid. To overcome this, a resource needs to be bound in a name frame which is persistent. (To avoid the deletion of a non-persistent protection deleting persistent persistent resources, the current implementation do not allow a persistent resource to be bound within a non-persistent Protection Domain.)

Core Name Frame and core root Name Frame

Every e-speak Core has a Core Name Frame with the following ESName.

```
es://<server>/core
```

The names of various core managed resources are bound within this name frame, including vocabularies, contracts and the metaresource. The following names are currently used.

```
/Core/MetaResource
/Core/ResourceFactory
/Core/SystemMonitor
/Core/Finder
/Core/CoreManagementService
/Core/DefaultVocabulary
/Core/BaseDistributorVocabulary
/Core/CoreDistributor
```

```
/Core/ConnectionManager
/Core/RemoteResourceManager
/Core/AccountManager
/Core/BaseAccountVocabulary
```

Every e-speak Core has a root Name Frame which is denoted:

```
es://<server>/
```

or, alternatively, assuming the <server> part is already known to the resolver:

```
es:/
```

The following names are bound in the e-speak Core root Name Frame:

```
/Core
/ContractContract
/VocabularyContract
```

Canonical ESName

The canonical is the ESName stored in the URL field in the Resource's ResourceSpecification (see Chapter 3, "Resource Descriptions, Resource Specifications and Resource Types"). This ESName is guaranteed always to be valid as long as the Resource is registered, it does not depend on any binding maintained by in a Client's Name Frame. Cannonical ESNames have the following form:

```
es://<hostport>/proc/resource/<type_string>/<unique_id>
```

The <hostport> field is of the form host name or IP address followed by a port number separeted by a ":" as specified in RFC 2396.

The following are the permissible values of <type_string>, they denote the Resource Type (see Chapter 3, "Resource Descriptions, Resource Specifications and Resource Types")

```
"Inbox"
"MetaResource"
"ProtectionDomain"
"ResourceFactory"
"ConnectionManager"
```

```
"RemoteResourceManager"
"Contract"
"CoreDistributor"
"ExternalResource"
"ExternalResourceContract"
"ImporterExporter"
"MappingObject"
"NameFrame"
"RepositoryView"
"SecureBoot"
"SystemMonitor"
"AccountManager"
"Account"
"Vocabulary"
"CoreManagementService"
"Finder"
```

The filed <unique_id> is the stringified form of a number (in the current implementation this is the repository handle).

Any Resource bound in the Core Name Frame es://<hostport>/Core/ cannot be assigned a canonical name of the form described above. Instead the name in the Core Name is assumed to be the Canonical Name.

Queries and fragments

Queries and fragments are also allowed in ESNames. A query is the data that follows the "?" in an ESName of the form:

```
es://<host>/<relative path> ? uric*
```

A fragment is the data that follows the "#" in an ESName of the form:

```
es://<host>/<relative path> # uric*
```

The character set uric is defined in RFC 2396 which also specifies constraints on the data in queries and fragments.

Queries and fragments are not used and name resolution and are never interpreted by the e-speak core. They are delivered to the resource handler as part of the message.

ESName class definition

```
class ESName{
   string hostPart;
   string[] pathPart;
}
```

The above class defines the hostPart consists of the portion of the ESname from “es://” to the first “/”. In an ESName the path separator is “/”. This separates elements of the path. Each element of pathPart consists of an element in the path, without and “/” character.

ESName BNF

Here is the BNF for ESNames. Please refer to RFC 2396 for any element not defined directly.

```
ESName = [ absoluteESname | relativeESname ] [ "#" fragment ]
absoluteESname = es ":" hier_part
relativeESname = ( net_path | abs_path | rel_path ) [ "?" query ]

hier_part = ( net_path | abs_path ) [ "?" query ]

net_path = "//" server [ abs_path ]
abs_path = "/"  path_segments
rel_path = rel_segment [ abs_path ]

rel_segment   = 1*( unreserved | escaped | ";" | "@" | "&" | "="
| "+" | "$" | "," )

server = host [ ":" port ]
host = hostname | IPv4address
```

```
hostname = *( domainlabel "." ) toplabel [ "." ]
domainlabel = alphanum | alphanum *( alphanum | "-" ) alphanum
toplabel = alpha | alpha *( alphanum | "-" ) alphanum
IPv4address = 1*digit "." 1*digit "." 1*digit "." 1*digit
port = *digit

path_segments = segment *( "/" segment )
segment = *pchar *( ";" param )
param = *pchar
pchar = unreserved | escaped | ":" | "@" | "&" | "=" | "+" | "$"
| ","

query         = *uric
fragment      = *uric
```

Session Layer Security Protcol (SLS)

The SLS (Session Layer Security) protocol tries to extend the capabilities of SSL "The TLS Protocol version 1.0 RFC 2246, IETF by T. Dierks and C. Allen January 1999." on page 169, a protocol that is supported by most modern web browsers, and is currently the default way to secure client/server interactions over the web. The motivation for changing SSL can be summarized as follows:

- Transport independence: SSL links a security session with a TCP socket. If the socket dies the security session dies with it, something undesirable when the life expectation of a security session is very different from the life expectation of the transport. Also, we cannot multiplex several security sessions onto the same socket, or perform dynamic load balancing of the end point without starting the session from scratch. Moreover, even though properties like reliability and in-order delivery of messages are critical for a security protocol, some TCP details are not, and this might put unnecessary restrictions on the applicability of SSL. Finally, in some cases we might want to use a different transport for sending and receiving messages (i.e., outgoing messages use a different firewall needing two sockets). SLS tries to make the minimum number of assumptions on the communication transport solving most of the issues above.

- Tunnelling support: During firewall traversal we might want the firewall to control the client access rights to the internal LAN for every packet. However, we might not want the firewall to see all the traffic in clear (therefore, losing the end-to-end security property). This is difficult to achieve with SSL because either we let the client open a direct socket to the service or the firewall sees all the traffic in clear. On the other hand, with SLS we can nest a secure session inside another one, possibly with different end points, allowing to achieve both goals simultaneously.
- Elliptic cryptography: Most implementations of SSL only support Diffie-Hellman key agreement algorithms based on exponentiation. SLS uses a faster algorithm based on Elliptic Curve Cryptography (ECC) developed in (ref to HPLabs report).
- Attribute certificates using SPKI. SSL only supports X509 name certificates, mainly to authenticate that the end-point "owns", according to a configured "trusted CA", the web address that we wanted to reach. Only one certificate by each party can be used, and in most cases only server authentication is performed. On the other hand, SLS performs a negotiation of tags that need to be proven represented by multiple SPKI certificates. This allows a fine grained control of security by mapping tags to actual permissions, raising the level of abstraction from "a stream of bytes" in SSL to a particular operation on service X in SLS, making it easier to integrate with application level security. Details on the use of SPKI certificates in SLS can be found in Chapter 6, "Access Control."
- Latency minimization: SLS allows the client to send application data after a round-trip negotiation has succeed. In SSL two round-trips are needed before the application data is sent. This can have important performance implications when network delays are large and we need a quick response from the server.

Functional Description of SLS

In this section we describe what is the expected behavior of the protocol.

Protocol message types

Every SLS message is embedded in a PDU (Protocol Data Unit) ("Protocol Data Unit PDU" on page 139) that contains some header information that allows the system to dispatch it to the correct security context, route it through the network, identify replies, ensure the protocol version and so on.Two fields of this header relevant to our discussion classify messages according to the type of encoding and their purpose:

Supported encoding types

- CLEAR_DATA: The message is not encrypted or protected against modification.
- PROTECTED_DATA: The message is not encrypted but it is protected against modification with a signed digest (MAC)
- SECURE_DATA: The message is encrypted and protected against modification using a MAC.

Supported message types

- HANDSHAKE: Message exchanged during the key-agreement protocol. There are four types of handshake messages that are discussed later.
- ALERT: Message that identifies an abnormal situation during the handshake or after the session has been established. ALERT messages can be fatal, forcing session termination, or just warnings, in which case what should be the response is implementation dependent.
- APPLICATION_MESSAGE: Message that communicates application data.
- TUNNEL: A message that contains another PDU in its payload. This is used to nest sessions, something important for firewall traversal.
- PING: A heartbeat message that is used by the session scavenger to know if the session is still active.
- REKEY: Forces a key offset of the instantiated cipher suite based on the previously negotiated shared secret. (REKEY is not supported in the current implementation.)

High level protocol state machine

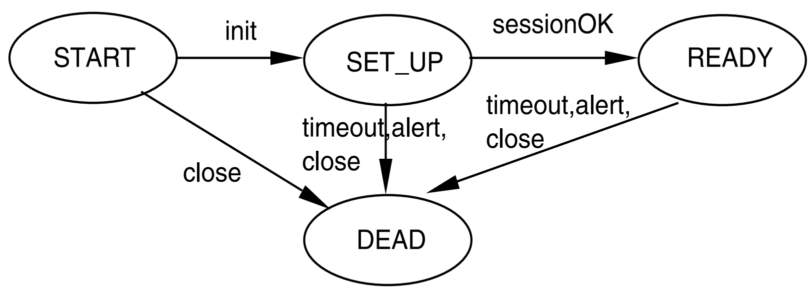


Figure 6 High level state transitions in SLS.

Figure 6 shows what are the possible states of a session, and what triggers transitions between them. There are four possible states:

- START: the session object has been created but it is not fully configured. Also, the key agreement protocol has not started.
- SET_UP: the key agreement protocol has started. We do not have a shared secret yet, so all the messages have CLEAR_DATA encoding. It is implementation dependent whether to pay attention to unauthenticated ALERT messages or not.
- READY: the key agreement protocol has successfully completed. We have a working session that we can use to encrypt/decrypt messages and validate whether the session has certain security tags associated with it. At this point ALERT messages are authenticated and should not be ignored.
- DEAD: the session is no longer operational, and will never be. We can safely scavenge it.

Also, we can see in Figure 6 the events that trigger state transitions:

- init: complete the initialization of the object and send a message to the other party (if needed) to start the key agreement.

- sessionOK: The key agreement has finished successfully and we have a new cipher suite to install in the session.
- timeout: a timeout expired. The different timeouts are explained in detail in "Timeouts description" on page 133.
- alert: an authenticated (or optionally non-authenticated) fatal alert was handled/ sent forcing a shutdown of the session.
- close: a client forced termination of a session by invoking the close() method.

Two important operations on a session are handling and sending a PDU.

SLS Handling a PDU

The handling operation assumes that the PDU has already been received from the transport and invokes some security processing on the PDU (i.e., decryption/ authentication).

Depending on the state the session, and the encoding/type of the PDU, the session behaves differently while handling PDUs. Table 7 shows the expected behavior of a

Table 7 PDU handling behavior depending on state

TYPE	ENCODE	START	SET_UP	READY	DEAD
HAND-SHAKE	CLEAR	Exception	HandleHsk	Ignore	Exception
	MAC	Exception	Ignore	Warning	Exception
ALERT	CLEAR	Exception	HandleAl Optional	Ignore	Exception
	MAC	Exception	Ignore	HandleAl	Exception
APPLICA-TION	CLEAR	Exception	Ignore	Ignore	Exception
	MAC	Exception	Ignore	HandleApp	Exception
TUNNEL	CLEAR	Exception	Ignore	Ignore	Exception
	MAC	Exception	Ignore	HandleTun	Exception

Table 7 PDU handling behavior depending on state (Continued)

TYPE	ENCODE	START	SET_UP	READY	DEAD
PING	CLEAR	Exception	Ignore	Ignore	Exception
	MAC	Exception	Ignore	HandlePin	Exception
REKEY	CLEAR	Exception	Ignore	Ignore	Exception
	MAC	Exception	Ignore	HandleRe	Exception

session when trying to handle a PDU. Let's explain the terms in that table:

- MAC: either PROTECTED_DATA or SECURE_DATA encoding type. Obviously, the important property is that the PDU is correctly authenticated, otherwise we always ignore the message, regardless of its claims.
- Exception: notify the client doing the handling that the session is not operational.
- Ignore: do not take any significant action based on that PDU (i.e., change session internal state). Optionally, an implementation could log the event that a PDU is being ignored.
- Optional: an action is considered optional if a particular implementation can decide to invoke a Ignore PDU instead.
- Warning: send a warning ALERT response to the other party.
- HandleHsh: handle a HANDSHAKE PDU by making progress in the key agreement protocol. This could involve an Internal Send of a HANDSHAKE or ALERT PDU to the other party.
- HandleAl: handle an ALERT PDU. This might involve closing the session if it is a fatal alert or logging the event otherwise
- HandleApp: handle an APPLICATION_MESSAGE PDU. Typically, the PDU is returned in clear text to the client if authenticates and/or decrypts correctly, otherwise it is ignored.

- HandleTun: handle a TUNNEL PDU. This could involve “peeling off” the outer PDU, returning the inner one (after decryption/authentication of the outer one) or calling a custom handler to deal with it.
- HandlePin: handle a PING PDU. This handling might require sending a reply PING PDU or just record that our previous PING has been replied successfully.
- HandleRe: handle a REKEY PDU. Forces the re-key of the handler part of the crypto suite.

SLS Sending a PDU

The sending operation first invokes the required security processing (i.e, encoding, MAC computation) and then it uses the underlying transport to deliver the message at the other end. Note that handling a PDU might have as a side effect that another PDU is sent to the other party, i.e., a response to a handshake message during the key agreement. We call that case a internal send as opposed to a external send directly invoked by the client.

Table 8 shows the expected behavior when trying to send a PDU through a session.

Table 8 PDU sending behavior depending on state

TYPE	MODE	START	SET_UP	READY	DEAD
HAND-SHAKE	Internal	NotApply	OK	OK	NotApply
	External	Exception	Retry	OK	Exception
ALERT	Internal	NotApply	OK	OK	NotApply
	External	Exception	Retry	OK	Exception
APPLICA-TION	Internal	NotApply	NotApply	NotApply	NotApply
	External	Exception	Retry	OK	Exception
TUNNEL	Internal	NotApply	NotApply	NotApply	NotApply
	External	Exception	Retry	OK	Exception

TYPE	MODE	START	SET_UP	READY	DEAD
PING	Internal	NotApply	NotApply	OK	NotApply
	External	Exception	Retry	OK	Exception
REKEY	Internal	NotApply	NotApply	OK	NotApply
	External	Exception	Retry	OK	Exception

Some terms in that table deserve further explanation:

- NotApply: it is an implementation error if the protocol tries to send this message. This is only relevant for internal messages because the implementation does not have control over possible external messages.
- OK: this means that the session performs the appropriate processing and try to deliver it to the lower layer. This does not mean that the message has been correctly sent, because this depends on the status of the underlying transport/ session.
- Retry: The client is informed that the session is currently unavailable to send messages but this might change in the future.
- Exception: The client is informed that the session is permanently unavailable.

The key-exchange protocol

The key-exchange protocol is an authenticated Diffie-Hellman key exchange. From the session key agreed in the Diffie-Hellman exchange further keys are derived for encryption and confidentiality.

Need more details here

- Elliptic Diffie-Hellman instead of modulus exponentiation: instead of choosing a group and checking its validity at the other end, we pick one of a pre-determined family of elliptic curves, and later check that the random point belongs to that curve.

- No support for multiple public keys of the same principal: this extension should be trivial by adding more than one signature in the handshake.
- Added tunneling support: we allow the responder to notify in its first handshake message that it wants to relay the session (tunnelling is described in "Support for tunneling" on page 137), so it might not have to prove the tags requested.
- Randomized secure channel identifiers (SPI): we want SPIs to be hard to guess to avoid a denial-of-service attack by flooding the client/server with fatal alert messages while being able to pay attention to non-authenticated alerts during the handshake (handling alerts is described in Section "Handling alert messages" on page 132)

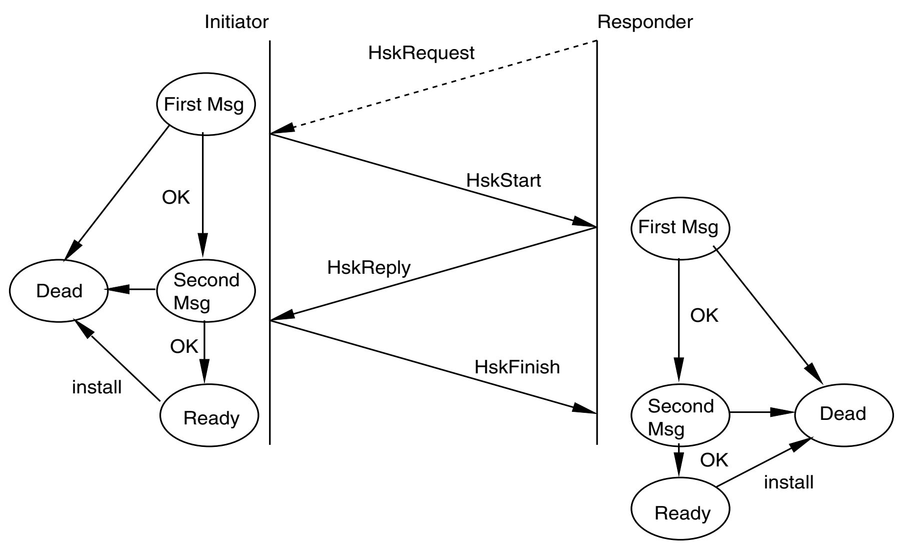


Figure 7 Key agreement protocol

Figure 7 shows the key agreement interactions and corresponding state machines of the handlers that control these interactions. These state machines are embedded in the state SET_UP in Figure 7. Sub-states READY and DEAD are not necessarily related to the ones in Figure 7. For example, killing the handler does not mean that the session dies, it is perfectly normal that the handler terminates when the key agreement finish successfully, and the cipher suite gets “instantiated” in the session.

A quick summary of the messages sent during the handshake:

- HskRequest: a request from the server (Responder) to the client (Initiator) to start a session
- HskStart: a request (or acknowledgement) from the client to the server to start a session. It contains the elliptic curve and cipher suite list suggested, the SPI at this end, a hint on the tags that the server should prove, a hint on the operations that we want to perform, and a public Diffie-Hellman (DH) key.
- HskReply: a reply to the previous HskStart from the server to the client. It contains the cipher suite chosen, the SPI at the server end, whether the server is a relay, certificates to prove the requested tags, a hint of tags that the client need to prove, a public DH key and a signature.
- HskFinish: last message from client to server. It contains certificates to prove the requested tags and a signature.
- Alert: notify the other end of a failure of the key agreement.

We made the Initiator and Responder state machines similar by always introducing a HskRequest message. If the protocol is started by the Initiator the HskRequest is generated locally. Otherwise it is generated by the Responder and transmitted through the network. We pay attention to alerts and “incorrect” messages with valid random SPI so we can kill the handler at any time.

Key generation algorithm

Discription of key generation algorithm : section 4.0 of Ferguson report

The record layer

What is the record layer. Need more details here - devTeam?

Cipher suite support

The encryption algorithms currently supported are Blowfish with a 128 bit key, and triple DES with three independent keys for encryption. Blowfish is the recommended cipher because of its speed but 3DES is the conservative choice. Also only CBC mode and PKCS 5 padding is supported.

Elliptic Curve Cryptography is used for the Diffie-Hellman shared secret negotiation instead of modulus exponentiation.

Our current hash algorithm is SHA-1 and our MAC algorithm is an HMAC construction based on SHA-1.

SPKI certificates are signed/verified using RSA with a 1024 bit key.

Handling alert messages

During the key agreement protocol the default is to pay attention to non-authenticated alert messages that have the correct random SPI. These identifiers are sent in clear so if the attacker listens to all our traffic and sends fatal alerts with the right SPI before the other party responds, it still stops the session set-up. However, this attack is trivial to do if we had a predictable SPI because the attacker can just flood the system with fatal alerts with typical SPIs. Note that in SLS this problem is more evident than in SSL because of the transport independence assumption. In this case we cannot make the assumption that it takes some effort to hi-jack the transport, as it is the case for a TCP socket in SSL

On the other hand, if we want to avoid this attack completely we could ignore all non-authenticated alerts and rely on timeouts to close failed sessions. This is not the default because the convenience of quick and detailed notification of session set-up failure was believed to be more important than an impractical denial-of-service attack, that can always be done at transport level anyway.

In any case, after a session is established only authenticated alerts are respected. At that point many messages with the SPIs in clear have been exchanged, and the randomization does not help much.

Also, alert messages can be fatal, forcing the other end to close the session, or warning, that in our first implementation are just logged. We also support internal alerts, i.e., generated as a side effect of a PDU handling, and external alerts, i.e., explicitly send by the client, but it is recommended to avoid sending external alerts and rely on internal ones as much as possible. The alert codes used in SLS are described in (“Alert” on page 141).

Timeouts description

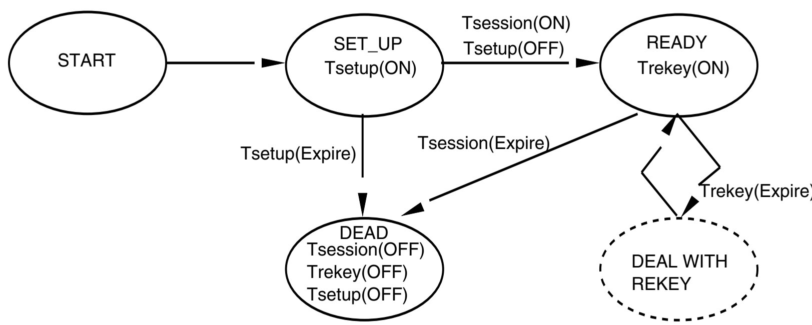


Figure 8 Session timeouts and state transitions

There are three built-in timeouts associated with a session:

- Tsetup: sets the maximum time taken by the key agreement protocol used in the SET_UP state. After that time the session becomes DEAD.
- Tsession: sets the maximum life expectancy of session. This values is the minimum of a fixed value (common for all sessions), and the life expectancy of the certificates negotiated during the key agreement. After that time the session becomes DEAD.
- Trekey: sets the maximum time allowed before forcing a rekey operation on the session (rekeying is current not supported).

Figure 8 describes the behavior of the timeouts in relation to the session states. Tsetup sets a limit on the time spent on the SET_UP state, but it is reset after we transition to the READY state. Tsession limits the maximum time spent on the

READY state. When Trekey expires we initiate the rekey and reset the timer, but this does not imply a state transition (see Section "Re-keying (not currently supported)" on page 134. Clearly, Trekey is only useful if it is smaller than Tsession, otherwise the session is never re-keyed.

Re-keying (not currently supported)

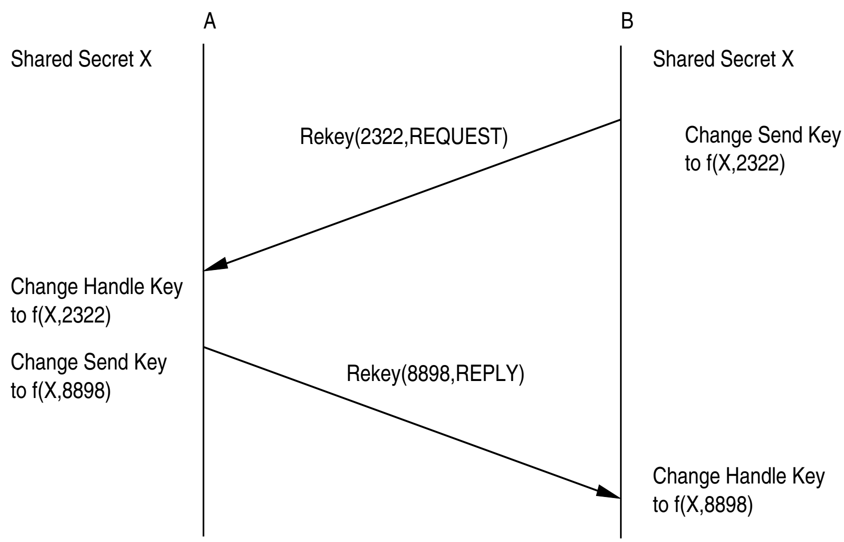


Figure 9 Rekey protocol.

After the session has been established it is possible to change the key used in the cipher and MAC operations by sending a REKEY message. However, this new key has to be based on the original shared secret negotiated using Diffie-Hellman (we do not re-run the key agreement protocol). Therefore, the re-key operation does not extend the life of the shared secret, only of the derived keys. In particular, the new key is obtained by exclusive-or of the first four bytes of the shared secret with a random integer before re-running the key generation algorithm. The one-way function used in that algorithm ensures that it is difficult to guess the next key, provided that you know the previous one. The integer added to the shared secret is transmitted inside the REKEY message.

Figure 9 shows the basic protocol to re-key a session. Node B decides that it wants to start a rekey, so it generates a random number (2322) and sends a REKEY message with it. After that it re-keys the “send part” of its cipher suite right away, so the next message sent is encrypted with the new key. When the REKEY message arrives to node A this node changes its “handle part” of the cipher suite to the new derived key. Then it checks that the message is a request and not a reply (the rekey was not initiated by him) and sends a REKEY reply message with possibly a different random integer (8898), changing the “send part” of its cipher suite too. When the reply message arrives to B, this node updates the “handle part” of their cipher suite but it does not rekey the “send part” again because the message was tagged as “reply”.

The important point of the protocol is that all the state is encoded in the messages, and provided that messages are not re-ordered, the receiver always has the right key to decrypt the message (it does not have to remember old keys or interrupt the service during re-keying). This avoids the need of an extra state in the protocol for re-keying.

Support for session scavenging

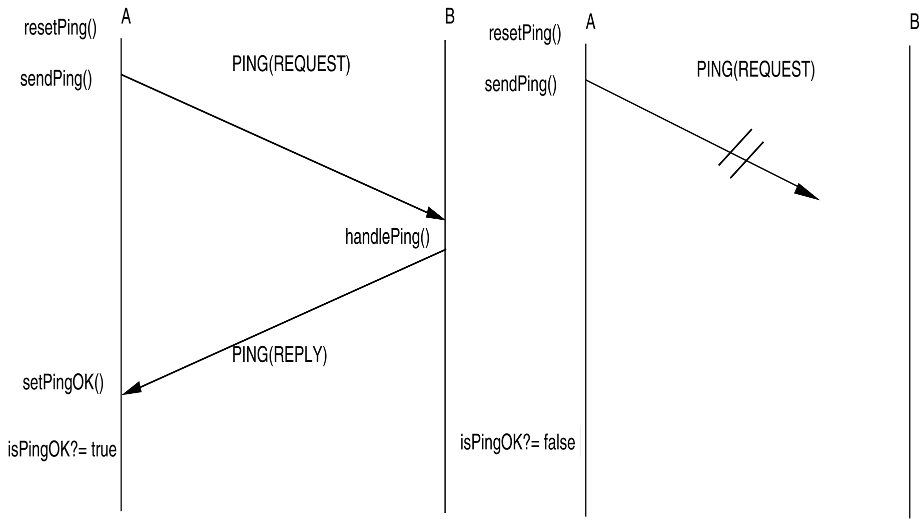


Figure 10 Ping protocol

SLS needs a external mechanism to detect that the other party in the session is not longer active. This is required because being transport independent implies that we cannot assume that the underlying transport detects a lost of the connection, i.e., TCP keep-alive message, so we have to provide that service at a higher level.

Figure 10 shows the basic support provided for session scavenging. An external client can check whether the session is active by forcing its endpoint to send a PING message and resetting a flag that indicates a reply ping arrived. If the reply ping arrives the flag is set. After a certain time the client checks whether the flag is set, hinting whether the other end is still alive or not.

Support for tunneling

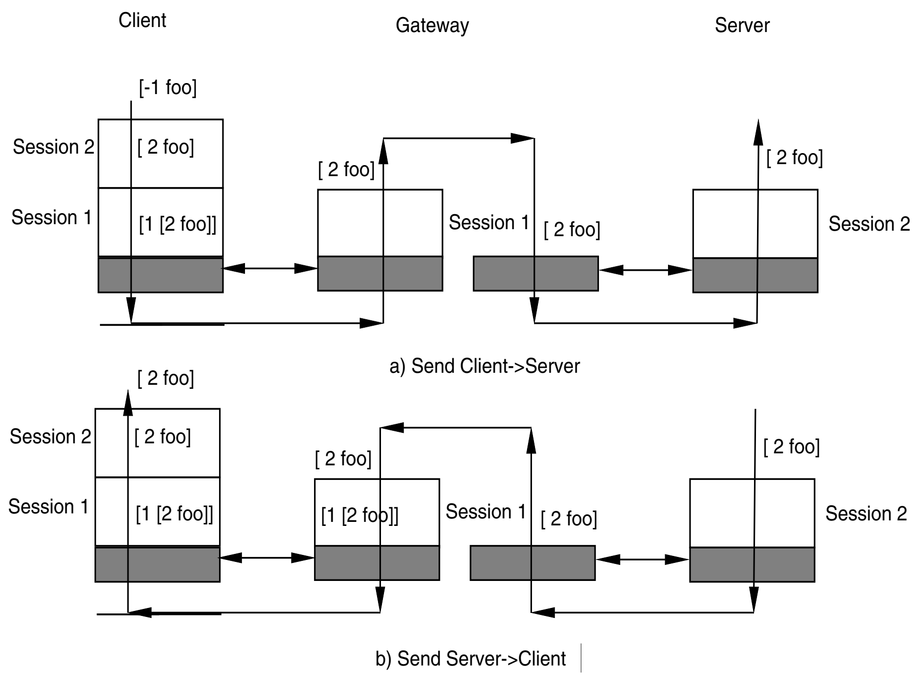


Figure 11 Tunnelling SLS sessions

In SLS tunneling a PDU contains in its payload another PDU, therefore messages are sent using another SLS session as “transport”. When the initiator sends the first protocol message the responder might reply that it is not the final end-point, so it cannot prove what was requested, and it wants to be a relay instead. At that point the client can decide whether to continue the key agreement or not. If it continues it gets a ready session that was negotiated as a “relay” and it can use that session to negotiate another one to the end-point. If this new end-point also wants to be a relay the process repeats. Typically the maximum depth of session nesting is limited to a fixed value to avoid a denial-of-service attack.

Figure 11 shows how to send messages from client to server and back after the nested session via a SLS gateway has been established. The diagram shows always sender SPIs. The session performs automatic wrapping of a PDU inside another PDU while sending when the sender SPI is valid (>0) and the sender SPI does not match the one the session is going through (we use sender and not receiver SPI because the receiver one is not guaranteed to be unique). By default a PDU is initialized with an invalid sender SPI so it only gets wrapped when it has already gone through another session (that sets the sender SPI to itself). During the wrapping the addresses of the inner PDU are copied into the header of the external PDU. Also, a TUNNEL PDU gets unwrapped (default behavior that can be overridden by a custom handler) when it is handled by a session (i.e., in Figure 11 session 1 in the gateway or client depending on the message direction). (**Check with devTeam, the above seems to be saying important things about the implementation, but is it relevant for the spec?**)

An important feature of the tunneling implementation is that a session can be used to tunnel messages regardless of whether the session was negotiated as a relay session or not. The opposite is also true, we can send non-tunnelled messages for a session that requested to be a relay session. In fact, how sessions are created is orthogonal to what type of messages can be sent through them. This simplifies the re-use of sessions but care has to be taken to avoid dangerous pitfalls like assuming that a received message in a “normal” session cannot be tunnelled or assuming that a tunnelled message is more “secure” or is coming from outside. (**Check with devTeam: why are these dangerous pitfalls?**)

Protocol Data Unit PDU

All messages exchanged between e-speak cores, and between e-speak cores and clients are PDUs. A single PDU is corresponds to a single Session Layer Security Message. The elements of PDU are marshalled in the order of member definition shown in the class declaration below. E-speak clients and resources use a sub-class of PDU for exchanging messages: ESPDU.

```
class PDU {
   int versionMajor;
   int versionMinor;
   int spi;
   int spiSender;
   int serial;
   int inReplyTo;
   int messageType;
   int encodingType;
   String toAddress;
   String fromAddress;
   byte[] route;
   byte[] data;
}
```

The current value for versionMajor is 1, and for versionMinor is 0.

SPI stands for Session Parameter Index. This is used by the two endpoints in the Session Layer Security protocol to indicate which session the message is being sent on. Since the sender and the receiver may identify the SPI separately, we have two fields: spi denotes the recepients SPI; spiSender denotes the senders SPI.

Two fields are used by SLS to protect againts replay attacks: serial is set by the sender; inReply to is the serial field of the message to which the sender is responding.

The following values for message type are defined: alert(0), handshake(1), application message(2), tunnel(3), ping(4). Alert, handshake, tunnel and ping are used in SLS to manage sessions.

The following encoding types are defined for a PDU: clear data (0); protected data(1); secure data(2). Protected data is authenticated and protected from tampering by a Message Authentication Code (MAC). Secure data is protected by a MAC and also encrypted for confidentiality.

The string toAddress is an absolute ESName and denotes the destination for the message. The string fromAddress is also an absolute ESName. It denotes the sender of the message and can be used for replies. The e-speak core attempts to resolve these names in its root name frame by finding the Mapping Object associated with each ESName. The Mapping Object is used by the e-speak core to refer to a Resource, a Search Recipe, or any combination. If the e-speak core cannot unambiguously identify the Resource Handler for the toAddress, it sends an exception message to the Client. The format for an exception message is described in Chapter 8, "Exceptions". The possible exceptions are described in Table 9.

Table 9 Exceptions for unresolved 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.

The byte array route can be used by applications to pass routing data. This is never encrypted or protected by a MAC.

The format of the byte array data is determined by the message encoding. If the encoding is clear data, the byte array is the message body whose contents is denoted by the message type.

If the encoding type is protected data, then data begins with the MAC in network byte order. The length of the MAC depends on the MAC algorithm negotiated in the SLS session setup. The remainder of the contents of data is the body of the message whose contents are denoted by the message type.

If the encoding is secure data, then it has been encrypted according to the cipher negotiated in the SLS session set up. Once it has been decrypted, it has the same format as protected data: a MAC, followed by the message body. The contents of the message body is denoted by the message type.

Check with devTeam. What else can we say about the format of MAC? What else can we say about the format of encrypted data. Presumably the encrypted data is just a byte array conssting of a 32 bit integer + data, but this is implicit because we have used [] in the definition of data. Is it sufficient to take section 5.0 of Niels Ferguson's report?

The type of the message contained in the data field is determined by the messageType, the following section described the permissible types.

PDU Message types

Each ESPDU message type has a different format as specified below. The elements of each message are marshalled in the order in which they appear in the class definition.

Alert

```
Class Alert{
   byte level;
   byte code;
   string detail;
}
```

The Alert message is used for SLS session management. Valid levels are:

- fatal(0x00)
- warning (0x01)
- debug (0x02).
- All codes are normall sent with a level of fatal, unless indicated. Valid codes are: CLOSE_NOTIFY (0x00) (warning)
- UNEXPECTED_MESSAGE (0x01)
- BAD_SPI (0x0A)
- BAD_SERIAL (0x0B)
- BAD_MAC (0x0C)
- HANDSHAKE_FAILURE (0x14)
- BAD_CERTIFICATE (0x15)
- UNSUPPORTED_CERTIFICATE (0x16)
- CERTIFICATE_REVOKED (0x17)
- CERTIFICATE_VERIFICATION_FAILED (0x18)
- ILLEGAL_PARAMETER (0x1E)= 30;
- BAD_PROTOCOL_VERSION (0x1F)
- INSUFFICIENT_SECURITY (0x20);
- NO_RENEGOTIATION(0x28) (warning)
- ERROR (0x32)

The detail string is intended for human consumption and is left unspecified.

Handshake

```
class Handshake{
   int type;
   byte data[];
```

```
}
```

The type member indicates denotes the contents of the data as follows:

- HANDSHAKE_REQUEST = 0
- HANDSHAKE_START = 1
- HANDSHAKE_REPLY = 2
- HANDSHAKE_FINISH = 3

The definition of the various handshake messages follows. In all cases the current value of ID is "SLS:HandshakeStart:v1.0".

In all cases the current value of majorVersion is 0x01 and of minorVersion is 0x00. This indicates the highest version of SLS supported by the sender.

The ADR type are s-expressions as defined in Chapter 6, “Access Control” “SPKI BNF Formats” on page 106, ADR stands for ASCII Data Representation.

Handshake request

```
class HandShakeRequest{
   string ID;
   boolean flag;
   PDU pdu;
}
```

Handshake request is used to request a renegotation of the session parameters.

The boolean flag is set to true if the request includes a PDU, otherwise no PDU is included.

Handshake start

```
class HandShakeStart{
   string ID;
   byte majorVersion;
   byte minorVersion;
   int spi;
   ADR group;
   ADR keyData;
   ADR cipherSuiteList;
```

```
    ADR tags;
    ADR query;
}
```

The SPI member is the session parameter index of the sender of this message.

The ADR type are s-expressions as defined in (Chapter 6, “Access Control” “SPKI BNF Formats” on page 106), ADR stands for ASCII Data Representation.

The group member is the definition of the Diffie-Hellman group. **(See devTeam for valid DH groups)**.

The keyData member is the senders part of the Diffie-Hellman key-exchange (**see devTeam for format of this s-expression**).

The cipherSuiteList member is the list of valid cipher suites in decreasing order of preference **(see devTeam for format of this s-expression, and list of valid cipher suites)**.

The tags member is the list of SPKI tags the sender is requiring the receiver to prove **(see devTeam for format of this s-expression)**.

The query member is the senders query on the recepient. The sender can use this field to declare the operations it wishes to invoke within the session once it is established. This can be used by the receipient to determine what tags it requires the sender to prove. **(see devTeam for the format of this s-expression)**.

Handshake reply

```
class HandshakeReply{
    string id;
    byte majorversion;
    byte minorversion;
    int spi;
    adr keydata;
    adr ciphersuite;
    adr proof;
    adr tags;
    boolean relay;
    string forwardaddress;
    adr signature;
```

```
}
```

The spi member is the Session Parameter Index of the sender.

The keydata member is the senders part of the Diffie-Hellman key exchange.

The member ciphersuite is the senders choosen cipher suite selected from the list of cipher suites in the initial HandshakeRequest message. **(check with devTeam for format of s-expression in this field.)**

The proof member is the list of certificates that proves the tags the sender of the HandshakeRequest message required. (**check with devTeam for format of s-expressions in this field)**.

The tags member is the list of tags the sender is requiring the receiver to prove. This may have been generated by having examined the query field in the intial HandshakeRequest message.**(check with devTeam for format of s-expressions in this field)**

The boolean relay is set to true if this message is to be forwarded (because we are setting up a tunnel). This allows the responder to notify in its first handshake message that it wants to relay the session , so it might not have to prove the tags requested in the HandshakeStart message.

If relay is set to true, the forwardAddress member contains the absolute ESName of the receipient to which this message is to be forwards **(check this with devTeam)**

The signature field is the signature of the hash of this message and the initial HandShakeRequest message. **(Check with devTeam for details of how the hash is calculated and what the format of the s-expression is - Also code suggest handshake is optional?).**

Handshake finish

```
class HandshakeReply{
   string id;
   adr proof;
   adr signature;
}
```

The proof field is the list of certificates that proves the tags that the HandshakeReply message require (in that message's tags field). (**check with devTeam for format of s-expressions in this field)**. The signature field is the signature of the hash of this message and the previous HandshakeReply and HandshakeRequest message.**(Check with devTeam for details of how the hash is calculated and what the format of the s-expression is - Also code suggest handshake is optional?)**

Application message

The e-speak PDU ESPDU is an application message currently specified “Protocol Data Unit PDU” on page 139

Tunnel

If the message type of PDU is tunnel, the data field of the PDU contains another PDU. The outer PDU is removed, the PDU unmarshalled from the data field and forwarded to the correct address denoted by the toAddress field of the inner PDU.

Ping

```
class ping{
   string ID;
   byte type;
}
```

Ping messages are used by SLS for session management.

The current value of ID is "SLS:Ping:v1.0".

Two values of type are defined:

- Request (0x00)
- Reply (0x01)

PDU Marshalling format

An PDU is transmitted as a 32 bit length (in network byte order) followed by the buffer containing the PDU itself.

All data inside a PDU is marshalled in network byte order.

- int is a 32 bit integer
- long is a 64 bit integer.
- short is a 16 bit integer
- char is a 16 bit character
- boolean ismarshalled as a single byte (0x01) for true (0x00) for false.
- float (**ask devTeam**)
- double (**ask devTeam**)
- string is marshalled as a 16 bit length followed by each character in the string as 16 bits per character
- ByteString - what is this (**ask devTeam**)
- byte[] - **ask devTeam can't figure what marshalBytes does**
- ADR are marshalled by converting them to ASCII canonical s-expressions defined in (x-ref SPKI BNF) and then marshalled as a byte array using marshalledBytes (**discuss with devTeam**).

E-speak Protocol Data Units (ESPDUs)

An E-speak Protocol Data Unit is a PDU with a message type of application message. The contents of the data field of the PDU is of type MessageForResource.

```
class MessageForResource{
   byte versionMajor;
   byte versionMinor;
   ADRList tags;
```

```
    short secondaryABIVersion;
    byte payloadType;
    boolean isVoid;
    byte payload[];
}
```

The current value of versionMajor is 2 and of versionMinor is 0. The current value of secondaryABIVersion is 0. This specifies the format of the data field when communicating with core-managed resources.

ADRList is a list of SPKI tags using the *-set form as defined in (Chapter 6, "Access Control" "SPKI BNF Formats" on page 106) **(Check with devTeam - what are these used for. - presenting more certificates?, the tags required for authorizing the operation?**)

The following are permissible values for payload type.

```
METHOD_CALL=0 (a method call)
METHOD_RESULT=1 (a method result)
EXCEPTION=2 (an exception)
EVENT=3 (an event)
OBJECT=4 (an arbitrary object)
```

The value of isVoid indicates whether or not there is a payload. If isVoid is true, there is no payload and it is not marshalled or unmarshalled.

MessageForResource is marshalled using the PDU marshalling defined in "PDU Marshalling format" on page 147.

The payload field contains the message for the resource which is serialized according to the e-speak serialization format "E-speak Serialization Format" on page 166.

The format of messages for core-managed resources is specified in "Format of Payload for Core-Managed Resource Messages" on page 150.

No payload format is specified if the payload type is set to OBJECT. This is for use by applications to communicate with external resources.

Client to Core Communication

Whenever a client sends a message to a resource, it sends an ESPDU message as defined above in “E-speak Protocol Data Units (ESPDUs)” on page 147) to the e-speak core. The e-speak core routes this message to the resource handler. When the e-speak core forwards the message to an external resource handler it places the following data in the route field of the PDU.

```
class routeData{
   string slot;
   boolean specificationNonNull;
   ESmap privateRSD;
   ADR mask;
   ADR serviceID;
}
```

The fields slot, specificationNonNull, mask and serviceID are marshalled using the format defined in “PDU Marshalling format” on page 147. The privateRSD field is marshalled using the e-speak serialization format defined in “E-speak Serialization Format” on page 166.

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 for.

If specifcationNonNull is set to false, privateRSD, mask and serviceID are not marshall.

The privateRSD field is the resource’s private RSD extracted from the resource’s metadate held by the e-speak core. The mask field and serviceID field are also fields from the resource’s metadata maintained by the e-speak core. The mask field tells the resource handler which methods have security disabled. The serviceID field is the service identity for the resource. Both these fields are <tag-expr> as defined in Chapter 6, “Access Control” “SPKI BNF Formats” on page 106.

Format of Payload for Core-Managed Resource Messages

E-speak specifies the payload format for messages sent to and received from Core-managed Resources (`the payload field in ESPDU`). It does not specify the payload format for non-Core-managed Resources. The class `MethodCall` defines the payload format of messages sent to Core-managed Resources. Note that that the messageForResource payload type is set to METHOD_CALL.

```
public class PayloadForCore
{
   string interfaceName;
   string methodName
   Ob[]arguments;
}
```

The first two fields define the interface and method to be invoked. These are marshalled in the e-speak ABI serialzation format. The type `Ob` is any type defined in the e-speak ABI serialization format.

The `methodResult` class specifies the payload format of messages received from Core-managed Resources in response to method invocation when no exception is thrown.

```
public class methodResult
{
   Ob result;
}
```

Where Ob is any object defined in the e-speak ABI serialization format. When this is returned the messageForResource payload type field is set to METHOD_RESULT.

The exceptionResponse class is used when an excpetion is thrown; ob is any object defined in the e-speak ABI serialization format. The MessageForResource payload type field is set to EXCEPTION.

```
public class exceptionResponse
{
   Ob result;
```

```
}
```

Core event messages

When the core generates an event, it sends an ESPDU with the date field contain a messageForResource. This contains a payload type field set to EVENT. The payload is an eventMessage; ob is any object defined in the e-speak ABI serialization format.

```
public class eventMessage
{
   Ob result;
}
```

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 can 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.

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.

```
class bootstrapReply{
   ESname Inbox;
   string InboxSlot;
   ESname CallbackResource;
   ESname ExceptionHandlerResource;
   String anchor;
}
```

Inbox InboxSlot is the ESname of the inbox and the slot allocated by the e-speak core to the client. The CallbackResource filed is the ESname to be used to send messages to the client. This should be used in the fromAddress field of PDU sent by the client.

The ExceptionHandlerResource is deprecated and should not be used.

The anchor field is the URL of the root name frame of the Protection Domain which has been created by the e-speak core for the client. (**Bug: this should probably be an ESname?**)

Core to core communication

Two core-managed resources handle communication between e-speak cores.

- The Connection Manager CM, sets up the initial connection, manages it and closes it down. The ESName for the CM on any given e-speak core is: es://<server>/CORE/ConnectionManager

- The Remote Resource Manager (RRM) is responsible for managing metadata: importing and exporting resources from the remote e-speak core. The ESName for the RRM on any given e-speak core is: es://<server>/CORE/RemoteResourceManager

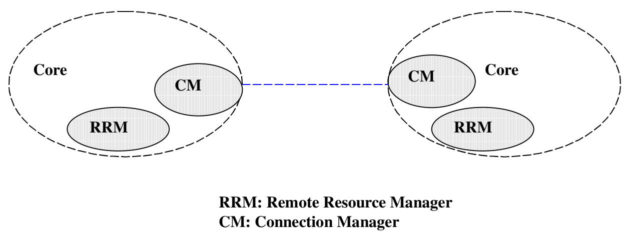


Figure 12 Core-core communication components

Connection Manager

The Connection Manager provides the core to core connection handling APIs. Each connection is associated with a Protection Domain, Outbox and an Inbox. The inbox and outbox along with the Router form the message forwarding subsystem to the remote core (see Figure 5).

The core to core connection can be a secured channel: SLS messages can be exchanged to set up a secure chanel.

```
class connectionManager{
   public  synchronized String openConnection(String coreUrl) 1
      throws UnknownHostException
   public synchronized void closeConnection(String conID)
      throws UnknownHostException
   public synchronized CMArg closeConnectionFromRemote(CMArg cmArg)
      throws UnknownHostException
   public synchronized ESArray getConnections()
}
```

1 The coreUrl should be type ESname, in the current implementation it is type String.

The openConnection() invocation is synchronous. The caller has to wait until the openConnection() returns or times out (the current default time out period is 10 seconds). The parameter is the URL of the remote core's root frame: URL of the form es://host

When the Connection Manager executes this function it sends an ESPDU message to the remote core. This messageForResource has a payload type of METHOD_CALL and the payload field is an instance of payloadForCore. The interface field in payloadForCore is "Core" and the method field is "bootstrap". The toAddress of PDU is set to es://host/CORE (note only es://host is passed as a parameter by the caller of openConnection. The fromAddress is set to es://<localCore>/CORE. Where <localCore> is the host and port for the Connection Manager's e-speak core.[2]

The messageType field of PDU is set to HANDSHAKE for this request.[3]

Once the message has been sent, the Connection Manager waits for a reply ESPDU message. The remote core receiving the message replies with a bootstrapReply message "Initial Connection to the Core" on page 152. In the current implementation this reply is ignored.

Next the Connection factory sends a "negotiate"message to the remote core. This consists of an empty messageForResource instance (payloadType is set to OBJECT and the payload contains the null object). The toAddress of the PDU is set to es://host/CORE/ConnectionManager. The fromAddress of the PDU is set to es://<localCore>/CORE/ConnectionManager.

After this message is sent the Connection Manager waits for a reply from the remote Connection Manager. In the current implementaiton this reply is ignored.

The negotiation message is a place holder for future extensions. The intended future behavior is as follows.

1.Initiator build the negotiation proposal, and sends the proposal to the remote core. The offer includes the parameters like Core Version, ABI version, PDU size (for buffering , fragmentation and reassembly)

2 In the current implementation all URLs created by the connectionManager begin with "tcp://" instead of "es://"

3 This is current implementation behavior, for consistency the messageType should be APPLICATION_MESSAGE

2.The remote core then builds the offer based on the proposal and send the offer to the initiator.

3.The initiator then builds the agreed upon offer and sends the final offer to the remote core.

The returned value of openConnection is a string denoting the server and port of the remote core to which a connection has been made. Thus a connection to es://foo.bar.com:8000/ returns a string: “foo.bar.com:8000”. This string can be used to identify the connection for later connection management operations.

Only one connection exist between 2 cores. Once the connection is estabished subsquent openConnection() requests with the same parameter have no effect

The closeConnection() function performs graceful close of connection between the cores. The closeConnection API sends a close message to the remote core and requests to cleanup the resources allocated to this connection. Upon receving the close request message from a remote core the Connection Manager initiated the clean-up process therby deallocating the resources assigned for the connection.

The parameter conID to closeConnection() is the string previously returned from openConnection(). When closeConnection() is invoked the Connection Manager sends a ESPDU containing messageForResource instance to the remote Connection Manager. The MessageForResource has a payloadType of METHOD_CALL. The interfaceName is “ConnectionManagerInterface” and the methodName is “closeConnectionFromRemote”. The parameter CMArg is marshalled in the e-speak ABI serialization format. It is defined as follows:

```
class CMArg{
   string localURL;
   string remoteURL;
   int type; // CLOSECONNECTIONREQUEST=1 CLOSECONNECTIONREPLY=2
}
```

The localURL field is set to the host+server port for the sending core for example: “initiator.bar.com:8080”. The remoteURL field is set to host+server port for the remote core. The type is set to CLOSECONNECTIONREQUEST. When the remote core receives this message, it can use the localURL, remoteURL pair to identify the connection. The remote core sends a MessageForREsoruce with a payloadType of METHOD_RESULT. The Ob field of the MethodResult class is an instance of

CMArg. The localURL and remoteURL are unchanged, but the type field is set to CLOSECONNECTIONREPLY. Having sent this, the remote core closes the connection. When the initiating core receives this reply, it closes the connection.

The function getConnections() returns the state of current connections. Each element of the returned ESArray is a string of the formed by the IP address of the remote host and the port number on which the remote e-speak core is located separated by a colon, e.g.: “host.foo.com:8000”

Remote Resource Manager

The Remote Resource Manager (RRM) handles metadata related functions. It provide the capability to export and import resources to and from the remote e-speak cores.

```
class payloadForRRM{
   int payloadType;
   boolean topLevel;
   int importExportMode;
   byte[] contextPDU;
   ESArray resourceTable;
   ESArray tablesArray;
}
```

The following are permissible values for payloadType in call payloadForRRM.

```
EXPORT_REQUEST=0; //Export Request
EXPORT_REPLY=1;   //Export Reply
IMPORT_REQUEST=2; //Import Request
IMPORT_REPLY=3;   //Import Reply
UPDATE_EXPORTED_RESOURCE_REQUEST=4; //Update Export Resource
Request
UPDATE_EXPORTED_RESOURCE_REPLY=5; //Update Export Resource Reply
UPDATE_IMPORTED_RESOURCE_REQUEST=6; //Update Import Resource
Request
UPDATE_IMPORTED_RESOURCE_REPLY=7; //Update Import Resource Reply
UNEXPORT_REQUEST=8; //Unexport Resource Request
UNEXPORT_REPLY=9;   //Unexport Resource Reply
```

```
IMPORT_ERROR=10;    //Import Error
EXPORT_ERROR=11;    //Export Error
class RemoteResourceManager{
   void exportResource(ESName esname, boolean topLevel,  int mode,
   String server)
      throws NameNotFoundException, StaleEntryAccessException

   PayloadForRRM importResourceFromMsg(PayloadForRRM rrmPayload)
      throws RemoteException

   void importResource(ESName esname, boolean topLevel, int type,
   String server)
      throws ImportFailedException

   PayloadForRRM exportResourceAsMsg(PayloadForRRM rrmPayload)
      throws StaleEntryAccessException,NameNotFoundException,
      RemoteException

   void unExportResource(ESName esname, String server)
      throws RequestNotDeliveredException

   PayloadForRRM unExportResourceFromMsg(PayloadForRRM rrmPayload)
      throws NameNotFoundException,
         StaleEntryAccessException,
         QuotaExhaustedException,
         InvalidNameException,
         PermissionDeniedException,
         RemoteException

   void updateExportedResource(ESName esname, boolean topLevel, int
   mode, String server)
      throws StaleEntryAccessException,
         NameNotFoundException,
         RequestNotDeliveredException,
```

```
      ExportFailedException

  PayloadForRRM updateExportedResourceFromMsg(PayloadForRRM
  rrmPayload ,
      String fromServer )
     throws NameNotFoundException,
                StaleEntryAccessException,
                PermissionDeniedException,
                InvalidValueException,
                UpdateFailedException,
                RemoteException

  void updateImportedResource(ESName esname, boolean topLevel, int
  type, String server)
    throws NameNotFoundException,
      StaleEntryAccessException,
      RequestNotDeliveredException

  PayloadForRRM updateImportedResourceFromMsg(PayloadForRRM
  rrmPayload)
       throws RemoteException

"EXPORTONCONNECTING"
}

  void exportResource(ESName esname, boolean topLevel,  int mode,
  String server)
    throws NameNotFoundException, StaleEntryAccessException
```

The function exportResource exports the resource identified by esname to the server identified by server. The server parameter is a string of the form host-name:port, for example “foo.bar.com:8080”. The boolean topLevel indicates whether this is to be a recursive export or not. A recursive export exports all resources that are references in the metadata of the resource identified by esname (vocabularies, contracts and the like). If topLevel is false, then the export is recursive. The mode parameter indicates whether this is to be export by reference or export by value. Export by reference, copies the metadata , but not the resource state, so any invocation of the exported resource invokes the same copy. Export by value exports the resource state as well as its metadata, so an invocation of the resource on the remote core results in an invocation on a remote copy of the resource. Recognized values for mode are BY_VALUE (0) and BY_REFERENCE (1). Export by value is only supported for Core-managed Resources. The RRM invokes importResourceFromMsg on a remote RRM to export the resource to that RRM. The payloadForRRM has a payloadType field set to EXPORT_REQUEST.

```
PayloadForRRM importResourceFromMsg(PayloadForRRM rrmPayload)
    throws RemoteException
```

The function importResourceFromMsg is invoked by a remote RRM to tell the RRM to import the resource(s) contained in the argument rrmPayload which is an instance of payloadForRRM. The payloadType field sets to EXPORT_REQUEST. The topLevel field sets to false if the import is recursive. The ImportExportmode field is set to BY_VALUE (0) and BY_REFERENCE (1). The contextPDU is the not used by the RRM, it is returned to the remote RRM. The intent of this field is to enable to remote RRM to identify the context (typically an attempt to send a message) that caused it to invoke importResourceFromMsg.

The ESName of each resource being exported is in resourceTable. If the export is not recursive, this has only one element: the ESName of the original resource passed to the remote RRM as the parameter to exportResource(). If the export is recursive, this ESArray contains all the ESNames of resources included in the metadata of the original resource and all the resources included in the metadata of these resources and soon on. If the export is BY_VALUE then resources include in resource state is also included in the resourceTable.

Each element of tablesArray is itself an ESArray. There is a corresponding ESArray in tablesArray for each element in resourceTable. Each ESArray in tablesArray consists of four elements, taken together they define the resource metadata and state for the resource identified by the ESName in resourceTable.

- short typeCode
- ResourceSpecification spec
- ResourceDescription desc

- Object resource

The following are permissible values for typeCode:

```
INBOX_CODE = 0
META_RESOURCE_CODE = 1
PROTECTION_DOMAIN_CODE = 2
RESOURCE_FACTORY_CODE = 3
CONTRACT_CODE = 100
CORE_DISTRIBUTOR_CODE = 110
IMPORTER_EXPORTER_CODE = 120
MAPPING_OBJECT_CODE = 140
NAME_FRAME_CODE = 150
REPOSITORY_VIEW_CODE = 160
SECURE_BOOT_CODE = 170
SYSTEM_MONITOR_CODE = 180
VOCABULARY_CODE = 190
CORE_MANAGEMENT_SERVICE_CODE = 200
DEFAULT_VOCABULARY_CODE = 210
DEFAULT_CONTRACT_CODE = 220
FINDER_SERVICE_CODE = 230
CONNECTION_MANAGER_CODE = 240
REMOTE_RESOURCE_MANAGER_CODE = 250
EXTERNAL_CODE = 1000
EXTERNAL_RESOURCE_CONTRACT_CODE = 1001
```

ResourceSpecification and ResourceDescription are defined in Chapter 3, “Resource Descriptions, Resource Specifications and Resource Types”.

The resource field is omitted if the resource identified by the ESName in resourceTable is an external resource (typeCode = EXTERNAL_CODE). Otherwise the resource field is the instance of the core-managed resource and contains the data members defined for each Core-Managed Resource type in Chapter 4, “Core-Managed Resources”. The resource field is included even if the export mode is BY_REFERENCE.

```
void importResource(ESName esname, boolean topLevel, int type,
String server)
    throws ImportFailedException
```

This method instructs the RRM to import the named resource from the server identified by the string server (the format of this string is host:port). The boolean topLevel is set to false if the import is to be recursive. Permissible types for type are BY_VALUE or BY_REFERENCE. When this function is invoked on the RRM it sens a message to the remote RRM on the core denoted by the server parameter. This is a messageForResource, containing an instance of payloadForCore with interfaceName “RemoteResourceManagerInterface”, methodName “exportResourceFromMsg” and a single element in the argument array of type payloadForRRM. The payloadType for payloadForRRM is IMPORT_REQUEST. The importExportMode is set to BY_VALUE or BY_REFERENCE. The topLevel field is set to false for a recursive import. The contextPDU field is used by the RRM to identify the context for the reply to this message. Typically it contains a serialized PDU. The resourceTable contains a single element: the esname parameter passed in the call of importResource.

```
PayloadForRRM exportResourceFromMsg(PayloadForRRM rrmPayload)
   throws StaleEntryAccessException,NameNotFoundException,
   RemoteException
```

The function exportResourceFromMsg is called from a remote RRM in response to a invocation of importResource on the remote RRM. The rrmPayload parameter contains the data defined in the description of importResource. The returned PayloadForRRM is sent in a messageForResource which contains an instance of methodResult. This PayloadForRRM has type IMPORT_REPLY, the importExportmode, topLevel and contextPDU fields are those contained in the original rrmPayload. The resourceTable and tableArrays contains the list of ESNames of resources, and there metadata and state as described in the description of importResourceFromMsg.

```
void unExportResource(ESName esname, String server)
   throws RequestNotDeliveredException
```

The function unExportResource causes the RRM to try to unexport the resource from the remote e-speak core identifed by sever (format “host:port”). It does this by sending and instance of payloadForCore with interfaceName “RemoteResourceManagerInterface”, methodName “unExportResourceFromMsg” and a single element in the argument array of type payloadForRRM. The payloadType for payloadForRRM is UNEXPORT_REQUEST. The topLevel and importExportMode fields are not used for this request. The resourceTable field contains a single element: the esname parameter to the function unExportResource. The tablesArray field is empy.

```
PayloadForRRM unExportResourceFromMsg(PayloadForRRM rrmPayload)
   throws NameNotFoundException,
      StaleEntryAccessException,
      QuotaExhaustedException,
      InvalidNameException,
      PermissionDeniedException,
      RemoteException
```

The unExportResourceFromMsg is invoked from a remote RRM in response to a call of unExportResourceFromMsg on the remote RRM. The rrmPayload contains the data defined in the description of unExportResourceFromMsg. The intent is that the RRM receiving an invocation of unExportResourceFromMsg should remove te resource contained in rrmPayload from its repository. ThePayloadFor-RRM instance returned contains a payloadType of UNEXPORT_REPLY and the contextPDU passed in the rrmPayload parameter. All other fields are unused.

```
void updateExportedResource(ESName esname, boolean topLevel, int
mode, String server)
   throws StaleEntryAccessException,
      NameNotFoundException,
      RequestNotDeliveredException,
      ExportFailedException
```

The effect of calling the updateExportedResource function is very similar to calling ExportResource. The difference is that the RRM invokes the updateExporte-dResourceFromMsg function on the remote RRM (instead of ExportResource-FromMsg) and the payloadType of the PayloadForRRM instance passed as a parameter in the remote invocation is set to UPDATE_EXPORTED_RESOURCE_REQUEST.

```
PayloadForRRM updateExportedResourceFromMsg(PayloadForRRM
rrmPayload ,
      String fromServer )
    throws NameNotFoundException,
```

```
                    StaleEntryAccessException,
                    PermissionDeniedException,
                    InvalidValueException,
                    UpdateFailedException,
                    RemoteException
```

The function updateExportedResourceFromMsg is invoked by a remote RRM when it wishes to update resources which have been exported previously. The intent is that the RRM receiving the call of this function replaces the metadata (and state in the case of an export BY_VALUE) of each resource in the resourceTable in rrmPayload with the metadata and state contained in tablesArray. Only resources that have been already registered have there metadata and state updated. The PayloadForRRM returned as the result have the contextPDU of the rrmPayload parameter and a payloadType of UPDATE_EXPORTED_RESOURCE_REPLY. All other fields are ignored by the remote RRM that invoked updateExportedResourceFromMsg when it receives this result.

```
void updateImportedResource(ESName esname, boolean topLevel, int
type, String server)
  throws NameNotFoundException,
    StaleEntryAccessException,
    RequestNotDeliveredException
```

The updateImportedResource function is similar to the importResource function. The major difference is that the RRM has previously imported the resource identified by esname. The RRM invokes the updateImportedResourceFromMsg on the remote RRM identified by the string server (host:port). The PayloadForRRM passed as a parameter in updateImportedResoruceFromMsg has payloadType of UPDATE_IMPORTED_RESOURCE_REQUEST and has a single element in resourceTable: the ESname of the resource that needs updating. Note that this can not be the same ESName that is received as a parameter to updateImportedResource, it must be the ESName used to identify the Resource when it was originally imported. So the RRM must remember this information about imported resources if it is to request updates of metadata.

```
PayloadForRRM updateImportedResourceFromMsg(PayloadForRRM
rrmPayload)
     throws RemoteException
```

The function updateImportedResourceMsg is invoked by a remote RRM when it has received an invocation of updateImportedResource and needs to update a resource's metadata (and possibly state). The rrmPayload contains the data described in the descripion of updateImportedResource above. The PayloadFor-RRM returned from the updateImportedResoruceFromMsg function has a payload-Type of UPDATE_IMPORTED_RESOURCE_REPLY. The topLevel, importExportMode and contextPDU fields are the same as in the rrmPayload parameter. The resourceTable field and tablesArray respectively contain the ESnames and metadata (and possibly state) of the resources to be updates. Note that even though a single resource ESname isall that is contained in the rrmPayload parameter, the result can contain many ESnames if the topLevel flag as set to false (indicating recursive import/export).

Restrictions on import and export of core managed resources

The following Core-managed Resources cannot be exported or imported.

- Protection Domain
- Meta Resource
- Resource Factory
- Inbox
- System Monitor
- External Resource Contract

The Account Manager cannot be exported by value, only by reference.

Othe Core-managed Resources have restrictions when exported by reference. In particular, such a Resource cannot be used as part of message processing as shown in Table 10 ..

Table 10 Core-managed Resource export restrictions

Resource	Pass by reference restrictions
Name Frame	Cannot be used as a component of an ESName sent to the Core for name resolution
Repository View	Cannot be used in a Search Recipe
Resource Contract	Cannot register a Resource in this Contract
Vocabulary	Cannot be used in a Search Recipe

Removing imported resources (informational)

The Connection Manager provides the closeConnection() function to perform a graceful shutdown of a connection. The Connection Manager builds a close connection message and sends the message to the Connection Manager of the remote core, requesting for the connection cleanup process. In the current implementation, the Connection Manager on the remote core removes the Protection Domain, Outbox and other resources allocated to the connection. Removing the Protection Domain used for the connection remove all resources imported from the connection, as they are registered in this Protection Domain.

The initiating core also performs similar clean up process. The Protection Domain, Outbox assigned to the connection are removed.

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.

All elements are sent a signal byte that indicates the type of the object that follows, followed by the data for that object.

Signal bytes are entirely single bytes. They are encoded by literal ASCII characters (e.g., A), literal ASCII characters but with the high byte set (char)('J'+128)).

Here is the Backus-Naur Form (BNF) for “Ob” an object serialized in the e-speak ABI format.are the signal bytes defined currently.

```
Ob = 'E', <RSD.class>|
'D', <ResourceDescription.class>|
'S', <ResourceSpecification.class>|
'[', <ESUID.class>|
'c', <SearchPredicate.class>|
'C', <SearchRecipe.class>|
']', <VocabularyDeclaration.class>|
```

```
'_', <Preference.class>|
'`', <FilterSpec.class>|
'T', <AttributeProperty.class>|
'A', <AttributePropertySet.class>|
'a', <Attribute.class>|
'B', <AttributeSet.class>|
'V', <Value.class>|
'Y', <ValueType.class>|
'F', <ESName.class>|
')', <ESString.class>|
'q', <AttributePredicate.class>|
'<', <NamedObject.class>|
'=', <ProfileAttributeSet.class>|
'>', <UserProfile.class>|
'N', <NameSearchPolicy.class>|
'.', <FinderResults.class>|
'/', <FinderContext.class>|
'Q', <CoreEvent.class>|
'e', <Event.class>|
't', <EventAttributeSet.class>|
'Z', <ESRuntimeException>|
'z', <ESException>|
(char)('z'+128), <ESArray.class>|
(char)('s'+128), <ESSet.class>|
(char)('I'+128), <Integer.class>|
(char)('J'+128), <Long.class>|
(char)('B'+128), <Boolean.class>|
(char)('y'+128), <Byte.class>|
(char)('C'+128), <Character.class>|
(char)('W'+128), <Short.class>|
(char)('F'+128), <Float.class>|
(char)('D'+128), <Double.class>|
(char)('S'+128), <String.class>|
(char)('Z'+128), <boolean[].class>|
(char)('b'+128), <byte[].class>|
(char)('c'+128), <char[].class>|
(char)('w'+128), <short[].class>|
```

```
(char)('i'+128), <int[].class>|
(char)('j'+128), <long[].class>|
(char)('f'+128), <float[].class>|
(char)('d'+128), <double[].class>|
(char)('v'+128), <Object[].class>|
(char)('H'+128), <ESMap.class>|
'I', <PayloadForRRM.class>|
':', <CMArg.class>|
'h', <RepositoryHandle.class>|
'r', <Contract.class>|
'W', <RepositoryView.class>|
'v', <Vocabulary.class>|
'm', <MappingObject.class>|
's', <NameFrame.class>|
'o', <Binding.class>|
'?', <ProtectionDomain.class>|
'%', <Inbox.class>|
'@', <ExternalResource.class>|
'^', <SystemMonitor.class>|
'+', <ResourceFactory.class>|
'y', <Finder.class>|
'n', <ConnectionManager.class>|
'd', <RemoteResourceManager.class>|
'-', <CoreManagementService.class>|
'~', <AccountManager.class>
```

The components for each of the remaining non primative type are defined in the relevant sections of this specification (**to do: need to pull these definitions in to complete the BNF**).

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

```
String        => string
Integer       => int
Long          => long
Boolean       => byte
```

```
Null          =>
ByteArray     => byte[]
ObjectArray   => Ob[]
ESMap         => map
ESArray       => Ob[]
ESSet         => Ob[]
ESList        => Ob[]
```

ESMap, ESArray, ESSet and ESList in the current implementation are marshalled using Java serialization.

References

1. The TLS Protocol version 1.0 RFC 2246, IETF by T. Dierks and C. Allen January 1999.
2. Recommendations for Elliptic Curve Cryptosystems by G. Seroussi and Nigel Smart HPL Technical Report 1999-148.
3. Simple Public Key Certificate. INTERNET-DRAFT September 98 by Carl M. Ellison et al. **This needs to be replaced with a reference to the RFC**

Chapter 8 Exceptions

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.

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 attempts to notify all Clients of its inability to continue operating, it also replies with this exception for as long as it can. The Core can 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 can be a terminal error for the service, in which case the service exits. Or it can simply mean that the request being processed caused an internal error that was not recoverable, and the service accepts 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.
- ConnectionFailedException is thrown when the Connection Manager fails to establish a connection, details are contained in the exception state.
- `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 Handler's Inbox is full. Recovery for this case can be as simple as resending the message.

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

`ESLibException` is the base clase for client library exceptions. It should not be thrown itself but rather a subclass exception.Currently one subclass is defined.

- `CoreNotFoundException` indicates that a core could not be found to connect to. Either change the specification of the core or insure the core is running to correct this exception.

`ESInvocationException` is a base class for all the exceptions that can 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.
- `LoopDetectedException` is currently unused.

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

`PermissionDeniedException` is thrown by any Resource Handler when the client is not authorized to access the Resource. The Client can recover by retrying with a different set of certificates. One subclass is defined

- `SessionRequiredException` is thrown when a client attempts to send it a message without first setting up a session. The service has security enabled and is performing access control checks. A secure session is needed so that the access control check can be made. This would normally be handled by the client library and is transparent to the application programmer. The client recovers from this exception by exchanging SLS messages with the service to establish a session.

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

`MethodNotImplementedException` is thrown when the Client attempts to invoke a method on a Resource that is not implemented even though the method is consistent with the type of the Resource. This is typically used to "stub-out" routines when a service is under development.

`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 metadata.

`TimedOutException` is thrown when a message being written to or received from a channel has not successfully completed within the caller defined time period.

`UndeliverableRequestException` is thrown when the message cannot be delivered to the Resource Handler. In the current implementation, this is not thrown with security enabled, the security subsystem silently ignores such messages (in case they are a denial of service attack) and the Client has to wait for a TimeOutException There are two subclass exceptions of `UndeliverableRequestException`:

- `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.

- ESNameFrameException is the super class of all name frame exceptions.This allows the client to catch this exception and handle all the name frame related exceptions in one catch block.
  - `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.
- `ESRemoteException` is thrown if the Remote Resource Manager operation failed for any reason, details are in the exception state.

Exception State

Each exception has the following state.

```
class ESException {
   int errno;
   Object[] info;
}
```

The field errno indicates the type of the exception as shown below.

```
NONE= 0
```

ESRuntimeExceptions (1-99 reserved)

```
INVALID_PARAMETER= 1
NULL_PARAMETER= 2
INVALID_VALUE= 3
```

```
INVALID_TYPE= 4
OUT_OF_ORDER_REQUEST= 5
CORE_PANIC= 6
SERVICE_PANIC= 7
REPOSITORY_FULL= 8
```

ESExceptions (100-999 reserved)

```
INVOCATION= 100
NAMING= 101
NAME_NOT_FOUND= 102
EMPTY_MAPPING= 103
UNRESOLVED_BINDING= 104
MULTIPLE_RESOLVED_BINDING= 105
PERMISSION_DENIED = 106
QUOTA_EXHAUSTED= 107
STALE_ENTRY_ACCESS= 108
RECOVERABLE_CORE= 109
PARTIAL_STATE_UPDATE= 110
REQUEST_NOT_DELIVERED= 111
UNDELIVERABLE_REQUEST= 112
UNRECOVERABLE_DELIVERY= 113
RECOVERABLE_DELIVERY= 114
TIMED_OUT= 115
METHOD_NOT_IMPLEMENTED=116
LOOP_DETECTED= 117
SESSION_REQUIRED= 118
CONNECTIONFAILED= 119
```

E-speak defined service exceptions

```
SERVICE= 200
```

Name frame service exceptions

```
NAMEFRAME= 201
INVALID_NAME= 202
NAME_COLLISION= 203
LOOKUP_FAILED= 204
```

Import/Export service exceptions

```
REMOTE= 210
```

Client Library defined exceptions

```
ESLIB = 950
CORE_NOT_FOUND= 951
```

Exception numbers 1000+ are reserved for application use

Exception hierarchy

Here is the exception hierarchy. Indentation indicates position in the hierarchy.

```
ESRuntimeException
      ServicePanicException
      OutofOrderRequestException
      CorePanicException
      ConnectionFailedException
      RepositoryFullException
      ConnectionFailedException
      InvalidParameterException
            InvalidTypeException
            InvalidValueException
            NullParameterException
ESException
      ESLibException
            CoreNotFoundException
      ESServiceException
            ESRemoteException
            ESNameFrameException
               NameCollisionException
               InvalidNameException
               LookupFailedException
      ESInvocationException
            StaleEntryAccessException
            PermissionDeniedException
               SessionRequiredException
            QuotaExhaustedException
            MethodNotImplementedException
            RecoverableCoreException
               PartialStateUpdateException
               RequestNotDeliveredException
            NamingException
               MultipleResolvedBindingException
               UnresolvedBindingException
               NameNotFoundException
               LoopDetectedException
               EmptyMappingException
            TimedOutException
            UndeliverableRequestException
               RecoverableDeliveryException
               UnrecoverableDeliveryException
```

Chapter 9 Core Generated Events and Event Distributor Vocabularies

The e-speak event service is described in the E-speak Programmer's Guide. The Core itself is an example of an Event Publisher. It sends Events to an external Client called the *Core Distributor* to signal state changes such as a change in a Resource's attributes. The Core Distributor can then distribute these Events to interested Clients that have appropriate authority.

Events

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 can 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:

```
class Event
{
   String eventType;
   EventAttributeSet eventAttrs;
   EventAttributeSet controlAttrs;
   Object payload;
}
```

Every event has a string that describes the event type and two AttributeSets (sets of name-value pairs). The first AttributeSet is the attributes that describe the event. The second AttributeSet are control attributes that intermediating entities (such as distributors) can insert into the event. Matching (filtering) can only be performed on the event attributes, not on the control attributes. An string is valid for eventType, the meaning of the string is application dependent.

`EventAttributeSet` contains an AttributeSet (xref to chapter on resourceDescriptions, section on Resource Description defines AttributeSet):

```
class EventAttributeSet extends AttributeSet
{
   AttributeSet attrs;
   String format;
}
```

The string format indicates the format of each `Attribute` in the AttributeSet attrs. "VOCAB" means that the attributes have to be valid in the vocabulary references in the AttributeSet attrs. "SIMPLE" means the attributes are simple (name, value) pairs and no valid vocabulary is specified in attrs.

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 string in the eventType field for events generated by the e-speak Core consists of a prefix indicating the component that generated the event, followed by further information (such as the name of the method being invoked).

The eventAttrs field consists of a set of name, string-value pairs. Two common examples are:

- name "Name", value is the stringified version of the ESUID of the Resource responsible for generating the event
- name "FailureDetail", value is a string indicating the nature of the failure

The format string of the EventAttributeSet eventAttrs is "SIMPLE".

The payload field is null for events generated by the e-speak Core.

The following is the list of prefix strings used by the current implementation.

```
"core.mutate.NameFrameInterface."
"core.mutate.MailBoxInterface."
"core.mutate.ProtectionDomainInterface."
"core.mutate.RepositoryViewInterface."
"core.mutate.VocabularyInterface."
"core.mutate.VocabularyToolBoxInterface."
"core.mutate.ResourceFactoryInterface."
"core.mutate.ResourceManipulationInterface."
"core.mutate.ImporterExporterInterface."
"core.mutate.SecureBoot."
"core.failure."
"core.failure.exception."
"notifySync"
"notify"
"publish"
"subscribe"
"unpublish"
"unsubscribe"
"net.espeak.services.events.intf.ESListenerIntf"
"net.espeak.services.events.intf.DistributorIntf"
"net.espeak.jesi.event.coredist.ESCoreDistributorIntf"
"net.espeak.infra.cci.events.Event"
"service.create"
"service.delete"
"service.mutate"
"service.access"
"service.pause"
"service.resume"
"service.panic"
"service.genericInfo"
"management.service.create"
"management.service.coldreset"
"management.service.warmreset"
"management.service.stop"
"management.service.start"
"management.service.shutdown"
"management.service.remove"
```

```
"management.service.error"
"management.service.info"
"management.service.illegalstate"
"management.servicemanager.newservice"
"management.servicemanager.deleteservice"
"management.servicemanager.servicechanged"
"resource.change_state"
"resource.invalid_state"
"resource.invalid_state_transition"
"resource.statistics"
"resource.proxy_created"
"coremanager.info"
"coremanager.warning"
"coremanager.serious"
```

Publication of Core-generated Events

The e-speak core sends events to the core distributor as a Protocol Data Unit containing a MessageForResource (xref to ESPDU section in communications chapter). The payload field of the MessageForResource is the event. The payloadType of MessageForResource is set to EVENT.

The e-speak Core does not subscribe to the Core Distributor (as an ordinary Client would).

Distributor Vocabulary

A vocabulary is defined in which Distributors can be registered.

Table 11 Distributor vocabulary

Attribute Name	Value Type	Comment	Meaning
Name	String	Default value "BaseDistributorVocabulary"	
Type	String		
ESGroup	String		
ContractNames	String		
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

Table 11 Distributor vocabulary (Continued)

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 can 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.

Events in a Distributed Environment (Informational)

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 is 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 can generate an Event that reaches the first Client. Not having any knowledge of the source of the Event, the Client makes the change again. These two Clients 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-speakcomponent can be guaranteed to break such cycles. Only the Clients have sufficient information to detect the problem.

Chapter 10 Management

Two concepts that underpin the manageability of e-speak Resources and e-speak Clients.

- Managed State: a defined service state embodying the life cycle of a service.
- Managed Variable Table: sets of values that can be affected by a manager for the purposes of configuration and control.

Managed Life Cycle

The full state transition diagram is as follows.

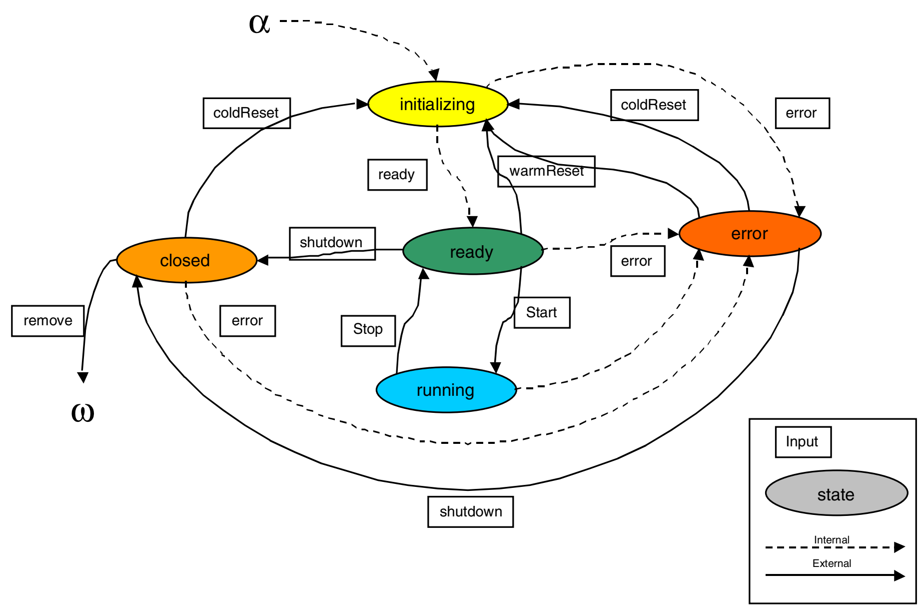


Figure 13 Managed Service Lifecycle

State Descriptions

Initializing

The internal dynamic state of the service is being constructed, for example: a policy manager is being queried for configuration information and resources are being discovered via search recipes or yellow pages servers. When the service finishes this work it moves asynchronously into the ready or error states.

Ready

The service is in a ready to run situation, this state is also equivalent to a stopped or paused state.

Running

The service is running and responding to methods invoked on its operational interfaces. If an error occurs which implies that the service cannot continue to run it should move into the error state.

Error

The service has some problem and is awaiting management action on what to do next.

Closed

The service has removed/deleted much of its internal state and awaits either a coldReset or remove transitions.

Inputs

An input is the trigger that causes a state transition to occur. In any given state there is a defined set of permissible inputs that are available, i.e. only those that are depicted in the diagram as leaving the current state and connecting with the next state. To attempt to perform any other transition is illegal. Note that many inputs can have the same name (e.g. error) yet there is no ambiguity as long as the originating state is different.

Clients can provide any input with impunity However a management agent can request only provide external inputs. For example the manager might reasonably request that a client perform a warm reset, but not to become ready, the client alone can provide this input i.e. when it's internal initialization process has completed.

The available inputs are as follows.

- start: move into the running state. Start to handle invocations on operational interfaces.
- stop: move into the ready state. Stop handling invocations on operational interfaces.
- ready: move into the ready state having finished initialization.

- error: move into the error state, this transition is valid from any state.
- shutdown: clean up any internal state required and move into the closed state. This transition should not cause the deregistering of resources from the repository.
- coldReset: cause a from complete reinitialization of the service and move into the initializing state. The only exemption is that resources that are already registered should not be reregistered.
- warmReset: cause a partial reinitialization of the service i.e. retaining some of the existing service state move into the initializing state.
- remove: cause the service to remove itself from existence. Any non-persistent resources should be deregistered from the repository.

Managed Variable Tables

A managed variable table is at it's simplest a table of name/string value pairs that exist within the client but to which a manager has some level of access. Thus a management agent can control those aspects of a services behavior that is affected by those variables to which it has access.

There is a degree of configurability associated with managed variables and their variables that permit something more sophisticated than the simple get and set operations one would expect to find.

Each table itself has a name to distinguish it from other tables. As we shall see later, the managed service model itself provides for two such tables.

There is a restriction on variable table usage: each name in a variable table must be unique within that table. It is not possible to implement lists by having many entries with the same name.

Configuration Parameter Table

The configuration parameter table is an instance of a managed variable table with a reserved name that identifies it as such. The table holds generic configuration data for the client.

Resource Table

The resource table is another instance of a managed variable table, identical in behavior to the configuration parameter table except that the names in the client's table refer to other services with which the client has some relationship. For example, if a particular client makes use of a mail service then this relationship can be made visible to a management agent through the resource table. Thus a management agent might reconfigure the client to use an alternative but equivalent service. While there might seem no obvious need to separate out this particular aspect of configuration, doing so makes it possible for a management agent to discover the topology and integrity of a network of connected services without the need for service specific interpretation of the variable table (all entries in the resource table are resources).

The name used for an entry in a resource table can be any symbolic name the client chooses, while the value must be the valid e-speak ESName of the actual service.

Managed Service Interface

All e-speak Resources that are manageable implement the ManagedService interface. This applies whether the Resources are external to the e-speak Core, or Core-managed.

```
interface ManagedServiceIntf{
   String getName()
      throws ESInvocationException;

   String getDescription()
      throws ESInvocationException;
```

```
String getOwner()
   throws ESInvocationException;

String getUptime()
   throws ESInvocationException;

String getVersion()
   throws ESInvocationException;

String getErrorCondition()
   throws ESInvocationException;

String getStaticInfo()
   throws ESInvocationException;

void coldReset()
   throws IllegalStateTransition,ESInvocationException;

void warmReset()
   throws IllegalStateTransition,ESInvocationException;

void start()
   throws IllegalStateTransition, ESInvocationException;

void stop()
   throws IllegalStateTransition, ESInvocationException;

void shutdown()
   throws IllegalStateTransition, ESInvocationException;

void remove()
   throws IllegalStateTransition, ESInvocationException;

int getState()
   throws ESInvocationException;

VariableEntry[] getVariableEntries()
```

```
    throws ESInvocationException;

  String[] getVariableNames()
    throws ESInvocationException;

  VariableEntry getVariableEntry(String name)
    throws ESInvocationException, NoSuchVariableName;

  void setVariable(String name, String value)
    throws ESInvocationException;

  ResourceEntry[] getResourceEntries()
    throws ESInvocationException;

  String[] getResourceNames()
    throws ESInvocationException;

  ResourceEntry getResourceEntry(String name)
    throws NoSuchVariableName, ESInvocationException;

  void setResource(String name, ESName resource)
    throws ESInvocationException;
  }
```

The method getName return String containing the service name. This name should be used when registering the service resource in the service vocabulary.

The method getDescription returns a human readable description of the service for display on a management console.

The method getOwner returns a string indicating the owner of the service.

The method getUptime gets the time for which the service has been running. The format of the string is “years.days.hours.minutes.seconds”.

The method getVersion returns a string indicating the version of the service.

The method getErrorCondition returns a string indicating the error condition. This returns null if the service is not in an error state.

The method getStaticInfo returns an XML document of the following form.

```
<staticInfo>
<name>the name of the resource </name>
<owner> the name nameof the onwning service </owner>
<description> the decription here </description>
<version> the version string </version>
<uptime> the uptime string </uptime>
</staticInfo>
```

The coldReset transistion function cause the service to move into the initializing state and completly reinitialize. The exception IllegalStateTransitionException is thrown if the state is not in the ready, error or closed states.

The warmReset transistion function cause the service to move into the initializing state and partially reinitialize. The exception IllegalStateTransitionException is thrown if the state is not in the ready or error states.

The start transistion function cause the service to move into the running state and service client requests. The IllegalStateTransitionException exception is thrown if the state is not in the ready state.

The stop transistion function cause the service to move into the ready state and stop serving client requests. The exception IllegalStateTransitionException is thrown if the state is not in the running state

The shutdown transistion function clean up any internal state required and move into the closed state. This transition should not cause the deregistering of resources from the repository. The exception IllegalStateTransitionException is thrown, if the state is already in the closed state.

The remove transition function causes the service to remove itself from existence. Any non-persistent resources should be deregistered. The exception IllegalStateTransitionException is thrown if the state is not in the closed state.

The method getState return the current state: an integer value from 0 to 4.

The value returned is interpreted as follows.

- Initializing(0) - the service is contructing its internal data structures and finding other services which is needs to function.

- Ready(1) - the service is fully constructed and ready to run.
- Running(2) - the service is running and handling methods on its operational interfaces.
- Closed(3) - the service has deleted much of its internal state and closed any open connections to files or other services.
- Error(4) - The service has encountered an error preventing the service from continuing to operate.

The Variable Table

Each manageable Resource maintains a table of name value pairs, which contains whatever information that Resource wishes to expose to the management agent. The table entries can be either read only or read write.

```
class VariableEntry {
   String name;
   String value;
   int updateType;
}
```

The updateType is interpreted as follows (**DevTeam needs to provide this information.**)

The method getVariableEntries returns the table as an array of VariableEntry's. Each VariableEntry object contains the name, the value & update information.

The method getVariableNames returns an array of strings - one element in the array for each variable.

The method getVariableEntry returns the entry in the table for variable identifed in the parameter name.

The method setVariable sets the variable identified by the parameter name to the string in the value parameter.

The Resource Table

The managed Resource maintains a table of name-Resource pairs. This table contains all the Resources that the element depends on i.e. uses. The table entries can be either read only or read write.

```
class ResourceEntry {
   String name;
   ESName resource;
   int updateType;

}
```

The method getResourceEntries returns the table as an array of ResourceEntry. Each entry contains a string that name for the resource, the ESName of the resource (URL) and the update information. The updateType is interpreted as **(Need information from DevTeam.)**

The method getResourceNames returns an array of strings, one element for each entry in the resource table.

The method getResourceEntry(String name) returns the entry in the table for the named resource.

The method setResource sets the Resource identified by the name parameter to the ESName supplied in the resource parameter.

Chapter 11 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 can 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. The following illustrate the methods that need to be supported in these interfaces. The exact signatures and functions vary from implementation to implementation. In the current implementation these interfaces can be found in net.espeak.infra.core.repository.Repository

```
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 can access permanent storage, but the protocol used for such access is not part of thee-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 can 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 can act as a server to a thin Repository. Clearly, such a Repository can 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 can use a fat Repository to fulfill its scalability needs, and a fat Repository can simultaneously serve many thin Repositories. However, many devices cannot 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.

The keyIndexType field: Efficient Repository Lookup

In DBMS, indexes are the primary means of reducing the volume of data that must be fetched and processed in response to a query. If there were no indexes used for resource description attributes in an e-speak repository, every resource defined against a particular vocabulary needs to be examined to see if it matches the constraints specified in the search recipe. This would cause very slow performance on lookups when large numbers of resources are registered in the e-speak Core. So there needs to be a way of specifying which attributes properties within a Vocabulary are the 'key' attributes so that some indexing scheme can be added.

It is not reasonable to index each and every attribute in a resource description. The more indexes that you have, the more overhead in registering descriptions and also the memory requirement becomes more for in-memory repository. It does not make sense to index attributes that are not going to be frequently used in constraints. Therefore, there needs to be a way of specifying which attribute properties within a vocabulary are the 'key' attributes so that some indexing scheme can be implemented on these 'key' attribute properties.

This is the purpose of the keyIndexType field in AttributeProperty (xref to core managed resource Vocabulary section). Valid values of keyIndexType are: NO_INDEX, HASH_INDEX and TREE_INDEX. If the value is HASH_INDEX or TREE_INDEX the attribute is used as an index by the DBMS.

Chapter 12 Localization

A key factor in global acceptance of a software package is its ability to be customized to the location running the software. It is very frustrating for a user to have to read messages in a language other than their own native language or interpret numbers using a foreign format. Imagine if you had to understand an error message written in German, or recognized that the string "06/01/99" really means January 6th and not June 1st.

This chapter describes how to support localizing e-speak for native language and locale-dependent number & date formats. This design is implemented in the current release of e-speak. However, currently all entities have the same underlying data catalog to get their localized strings.

Current Implementation

The current code has hard-coded strings for all display messages and exception details. It also hard-codes the format of number and date/time representations.

For example:

- net.espeak.infra.core.startup.StartESCore prints a message when the core is initialized using System.out.print("Starting ES Core Server with Rendezvous of \"" + popURL + "\". "); and System.out.println("started.");.
- Value.getString() simply calls the toString method of the data type object represented by the Value class.

Requirements

String Messages

There are three requirements for string messages:

- A framework implementation which supports the use of localized string messages.
- A English implementation of all string messages within the core, cci and client packages, using the framework created above.
- Additional language implementations as required by our customers.

Framework

- Any time the message text is moved away from the code that produces the message, confusion and incorrect messaging is likely. It is important, therefore, that the framework minimized the confusion and makes it difficult to issue the incorrect message. A hierarchical structure must be supported for the specification of the message to be issued.
- Messages are rarely static, i.e., they often contain concatenations of variable values in the middle of the message. The framework must support the substitution of variable values in the body of the message.
- The framework must support the specification of the location and language of the user. If support for the requested location and language are not implemented, the framework should provide the closest match to the requested location and language available.
- A likely scenario includes the core running in one locale and the client running in a different locale. The framework must support a core issuing a message in the client's locale language.
- During development phases, the framework should throw exceptions if the invocation of the messaging methods are coded incorrectly (e.g., a message id that is not valid), but in the release the framework should make a best attempt to format the message for the user.

- The framework must be initialized during the startup of the e-speak processes.

English implementation

1. The current code base must be examined for each string message that is produced. Unless there is a good reason for keeping the message definiton local (e.g., a debug message), the text of the message should be placed in the English string implementation file and the reference changed to retrieve the message text.
2. This English implementation becomes the base implementation and is shown to the user as the default language if their specified language is not implemented.

Additional language implementations

- After a good English implementation has been developed additional language implementations can be created translating the message text from English to the new language.
- After an additional language is created, changes to existing messages must also be changed in each of the additional languages. This is a development process issue that is addressed further here.
- If a new message is created in the base implementation (English), the new message does not need to be implemented in all other languages, however, if this is the case the user sees the English version of the message.

Number & Date Formats

The requirements for non-string formats is broken into two categories:

- Vocabulary attributes
- Value class string representations

Vocabulary Attributes

Three new data types should be supported which provide for locale-defined formats. They are:

- Decimal
  This data type provides for a decimal representation of a number in a user-defined pattern. The pattern can be derived from the locale-defined format or customized by the user. For example: Decimal number = new Decimal("###,###.##");
- Currency
  This data type provides a specialized Decimal format that includes the currency symbol and format defined by the user's locale.
- Percent
  This data type provides a specialized Decimal format for percentages using the symbols and format defined by the user's locale.

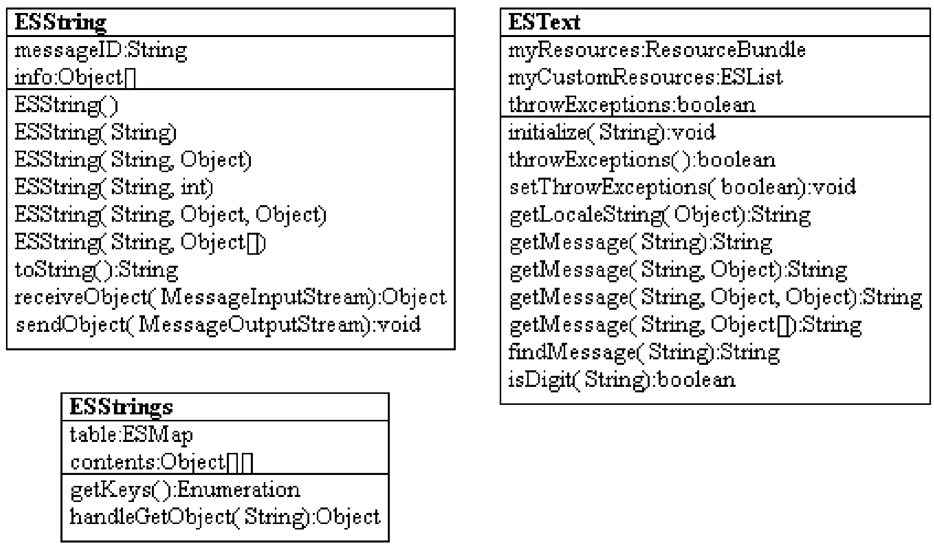
Value class string representations

The Value class getString method should return a string representation that is customized by the user's locale formats. Specifically:

- Timestamp
- Time
- Currency
- Numeric data type (Long, Double, Float, etc.)
- Date
- Decimal
- Percent

High-level Design

The implementation for the Number and Date formats are left to the Vocabulary team. This document only addresses the String Message requirements. Shown below is the class diagram for the classes implementing localization.



ESText, ESStrings & ESString classes

Three new classes are defined. ESText is the retrieval class and ESStrings is the language dependent implementation class. ESString is a logical extension to String which performs the localization at the last possible moment (client in most cases).

The initalize() method needs to be called by each process that uses the ESText facility. If this method is not called, the first invocation of getMessage defaults to the e-speak base class. The initalize method uses the java.util.ResourceBundle class to discover the language defined strings. ESText supports mutiple base classes. After it is initalized, it can be called multiple times with different base class name parameters. When ESText looks up a message, it searches all the supplied base classes to resolve the message ID.

The throwExceptions and setThrowException methods return and specify if exceptions are thrown for detected problems (see below).

Additional getMessage prototypes can be created with multiple params if the need arises.

The base implementation for ESStrings looks like the following:

```
public class ESStrings extends ListResourceBundle
{
        public Object[][] getContents() {
           return contents;
        }
        static final Object[][] contents = {
           {"net.espeak.startup.Hello", "template for ID1"},
           {"net.espeak.eslib.ESFolder.dup", "template for ID2"}
        };
      }
```

Each additional language looks like the following:

```
public class ESStrings_de extends ListResourceBundle
{
        public Object[][] getContents() {
           return contents;
        }
        static final Object[][] contents = {
           {"net.espeak.startup.Hello",
              "German override for ID1"},
           {"net.espeak.eslib.ESFolder.dup",
              "German override for ID2"}
        };
      }
```

Class names are searched in the following order:

1 baseclass + "_" + language + "_" + country + "_" + variant

2 baseclass + "_" + language + "_" + country

3 baseclass + "_" + language

4 baseclass

ResourceBundle automatically defers to this search order for any message ID that is not found in the specific language implementation or if the specific language implementation is missing.

The optional param values are substituted in the message text by the following rules:

- For each occurance of the string "%n", the string is replaced by the object[n].toString() value. Note that this is a zero-based index.
- If "n" is out of bounds for the supplied params, the string "n/a" should be substituted. Note: during development, this situation threw an exception.
- If a param is not referenced by the message, the param should be ignored. Note: during development this stituation is thrown an exception.
- Multiple references to the same param should be valid (e.g., "%0 blah %0").
- If the included object is a localizable object (Timestamp, Date, Time or Number) the locale-defined formatting rules are applied to this object.
- To code a percent sign in the message, code two percent signs (%%).
- To include all passed parameters, code "%all" in the message template. This is be replaced by [arg0, arg1, ...].

The last class is ESString. This class logically extends the java.lang.String class for localization. It accepts a message ID and optional data objects in the same way as ESText does. The toString method localizes the message when it is called rather than when it is constructed.

Usage example

Below is an example of how the StartESCore message can be coded. The ESStrings.java class contains the following:

```
public class ESStrings extends ListResourceBundle
{
        public Object[][] getContents() {
           return contents;
        }
        static final Object[][] contents = {
           {"es.core.startup.Hello1",
              "Starting ES Core Server with " +
              "Rendezvous of \"%0\"..."},
           {"es.core.startup.Hello2", " started."}
        };
     }
```

The StartESCore.java class contains the following:

```
System.out.print(ESText.getMessage(
        "es.core.startup.Hello1", popURL));
System.out.println(ESText.getMessage(
        "es.core.startup.Hello2"));
```

The ESString class is used as follows:

```
throw new StaleEntryAccessException(
        new ESString("es.core.startup.Hello2"));
```

Additional design considerations

ID specifications

To simplify the ID generation and reduce the chances of duplications, the IDs should follow the following convention:

- Use the dot format to specify the hierarchy. For example, message IDs for the StartESCore class should be "es.core.startup.StartESCore.*".

- The last node should be a short description string denoting the message. For example, the startup message issued today by StartESCore would have the ID of “es.core.starStartESCore.Hello”.
- Messages are defined in the ESStrings class in sorted ID order.

Core generated exceptions

Because it is possible that the core is running in a different locale from the client, any message text produced in the core that is destined for a client should be specified in the client language. To do this, modify the exception classes to pass the additional objects instead of the text. The exception.getMessage code on the client side uses the ESText class to map the exception number to a message ID and performs the substitutions in the client’s language.

Client usage

Client applications can use these classes as well. They need only call ESText.initialize() with the base class name for their ESStrings equivalent.

ESString Wire Format

```
class ESString{
String messageID;
Object[] info;
}
```

The message ID specifies the text of the message in either the service-defined message catalog or the e-speak defined catalog. The message ID is used to retrieve a message template from the catalog (ESStrings). The optional Objects are substituted into the message based upon the syntax of the message template.

Message templates can contain the "%" (percent sign) symbol followed by a number. The number the index info object. The percent sign (and the number following it) are substituted with the toString value of the associated object.

An example entry from the current e-speak message catalog (net.espeak.util.ESStrings)

```
MessageID: "net.espeak.exception.4"
Message template: "Parameter '%0' invalid type, expected '%1'"
```

Chapter 13 Future Developments

The next release of e-speak will integrate the current e-speak Core with the Web Access architecture (see e-speak Web Access Architecture). It will also implement the localization architecture described in Chapter 12, “Localization”.

Chapter 14 Glossary

This chapter needs check to make sure we are not using terms no longer needed. Terms to do with security (keys and locks) need to be removed. New terms need to be added: certificate, key, PKI, ACI, Principal, URL, ESPDU..... probably others.

Term	**Meaning**
Advertising Service	A service for looking up resources not registered in the local Repository. It returns zero or more Connection Objects.
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.
Certificate	A data structure assigning a Tag or name to a Subject. Certificates are signed using cryptographic techniques so they cannot be tampered with.

Term	Meaning
Certificate Issuer (CI))	A service issuing certificates to Subjects.
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 Manager	A Logical Machine's component that does the initial connection with another Logical Machine.
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.
Distributor Service	A service that forwards published Events to subscribers.
Event	A message that results in the recipient invoking a registered callback.

Term	Meaning
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.
Explicit Binding	An accessor that contains a Repository Handle.
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.
Issuer	An entity issuing a certificate. The Issuer is denoted in a certificate by its Public Key
Logical Machine	A Core and its Repository.
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.

Term	Meaning
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.
Principal	The entity holding the Private Key corresponding to a given Public Key
Private Key	This is secret data. An entity demonstrates knowledge of this secrete data by cryptographic technicques to authenticate itself. Private Keys must be kept secret
Private Security Environment (PSE)	A cryptographically secure store for Private Keys.

Term	Meaning
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.
Public Key	Non-secret data that is associated with a given Private Key by cryptographic techniques
Public Key Infrastructure (PKI)	A set of services and protocols that support the use of public and private key pairs by applications for security.
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.
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.

Term	Meaning
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.
Session Layer Security Protocol (SLS)	The low level message protocol used by all e-speak Cores and Clients for remote communication.
Service Identity (ServiceID)	A field in the metadata that identifies a service or Resource
Simple Public Key Infrastructure (SPKI)	A specific variant of PKI developed within the Internet Engineering Task Force and used by e-speak.
State	Data a Resource needs to implement its abstraction.

Term	Meaning
Subject	The entity to which the access right or name has been issued. In a certificate the Subject is denoted by its Public Key.
Vocabulary	A Resource that contains the set of attributes and value types for describing Resources.
Tag	The field in a certificate expressing an access right
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.