e-speak

Programmer's Guide

Developer Release 3.01
June 2000

HEWLETT® PACKARD

Contents

Chapter 1 The Basics

About This Document

This document describes the *E-speak Service Interface* for Java programmers (J-ESI), pronounced J-Easy, that is used by Clients to program on the e-speak infrastructure. The document explains the programming model for Clients of the e-speak infrastructure. This document contains the following chapters:

- Chapter 1 provides an overview of the J-ESI, the programming model, and the different Services provided in e-speak.
- Chapter 2 explains how to connect with e-speak, and how the Client and the Service provider create and use Services.
- Chapter 3 explains folders, communities, categories, security, views, and events.
- Appendix A explains thread-safe programming concepts.
- Appendix B describes the messaging Application Programming Interfaces (APIs).
- Appendix C describes the command line invocation of the IDL compiler.
- Appendix D describes the interceptor mechanism in J-ESI.
- Appendix E describes the account manager in J-ESI.
- Appendix F describes the security bootstrap mechanism.
- Appendix H describes how services can be deployed and accessed across firewalls.
- Appendix H describes how to create managed e-speak services and monitor them.

Introduction

Traditionally, distributed applications have been viewed as applications that are run over multiple computers in a local domain. As a result, distributed environments have been geared toward tightly coupled applications. However, these environments do not meet the requirements of application deployment and interaction in the Internet domain.

Applications in the Internet domain are better characterized as Services. A Service is a piece of software that is not tightly coupled with Client applications. Services are dynamically discoverable and composable entities. E-speak allows building such loosely coupled, distributed services by supporting the notion of a *Service interface*, a *Service description* that enables Services to be discovered, and by *mediating access* to Services.

The e-speak infrastructure provides clear segregation of the roles played by different entities in enabling a robust Service ecosystem. The entities involved in defining Service ecosystems are categorized as follows:

- Service standards bodies: Standardization bodies define and publish standardized interfaces for different Service categories that are used by Service developers to write Services conforming to the standardized interface.

  For example, a standards body may define a generic printer Service interface to contain the following invocation points: print(), and status(). Any Service provider interested in providing a printer Service writes their Service to support this interface.

  In addition, the standardization body is responsible for identifying the Vocabulary used to describe a Service. The Vocabulary describes the attributes that are used to uniquely describe a Service. For example, a printer Vocabulary includes attributes such as Manufacturer, Modelname, DPI and so on (perhaps using XML) that is used by service deployers to advertise this Service, and queried by Clients to discover it.

- Service developers: These are the developers of the actual Service implementation. Service developers design their Service to comply with the programmatic Service interfaces published by the standards bodies. Service developers are also able to wrap existing legacy applications so that they can be made available as services in the e-speak infrastructure.
- Service deployers: Service deployers are responsible for deploying the Services created by the Service developers. They are responsible for advertising the Services in the appropriate Vocabulary and also for handling requests to the Service.
- Service administrators: These administrators are responsible for monitoring, administering, and controlling the Service deployment.
- Service Directories: These directories are places where service providers and clients can advertise and find other services that are of interest to them.

This segregation of roles allows the Service developers to concentrate on the business logic of the Service that they are developing, the Service deployers to concentrate on advertising and handling requests to the Service, and the administrators to administer the Service. Furthermore, the existence of service directories greatly simplifies the process of building new composite services by providing a place where services can find other services. An example of such a service directory is accessible from the e-speak.hp.com site that each e-speak installation points to by default.

J-ESI acts as the Java system call interface to the e-speak *Core* for all the above entities. The Core is the active entity of e-speak that acts as the mediation layer and routes messages to Services.

Figure 1 shows a typical interaction between Clients and Service providers in an e-speak community. An e-speak community consists of a number of Cores connected to each other. Clients and Service providers join the community through these Cores.

Clients search for Services registered in the community, and when they are successful, they gain access to the Service using a Service proxy (stub) of the discovered Service.

The Clients need to know only the interface of the Service in order to invoke object-oriented functions (methods) on it.

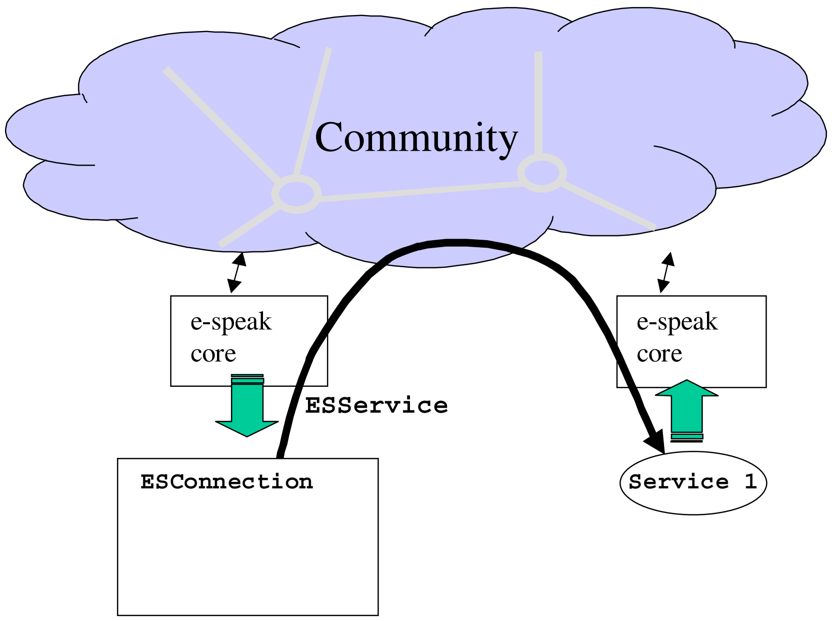


Figure 1 Typical interaction between Clients and Service providers

Unlike other distributed computing infrastructures, where the infrastructure is used only to locate the remote stub, in e-speak all subsequent accesses are mediated by the infrastructure. This mediation allows the infrastructure to enable very flexible security mechanisms that can be used to implement a wide variety of security policies. It also permits better management of the Services in the community for accounting, auditing, and billing purposes. The e-speak architectural specification has a description of the security features in e-speak. However, this release of J-ESI exposes only few security-related APIs.

Programming Model

The primary goal of e-speakk is to simplify the development, deployment, and management of communities of Services on the Internet. To accomplish this, e-speak provides three basic abstractions:

- Services
- Service Contracts
- Service Vocabularies

In addition to these abstractions, e-speak provides system Services that enable Clients to write e-speak Services using J-ESI.

E-speak Services

Essentially, Services are programs written in a programming language such as Java, C, Perl, and C++. Although J-ESI makes use of some Java features, the e-speak infrastructure itself is not limited to any particular language. For example, a Service registered using J-ESI can be accessed from a Perl/Python script by using the appropriate e-speak library. Figure 2 illustrates the relationship between Services, their Contracts, and their Vocabularies.

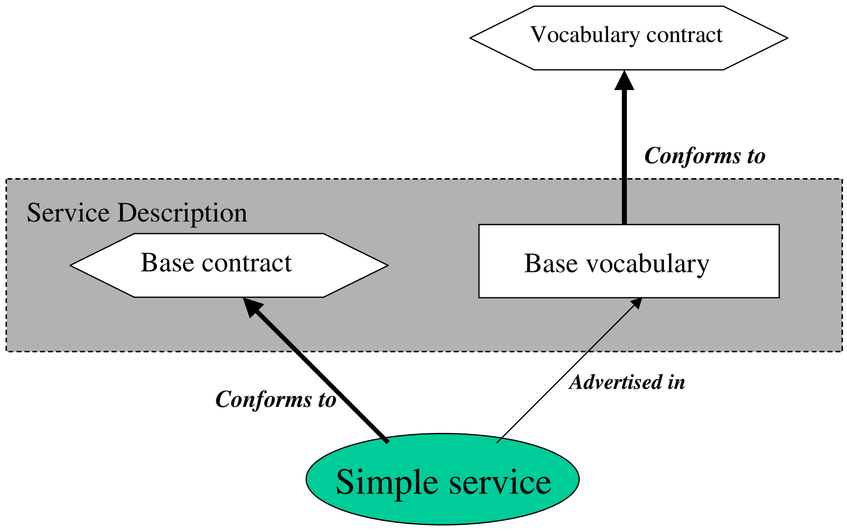


Figure 2 Relationships among Services, Contracts, and Vocabularies

Any e-speak Service conforms to its Contract because by design, it implements the interfaces that are outlined in its Contract. In addition, the Service is advertised in an appropriate Vocabulary.

Service Vocabularies essentially provide the scheme for constructing the description of the Service. All three abstractions, Services, Contracts, and Vocabularies, are first-class entities in e-speak. By decoupling Services from the Contract and the Vocabulary, e-speak allows Service implementors (who develop the actual Service implementations conforming to certain Service Contracts) to be independent of Service deployers (who simply advertise the Service in a well-known Vocabulary).

Clients are not dependent on the Service developer and deployer to communicate the Service Contracts and Vocabularies with which these Services have been described.

The Service deployers register their Services with a group of e-speak Cores. On registering a Service, this Service is discoverable by others in the same e-speak group of Cores.

Clients use the default e-speak finders that are provided with J-ESI, or write their own finders to find Services that meet their requirements. The access permissions of the Service determine the Services that will be visible to the Clients. On finding Services, Clients receive a remote stub to the Service.

Service Contracts

Each e-speak Service implements a set of interfaces defined in a Service Contract. These interfaces are often defined using the e-speak IDL, which is similar to the Java-RMI IDL. (See Appendix B, "Messaging Classes", for additional information.) Service Contracts are first-class entities that can be discovered and used like any other Service. All Service Contracts support a base set of interfaces that provide mechanisms to create and query them. As a result, Contracts also can be advertised in a well-known Vocabulary—typically by a standardization body.

Service Vocabularies

A Vocabulary consists of a set of associated attributes and properties. The only properties associated with any attribute are the name of the attribute and the type of values to which these attributes are assigned. The structure of a Vocabulary is dependent on the vertical market for which it is defined.

For example, the Vocabulary used to define a printer with attributes such as manufacturer, model name, DPI, speed, color, and cost is quite different from the Vocabulary that defines apparel merchandise that has attributes such as size, color, material, and cost.

Though some of the attribute names may be the same across Vocabularies, they may have different semantics. The attribute color in the Printer Vocabulary may be a boolean expression that indicates whether the printer supports color printing; however, the attribute color in the Apparel Vocabulary may indicate the actual color of the piece of clothing.

Because Vocabularies are first-class entities, Clients and Services alike can find these Vocabularies registered in the community and subsequently make use of them. New Vocabularies are typically advertised using the Base Vocabulary that is available in the e-speak infrastructure.

This Base Vocabulary defines a set of attributes that are generic and can be used by generic Services in markets that do not have well-defined Vocabularies. All Services registered as a Vocabulary conform to the Vocabulary Contract that defines the operations that can legally be performed on Vocabularies.

E-speak System Services

In addition to the base abstractions of Vocabulary, Contracts, and Services, J-ESI supports a set of base and extended services that makes the job of deploying and using Internet-wide Services simple.

Basic Services

These are the basic set of Services that are required to get connected to the e-speak infrastructure and to create and find new user-defined Services. These Services include the following:

- **Connection—**A connection Service is used to connect and disconnect from the e-speak infrastructure.
- **Vocabulary—**A Vocabulary Service allows the creation of new Vocabularies and queries the properties of existing Vocabularies.
- **Contract—**The Contract Service is used to create and use Contracts.
- **Elements and Finders—**Service elements are used to register Services with the Core, while Service finders are used to find Services.

Extended Services

- **Events—**This Service defines a distributed Event Publisher-Subscriber model for Events. Events are distributed by a Distributor across any number of connected Cores.

- **Community—**This Service enables discovery of Services across multiple e-speak cores. Communities are Client-side abstractions of collections of groups that form the search domain. The Community is a collection of member core groups identified by group names.
- **Folders—**The Folder Service allows users to manage their Services (discovered or created) similar to how they manage local files in a standard operating system. Persistent folders appear on reconnecting and act very much like organized hierarchically persistent bookmarks.

Client-Service Interaction

As described earlier, Clients find a Service using attributes described in the Service Vocabulary. The return value of the `find()` call is a Service stub that extends `ESService` and implements the operational interface specified in the find.

When Clients discover a Service, they have to specify the interface they want to use. The stub is the Client-side abstraction of a Service that implements this interface. The Service provider's abstraction of the Service is a Service element as represented by the class `ESServiceElement`.

The Service element has, among other things, the description of the Service, the description accessor of the Service to mutate the description, and the actual implementation of the Service. It also has information about the handler of the Service, including the queue on which messages to this Service will be sent and the number of threads servicing requests to this Service.

For example, if a Service provider wants to make an existing, stand-alone application (such as `PrinterServiceImpl`), an e-speak Service, performs the following actions:

1. Define or modify the interface that describes the Service interface to conform to e-speak IDL.
2. Create a new `ESServiceElement`.
3. The Service deployer provides the description of the Service in a Vocabulary, along with the implementation of the Service such as the object `PrinterServiceImpl` to the Service element. The element also contains the description

accessor of the Service that can be used by the deployer to query or modify the description of the Service.

4 After performing these actions, the deployer can use the Service element to start the Service. The deployer can also control the concurrence of the Service by changing the number of threads that process the requests for this Service. The message queue and threads are controlled by a Service handler.

Clients, on the other hand, create instances of a stub to the Service when they do a find by using the e-speak finder service. They typically refer only to the interface that the stub implements and invoke methods in that interface. Clients could also choose to use the messaging APIs to communicate with Services.

A Simple Example

The following section provides a simple example of an e-speak Service written using J-ESI that illustrates some of the basic ideas in the e-speak infrastructure.

Client Service Discovery

A Client first creates a new connection to an e-speak Core. After connecting to the Core, the Client can look up or register Services. The Client locates a Service that satisfies a constraint expressed with attributes in the default Vocabulary. The result of the find Service is a stub (or proxy) to the Service provider's Service. Clients can use this stub as a network object reference and directly invoke methods on the Service (see Figure 3).

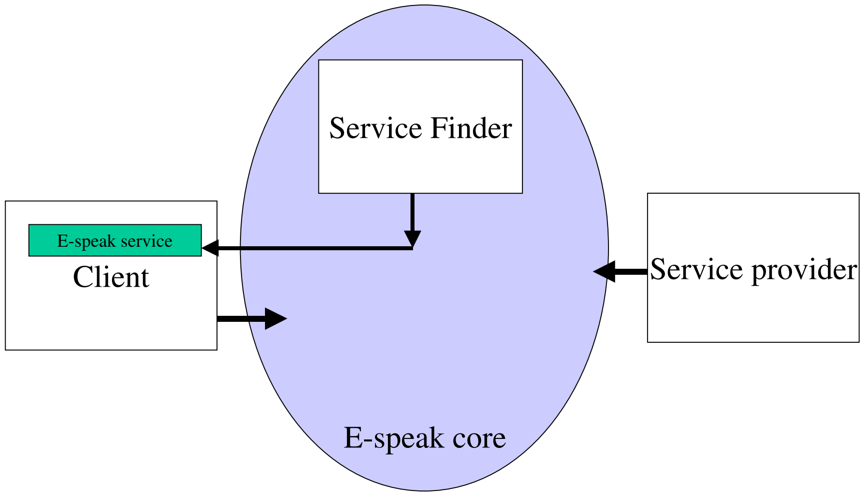


Figure 3 Client and provider Core relationship

Client Service Usage

Clients interact with the Service with the set of interfaces for which stubs are available in the Client address space. Clients can preinstall the Service stubs that are generated using the e-speak IDL stub generator, or they may acquire the stub class from the Service provider by other means. From the Client's perspective, the nature of the Service provider is insignificant beyond the requirement that it implement the interface and that its attributes satisfy the query that the Client made.

When a Client invokes an operation, a well-defined e-speak custom serialization is used to ship the invocation to the target Service through the mediating e-speak infrastructure. In so doing, all method invocations are effectively mediated. This remote invocation will work across languages and platforms, because the basic architecture does not depend on Java. The following code fragment illustrates a very simple find-and-use scenario. The Client finds a Service whose name is `printer` and invokes the `print` method on it.

```
ESConnection coreConnection = new ESConnection();
String intfName = PrinterServiceIntf.class.getName();
ESServiceFinder printFinder =
   new ESServiceFinder(coreConnection, intfName);
ESQuery printQuery = new ESQuery("Name == 'printer'");
PrinterServiceIntf printer =
   (PrinterServiceIntf) printFinder.find(printQuery);
printer.print(document);
```

Figure 4 shows that the e-speak Core is located between the Client and the Service provider. In general, many Cores may lie between the Client's Core and the Core where the Service provider is registered.

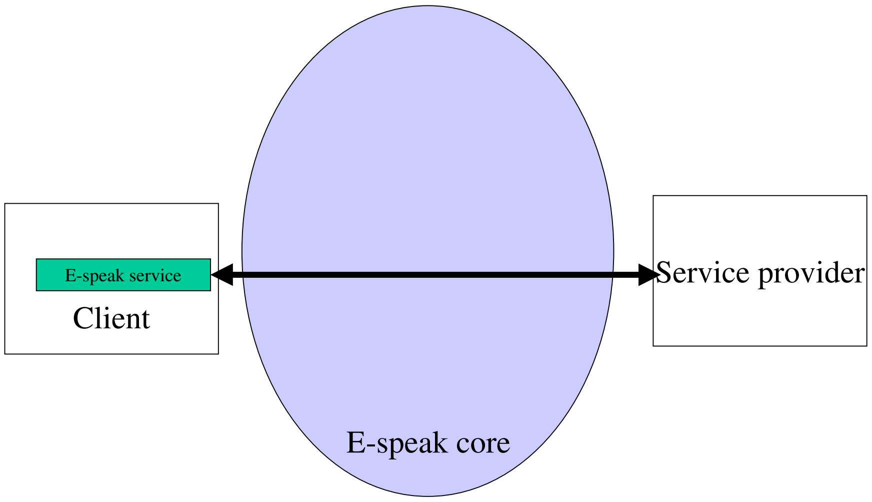


Figure 4 The e-speak Core

Service Creation

A Service provider is primarily interested in implementing a Service interface to comply with the Contract and in advertising the Service in an appropriate Vocabulary. The Service provider first connects to the e-speak Core. The provider then creates a Service element with a description of any Service Vocabularies and Contracts.

The `register()` method of ESServiceElement is used to register this Service with the e-speak Core. The Service element object also supports certain management APIs that allow the Service provider to start, stop, and query the status of the Services encapsulated in this Service element:

```
ESConnection coreConnection = new ESConnection();
ESServiceDescription printDescription = new ESServiceDescription();
printDescription.addAttribute("Name","printer");
ESServiceElement printElement =
   new ESServiceElement(coreConnection,printDescription);
PrinterServiceImpl printerImpl = new PrinterServiceImpl();
printElement.setImplementation(printerImpl);
ESAccessor printAccessor = printElement.register();
printElement.advertise();
printElement.start();
```

This code fragment creates a simple Service called `printer` that is implemented by a class called `PrinterServiceImpl`. The only attribute used in this example is the name of the Service that is set using the `addAttribute` method of `ESServiceDescription`. Clients can create more complex descriptions using custom Vocabularies, which are described in later chapters.

Once the `start` method in the Service element has been called, the printer Service is available for other Clients in the community to access. The actual implementation of the Service is independent of the e-speak infrastructure. In fact, the implementation can very well be legacy code potentially written in any language.

The advertise call in the code causes the description of the service to be advertised in the advertising directories that have been set up. By default, J-ESI advertises the description of the service to the e-services village service directory. Note that the advertising service has to be started separately in order for the advertise call to function appropriately.

Complete Example

This section puts together the basic ideas of e-speak into a simple end-to-end example that illustrates the sequence of steps that have to be taken on the Service provider and Client side to provide and access Services.

Typically, users have access to the IDL that defines the interface that the Service implements. In this example, the Clients and Service providers are assumed to have installed the stubs by virtue of having access to the IDL file as well as the IDL compiler. Although dynamic loading of the stubs is not directly supported, it can be implemented in user applications using standard Java class loaders, and will also be supported in the next J-ESI release that incorporates the e-speak security features.

The e-speak IDL is similar to the Java-RMI IDL. Therefore, the contents of the IDL files look similar to Java interface files. These IDL files must have a .esidl extension for the IDL compiler to recognize them as e-speak IDL files. The IDL specifications are found in Appendix B, “Messaging Classes”.

PrinterServiceIntf.esidl

The following is a sample esidl file that defines the interface to a printer Service that has two methods: `status` and `print`.

```
public interface PrinterServiceIntf {
   public String status()
   public void print(String text)
}
```

The IDL compiler generates the following files, all of which are used by J-ESI:

```
PrinterServiceIntf.java,
PrinterServiceStub.java, and
PrinterServiceMessageRegistry.java
```

Except for some minor changes, such as marking the generated interface as an e-speak interface, the Service `PrinterServiceIntf.java` is a copy of `PrinterService.esidl.`

`PrinterServiceStub.java` is the stub class that the finder returns to the Client when it finds a Printer Service.

For every method defined in the interface, the stub class contains code to create a message, marshal parameters, and send it to the Service provider. `PrinterServiceMessageRegistry.java` does not need to be used by the Client directly, but is used by J-ESI.

PrinterServiceImpl.Java

When the IDL file is passed through the IDL compiler for Java, it produces an equivalent Java interface file. The Service implementor implements this Java interface. Therefore, the Service implementor's class (using `PrinterServiceImpl` as an example) looks as follows:

```
public class PrinterServiceImpl implements PrinterServiceIntf
{
   public String status()
      throws ESInvocationException
   {
      // Implementation to return the printer status
   }

   public void print (String text)
      throws ESInvocationException
   {
      // Implementation to print the document sent by user
   }
}
```

The Service deployer advertises the Service and handles requests to the Service. The following code fragment is written by the Service deployer:

```
public class PrintServer
{
public static void main(String [] args)
{
    try
    {
        ESConnection coreConnection = new ESConnection(“file.pr”);
        ESServiceDescription printDescription =
            new ESServiceDescription();
        printDescription.addAttribute(“Name”,”printer”);
        ESServiceElement printElement =
            new ESServiceElement(coreConnection, printDescription);
        printElement.setImplementation(new PrinterServiceImpl());
        printElement.register();
        printElement.start();
        System.out.println("Started printer Service ");
    }
    catch (Exception e)
    {
        // handle the exception
    }
}
}
```

The Client also runs the IDL compiler to generate the interface and stub files, and makes use of the interface in the program. For instance, a Client discovers and uses the printer as follows:

```
public class PrintClient
{
    public static void main(String [] args)
    {
        try
        {
            ESConnection coreConnection = new ESConnection(“file.pr”);
            String intfName = PrinterServiceIntf.class.getName();
            ESServiceFinder printFinder =
                new ESServiceFinder(coreConnection, intfName);
            ESQuery printQuery =
                new ESQuery(“Name == ‘printer’”);
            PrinterServiceIntf printer = (PrinterServiceIntf)
               printFinder.find(printQuery);
            String document = getDocument();
            printer.print(document);
            System.out.println(printer.status());
         }
         catch (Exception e)
         {
            // handle the exception
         }
      }
   }
```

This class is all that is required of a client application developer to find and use an example printer service.

Chapter 2 Basic Services

This chapter explains the basic Services available within e-speak. The chapter contains the following sections:

- Getting Connected to E-speak
- E-speak Services
- Client: Finding Services
- Service Deployer: Creating Services
- Service Description
- Accessing Descriptions: ESAccessor

Getting Connected to E-speak

The simple example in Chapter 1 shows that pure Clients (e-speak Clients who use found Services rather than creating their own Services) and Service providers who provide Services have to connect to the e-speak infrastructure to access or provide Services. Because both pure Clients and Service providers are Clients of the e-speak infrastructure, they are often referred to collectively as e-speak Clients.

ESConnection

In J-ESI, the connection between Clients and the e-speak infrastructure is represented by an `ESConnection`. Clients create a new instance of `ESConnection` when they want to connect to the e-speak infrastructure. Clients insert all of the relevant information needed for connecting to the e-speak infrastructure in a configuration file that is passed as an argument to the constructor of the `ESConnection`.

The default constructor for the `ESConnection` connects to an e-speak Core that is running on the local machine and is listening to requests on a port defined in the `ESConstants` file. Clients can also insert the name of a properties file that has the values of the properties that are relevant to establishing the connection. The most common fields in this property file are listed in the next subsection.

When a connection is established between a Client and the e-speak Core, the capabilities granted to operations performed on this connection depend on the credentials that are presented. For example, if a user connects to a Core as a guest, the user receives restricted access and privileges compared to a user logged in as an administrator while operating from that connection.

`ESConnection` supports APIs that return the Base Vocabulary and Contract Services. In addition, there are other APIs for getting management-related information that identifies users and the Core. The `ESConnection` also encapsulates the context of the actions that are performed by the Client.

Property Files

The property file parameter passed to the `ESConnection` constructor provides values for the properties that are required to connect to the e-speak infrastructure. For example, the property file has entries for the following properties:

- `username`

  This property specifies the name of the user who is connecting to the e-speak Core. If the property file does not provide a value for this, the default value that is used is `guest`.

- `password`

  This is the password of the user. The password along with the username form the user credential.

- `esurl`

This is a colon-separated string that specifies the communication channel between the Client and the Core. For example, `tcp:rgelpc032.rgv.hp.com:12346` specifies that the Client connects to the Core running on host `rgelpc032.rgv.hp.com` on port `12346` using `tcp` as the communication protocol.

- `protocol`
  `hostname`
  `portnumber`

  When the `esurl` is not specified, the Client can alternatively specify the elements of the connection information by explicitly specifying the protocol, host name, and port number. If the configuration file does not have the protocol, the default value for the protocol is `tcp`. Furthermore, if the property file does not have a value for the host name, the default value for the host name is `local-host`, and the default port number is `12346`.

- `accountname`

  Clients can provide names for their accounts so that they can reconnect to the same account when they connect the next time.

- `homefolder`

  Applications can specify their home folder. This folder is the current working folder when the connection is made to the e-speak Core.

- `Timeouts`

  Applications may be able to specify timeouts for various calls. Synchronous call timeouts can be specified with `synchronous_call_timeout`. Asynchronous call timeouts can be specified using `asynchronous_send_timeout`. Service providers can specify timeouts for messages they receive using `asynchronous_receive_timeout`. Clients can time out their find calls using `finder_timeout.` Timeouts are specified in milliseconds. Timeouts can also be changed dynamically. This can be done by calling the following methods on `ESConnection.`

```
- setCallTimeout(int value);
   - setAsyncSendTimeout(int value);
```

```
      - setAsyncRecvTimout(int value);
      - setFinderTimeout(int value);
```

Connection Configuration

Typically, Clients create an instance of an `ESConnection` as follows:

```
ESConnection coreConnection = new ESConnection();
```

In this case, a transient guest account is created and as soon as the connection is closed, this instance of the guest account is removed. Alternatively, the client can connect to the Core using

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
```

When the property file is used for creating a connection and the account name property in the property file is used, the following account creation rules are followed:

- Where the account name property is not specified, a transient guest account is created. However, if the account name is provided and no previous account of this name exists, a new account with the user and password is created.
- Where the account already exists and the user credentials are validated, the connection returns a connection with the last saved state associated with the same user credential.

After having connected to the Core, the Client can retrieve the parameters of the connection by entering

```
ESConfiguration config = coreConnection.getConfiguration();
```

The following methods in the `ESConfiguration` class allow the Client to retrieve the various parameters of the connection:

```
public class ESConfiguration
{
   public ESUserCredential getUserCredential();
   public String getHostName();
   public int getPortNumber();
   public String getProtocol();
public String getURL();
public void setHostName(String hostName);
public void setPortNumber(int portNumber);
public void setProtocol(String protocol);
public String getConnectionName();
public void setConnectionName(String name);
public String getGroupName();
public void setURL(String url);
public int getCallTimeout();
public int setCallTimeout( int newVal );
// .. refer to javadoc for other methods
}
```

E-speak Services

There are two abstractions for a Service:

- **Service providers—**Service elements (ESServiceElement) that are used by Service providers to deploy and manage a Service.
- **Clients—**Service stubs (ESService) that are used by Clients to access an e-speak Service.

Service providers take the following steps to create a Service:

1. The Service developer provides implementation of the Service interface. These interfaces are encapsulated in the Contract to which the Service conforms. The actual implementation need not contain any e-speak-specific code.
2. The Service deployer then describes the Service in a chosen Vocabulary by setting the attribute values appropriately. The `ESServiceDescription` class is used to set these attribute values.
3. The Service deployer and/or administrator then creates a new `ESService-Element` that registers and starts the Service so that other Clients of the e-speak infrastructure can find and use the Service. (See Figure 5.)

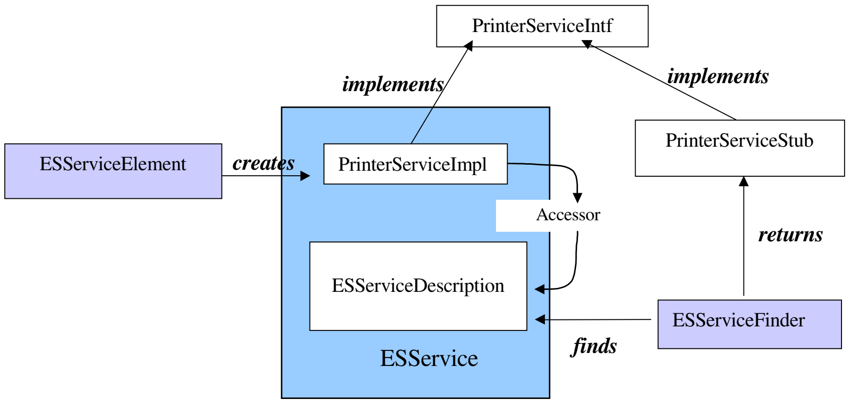


Figure 5 Creating an `ESServiceElement`

Clients typically take the following steps to find and use a Service:

1. Clients find Services using a Service finder that accepts queries in the Vocabulary that was used to advertise the Service. Clients also specify the interface that they expect to use to invoke the Service.

2 On finding a match, the Clients get a handle to a Service stub that can be used to invoke the methods that are implemented by the Service provider (see Figure 5).

The Client or Service deployer can choose to examine or modify attribute values. This is accomplished using the `ESAccessor` class.

All the three basic entities (Vocabularies, Contracts, and other Services) in e-speak can be created or found in the same manner.

The main difference between generic Services and Contracts or Vocabularies is that the interface for Vocabularies and Contracts is defined by the e-speak Core. The Core is the handler for any Vocabulary or Contract method.

Generic Services on the other hand, are not handled by the Core, and can have arbitrary interfaces that meet the requirements of the specific Service. The rest of this section describes the well-defined interfaces supported by Contracts and Vocabularies in the e-speak Core.

ESContract

Just as a Vocabulary determines the scheme for describing the attributes of Services, the Contract determines the type of Service as represented by the interfaces that it implements. Just as the Core is the handler for Vocabularies, the Core is the handler for Contracts as well.

The following method returns the name of the Contract; the name of the interface that this Contract encapsulates:

```
public String getInterfaceName();
```

The following method returns the IDL string associated with the Contract:

```
public byte [] getInterfaceDefinition();
```

In addition, the ESContract has methods that return the conversations that the service supports, the terms and conditions of use, and the license policy. The following entry points in ESContract are the relevant entry points.

```
public String getConversationScheme();
public String getTermsOfUse();
public String getLicense();
```

Base Contract

A Base Contract is available in the Core that Clients can obtain by invoking the `getBaseContract` method in the connection object. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESContract baseContract = coreConnection.getBaseContract();
```

ESVocabulary

In the e-speak infrastructure, Vocabularies are also Services that are discovered by Clients. When the Client discovers a Vocabulary, the Client receives a stub that conforms to the `ESVocabulary` interface. However, the Service provider for the Vocabulary is the e-speak Core itself.

After a Vocabulary has been defined, Clients of the Vocabulary can only query its contents to determine what properties this Vocabulary defines. The Clients cannot change the properties of the Vocabulary. Furthermore, they can query the core for all the services that are registered in this vocabulary. There are two main APIs in ESVocabulary:

```
public ESProperty[] getProperties();
public ESAccessor[] getServices();
```

The conversation scheme, terms of use, and licensing policy are not available for the base contract.

Base Vocabulary

The Base Vocabulary comes preloaded with e-speak. This Base Vocabulary has pre-defined properties that can be used to describe Services. Service providers do not have to invent their own Vocabulary to describe their simple Services. The Base Vocabulary has basic attributes that can represent the name of the Service, the type of Service, and other similar characteristics that are universal to many Services.

The base Vocabulary can be retrieved from an `ESConnection` using the `getBaseVocabulary()` method. The following code fragment shows how Clients get the Base Vocabulary:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESVocabulary baseVocab = coreConnection.getBaseVocabulary();
```

The following table shows some of the important properties that are defined in the Base Vocabulary. No other vocabulary should define and use 'Name', 'Type' and 'ESGroup' as these attributes are reserved and are used by the J-ESI library internally.

Table 1 Base Vocabulary properties

Attribute name	Type	Description
`Name`	String	Name of the Service
`Type`	String	Type of Service
`ResourceSubType`	String	Subtype of Service
`Description`	String	Description of the Service
`Version`	String	Version of the Service
`ESDate`	Date	Date associated with the Service
`ESGroup`	String	Group associated with the Service
`ESTimestamp`	Timestamp	Timestamp associated with the Service
`ESCategory`	String	Stringified list of categories that any service is advertised in.

Client: Finding Services

Clients to the e-speak Core typically find Services that match certain constraints. The kinds of finders are visible to the Client are:

- Vocabulary finders
- Contract finders
- Service finders
- Folder finders
- View finders

You can extend these finders to suit your needs or to use the APIs supported in these classes directly.

ESContractFinder

The `ESContractFinder` class is used by Clients to find Contracts. Contracts are Services that are registered with the e-speak Core just like any other Service. The Contract Finder finds only Services that have been registered as Contracts. The attribute that is used to distinguish between Contracts is the `Name` attribute in the Base Vocabulary. There are three important entry points in ESContractFinder. These are:

```
public ESContract find(ESQuery query);
public ESContract[] findAll(ESQuery query);
public ESContract [] findNext();
public void setMaxToFind(int number);
```

The find method finds a single contract that matches the constraints and preferences in the query. The findAll finds all contracts that match the query subject to the maximum number of services set in the finder.

For example, to find a Contract named `PrintContract`, use the following code fragment:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESContractFinder printContractFinder =
   new ESContractFinder(coreConnection);
ESQuery printContractQuery =
   new ESQuery("Name == 'PrintContract'");
ESContract printContract =
   printContractFinder.find(printContractQuery);
```

ESVocabularyFinder

The `ESVocabularyFinder` class is used by Clients to find Vocabularies. Finding a vocabulary is done more often than creating a vocabulary. Vocabularies are usually defined by some standard body. For example, a standard body might define a printer vocabulary which contains attributes like the speed of the printer, its location, etc. The definition of the vocabulary happens only once. It is likely that this vocabulary is registered in a community of interest. Any printer which wants to conform to this standard body finds the vocabulary and registers the printer service using this vocabulary.

Because a vocabulary is also a service, it needs to be advertised as well. Usually vocabularies are advertised in the default vocabulary. It is also possible to advertise a vocabulary in another non default vocabulary.

```
public ESVocabulary find(ESQuery query);
public ESVocabulary[] findAll(ESQuery query);
public ESVocabulary[] findNext();
public void setMaxToFind(int number);
```

For example, to find a Vocabulary named `printerVocab`, use the following code fragment:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESVocabularyFinder printVocabFinder =
   new ESVocabularyFinder(coreConnection);
```

```
ESQuery printVocabQuery =
   new ESQuery("Name == 'printerVocab'");
ESVocabulary printVocab = printVocabFinder.find(printVocabQuery);
```

Instead of using ESQuery, you can find a vocabulary using an XML query using ESXMLQuery.

ESServiceFinder

The `ESServiceFinder` class is a generic finder class that is used to find user-defined Services. The interfaces here look similar to the ESContractFinder and ESVocabulary finder.

```
public ESService find(ESQuery query);
public ESService[] findAll(ESQuery query);
public ESService [] findNext();
public void setMaxTofind(int number);
```

The `ESServiceFinder` class has two constructors. One constructor which takes only `ESConnection` as parameter. `find/findAll` in this case return the base stub (`ESService`) from which all other stubs derive. The second constructor takes `ESConnection` and the interface name (or `ESContract`) as parameters. `find/findAll` in this scenario returns the stub for the Service provider. When the interface name is specified for doing a find, the client library sends an introspection request to the potential service providers. The service providers that respond correctly to the introspection request are returned as results. This means that when the interface name is used in the find, the services that the finder returns were known to be up and ready at the time of the find. No such guarantee can be made when the finder is used without the interface name. If the Service provider is not running, the call timesout if the finder timeout is set. More details on how to set the timeouts is available in the later sections.

The `ESServiceFinder` class can return multiple Services as the result of a query. However, Clients can control the maximum number of Services that are returned as the result of a find by entering the following code:

```
public void setMaxToFind(int number);
public int getMaxToFind();
```

Chapter 1 presented an example of a Client that searches for a simple printer that is described in the base Vocabulary. A slightly more complex example of a Printer that has been advertised in a non-Base Vocabulary, called a `printerVocab`, requires the following code to find the printer:

```
public class PrintClient{
   public static void main(String[] argv){
      try  {
         String propertyFileName =
            new String("/users/connection.prop");
         ESConnection coreConnection =
            new ESConnection(propertyFileName);
         // find vocabulary
         ESVocabularyFinder printVocabFinder =
            new ESVocabularyFinder(coreConnection);
         ESQuery printVocabQuery =
            new ESQuery("Name == 'PrintVocab'");
         ESVocabulary printVocab =
            printVocabFinder.find(printVocabQuery);
         // find contract
         ESContractFinder printContractFinder =
            new ESContractFinder(coreConnection);
         ESQuery printContractQuery =
            new ESQuery("Name == 'PrintContract'");
         ESContract printContract =
            printContractFinder.find(printContractQuery);
         // find service described in vocab that conforms to contract
         ESServiceFinder printFinder =
            new ESServiceFinder(coreConnection, printContract);
         ESQuery printQuery = new ESQuery(printVocab);
         printQuery.addConstraint("DPI == 1400");
         PrinterServiceIntf printer = (PrinterServiceIntf)
            printFinder.find(printQuery);
         String document = getDocument();
         printer.print(document);
      } catch (Exception e)
      {
        // handle the exception.
      }
   }
}
```

The finder class also supports a cursor like mechanism for clients to traverse the result set obtained as a result of performing a `findAll()` operation. The cursor mechanism is quite common in relational database APIs such as JDBC. In JDBC, the result of executing a query is a result set, and the client program can loop based on a condition that depends on whether there are additional rows in the result set. In J-ESI, on the other hand, the finder itself acts as the result set. On invoking a `findAll()` operation on the finder, the finder acts as a cursor. The client program can determine whether there are other services found by the finder by invoking the `hasMoreResults()` method on the finder. In addition, the next set of results can be obtained from the finder by invoking the `findNext()` method. In the example below, the query essentially finds all the services that are registered in the printer vocabulary. It then proceeds to retrieve the services that are found two at a time.

```
public static void main( String [] args ) {
   try {
      ESConnection    conn = new ESConnection();
      ESVocabularyFinder vf = new ESVocabularyFinder(conn);
      ESVocabulary v = null ;
      try {
      v = vf.find( new ESQuery("Name == 'printerVocab'"));
   } catch( LookupFailedException ffe ) {
      ffe.printStackTrace();
      return;
   }
   ESServiceFinder finder = new ESServiceFinder(conn);
   ESQuery query = new ESQuery(v,"(Name == 'printer') or (Name != 'printer')");
   ESService[]  serviceList = null;
   int total_findall_services = 0;
   finder.setMaxToFind(2);
   serviceList = finder.findAll(query);
   total_findall_services += serviceList.length;
   while (finder.hasMoreResults()){
      serviceList = finder.findNext();
      // .. do stuff with services found in service list.
      System.out.println("number of services got: " +
                 serviceList.length);
       }

       } catch( Exception ex ) {
            ex1.printStackTrace();
```

```
        }
return;
    }
```

ESQuery

In general, Clients use the `ESQuery` class to construct queries that are either not simple name lookups in the Base Vocabulary or are constraints that are expressed in a Vocabulary that is different from the default Vocabulary. The ESQuery class has methods that allow the client to add constraints that make up the query. These constraints are string constraints whose syntax is similar to the syntax of constraint specification in OMG's OTS specification. Essentially, every constraint is parsed as if it were a disjunction of conjunctions, i.e., the contraint expression is expected to be in disjunctive normal form.

More formally, the context-free grammar that defines the syntax of the contraint strings is as follows:

```
<Expr> ::=   <OrExpr>
<OrExpr> ::=   <AndExpr  >
       |  <OrExpr> 'or' <AndExpr>
<AndExpr> ::=   <NotExpr >
       | <AndExpr> 'and'  <NotExpr>
<NotExpr> ::=   <EqualityExpr>
       |  'not'  <EqualityExpr>
<EqualityExpr> ::=   <RelationalExpr>
        |  <RelationalExpr> ( <OpEqual> | '!=' )  <RelationalExpr>
<RelationalExpr> ::=   <InExpr>
       |  <InExpr> <OpRelational> <InExpr>
<InExpr> ::=   <AdditiveExpr>
        |  <AdditiveExpr> 'in'  <AdditiveExpr>
                         |  <AdditiveExpr> 'in'  <ListExpr>
<AdditiveExpr> ::=   <MultiplicativeExpr>
       |  <AdditiveExpr> <OpAdditive> <MultiplicativeExpr>
<MultiplicativeExpr> ::=   <UnaryExpr>
                                        |  <MultiplicativeExpr> <OpMultiplicative>
<UnaryExpr>
<UnaryExpr> ::=   <UnionExpr>
       |   '-'  <UnaryExpr >
       |   'exist'  <PathExpr>
<UnionExpr> ::=   <PathExpr>
       |   <UnionExpr> '|'  <PathExpr>
<ListExpr> ::=   '['  <Arguments> ']'
<PathExpr> ::=    '/'
                                            |   '/'  <RelativeLocationPath>
                                            |    '//'  <RelativeLocationPath>
                                            |   <RelativeLocationPath>
                                        |   <FilterExpr>
                                        |   <FilterExpr> '/'  <RelativeLocationPath>
                                        |   <FilterExpr> '//'  <RelativeLocationPath>
<FilterExpr> ::=    <PrimaryExpr>
                                        |   <FilterExpr> <Predicate>
<RelativeLocationPath> ::=   <Step>
                                        |   <RelativeLocationPath> '/'  <Step>
                                        |   <RelativeLocationPath> '//'  <Step>
<PrimaryExpr> ::=   '$'  <NCName>
                                        |   '('  <Expr> ')'
                                        |   <Literal>
                                        |   <FunctionCall>
<FunctionCall> ::=   <FunctionName> '('  <Arguments> ')'
<Arguments> ::=   <Expr>
                                        | <Arguments> ',' <Expr>
<Predicate> ::=    '['  <Expr> ']'
<Step> ::=  <StepPredicate>
                                        |   '.'
                                        |   '..'
<StepPredicate> ::=  <AxisSpecifier> <NodeTest>
                                        |   <StepPredicate> <Predicate>
<AxisSpecifier> ::=  <AxisName> '::'
                                        |   '@'
                                        |   e

<NodeTest> ::=   '*'
                                        |   <NCName> ':'  '*'
                                        |   <QName>
                                        |   NodeType '(' ')'
                                  |   'processing-instruction'  '('  String_Lit  ')'
<NodeType> ::=   'comment'  |  'text'  |  'node'
<AxisName> ::=   'ancestor'   |  'ancestor-or-self'  |  'attribute'
                                   | 'child' | 'descendant' | 'descendant-or-self'
                                  | 'following' | 'following-sibling' | 'namespace'
                                   | 'parent' | 'preceding' | 'preceding-sibling'
                                        | 'self'
<Qname> ::=   <NCName>
                                        |   <NCName> ':'  <NCName>
<Literal> ::=   <Number_Lit>
                                        |   <String_Lit>
                                        |   <Boolean_Lit>
                                        |   <Date_Lit>
                                        |   <Time_Lit>
                                        |   <TimeStamp_Lit>
<OpEqual> ::=  '=' | '=='
<OpRelational> ::= '<' | '<=' | '>' | '>=' | '&lt;' | '&lt;=' | '&gt;' | '&gt;='
<OpAddtive> ::= '+' | '-'
<OpMultiplicative> ::=  <MultiplyOperator> |  'div' | 'mod'
<NCName> ::=   ( <Letter> |  '_' )?  <NCNameChar>*
<NCNameChar> ::=   <Letter> |  <Digit> |  '.'  |  '-'  |  '_'
                                        | <CombiningChar> | <Extender>
<Number_Lit> ::=   <Digit>*  '.'  <Digit>+  (  ( 'E' | 'e' )   ( '+' |  '-' )
<Digit>+ )
```

```
<String_Lit> ::=  `"'  [ <AnyChar>^'"' ]*  `"'
                                       |   `'' [ <AnyChar>^''' ]*  `''
<Boolean_Lit> ::=   true  |  false
<Date_Lit> ::=   `{`  ( `d' | `D' )  WhiteSpace*  (CC | `-`) YY? `-` MM? `-` DD `}'
<Time_Lit> ::=   `{`  (`t' | `T') WhiteSpace* HH `:' MM `:` SS
                                                                          (`.`
SSS S* )? (`Z' | `-` MM `:' SS )? `}'
<TimeStamp_Lit> ::=   `{`  ( `t' | `T' )  ( `s' | `S' )  WhiteSpace*  CCYY `-` MM
`-` DD `T' <Time_Lit>
```

```
Example query strings include:

   "TestStr1 + 'String2' == 'String1String2' and TestInt - 5 < 10 and TestDouble
< 30"
   "TestDate > {d 1980-08-21} and TestTimestamp <= {ts
1992-05-12T05:33:44.111111000}",
   "TestInt * 2 >= 22 and not (TestDouble >= 55.550)",
   "TestInt > 10 or TestDouble == 10 or TestFloat > 10.0",
   "TestBool != TRUE or TestInt -10 <= 10 and 2345.99 >= TestBigDecimal1",
```

The specialization of `ESQuery` called `ESXMLQuery` can be initialized from a file that contains a query for a Service expressed in XML:

```
public ESXMLQuery(ESConnection coreConnection,
   ESXMLFile xmlQueryFile);
```

Finding Services Using XQL

XQL can be used to find a printer described in `printervocab`. See Appendix C, “IDL Compiler”, for more details about XML DTDs used in e-speak.

The following XML page describes an XQL-based search that is passed to the `ESQuery` constructor:

```
<?xml version="1.0"?>
<ESpeak version="E-speak 1.0" operation="FindService">
   <resource>
      <!-- The search query-->
      <query>
         <queryBlock>
            <WHERE>
               <!-- Begin: Specify printer Vocabulary -->
```

```
                  <query>
                     <queryBlock>
                        <WHERE>
                           <condition>
                              <IN>
                                 <pattern>
                                   <Name>printervocab</Name>
                                 </pattern>
                              </IN>
                           </condition>
                        </WHERE>
                     </queryBlock>
                  </query>
                 <!-- End: Specify printer Vocabulary -->
              <!-- Begin: search conditions -->
                  <condition>
                      <predicate lexpr="DPI" rexpr="1400" RelOp="eq"/>
                  </condition>
                 <!-- End: search conditions -->
               </WHERE>
            </queryBlock>
         </query>
      </resource>
</ESpeak>
```

The Service to be discovered is specified by means of a query. An embedded query specifies the attribute Vocabulary in which attributes for the query conditions are specified. In this case, assume that this XML fragment is in a file called `printFinder.xml`.

The query searches in the Vocabulary called `printervocab` for a printer whose DPI attribute has a value of `1400`.

```
PrintClient.java
public class PrintClient
{
   public static void main(String[] args)
   {
     try
     {
        String propertyFileName =
           new String("/users/connection.prop");
        ESConnection coreConnection =
           new ESConnection(propertyFileName);
        ESXMLFile xmlQueryFile =
           new ESXMLFile("/users/printerQuery.xml");
        ESXMLQuery printQuery = new ESXMLQuery(coreConnection,
           xmlQueryFile);
        String intfName = PrinterServiceIntf.class.getName();
        ESServiceFinder printFinder = new
           ESServiceFinder(coreConnection, intfName);
        PrinterServiceIntf printer = (PrinterServiceIntf)
           printFinder.find(printQuery);
        String document = getDocument();
        printer.print(document);
     } catch (Exception e)
     {
        // handle the exception
     }
  }
}
```

The XML fragment above was the XML used in release 2.2 of the e-speak code. In the 3.0 release, a shorter more concise XML is supported that achieves the same purpose. The XML fragment for the query now looks as follows:

```
<?xml version='1.0'?>
<esquery xmlns="http://www.e-speak.net/Schema/E-speak.query.xsd">
  <result>$serviceInfo</result>
  <where>
    <vocabulary name="printervocab" src="printervocab"/>
    <condition>
      printerVocab:DPI = 1400
    </condition>
  </where>
  <preference>
   </preference>
  <arbitration>
    <operator>first</operator>
    <cardinality>all</cardinality>
  </arbitration>
</esquery>
```

J-ESI supports both the formats of XML documents. However, the older format is deprecated and clients should consider moving to the newer XML format. In addition, there is a new constructor for ESXMLQuery that takes two ESXMLFiles as arguments:

```
public ESXMLQuery( ESConnection connection, ESXMLFile xmlHeader,
               ESXMLFile xmlRequest );
```

In the example above, in the line where printQuery is constructed, this constructor is used instead of the constructor that takes in a single xml file. In addition, the contents of the xmlHeader file look as follows:

```
<?xml version='1.0'?>
<header xmlns="http://www.e-speak.net/Schema/E-speak.header.xsd">
  <communication>
    <to>es://localhost:12346/WebAccess/FindService</to>
  </communication>
</header>
```

Queries in Multiple Vocabularies

You can specify a query using constraints in multiple vocabularies. Essentially, the client can create local names for vocabularies in a query. The constraint expression can then involve these vocabularies. This feature of being able to specify queries in multiple vocabularies is similar to the notion of formulating queries in SQL that query multiple tables. In addition, one can also specify preferences and arbitration policies. The specification of preferences is similar to the notion of the “order by” construct in SQL. Essentially, you can get the finder to return services using the order specified in the preference parameter. The specification of the parameter is done using the addSorter entry point in ESQuery. It takes two parameters, the first being the kind of sorting to be done, and the second, the expression whose value determines the order. In the example below, the code fragment constructs an esquery, that returns the services registered in the “printervocab” sorted by their DPI with the printer with the highest DPI appearing first.

```
ESQuery query = new ESQuery("(printervocab:Name == 'AnyName')");
query.addVocabularyKey("printervocab", v);
query.addOrConstraint("(printervocab:Name != 'AnyName')");
query.addConstraint("(printervocab:Name != 'XXXX')");
query.setArbitPolicy(ESConstants.ARBITRATION_POLICY_ALL);
query.addSorter(ESConstants.SORTER_OPERATOR_MAX,
               "printervocab:DPI");
```

Using Introspection

When finding Services, sometimes the Client wants to know all the Service interfaces supported by the Service the Client has found. The most likely use of this scenario is in browsers, where the user is presented with a set of choices. This is possible in an e-speak environment using the introspection facility. The Client can get a handle to a service without knowing the interface it supports, and after getting the handle can query the Service to find all the Service interfaces the Service supports. The Client can also get a Service stub corresponding to a particular interface and can invoke operations on it. An example follows:

```
ESServiceFinder printFinder = new ESServiceFinder(coreConnection);
ESService printer = printFinder.find(printQuery);
ESHashMap allSupportedInterfaceMap =
   ((ESIntrospectionIntf)printer).getInterfaces();
```

`allSupportedInterfaceMap` contains a map of interface names and the actual class of the interface. The Client can find whether the Service supports a particular interface of interest and then get the stub corresponding to it using

```
PrinterManagementIntf mg = ((ESStubFactoryIntf)
   printer).getServiceStub(“PrinterManagementIntf”);
```

Another use of introspection is in the development of lightweight Clients. The Clients do not need to have any of the class files like the interface, stub files, and so forth. All these can be obtained using introspection and stub factory interfaces, and methods can be invoked using Java reflection. Thus introspection and stub factory interfaces allow building truly dynamic, lightweight, and scalable solutions.

Service Deployer: Creating Services

The previous section explained how a Client gains access to Services created by others. This section explains how Service providers create Services. As described earlier, Service providers perform the following sequence of operations in creating and managing the Service:

1. Create a description of the Service that uses the chosen Vocabulary.
2. Create a Service element that encapsulates the description and associates an implementation and a handler for the Service.
3. Register and start the Service.
4. If necessary, mutate the metadata of the Service.

As with the finders, there are three kinds of elements:

- `ESContractElement`
- `ESVocabularyElement`
- `ESServiceElement`
- ESFolderElement
- ESViewElement

A similar set of classes encapsulates the descriptions of Services:

- `ESContractDescription`
- `ESVocabularyDescription`
- `ESServiceDescription`
- ESViewDescription
- ESFolderDescription

Service Description

Service descriptions are sets of attribute-value pairs expressed in a certain Vocabulary describing the Service. The Vocabulary determines the names and types of the attributes used in the description of the Service. For instance, a printer Service can be described in a Printer Vocabulary. The Printer Vocabulary in turn can contain attributes such as `DPI` that takes on an integer value, `Manufacturer,` that takes on a string value, and `Modelname` that takes on a string value. The printer itself is identified by the fact that it is manufactured by a particular manufacturer, has a particular model name, and prints at a particular DPI value.

Vocabularies, in turn, have to be described, and e-speak breaks the recursion using a Base Vocabulary that has predefined attribute properties described earlier.

All Service Descriptions are associated with two kinds of attributes:

- **Searchable attributes—**Those used in queries by Clients.These are also the attributes listed in each Vocabulary.
- **Service-specific data (Data)—**This data, which is stored along with the description of the Service, cannot be searched for, but can be accessed. This data can be as simple as String entries, or as complex as arbitrary byte code. For example, one Datum in the printer description may be the administrator's contact information (such as a phone number).

In the current API, there are two ways of specifying the descriptions:

- Using XML (This is the recommended method for specifying descriptions.)
- Using the e-speak description of objects provided in the Client library

Searchable Attributes

The attributes that can be part of any Vocabulary can be in one of the 14 e-speak-base types:

- String
- byte
- short
- long
- double
- Date
- Timestamp
- boolean
- char
- int
- float
- BigDecimal
- Time
- byte []

The `ESBaseDescription` class supports the addition of attributes by the `addAttribute` method. These calls take two parameters, the first is the name of the attribute, and the second parameter is the value of the parameter.

ESContractDescription

Contracts are also Services in the e-speak infrastructure. Contracts are typically described using the Base Vocabulary. Contracts encapsulate the Service IDL to which the Service is guaranteed to conform.

The `setInterfaceDefinition` methods set the IDL associated with the Contract:

```
public void setInterfaceName(String intfName);
public void setInterfaceDefinition(String idlString);
public void setInterfaceDefinition(byte[] intfClass);
public void setConversationScheme(String scheme);
public void setTermsOfUse(String terms);
public void setLicense(String license);
```

The following code segment explains how to create a contract using the contract description.

```
String propertyFileName = new String("/users/conection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);

ESContractDescription printContractDescription =
   new ESContractDescription();
printContractDescription.addAttribute("Name","PrintContract");
printContractDescription.setInterfaceName
   ("printer.PrinterServiceIntf");
String idl = getIDLFromFile("PrinterService.esidl");
printContractDescription.setInterfaceDefinition(idl);

ESContractElement printContractElement =
   new ESContractElement(coreConnection, printContractDescription);
ESContract printContract = printContractElement.register();
```

ESVocabularyDescription

This class is used to describe a new Service Vocabulary that is used in descriptions of other Services. As noted earlier, the printer Service is described in the Printer Vocabulary.

There are two parts to describing a Vocabulary. First, because Vocabularies are Services that can be discovered, the attributes of the Vocabulary need to defined so that Clients and Service providers can discover the Vocabulary. Second, the properties of the Vocabulary need to be defined so that any Service that is registered in this Vocabulary has the same properties. These properties can also be used to build queries to find Services.

Vocabularies are somewhat analogous to a table schema in relational databases. The properties that make up the Vocabulary are the columns of the table, and each Service that is registered in the Vocabulary becomes a row in the table. In keeping with the database analogy, it is assumed that the names of the properties in a Vocabulary are case insensitive.

Clients who are creating Vocabularies can set the name of the new Vocabulary they want to create using the addAttribute() method in the ESBaseDescription class.

Just as Service descriptions contain 14 basic attributes in e-speak, the Vocabulary descriptions can contain properties in those 14 types:

- String
- boolean
- byte
- char
- short
- int
- long
- float
- double
- BigDecimal
- Date
- Time
- Timestamp
- byte []

Typically, Clients make a series of calls that add various named properties. For example, to create a description of a printer Vocabulary that has three properties corresponding to the manufacturer, model name, and DPI and that indicates that a particular property is mandatory (meaning that any Service registered in this Vocabulary has to provide a value for this property), use the following code fragment:

```
String propertyFileName = new String("/users/conection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);

ESVocabularyDescription printVocabDescription =
   new ESVocabularyDescription();
printVocabDescription.addAttribute("Name","PrintVocab");
printVocabDescription.addStringProperty("Manufacturer");
printVocabDescription.addStringProperty("Modelname");
printVocabDescription.addIntegerProperty("DPI");
```

```
ESVocabularyElement printVocabElement =
   new ESVocabularyElement(coreConnection, printVocabDescription);
ESVocabulary printVocab = printVocabElement.register();
```

ESServiceDescription

The preceding example describes how a printer Vocabulary is defined. This Vocabulary description can be used by a standards body to register a Printer Vocabulary. Now, a printer manufacturer can choose to use this Vocabulary to advertise a printer with appropriate attribute values.

For example, if a Printer Service provider were interested in offering an “Acme” printer as a Service, it can choose to describe the printer as follows:

```
//describe printer in printervocab
String propertyFileName = new String(“/users/connection.prop”);
ESConnection coreConnection =
   new ESConnection(propertyFileName);
ESVocabularyFinder printVocabFinder =
   new ESVocabularyFinder(coreConnection);
ESQuery printVocabQuery =
   new ESQuery(“Name == ‘PrintVocab’”);
ESVocabulary printerVocab =
   printVocabFinder.find(printVocabQuery);

ESServiceDescription printDescription =
   new ESServiceDescription(printerVocab);
printDescription.addAttribute(“Manufacturer”,”HP”);
printDescription.addAttribute(“Modelname”,”LP 5”);
printDescription.addAttribute(“DPI”, (int) 1400);
```

XML Descriptions of Services

So far, APIs have been used to create and manage descriptions in Java. In addition to these APIs, J-ESI supports XML descriptions of Vocabularies and Services using the constructors described here. XML is the preferred mode for describing Vocabularies and Services. Typically, XML is used for data descriptions passed as parameters to the description constructors.

XML Vocabulary Description

The `ESVocabularyDescription` class has a constructor that takes in an `ESXMLFile` describing the Vocabulary encapsulated in an XML page:

```
public ESVocabulary(ESXMLFile xml_stream, ESConnection conn);
```

This example creates a new Vocabulary with three attribute properties: `Manufacturer`, `Modelname`, and `DPI`:

```
<?xml version="1.0"?>
<ESpeak version="E-speak 2.0" operation="CreateVocab"
xmlns="http://localhost/e:/Esxml/Schemas/espeak.xsd">
   <resource xmlns="">
      <resourceDes xmlns="">
         <pattern>
            <Name xmlns=””>
               printervocab
            </Name>
            <Type>
               Vocabulary
            </Type>
         </pattern>
      </resourceDes>
      <resourceData xmlns="">
         <attrGroup name="my printer Vocabulary" xmlns="">
            <attrDecl xmlns="" name="Manufacturer" required="no">
               <datatypeRef xmlns="" name="string"/>
            </attrDecl>
            <attrDecl xmlns="" name="Modelname" required="no">
               <datatypeRef xmlns="" name="string"/>
            </attrDecl>
            <attrDecl xmlns="" name="DPI" required="no">
               <datatypeRef xmlns="" name="integer">
               </datatypeRef>
            </attrDecl>
         </attrGroup>
      </resourceData>
   </resource>
</ESpeak>
```

All XML requests in e-speak have E-speak as the root element, which specifies the version information as well as the intended operation for the Services described in the request. As noted earlier, two aspects are required to describe Vocabularies.

First, how Clients can find the Vocabulary has to be described. It is assumed that the Vocabularies are themselves defined in the default Vocabulary that allows specifying, among other things, the name and type of Service. In the preceding XML description, the element `resourceDes` specifies the `Name` of this Vocabulary to be `printervocab`. Because this is a Vocabulary, its `Type` is Vocabulary.

Second, the properties of the Vocabulary must be specified. The properties that make up the printer Vocabulary are contained in an `attrGroup` element. The `attrGroup` element contains a list of `attrDecl` elements, each describing the specifics of an attribute property, including its name and data type. The XML file defines a Printer Vocabulary that has three properties: `Manufacturer`, `Model-name`, and `DPI`.

As with queries, J-ESI 3.0 supports a new XML format for creating vocabularies. The new format involves constructing ESVocabularyDescription with two ESXMLFile arguments

```
public ESVocabularyDescription(ESConnection connection,
          ESXMLFile xmlHeader, ESXMLFile xmlRequest)
```

The contents of xmlHeader looks as follows:

```
<?xml version='1.0'?>
<header xmlns="http://www.e-speak.net/Schema/E-speak.header.xsd">
  <communication>
```

```
    <to>es://localhost:12346/WebAccess/CreateVocab</to>
  </communication>
</header>
```

The contents of xmlRequest looks as follows:

```
<?xml version='1.0'?>
<resource xmlns="http://www.e-speak.net/Schema/E-speak.register.xsd">
  <resourceDes>
    <vocabulary>http://www.e-speak.net/Schema/E-speak.base.xsd</vocabulary>
    <attr name="Name">
      <value>printervocab</value>
    </attr>
    <attr name="Type">
      <value>Vocabulary</value>
    </attr>
  </resourceDes>
  <attrGroup name="printervocab"
xmlns="http://www.e-speak.net/Schema/E-speak.vocab.xsd">
         <attrDecl name="Model" required="true">
            <datatypeRef name="String"/>
         </attrDecl>
         <attrDecl name="DPI" required="true">
            <datatypeRef name="String"/>
         </attrDecl>
         <attrDecl name="Manufacturer" required="true">
            <datatypeRef name="String"/>
         </attrDecl>
  </attrGroup>
</resource>
```

XML Service Description

The `ESServiceDescription` class has a constructor that takes in an ESXML file describing the Vocabulary encapsulated in an XML page, as follows:

```
public ESServiceDescription(ESXMLFile xmlDescriptionFile,
   ESConnection coreConnection);
```

This is the recommended mode of creating and registering Services for compatibility with existing or emerging Vocabularies. The following example registers a new Service using the Vocabulary just created:

```
<?xml version="1.0"?>
```

```
<ESpeak version="E-Speak 1.0beta" operation="RegisterService"
xmlns="http://localhost/e:/Esxml/Schemas/espeak.xsd">
  <resource>
   <resourceDes xmlns="" name="Printer">
      <!-- Specify printer Vocabulary -->
      <query xmlns="">
         <queryBlock xmlns="">
            <WHERE xmlns="">
               <!-- absence of query implies Base Vocabulary -->
               <condition xmlns="">
                  <IN xmlns="">
                     <pattern xmlns="">
                        <Name
                            xmlns="">printervocab</Name>
                        <Type
                            xmlns="">Vocabulary</Type>
                     </pattern>
                  </IN>
               </condition>
            </WHERE>
         </queryBlock>
      </query>
      <!-- Begin: attributes -->
      <attrSet xmlns="">
         <!-- End: Use printerVocabulary -->
         <attr xmlns="" name="Manufacturer" required="true">
            <value xmlns="">HP</value>
         </attr>
         <attr xmlns="" name="Modelname" required="false">
            <value xmlns="">LP 5</value>
         </attr>
         <attr xmlns="" name="DPI">
            <value xmlns="">1400</value>
         </attr>
         <!-- End: attributes -->
      </attrSet>
   </resourceDes>
  </resource>
</ESpeak>
```

In the preceding example, XML has been used to describe a specific printer. The XML description has two parts. The first part of the XML document, the Vocabulary in which the rest of the description is to be interpreted, is specified. For example, the query specifies that the following description is in a Vocabulary whose name is `printervocab`:

```
<?xml version="1.0"?>
<ESpeak version="E-Speak 1.0beta" operation="RegisterService"
xmlns="http://localhost/e:/Esxml/Schemas/espeak.xsd">
  <resource>
   <resourceDes xmlns="" name="Printer">
      <!-- Specify printer Vocabulary -->
      <query xmlns="">
         <queryBlock xmlns="">
            <WHERE xmlns="">
             <condition xmlns="">
                  <IN xmlns="">
                     <pattern xmlns="">
                        <Name>printervocab</Name>
              <Type>Vocabulary</Type>
                     </pattern>
                  </IN>
               </condition>
            </WHERE>
         </queryBlock>
      </query>
```

In the second part, the element `attrSet` actually contains the set of attribute value pairs that describe the particular Service in question. The following example describes a printer whose Manufacturer is `HP`, whose Model name is `LP 5`, and whose DPI is `1400`:

```
<!-- Begin: attributes -->
      <attrSet xmlns="">
         <!-- End: Use printer Vocabulary -->
         <attr xmlns="" name="Manufacturer" required="true">
```

```
            <value xmlns="">HP</value>
         </attr>
         <attr xmlns="" name="Modelname" required="false">
            <value xmlns="">LP 5</value>
         </attr>
         <attr xmlns="" name="DPI">
            <value xmlns="">1400</value>
         </attr>
         <!-- End: attributes -->
      </attrSet>
   </resourceDes>
  </resource>
</ESpeak>
```

Just as the 3.0 version of e-speak supports a new format for XML queries, it also supports a new format for service descriptions. The XML that represents the registration request now looks as follows:

```
<?xml version='1.0'?>
<resource
xmlns="http://www.e-speak.net/Schema/E-speak.register.xsd">
  <resourceSpec>
    <locator>
   http://www.hp.com
    </locator>
  </resourceSpec>
  <resourceDes>
    <vocabulary>
   printervocab
    </vocabulary>
         <attr name="Manufacturer">
            <value>HP</value>
         </attr>
         <attr name="Modelname">
            <value>LP 5</value>
         </attr>
         <attr name="DPI">
```

```
            <value>1400</value>
         </attr>
  </resourceDes>
</resource>
```

J-ESI does support the older XML schemas for registering services, but they should be treated like deprecated schemas and clients are encouraged to move to the newer schemas. The header for the service description creation looks as follows:

```
<?xml version='1.0'?>
<header xmlns="http://www.e-speak.net/Schema/E-speak.header.xsd">
  <communication>
    <to>es://localhost:12346/WebAccess/RegisterService</to>
    <context>
      <!--  session token is inserted by the requesting appl. -->
    </context>
  </communication>
</header>
```

The ESServiceDescription constructor that uses the new XML entry points looks as follows:

```
public ESServiceDescription(ESConnection connection,
             ESXMLFile xmlHeader,
             ESXMLFile xmlRequest)
```

Example: Creating a Printer Description

The following example shows how to make an XML description of a printer registered in the e-speak Core. This code assumes that a `printervocab` Vocabulary is found in the registry:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESXMLFile xmlDescriptionFile =
   new ESXMLFile("/users/printer.xml");
ESServiceDescription printDescription =
   new ESServiceDescription(xmlDescriptionFile,
      coreConnection);
```

where `printer.xml` is the XML file that has the contents in the last example. This is much more succinct than explicitly adding all the properties by hand. A typical usage scenario reads a list of products described in a merchant's catalogs, described in XML, to be read in and automatically registered as Services in e-speak.

Now that various methods have been presented for describing Services, the next section describes other operations that Service providers have to perform to make their Service available to other Clients.

Registering and Starting Services

Typically, Service providers have to associate an actual implementation with their Service, register their Service, and start a handler that will handle requests directed to this Service. J-ESI provides an abstraction of a Service element represented by the `ESServiceElement` class for performing these operations.

Consider the steps required to register a printer Service (`PrinterServiceImpl`) that implements the `PrinterServiceIntf`. The following code fragment creates a Service described in the default Vocabulary with the name `MyPrinter`, registers it, and starts a thread that can process requests to the Service. The `register()` call puts the description of the Service into the e-speak Repository so that other Clients can discover it. The `start()` call starts a thread that processes requests to this Service.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESXMLFile xmlDescriptionFile =
   new ESXMLFile("/users/printer.xml");
ESServiceDescription printDescription =
   new ESServiceDescription(xmlDescriptionFile,
      coreConnection);
ESServiceElement printElement =
   new ESServiceElement(coreConnection,printDescription);
printElement.setImplementation(new PrinterServiceImpl());
ESAccessor printAccessor = printElement.register();
printElement.start();
```

In this example, a single thread is started that handles the requests to the Service. All requests destined to this Service are dispatched to an instance of the `PrinterServiceImpl` object.

There are situations where Clients may want to associate multiple Services with the same thread, or have multiple threads for the same Service. The current Client library allows Service providers to control the number of threads through the notion of a Service handler, represented by the `ESServiceHandler` class.

Every connection with the e-speak Core has a default Service handler associated with it. This default Service handler can be retrieved using the `getDefaultServiceHandler` call in the connection, as shown by the following code fragment:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceHandler defaultHandler =
   coreConnection.getDefaultServiceHandler();
```

In essence, each Service handler encapsulates a channel of communication through which messages are received. A single Service handler can serve as a channel for getting messages to multiple Services. Each Service handler also has control over the number of threads that process the messages that are delivered on the communication channel.

In the following example, a print server maintains a message queue for multiple printers. This is made possible by using the same handler for all the printers. The print server has 32 threads that process the requests of the Clients. The following code sample shows how this is accomplished:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceHandler printHandler =
   new ESServiceHandler(coreConnection);
printHandler.setNumThreads(32);

ESServiceDescription printDescription1 =
   new ESServiceDescription();
printDescription1.addAttribute("Name","Printer1");
ESServiceElement printElement1 = new
```

```
   ESServiceElement(coreConnection,printDescription1);
printElement1.setImplementation(new PrinterServiceImpl());
printElement1.setHandler(printHandler);
ESAccessor printAccessor1 = printElement1.register();
printElement1.start();

ESServiceDescription printDescription2 =
   new ESServiceDescription();
printDescription2.addAttribute("Name","Printer2");
ESServiceElement printElement2 =
   new ESServiceElement(coreConnection,printDescription2);
printElement2.setImplementation(new PrinterServiceImpl());
printElement2.setHandler(printHandler);
ESAccessor printAccessor2 = printElement2.register();
printElement2.start();
```

The Service elements for each printer share the same handler. This way, there is a single queue for requests for both printers, and there are 32 threads that handle requests to the printers.

Service-Specific Data

As described earlier, Service-specific data is arbitrary data that can be associated with Services. Service-specific data comes in two forms: public and private.

PublicData is data that can be accessed by any Client that can find the Service.

PrivateData is data that is sent back to the Service handler along with each request to the Service. For example, this data can be used by the Service handler to distinguish among various Services that share this handler.

The following two API calls allow Clients to add public and private data to the descriptions:

```
public void addPublicData(String dataName, byte[] data);
```

This API call adds public Service-specific data to the Service element

```
public void addPrivateData(String dataName, byte[] data);
```

This API call adds private Service-specific data to the Service element.

For example, in a file system implementation, where the file store and the files themselves are Services, the file store may choose to identify the files it creates using private Data. This allows the file store to be very generic because it uses a flexible implementation for storing the bytes that make up a file.

If the bytes are stored on the local disk directly, the file store can use the private Data to keep track of the directory path on the local disk for the file. If the file store instead stores the bytes making up a file in a database table, the information needed to access the row in the table where the bytes of the files are stored is in the private Data.

This allows the file handler to determine where the bytes of the file are stored when a request for this file is received. The names for the data elements are selected by the creator of the Service based on need. Service creators can choose to associate multiple Data elements with every Service they create.

Creating Services Described in Multiple Vocabularies

In the current version of J-ESI, services can be advertised in multiple vocabularies. In essence, the ESServiceElement can be constructed with an array of service descriptions, each of which describes the service in one vocabulary. For instance, suppose that a printer service provider wanted to advertise the printer in two vocabularies. Suppose one of the vocabularies enabled the service provider to advertise the speed of the printer and the second allowed the service provider to advertise the printer quality measured in terms of the dots per inch (DPI). Furthermore, the properties that one can add to a vocabulary can be flagged as primary key properties so that the e-speak repository can build indices using such attributes. This allows the searches that use these properties to be quite efficient.

```
ESServiceDescription sd[] = new ESServiceDescription[2];
try{
   ESConnection connection = new ESConnection("file.pr");

   //Creating the vocabulary........
   ESVocabularyDescription vocabDesc = new
ESVocabularyDescription();
   ESVocabularyDescription vocabDesc1 = new
ESVocabularyDescription();
```

```
   //Setting the vocabulary name
   vocabDesc.addAttribute("Name","vocab");
   vocabDesc1.addAttribute("Name","vocab1");

   //Adding vocabulary properties
   vocabDesc.addStringProperty("printerSpeed");
   vocabDesc1.addStringProperty("printerDPI");

   //Registering the vocabulary
   ESVocabularyElement vocabElem = new
ESVocabularyElement(connection,vocabDesc);
   ESVocabularyElement vocabElem1 = new
ESVocabularyElement(connection,vocabDesc1);
   ESVocabulary vocab = vocabElem.register();
   ESVocabulary vocab1 = vocabElem1.register();
       ESServiceDescription sd = new ESServiceDescription[2];
   //Creating the service description
   sd[0] = new ESServiceDescription(vocab);
   sd[1] = new ESServiceDescription(vocab1);

   //Setting the service name
   sd[0].addAttribute("Name","myPrinter");
   sd[0].addAttribute(“printerSpeed”, “10”);
   sd[1].addAttribute("Name","myPrinter");
   sd[1].addAttribute(“printerDPI”, “1000”);
   //Create the service element
   ESServiceElement se = new ESServiceElement(connection,
               sd);
```

Restarting Existing Services

Just as Service providers can start new Services, J-ESI allows Service providers to restart existing Services. Clients may want to restart Services in order to resume a Service after recovering from a Service shutdown or outage. Clients can restart Services in three ways: (i) They can construct a Service element from the description and call `restart()` on the element, (ii) they can store away the accessor for the service in a persistent folder and use the accessor to restart the service, or (iii) they can store away the accessor of the handler in a persistent folder and use that accessor to restart the service. Modes (ii) and (iii) are discussed in the next chapter in the section on folders.

The element attempts to find a Service whose description matches the description currently provided in the element. If such a Service is found, it activates that Service so that requests to that Service are now handled. However, if no such Service is found, an exception occurs and the Client is expected to catch the exception and then register and start the Service.

However, the recommended mode for restarting Services is with the use of Folders. The Client is expected to have created a local name for the Service in their folder hierarchy. See "Managing Bindings Using Folders" on page 71 to determine how to restart Services using the binding.

The following example shows the typical use of the restart method without the use of Folders:

```
public static void main (String [] args)
{
   try
   {
      String propertyFileName =
         new String("/users/connection.prop");
      ESConnection coreConnection =
         new ESConnection(propertyFileName);
      ESXMLFile xmlDescriptionFile =
         new ESXMLFile("/users/printer.xml");
      ESServiceDescription printDescription =
         new ESServiceDescription
            (xmlDescriptionFile, coreConnection);
      ESServiceElement printElement =
         new ESServiceElement(coreConnection, printDescription);
      printElement.setImplementation(new PrinterServiceImpl());
      ESServiceHandler printHandler =
         new ESServiceHandler(coreConnection);
      printHandler.setNumThreads(2);
```

```
        printElement.setHandler(printHandler);
        try
        {
          printElement.restart();
          System.out.println("XMLPrintServer re-started");
        }
        catch (ESInvocationException esie)
        {
          printElement.register();
          printElement.start();
          System.out.println("XMLPrintServer started");
        }
      }
      catch (Exception e)
      {
        // handle the exception
      }
    }
```

Registering Vocabularies and Contracts

Vocabularies and Contracts are themselves Services that are registered with the e-speak infrastructure. However, the Service provider for Vocabularies and Contracts is the e-speak Core itself. Because of this, Clients who create Vocabularies only register them, and do not start a thread in order to serve requests to the Vocabulary or Contract.

The classes that allow Clients to register Vocabularies and Contracts are `ESVocabularyElement` and `ESContractElement`, respectively. The following code fragment registers a printer Vocabulary that has three properties: `DPI`, `Manufacturer`, and `Modelname`:

```
public class VocabularyCreator
{
  public static void main(String[] argv)
  {
      try
      {
         String propertyFileName =
            new String("/users/connection.prop");
         ESConnection coreConnection =
            new ESConnection(propertyFileName);

         ESVocabularyDescription printVocabDescription =
            new ESVocabularyDescription();
         printVocabDescription.addAttribute
            ("Name","printervocab");
         printVocabDescription.addIntegerProperty("DPI");
         printVocabDescription.addStringProperty("Manufacturer");
         printVocabDescription.addStringProperty("Modelname");

         ESVocabularyElement printVocabElement =
         new ESVocabularyElement(coreConnection,
               printVocabDescription);
         ESVocabulary printVocab = printVocabElement.register();

         Property[] propertyList = printVocab.getProperties();
         for (int i = 0; i < propertyList.length; i++)
         {
            System.out.println(propertyList[i].getPropertyName());
         }
      }
      catch (Exception e)
      {
         // handle the exception
      }
   }
}
```

Creating Contracts is very similar to the creation of Vocabularies. The following code fragment shows the creation of a simple Contract:

```
public class ContractCreator
{
   public static void main(String[] argv)
   {
      try
      {
         String propertyFileName =
            new String("/users/connection.prop");
         ESConnection coreConnection =
            new ESConnection(propertyFileName);

         ESContractDescription printContractDescription =
            new ESContractDescription();
         printContractDescription.addAttribute
            ("Name","PrintContract");
         String printIDL = getIDLFromFile();
         printContractDescription.setInterfaceDefinition(printIDL);

         ESContractElement printContractElement =
            new ESContractElement(coreConnection,
               printContractDescription);
         ESContract printContract = printContractElement.register();

         String receivedPrintIDL =
            printContract.getInterfaceDefinition();

         System.out.println(receivedPrintIDL);
      }
      catch (Exception e)
      {
         // handle the exception
      }
   }
}
```

On registering a Vocabulary or Contract, the Clients receive a stub to the created Vocabulary or Contract. This is in contrast to other Services that the Client creates, where they receive an accessor to the Service that they create. (`ESAccessor` is explained in detail in the later sections) This is because the Client that is registering the Vocabulary or Contract is not the handler for that Vocabulary or Contract.

A Bank Service Example

The following example is a complete Bank Service example that makes use of many of the concepts introduced in this chapter. Figure 6 shows the relationship between a Bank Service, a Bank Service Contract, and the Bank Service Vocabulary. Typically, the Bank Vocabulary, Bank Contract, and Bank Service are not defined by the same programmer; however, for the sake of simplicity in this example, we can assume that they are defined by the same person.

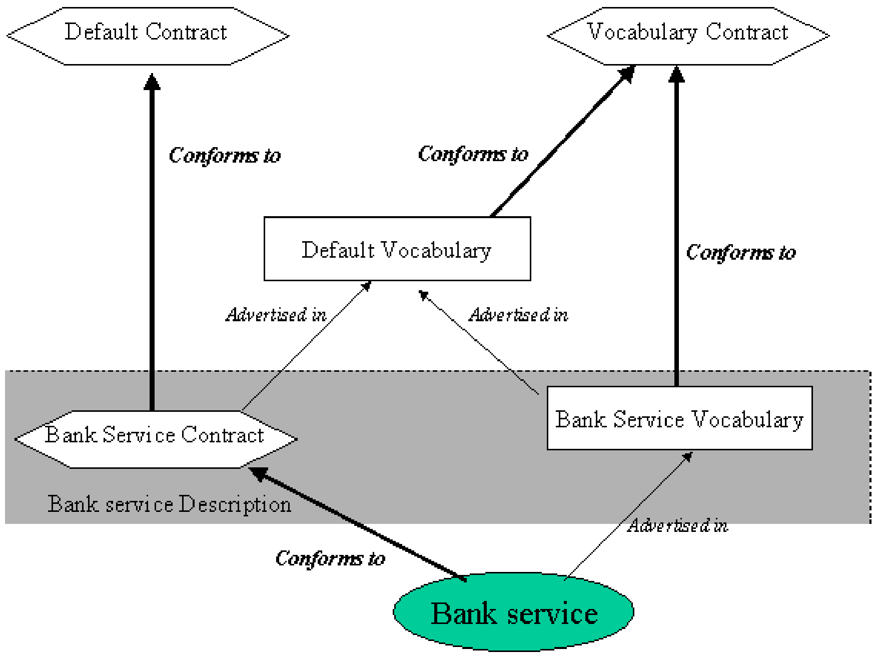


Figure 6 Example of a Bank Service

The following example lists the sequence of actions performed by a Bank Service (assuming the Vocabulary, Contract, and Service are all defined by the same piece of code):

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESContractDescription bankContractDescription =
   new ESContractDescription();
bankContractDescription.
   addAttribute("Name","BankContract");
ESContractElement bankContractElement =
   new ESContractElement(coreConnection,
      bankContractDescription);
ESContract bankContract = bankContractElement.register();

ESXMLFile xmlDescriptionFile =
   new ESXMLFile("/users/bankVocab.xml");
ESVocabularyDescription bankVocabDescription = new
   ESVocabularyDescription(xmlDescriptionFile,
      coreConnection);
ESVocabularyElement bankVocabElement = new
   ESVocabularyElement(coreConnection,
      bankVocabDescription);
ESVocabulary bankVocab = bankVocabElement.register();


xmlDescriptionFile =
   new ESXMLFile("/users/bank.xml");
ESServiceDescription bankDescription =
   new ESServiceDescription(xmlDescriptionFile,
      coreConnection);
bankDescription.setContract(bankContract);

ESServiceElement bankElement =
   new ESServiceElement(coreConnection, bankDescription);
bankElement.setImplementation(new BankServiceImpl());
ESAccessor bankAccessor = bankElement.register();
```

```
bankElement.start();
```

The following example shows the contents of the `bankVocab.xml` file that is used to describe the Vocabulary:

```
<?xml version=”1.0”?>
<ESpeak version=”E-speak 1.0” operation=”CreateVocab”>
   <resource>
      <!-- Begin: Specify the Vocabulary description -->
      <resourceDes>
         <attrSet>
            <attr name=”Name”>
               <value>bankvocab</value>
            </attr>
         </attrSet>
      </resourceDes>
      <!-- End: Specify the Vocabulary description -->
      <resourceData>
         <!-- Begin: Specify the attribute property set -->
         <attrGroup name=”Bank Vocabulary”>
            <attrDecl  name=”Name”>
               <datatypeRef  name=”string”/>
            </attrDecl>
            <attrDecl name=”createAccount”>
               <datatypeRef  name=”string”/>
            </attrDecl>
            <attrDecl name=”tradeStock”>
               <datatypeRef name=”string”/>
            </attrDecl>
         </attrGroup>
         <!-- End: Specify the attribute property set -->
      </resourceData>
   </resource>
</Espeak>
```

The following example lists the contents of the `bank.xml` file that is used to describe the particular Bank Service:

```
<?xml version=”1.0” ?>
```

```
<ESpeak version="E-Speak 1.0" operation="RegisterService">
   <resource>
      <resourceDes name="Bank Description">
         <!-- Begin: Specify bank Vocabulary in a query-->
         <query>
            <queryBlock>
               <WHERE>
                  <condition>
                     <IN>
                        <pattern>
                           <Name>bankvocab</Name>
                         </pattern>
                     </IN>
                  </condition>
               </WHERE>
            </queryBlock>
         </query>
         <!-- End: Specify bankVocabulary -->
         <!-- Begin: the attribute list -->
         <attrSet>
            <!-- End: Use bank Vocabulary -->
            <attr name="Name">
               <value>Acme</value>
            </attr>
            <attr name="createAccount">
               <value type="string">"Acme 1"</value>
            </attr>
            <attr name="tradeStock">
               <value type="string">"hwp"</value>
            </attr>
         </attrSet>
         <!-- End: the attribute list -->
      </resourceDes>
   </resource>
</ESpeak>
```

Accessing Descriptions: ESAccessor

`ESAccessor` is a reference to a Service with which the user can do the following.

- Obtain or change the attributes of a Service
- Send messages to the Service (More details are in Appendix B)

Typically, a Client finds a Service that matches some desired attributes, but after finding the Service, the Client may be interested in checking the values of other attributes as well.

For example, a Client may find a list of printers meeting a certain DPI value, but after finding this list, the Client may want to determine the manufacturer name before proceeding further. In such situations, the Client can get the accessor for each Service and query the accessor for the manufacturer attribute's value.

Similarly, an administrator who manages the printer may want to mutate one of the searchable attributes or add new Data to reflect some upgrade to the printer. In this case, the `ESAccessor` is used to mutate the Service description as well. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESXMLFile xmlDescriptionFile =
   new ESXMLFile("/users/printerQuery.xml");
ESXMLQuery printQuery = new ESXMLQuery(coreConnection,
   xmlDescriptionFile);
ESServiceFinder printFinder = new ESServiceFinder(coreConnection);
ESService printService = printFinder.find(printQuery);
ESAccessor printAccessor =
   ((ESAccessorHandle)printService).getAccessor();
```

On the Service provider side, when a Service is registered, an accessor is returned as a result of doing the register call. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceDescription printDescription =
   new ESServiceDescription(printVocab);
printDescription.addAttribute("Modelname", "HP1000");
```

```
ESServiceElement printElement =
   new ESServiceElement(coreConnection, printDescription);
printElement.setImplementation(new PrinterServiceImpl());
ESAccessor printAccessor = printElement.register();
printElement.start();
```

The Service provider can now use the accessor to access and mutate the description of the Service.

Consider a printer Client who, after finding a printer with `DPI == 1400`, decides to list all the attributes of the printer, perhaps in an effort to find out its manufacturer, speed, and so on.:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceFinder printFinder =
   new ESServiceFinder(coreConnection);
ESQuery printQuery = new ESQuery(printervocab);
printQuery.addConstraint("DPI==1400");
ESService printService = printFinder.find(printQuery);
ESAccessor printAccessor =
   ((ESAccessorHandle)printService).getAcessor();
System.out.println("Attributes of this printer are:\n");
ESAttribute[] attrList =printAccessor.getAttributes();
for(int i=0; i<attrList.length; i++)
{
   System.out.println(attrList[i].toString());
}
```

A Service provider who has created a Service or a Client and who has the appropriate permissions can change the attributes of the Services. For example, if the administrator upgrades the printer to the latest model that supports a higher DPI, the administrator can update the description of the printer with the new values for the attributes as follows:

```
// code to find print service from above
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESVocabularyFinder printVocabFinder =
   new ESVocabularyFinder(coreConnection);
ESQuery printVocabQuery =
   new ESQuery("Name == 'printervocab'");
ESVocabulary printVocab = printVocabFinder.find(printVocabQuery);

ESQuery printQuery = new ESQuery("Manufacturer == 'HP'");
ESService printService = printFinder.find(printQuery);
ESAccessor printAccessor =
((ESAccessorHandle)printService).getAccessor();

ESAttribute attr1 = new ESAttribute("Manufacturer");
attr1.setValue("HP");
ESAttribute attr2 = new ESAttribute("DPI");
attr1.setValue((int) 1400);

printAccessor.setAttribute(attr1, printVocab);
printAccessor.setAttribute(attr2, printVocab);
```

The ESAccessor class also has other methods that can be used to mutate the meta-data of services. Examples of such methods are the following. Note that though some of these signatures refer to ESBaseDescription, in practice, these return values or arguments to these functions are instances of ESVocabularyDescription, ESContractDescription, ESViewDescription, ESServiceDescription, etc.

```
public void setDescriptions(ESBaseDescriptions [] desc);
public void setDescription(ESBaseDescription desc);
public void setAttribute(ESAttribute attr, ESVocabulary vocab);
public void setAttributes(ESAttribute [] attrs, ESVocabulary vocab);
public ESBaseDescription [] getDescriptions();
public ESBaseDescription getDescription();
public ESAttribute getAttribute(String name, ESVocabulary vocab);
public ESAttribute [] getAttributes(String name, ESVocabulary
vocab);
```

Chapter 3 Extended Services

The previous chapters described the programming model, and how e-speak Clients and Services interact. In addition to support for these basic functions, J-ESI supports extended Services that enable support for persistent bindings, Event-based interaction semantics, and loosely coupled distributed Communities. This chapter shows how these extended Services can be used to create sophisticated Service interactions.

The chapter is divided into the following sections:

- Managing Bindings Using Folders
- Repository Views
- Categories
- Communities
- Security
- Events

Managing Bindings Using Folders

One of the main reasons why e-speak can support loosely coupled distributed Services is that it does not rely on global naming. In many traditional distributed systems, the distribution is possible because Clients know the name of the Service providers and they use the name to access the Service.

In e-speak, on the other hand, there are no global names. Clients can make up names for the Services they find or create, and that name is independent of the name that another Client uses for the same Service.

For instance, one Client may refer to a Printer Service as “Marketing Printer,” while another Client may refer to the same Printer Service as “Engineering Printer.” This allows Services to be migrated or upgraded, without having to change the Client-side programs that use these Services. Figure 7 illustrates the naming process in e-speak.

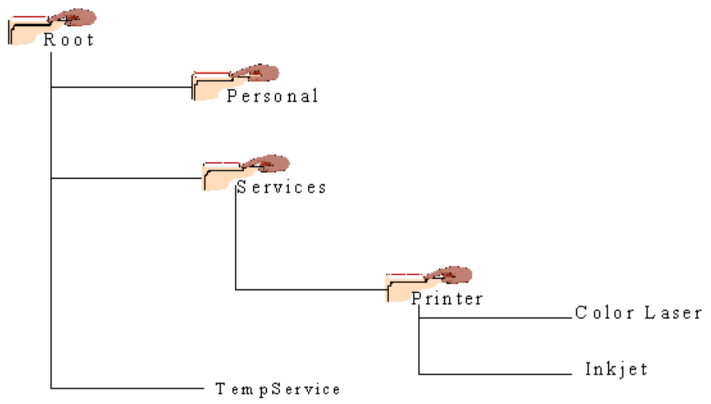


Figure 7 Example of folder names using e-speak

In J-ESI, Clients manage their local name spaces using folders. Folders are analogous to directories in traditional file systems. Clients can create bindings between the names and Services they find and then put the bindings in a folder.

Essentially, folders enable Clients to build a local hierarchical name space. Every user account in an e-speak Core has a folder that is its root (typically denoted by /). Clients can get at the root folder in their session by invoking the `getRootFolder` method in `ESConnection`. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESFolder myRoot = coreConnection.getRootFolder();
```

Creating Folders

Folders created by Clients are either persistent or transient. A transient folder is a folder that does not survive beyond the lifetime of the connection in which it is created. A persistent folder, on the other hand, can survive beyond the lifetime of the connection in which it is created. If the Core is backed up in a database, persistent folders also survive Core reboots. This is because, although the folders are a Client-side abstraction, their state is maintained in the e-speak Core.

Creating Transient Folders

Currently, Clients can create transient or persistent folder hierarchies under the root folder. To create a transient folder hierarchy, they create a subfolder of the root folder that is transient. For example, to create a transient folder under the root folder called `bookmarks`, use the following constructor of `ESFolder`:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESFolder bookMarkFolder = new ESFolder(coreConnection, "bookmarks");
```

This call creates a transient folder under the root folder called `/bookmarks`. The folder constructor is used to create subfolders of the root folder, while the `createSubFolder` method is used to create subfolders of all non-root folders.

Creating Persistent Folders

The above ESFolder constructor cannot be used to create persistent folders. To create a persistent folder called `/myServices` under the root folder, the following call is used:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESFolder ownFolder = new ESFolder(coreConnection, "myServices",
   true);
```

By default, a root folder is persistent. Hence, the user invokes the `createSub-Folder` method on the root folder to create persistent folders. Because the folder `/myServices` is a persistent folder, any subfolder of `/myServices` is also persistent.

As in most operating systems, there is the idea of a current folder in J-ESI. Clients can get their current folder through the Service context, as shown by the following code fragment:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceContext connectContext =
   coreConnection.getServiceContext();
ESFolder currentFolder = connectContext.getCurrentFolder();
```

When a new connection is created, the name of the home folder is read from the properties file that is passed to the constructor of `ESConnection`. The value of the property named `homefolder` in the properties file is used as the name of the home folder. If no properties file is passed to the constructor of `ESConnection`, the default home folder is `/home`. The home folder can be obtained as follows:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESFolder homeFolder = coreConnection.getHomeFolder();
```

The properties of the current folder that the Client is located in are important. If a Client creates Services while in a persistent folder, the Service is created as a persistent Service. This means that the description of the Service is registered with the e-speak Core even if the handler of the Service disconnects from the e-speak Core.

Naming Found Services

Clients typically create folders to manage name bindings on Services they have discovered or created. For example, an administrator may add bindings to discovered print Services in a persistent folder (such as `/home/services/printers`). Because the bindings are stored in a persistent folder, anytime the administrator reconnects, they continue to have access to previously discovered printers by simply looking into the persistent `/home/printers` folder.

The following code fragment shows the creation of a subfolder `'printers'` of the Client's home folder (`/home/services`). After `printers` is created, the Client finds a printer and places a name binding for it in this folder.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceContext connectContext =
   coreConnection.getServiceContext();
ESFolder currentFolder = connectContext.getCurrentFolder();
// The current folder is assumed to be /home/services.
// This can be set in the properties file
ESFolder newFolder = null;
try {
   newFolder = currentFolder.getSubFolder("printers");
}
catch (InvalidNameException ine){
   newFolder = currentFolder.createSubFolder("printers");
}
connectContext.setCurrentFolder(newFolder);
PrinterServiceIntf printer = null;
try{
   printer = (PrinterServiceIntf)
      newFolder.getService("myprinter", "PrinterServiceIntf");
}
catch (ESInvocationException esie){
   ESXMLFile xmlQueryFile = new ESXMLFile("/users/printerQuery.xml");
   ESXMLQuery printQuery =
      new ESXMLQuery(coreConnection, printQuery);
   String intfName = PrinterServiceIntf.class.getName();
   ESServiceFinder printFinder =
      new ESServiceFinder(coreConnection, intfName);
   printer = (PrinterServiceIntf) printFinder.find(printQuery);
   newFolder.add("myprinter", printer);
}
// get the contents of the folder
String[] contents = newFolder.listNames();
Naming Created Services
Naming a Service which is created and adding it to a folder is similar to naming a
found Service. The following code shows an example of how to name a created Service.
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceContext connectContext =
   coreConnection.getServiceContext();
ESFolder currentFolder = connectContext.getCurrentFolder();
// The current folder is assumed to be /home/services.
// This can be set in the properties file
ESFolder newFolder = null;
try {
   newFolder = currentFolder.getSubFolder("printers");
}
catch (InvalidNameException ine){
   newFolder = currentFolder.createSubFolder("printers");
}
connectContext.setCurrentFolder(newFolder);
if (!(newFolder.containsName("fastprinter"))){
   ESXMLFile xmlDescriptionFile =
      new ESXMLFile("/users/printer.xml");
   ESServiceDescription printDescription =
      new ESServiceDescription(xmlDescriptionFile, coreConnection);
   ESServiceElement printElement = new ESServiceElement(
      coreConnection, printDescription);
   printElement.setImplementation(new PrinterServiceImpl());
   ESServiceHandler essh = new ESServiceHandler(coreConnection);
   printElement.setHandler(essh);
   ESFolder homeFolder = coreConnection.getHomeFolder();
   homeFolder.add(HANDLER_NAME, essh.getAccessor());
   ESAccessor printAccessor = printElement.register();
   printElement.start();
   newFolder.add("fastprinter", printAccessor);
}
else
{
   ESServiceElement printElement = new
      ESServiceElement(newFolder.getAccessor("fastprinter"));
   printElement.setImplementation(new PrinterServiceImpl());
   printElement.restart();
}
```

When the Service provider goes off-line and logs back into the e-speak Core, they do not have to re-create the Service. The provider looks for the name binding corresponding to the printer that they created and performs a restart. This allows the print Service to go online and to respond to requests.

An alternative way to restart services is to use the accessor of the service handler. A service provider can store away a binding to the service handler that was created to handle requests to her services in a folder. On restarting, the service handler can be recreated from this binding, and a service element can be recreated from the service handler. This allows multiple services that share a handler to be restarted without creating a service element for each service all over again. The following code snippet shows how to do this.

```
 ESConnection conn = new ESConnection("propfile");
 ESFolder home = conn.getHomeFolder();
  ESAccessor da = home.getAccessor(HANDLER_NAME);
  ESServiceHandler handler = new ESServiceHandler(da);
  handler.setNumThreads(numThreads);
ESServiceElement element = new ESServiceElement(handler);
  element.setImplementation(new PrintServiceImpl());
  element.restart();
```

Creating and Finding Folders with Descriptions

Clients can create folders and describe them in any Vocabulary just like any other Service. By doing this, the folders can be found by other Clients using the queries that are in the Vocabulary in which the folder was advertised. Creating the folders with descriptions allows other Clients to discover the folders and use the name bindings that have been created within the folders.

For example, if Client A creates a folder with bindings for print Services that they want to share with Client B, Client A can create the folder with a description and advertise it so that Client B can discover the folder. Client B can then use the bindings without also having to create their own bindings.

Some of the constructors of `ESFolder`, as well as some of the `createSubFolder` calls in `ESFolder`, take an additional `ESServiceDescription` that describes the attributes of the folder that is being created. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESXMLFile xmlDescriptionFile = new ESXMLFile("/users/folder.xml");
ESServiceDescription folderDescription =
   new ESServiceDescription(xmlDescriptionFile, coreConnection);
ESFolder newFolder = new ESFolder("services", folderDescription);
```

The preceding constructor of `ESFolder` causes the description of the folder to be registered in the Repository of the Core so that other Clients can discover the folder description. Folders are found using the `ESFolderFinder` class. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESFolderFinder fFinder = new ESFolderFinder(coreConnection);
ESXMLFile xmlQueryFile =
   new ESXMLFile("/users/folderQuery.xml");
ESXMLQuery folderQuery =
   new ESXMLQuery(coreConnection, xmlQueryFile);
ESFolder folder = fFinder.find(folderQuery);
```

Navigating Folders

The `ESFolder` class has methods for navigating a folder hierarchy. For example, to get the subfolder of a given folder, there is a `getSubFolder` call. On the other hand, to get the parent of any folder, a `getParent` call is issued. In addition, there are other calls that allow the Client to list the contents of the folders.

Scopes

Scopes are used to mark the lifetime of transient and persistent Services.

Before scopes can be used, boundaries need to be defined for them. The following method in `ESServiceContext` is used to signify scope boundaries:

```
   public void beginTransientScope();
   public void endTransientScope();
```

Ending a scope results in all the transient/persistent Services created in the scope being deleted. Furthermore, all the transient bindings for Services found by the Client are also deleted. For example:

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
ESServiceContext connectContext =
   coreConnection.getServiceContext();
…
while(notDone())
{
   connectContext.beginTransientScope();
   …
   createFindAndUseServices(coreConnection);
   …
   connectContext.endTransientScope();
}
```

In this example, the `createFindAndUseServices` is a method that creates and uses a transient Service. The end scope call following this method invocation automatically deletes all transient bindings and Services created in the method.

Repository Views

The repository view is a feature in e-speak that allows clients to restrict the scope of their searches to within the view. These views are collections of e-speak registered e-speak services. Clients can add and remove services from these views. These views are maintained by the e-speak core engine, therefore they are core-managed services. These repository views are described using ESViewDescription just as vocabularies are described with ESVocabularyDescription and other services are described with ESServiceDescription. Similarly, clients use the ESViewFinder to find these repository views and create ESViewElement instances in order to register new repository views with the core repository.

The following code snippet shows how a client can create a repository view, find it, and add elements to it, etc.

```
public class testView
{
    public static void main(String args[])
    {
        ESConnection es = null;
        try {
     es = new ESConnection();
     ESViewDescription esd = new ESViewDescription();
     esd.addAttribute("Name","TestView");
     ESViewElement ese = new ESViewElement(es,esd);
     ESView v = ese.register();
     ESViewFinder esf = new ESViewFinder(es);
     ESView[] ess = esf.findAll(new ESQuery("Name == 'TestView'"));
     ESAccessor esa = ((ESAccessorHandle) ess[0]).getAccessor();
     ((ESViewStub)ess[0]).add(esa);
     boolean result = false;
     result = ((ESViewStub)ess[0]).contains(esa);
     // get all the accessors in the view
     ESAccessor[] e = ((ESViewStub)ess[0]).list();
     es.close();
   }catch(Exception e) {
   }
   }
}
```

Restricting searches with views

The typical use of such repository views is to restrict the search results from queries. This is accomplished by associating a view with a query. When an ESQuery with an associated view is executed by the e-speak repository, the search results are guaranteed to be from the elements within the repository view.

```
ESConnection es = new ESConnection();
  System.out.println("Connected to core");
ESViewFinder esf = new ESViewFinder(es);
  ESView view = esf.find(new ESQuery("Name == 'myPrintView'"));
ESServiceFinder finder = new ESServiceFinder(es,
“PrinterServiceIntf”);
ESVocabulary pVocab = // .. code to find vocabulary
ESQuery query = new ESQuery(pVocab, “printerSpeed == ‘10’”);
query.setView(view);
try {
          ESService[] list = finder.findAll(query);
          for (int i = 0; i < list.length; i++) {
             ESServiceStub printer = (ESServiceStub)list[i];
             ESAccessor desc = file.getAccessor();
          }
}
  catch (LookupFailedException lfe) {
        // No entries were found that matched the search criteria
      }
```

Categories

J-ESI supports the notion of categories. These categories provide a way to classify services so that service providers can advertise in multiple categories and clients can find services of interest to them in the categories of interest to them. Each category is qualified by its name. Furthermore, one can create sub-categories of existing categories, and provide descriptions for them. When clients search for services, they too can set the categories that they want the search to be performed in. This becomes part of their query and picks out the services that are advertised in the categories of interest to the client.

The category list of interest to the client/service provider is maintained in the ESServiceContext that is associated with the ESConnection. The following example shows the registration of the print service in the “high speed printer” category. Essentially, the service provider sets the current category list in the ESServiceContext and these categories are used when registering and advertising the service.

```
public static void main(String[] args)
    {
        try {
     ESConnection    connection = new ESConnection();
ESCategoryFinder cf = new ESCategoryFinder(connection);
     ESCategory root = cf.findRootCategory();
     ESCategory cat1 = root.createCategory("high speed printer",
"high ppm");
     ESCategory[] catList = new ESCategory[2];
     catList[0] = root;
     catList[1] = cat1;
     connection.getServiceContext().setCategory(catList);
     ESServiceDescription sd = new ESServiceDescription();
     sd.addAttribute(ESConstants.SERVICE_NAME, "printer");
     sd.addAttribute("Description", "my hp printer");
     ESServiceElement se = new ESServiceElement(connection, sd);
     se.setImplementation(new PrinterServiceImpl());
     se.register();
     se.advertise();
     se.start();

   } catch (Exception e1) {
   }
}
```

Now, on the client side, the finder for the printers also uses the category list that is set in the service context. The code snippet below shows how to search for printers in the “high speed printer” category. Note that the client has to construct a category list that represents the path from the root category to the category of interest.

```
public static void main(String [] args) {

try {
   String propFileName = args[1];
   ESConnection connection = new ESConnection( propFileName );
    ESCategoryFinder cf = new ESCategoryFinder(connection);
   ESCategory root = cf.findRootCategory();
   ESAccessor[] cat1Acc = cf.find("high speed printer");
   ESCategory cat1 = new ESCategoryStub(connection, cat1Acc[0]);
   ESCategory[] catList = new ESCategory[2];
   catList[0] = root;
   catList[1] = cat1;
   connection.getServiceContext().setCategory(catList);

   ESQuery q = new ESQuery (ESConstants.SERVICE_NAME +
"=='printer'");
   ESServiceFinder sf = new ESServiceFinder(connection,
                                            "PrinterServiceIntf");
   ESService[] myObjs = sf.findAll(q);

   if(( myObjs == null ) || (myObjs.length == 0))
      System.out.println( "ERROR IN CREATING STUB/FINDING" );
      System.out.println("Found " + myObjs.length + " services");

   for(int i=0; i<myObjs.length; i++) {
      PrinterServiceIntf myObj =
                          (PrinterServiceIntf)myObjs[i];
             String junk = myObj.print();
          }
    connection.close();
          return;
   } catch (Exception e1) {
   }
}
```

Delegators

Often, there is a need for the implementation objects associated with services to have access to some of the metadata of the services that the implementation object is associated with. For example, a file implementation is to be shared amongst multiple files that are registered as services with e-speak. Suppose furthermore, that the private service specific data is used to store the actual path to the contents of the file on disk. In particular, suppose that the name of the service specific data is “RealFileName”. Essentially, when the implementation object extends the ESDelegatorImpl, the service specific data of the service is passed onto the implementation object. This allows the implementation object to determine the exact service for which this request is meant.

```
public class VFSFileImpl extends ESDelegatorImpl{
final static String REAL_FILE_NAME = "RealFileName";

public VFSFileImpl(ESConnection connection, ESVocabulary vocab, ESLogClient log,
                       int numThreads) {

// constructor....
}

    public byte[] fetchBuffer(int offset, int size)
    throws ESInvocationException {
          try {
            String fileName = getFileName();
             File file = new File(fileName);
            int count = size;
            int whatsLeft = (int)file.length() - offset;
            if (whatsLeft < count) {
                count = whatsLeft;
             byte [] fileBuffer = new byte[count];
            FileInputStream in = new FileInputStream(file);
            in.skip((long)offset);
            in.read(fileBuffer);
            in.close();
            return fileBuffer;
        } catch (Exception ioe) {
          } finally {
          }
    }

private String getFileName() {
      try {
            byte[] entry = getPrivateData(REAL_FILE_NAME);
            String fileName = new String(entry);
              return fileName;
        } finally {
          }
    }

}
```

Communities

So far, the discussion has focused on how e-speak Clients can register and discover new Services while using folders to manage Service bindings. No mention has been made yet of multiple connected Cores, the distributed nature of deployments, or the impact of distribution on failure semantics, latency, and concurrency.

J-ESI makes it easy for programmers to write distributed Services. Although all Services are registered in the local Core by default, these can be made more widely visible by advertising this Service across multiple Cores. In J-ESI, the possible domains in which a Service is visible are as follows:

- **Only the e-speak Core—**In this deployment scenario, all Clients that want to use this Service are also connected to the same Core, loosely representative of a classic Client-server deployment of Services.
- **In an e-speak group—**An e-speak group is a collection of e-speak Cores that are closely connected to each other, such as in an administrative domain. These Cores typically can find all Services registered in any of the other Cores, and they may all share the same back-end server (possibly an LDAP server) for storing all Services registered in the group. E-services Village, a HP hosted service directory is an example of such a service directory. Such deployments are analogous to lookup or naming servers used in other solutions. Note: The advertising services without LDAP should use the same group name in the command line -group <groupname> option if they want to be part of the same group.
- **In an e-speak community—**An e-speak community is a Client-defined named set of e-speak groups that is created by Clients to enable them to search through different related sets of Services easily. The default community in J-ESI includes www.eservicesvillage.com. Therefore, every query that any client executes returns results that are not only in the local core, but also in eservices village.

In Figure 8, the top left corner shows a single Core case where all Clients and Service providers are connected to the same Core. The top right corner shows an e-speak group in which a closely connected group of Cores share an LDAP server or use other mechanisms to advertise all their Services to each other. The bottom figure shows an e-speak community that can be formed between groups A and B. A Client in group A can select to find Services in group B, by setting the community list to include group B.

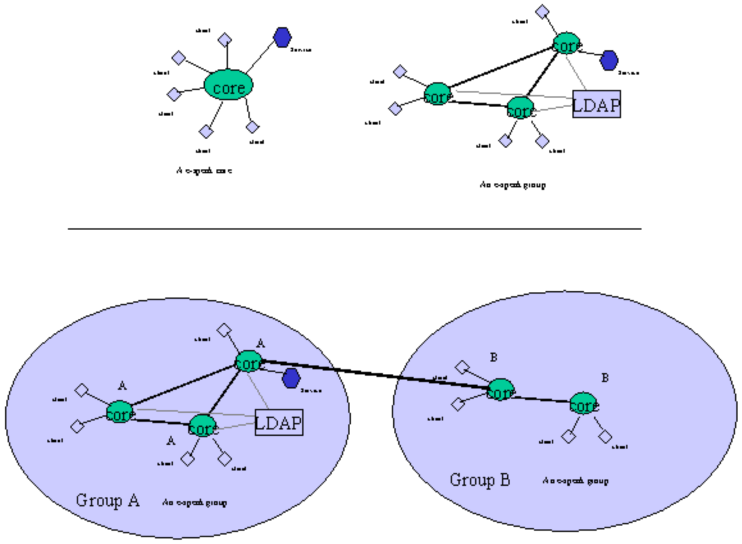


Figure 8 E-speak Core groups

A Service provider can advertise the Service in the local advertising service or can advertise it to different groups. The Service provider can do so by using either the `advertise` call or `advertiseInOtherGroups` method in `ESServiceElement.`

Consider an example where there are a number of printers organized according to administrative groups. Additionally, assume that each administrative group is associated with a different e-speak group. A user wanting to find printers across all administrative groups but still located closest to the user performs the following steps:

1 Create their own community that lists the groups of all the relevant administrative domains. Each group is identified by a string with the following format:

`<host_name>:<port_num>/<group_name>`

The host name and port number identify the host machine in which an advertising Service for the group of interest can be found. In addition, the name of the group of interest is specified as well. Note: The port is that of the connection factory and not of the Core. The advertising service should have been started with the same group name, with the command line option `-group <group-name>`.

```
ESCommunity newCommunity = new ESCommunity(coreConnection);
newCommunity.add("host1:22020/auction_site1");
newCommunity.add("host2:22021/auction_site2");
```

Another way to create a community is to specify it in the properties file used on start-up. The properties file takes in a **community** property that is a comma-delimited list of groups such as:

```
printcommunity.prop file:
…
community = host1:22020/auction_site1, host2:22021/auction_site2
…
```

2 Set the current community to this community name using the `ESServiceContext` class:

```
ESServiceContext  connectContext =
  coreConnection.getServiceContext();
connectContext.setCommunity(newCommunity);
```

3 Perform a search as follows:

```
ESServiceFinder anyFinder =
  new ESServiceFinder(coreConnection);
anyFinder.find("Name == 'AnyService'");
```

A mobile user can simply switch the community profile to find the appropriate Services in the currently relevant community.

In fact, because e-speak is geared toward Service deployment, it is natural to simplify the process of advertising a Service across a wide geographical area, simplify searching, and provide distributed access control.

To simplify the programming of distributed Services, the following support is built into J-ESI:

- Advertising a Service automatically makes the Service visible to the group in which it has been advertised.
- Because each Core has a registry, an e-speak deployment does not require a centralized naming or lookup Service to obtain information about all Services registered. A centralized scheme can be implemented as a special case within a more general scheme enabled by e-speak finder Services. The search for a Service is automatically performed in parallel across multiple registries.

Advertising across internet and locally

Advertising service across the Internet using HP hosted global service directory

Service providers can advertise the services throughout the world using HP hosted global service directory by selecting default values when starting the advertising service. In this case, when user invokes the advertise() call, the service gets advertised in the global service directory. This allows a service provider located in Los Angles to advertise a service to the global directory, accessable by any clients in New York. Clients and service providers can use the hosted gateway/connector service at eservicesvillage.com to access and provide services from behind a firewall.

For services hosted behind a firewall to advertise themselves at eservicesvillage.com (or any other service directory on the internet), the service provider has to set up the configuration files appropriately. The sections of the e-speak configuration file that pertain to web-proxy configuration and advertising service configuration may have to be modified in order for this to work correctly. The following is a sample section from an e-speak configuration file with the rules for web-proxy and advertising service configuration.

```
!-----------------------------------------------------------------
! Web proxy configuration
!-----------------------------------------------------------------
! webproxyname is a single value being the fully qualified hostname
! of the http-proxy used to traverse a firewall
net.espeak.infra.core.connector.webproxyname=web-proxy.efgijk.com

! webproxyport is the port on which the proxy listens
net.espeak.infra.core.connector.webproxyport=8088
```

```
! domain for which direct connection should be done.
! only one entry is supported currently
! Syntax is the end of the domain that should be match
! the '*' wildcard is not supported.
net.espeak.infra.core.connector.noproxydomain=efgijk.com


!------------------------------------------------------------------
! Advertising Proxy configuration
!------------------------------------------------------------------
!host at which the core for advertising proxy is running
net.espeak.services.advertise.esv_proxy_host=XXXXXXXX
(needs to be fixed*****************************************)

!port at which the core for advertising proxy is running
net.espeak.services.advertise.esv_proxy_port=23456

!default username and password to be used at the e-services village
net.espeak.services.advertise.esv_user_name=%esv_username%
net.espeak.services.advertise.esv_password=%esv_password%
```

If service providers who advertise their service have created an account at eservicesvillage.com, they can edit the esv_user_name and esv_password fields in the configuration file above to the appropriate values. This allows them to login to eservicesvillage and manage/edit their services.

Advertising a service within an enterprise

Service providers can advertise the services within an enterprise in two ways.

- Using a service directory (say LDAP)
- Using only the e-speak core repository

The first scenario is similar to using HP hosted global service directory except that the advertising service connects to the service directory specifed by the service provider. This service directory can be located within the enterprise spread in different locations. Specifically, the advertising service is started by specifying (`-beproto <protocol>`), (`-behost <hostname>`) and (`-beport <port-number>`) command line options. If (`-beproto`) command line option is not specified, then global service directory hosted by HP is selected.

In the second scenario, service providers who do not want to use service directory like LDAP can still achieve the same results by starting the advertising service in 'With repository mode'. When the user invokes the advertise() call, the service is placed in the local advertising service.

In this case, clients doing a search in a community, can specify fully qualified group names (group name+host name+port number). The infrastructure automatically connects and makes all the services available in the advertising service identified by host name and port number visible to the client, even when not using any service directory.

Advertising a service in local domain

Service providers advertising service in the local domain can do so in the following two ways.

- Using a service directory (say LDAP)
- Using only the e-speak core repository

The first scenario is same as described above.

In the second scenario, the users can use the spontaneous discovery mechanism in the e-speak system. Service provider's advertisements are automatically transfered to all the advertising services belonging to the same group.

Clients doing a search in a community, need to specify only the group names. The hostname and portnumber are no longer necessary in the local domain, as the advertising services spontaneously talk with each other in the local domain and exchange information.

Multiple groups in a single service directory

Multiple groups can be in a single service directory (say LDAP). In this case, different advertising services belonging to different groups can connect to the same service directory and advertise services.

Selecting a group name

Two different service providers can advertise services with exactly the same descriptions (attribute values). If the service provider wants to prevent collisions across the advertised services or wants to protect the access to the services advertised, the service provider can specify sufficiently unique group name for the advertising service. Specifically the service provider uses (`-group <groupname>`) command line option of advertising service to achieve this. It is the service provider's and client's responsibility to select a sufficiently unique name for the groups to prevent collisions.

A client doing a search can specify a community which is a collection of group names. In this case, only services registered in the those groups are returned to the client. Services in other groups, even if matching client's query are not returned.

A service provider wishing to advertise the service to the whole world can do so by starting the advertising service with group name `'speaktome'` and using the HP hosted global service directory. Specifically, the service provider starts the advertising service with command line option (`-group "speaktome"`)

Setting Current Community

The following two Application Programming Interfaces (APIs) in the `ESServiceContext` class are used to create the community in which a search or a registration of a Service is to be performed.

The `getCurrentCommunity` and `setCommunity` methods are as follows:

```
public ESCommunity getCurrentCommunity();
public void setCommunity(ESCommunity community);
```

ESCommunity

To add an entry to the list of groups, use the following code:

```
public void add(String groupName);
```

The default community is defined by the start-up file.

To remove a group from the community, use the following code:

```
public void  remove(String groupName);
```

NOTE: Note: This is included in future releases.

To return the enumeration of group servers, use the following method:

```
public Enumeration listGroupServers();
```

Example: `csldemo5.rgv.hp.com:22022/group`.

Advertising to Groups

As mentioned earlier, advertisement of Services is managed by `ESServiceElement`. An advertise call makes the Service visible to the entire group. To advertise a Service, use the following method:

```
public void advertise();
```

If a Service needs to be advertised to a different set of groups, use the following method:

```
public void advertiseToOtherGroups(String[] groupNames)
```

where each element in `groupNames` is in the form `<hostname>:portnumber/name`.

Finding in Communities

All the user needs to do to find a Service in a community is to set the current community to the one of interest and invoke the finder method to find the Service.

```
ESServiceContext connectContext =
   coreConnection.getServiceContext();
connectContext.setCommunity(owncommunity);
.....
finder.find(query);
.....
```

Security

The most basic notion in security is the notion of identity that is determined by its key-pair. In order to invoke operations on a security-enabled service, a client requires an appropriate certificate. This certificate must be signed either by the service provider's principal himself or by another principal who is linked by a chain of delegation to a principal listed in the trust assumptions of the service provider. With security enabled in the core, service providers and service client require certificates to enable them access to core apis. For instance, to register a service and to perform find operations. The principal representing the core issues certificates to both client and provider enabling such access.

When security is enabled, start up of any e-speak client (service client or service provider) reads two files: the trust assumptions and the certificates. Both these are actually just lists of certificates: the former being used by a service provider and the latter being used by a service client. In the base case, the service provider requires only one certificate: that for accessing the core. The service client, on the other hand, requires one for accessing the core and one or more for accessing the service itself. The client certificate list is merely a concatenation of these. An example of a certificate that the core can provide a service provider looks as follows, the binary data has been truncated for brevity.

```
(signed (cert
   (issuer (public-key elgamal-pkcs1
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\256\375\032\217:\351\024\327\
304\342\312B\'\311\016\007\304^\2052\322\271@\304`<\370\204\036j\220\030\2217aD*\
242\335|\233H\334N\201?.Uq\236)"))
   (subject (public-key "elgamal-pkcs1"
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\256\375\032\217:\351\024\327\
304\342\312B\'\311\016\007\304^\2052\322\316r+"))
   (propagate)
   (tag (net.espeak.method (*) (*) (*)))
   (not-before 2000-05-25_10:15:28)
   (not-after 3000-05-25_10:15:28)
   ) (signature (hash SHA-1
"\032\356V\317\260\t\347\331\277]\'\0278\237\0301t\212#\210") (public-key
elgamal-pkcs1
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\256\375\032\217:\351\024\327\
304\342\312B\'\311\016\007\304^\2052\201?.Uq\236)")
"\003\017`97\317`\377\'\303\347yC\337+\221$Sm\202\242\201\214w\004\352&\310]U-/\2
72\\342\301\330\034"))
```

A certificate that a service provider, a print service in this example, provides to a client that allows the client to invoke an operation, such as print, on it looks as follows (the binary data has been chopped):

```
(signed (cert
   (issuer (public-key elgamal-pkcs1
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\256\364p\255|\335"))
   (subject (public-key "elgamal-pkcs1"
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\256\311\301\273\307l\323\316r
+"))
   (propagate)
   (tag (net.espeak.method PrinterServiceIntf (* set print) (*)))
   (not-before 2000-05-25_10:15:28)
   (not-after 3000-05-25_10:15:28)
   ) (signature (hash SHA-1
"\0237\340}\'\310I\2638mq\207V\265\357\342\201\2702\274")
(public-key elgamal-pkcs1
"\003\017v\245\235b\004\345\211\225\021[\203=\256/K\246\374\032\020\244\004\004T\
307\221\037\247\034\332\365\3648~\300\274\362"))
```

When a message invoking an operation is received, J-ESI extracts the interface and method from it, and gets the service identifier from the information passed to the service handler by the core. From this it constructs the tag required to authorize the operation. The authorizer first checks to see if the tag is contained in the resource mask, and if it does, the operation is permitted. If the operation is not in the mask, J-ESI looks for a valid certificate (or certificates) that contain the tag needed to invoke the operation. The tag matching rules used for authorization are explained in the E-speak Architecture Specification chapter on “Access Control”. The certificates checked by J-ESI are those presented by the client to establish the session

J-ESI provides service providers with means to set metadata and resource masks that can mask access control to their services. See Appendix H for details on how to set up your security environments in e-speak.

The default behaviour when security is enabled is to require authorization for all operations. Service providers can create masks and associate them with service elements that they create. Masks enable Service providers to allow operations without any authorization. Any client is allowed to perform any operation that is specified in the mask for the service. If an operation is not included in a mask, only clients that have been given certificates authorizing access are allowed to invoke the operation. The security infrastructure examines the certificates that have been presented by the client to see if they contain a tag authorizing the operation. The rules for how tags authorize operations are explained in the E-speak Architecture Specification “Access Control” chapter.

Masks

There are two types of masks: the metadata mask for metadata operations and the resource mask for resource specific operations. For example, service providers can control who can mutate the metadata of the service that they have created using the metadata masks, and they control who can invoke operations on services provided by them using the resource specific masks.

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

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

The tag format is explained in detail in the E-speak Architecture Specification “Access Control” chapter. In the metadata mask, the interface name is the core interface being specified, and the method name is the operation in that interface. For metadata, the interface is likely to be ResourceManipulationInterface, and the method name one of its methods.

In the resource mask for a J-ESI service the interface name is the fully-qualified name of the interface class. 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, may 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 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.

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

```
    (*)
```

The full form of the method tag is actually:

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

In the normal case, the service handler is only interested in its own operations, so it does not care what the service field is. Since omitting a trailing field is equivalent to giving it the value (*), we omitted this detail above.

Authorizing Access

General tags can be constructed using the following method in ESSecurityEnv:

```
ADR createTag(String s) throws IOException
```

The IOException subclass net.espeak.security.adr.ADRParseException is thrown on a parse error. The parameter s is a string containing the input syntax for the tag.

Method tags can be created using

```
ADR createMethodTag(String interfaceName,
           String methodName,
           ADR service)
```

Clients can retrieve their current security environment from the connection. For example:

```
ESConnection conn= new ESConnection("config.file");
ESSecurityEnv secEnv = conn.getSecurityEnv();
```

For the purposes of resource masks, it is usual to use a tag containing simply (*) as the service parameter. In advanced applications, the service may want to set the service parameter to its service id, but this is not necessary.

After a mask tag has been constructed, it is used in ESServiceElement methods:

```
void setResourceMask(ADR tag) throws ESException
void setMetadataMask(ADR tag) throws ESException
```

Before a service is registered, these simply affect the local state. After registration, these set the local state and update the service metadata.

Masking can be turned on or off using ESAuthorizer:

```
void setMasking(Boolean x)
```

When masking is off, the resource mask is ignored by the service authorizer even if set. Setting masking off in the authorizer has no effect on the resource metadata, or the in-core metaresource handling metadata operations. Masking can be turned off completely, in the core and handler, by setting a mask to null.

An ESAuthorizer is associated with each ESServiceElement, and one can obtain the authorizer associated with an ESServiceElement using the getAuthorizer() call in ESServiceElement.

ESConnection has methods for controlling the default resource and metadata masks used when services are registered:

```
void setDefaultResourceMask(ADR mask)
ADR getDefaultResourceMask()
void setDefaultMetadataMask(ADR mask)
ADR getDefaultMetadataMask()
void setMasks(ADR metadataMask, ADR resourceMask)
```

After a default mask is set, all resources registered use it until it is changed. Unless the default masks are set explicitly, ESConnection uses null for them, causing authorization to be checked for all operations.

Example

In the example below, the service provider sets up a mask for the print method and a mask for the checkStatus method. Clients who present tags that match the print method are allowed to print and all clients are allowed to invoke the checkStatus method. The access control for the checkStatus method is disabled because of the setResourceMask method invocation in ESServiceElement.

```
public static void main(String[] args)
    {
        try {
       ESConnection    conn = new ESConnection("config.cfg");
       ESSecurityEnv sEnv = conn.getSecurityEnv();
       String m2 = "(net.espeak.method PrinterServiceIntf checkStatus)";
       ADR adr2 = sEnv.createTag(m2);
       ESServiceDescription sd = new ESServiceDescription();
       sd.addAttribute(ESConstants.SERVICE_NAME, "printer");
       sd.addAttribute("Description", "my hp printer");
       ESServiceElement se = new ESServiceElement(conn, sd);
       se.setResourceMask(adr2);
       ESAuthorizer esa = (ESAuthorizer) se.getAuthorizer();
       esa.setMasking(true);
       se.setImplementation(new PrinterServiceImpl());
       se.register();
       se.advertise();
       se.start();
    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

Clients who want to invoke this print service must obtain the tags to invoke this service and install them in their security environment.

Remote Connection Manager

Using advertising service, groups and communities is the prefered way of exchanging service metadata between various cores. However, sometimes more fine grained control is required where you want to make the service available to a specific core or engine. This can be done with the aid of remote connection manager and remote service manager.

Remote connection manager has set of simple methods that allows the application programmer to connect to the specified core, to disconnect from a core and to get a list of all connections that the local core maintains with other cores. To do this, first obtain the remote connection manager object from ESConnection.

```
ESConnection conn = new ESConnection("file.prop");
ESRemoteConnectionManager connMgr = conn.getConnectionManager();
```

To open a connection to a remote core, the client can invoke the openConnection() method on the connection manager. Typically, the client specifies the address and port number on the remote machine on which the remote core is running.

```
String url = "tcp:abc.foo.com:12345";
String id = connMgr.openConnection(url);
```

Clients can close existing connections with remote cores by invoking the closeConnection method in the connection manager. For example:

```
    connMgr.closeConnection(id);
```

where id is the string that represents the id of the connection that is returned by the openConnection() call. To get a list of connections that are currently open with the connection manager, use the following call.

```
    String[] ids = connMgr.getConnections();
```

Remote service manager is used to control transfer of service metadata from one core to another. This allows clients of one core to export their service descriptios to other cores on a selective basis. The remote service manager is obtained from the ESConnection by

```
ESConnection conn = new ESConnection("file.prop");
ESRemoteServiceManager servMgr = conn.getRemoteServiceManager();
```

Now, suppose the service provider has registered a series of services with the e-speak core and has the accessors for the service in an array of ESAccessors called accessors. Furthermore, suppose that the id of the core to which the service provider has opened a connection is id. Additionally, the exporter has to specify whether the export should occur by value or by reference, and in the case of accessors corresponding to folders, the exporter has to determine whether the export should recursively export sub folders of the current folder, or should it export only the current folder. The invocation of the exportResource method looks as follows:

```
servMgr.exportResource(accessors, id, exportType, level);
```

The exportType is an integer that is 1 if the services are to be exported by value and 2 if the services are to be exported by reference. In addition, set the boolean flag, level to false if the contents of a folder are to be recursively exported, and to true if only the top level of the folder is to be exported. A similar interface can be used to import resources.

```
servMgr.importResource(accessor, id, importType, level);
```

To unexport an exported resource from a remote core, the service can invoke:

```
servMgr.unexportResource(accessor, id);
```

To unimport an imported resource, the service provider can invoke:

```
servMgr.unimportResource(accessor, id);
```

Events

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

Event Model

E-speak supports an extended form of the familiar publish-subscribe Event Model. There are four logical entities in the e-speak Event Model whose interactions are shown in Figure 9. These entities are the *Publisher*, *Listener*, *Distributor*, and *Subscriber*.

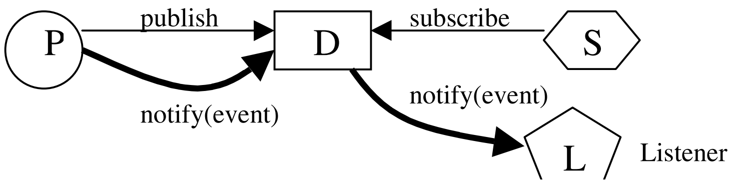


Figure 9 Interactions in the Event Model

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

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

Interaction Sequence

Figure 10 shows a typical Event notification process where the Subscriber and Listener are folded into a single Client.

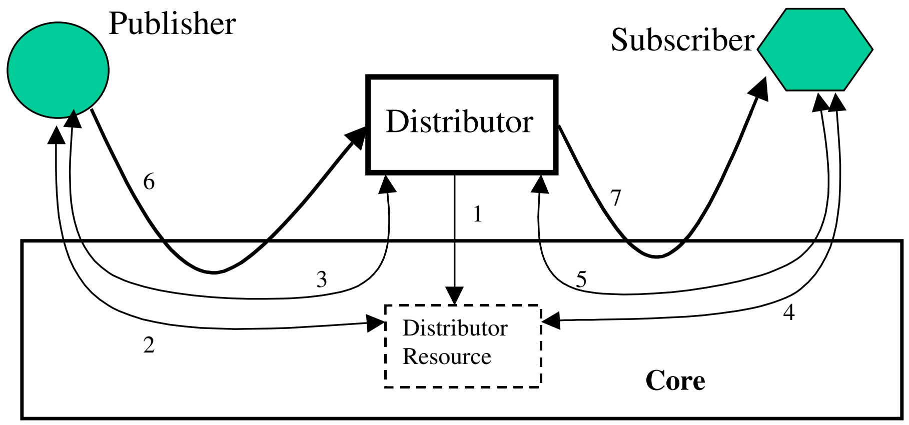


Figure 10 Typical Event notification process

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

1 Distributor registers with the Core.

2 Publisher discovers the Distributor.

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

4 Subscriber discovers the Distributor.

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

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

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

Subscribing to Events

The Event APIs provide simple mechanism by which Clients can express interest in various Events and handle them. The following code shows how a printer Client subscribes to outofpaper and paperjam Events and handle them subsequently.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
....
....
ESSubscriber printerEventSubscriber =
   new ESSubscriber(coreConnection);
printerEventSubscriber.addEvent
   ("hp.headoffice.firstfloor.printer.outofpaper");
printerEventSubscriber.addEvent(
   ("hp.headoffice.firstfloor.printer.paperjam");
printerEventSubscriber.setImplementation
   (new PrinterEventHandler());
ResultSet rs = printerEventSubscriber.subscribe();

public class PrintEventHandler implements ESListnerIntf
{
   ...
   ...
   public String notifySync(Event evt)
   {
      System.out.println(evt.getPayload());
      return "Notified" ;
   }

   public void notify(Event evt)
   {
      System.out.println(evt.getPayload());
   }
}
```

The event subscriber, after connecting to the e-speak core, creates an instance of `ESSubscriber` and expresses interest in certain event using `addEvent` call. Furthermore, the subscriber sets the handler for the Events in which it is interested using `setImplementation()`. The handler should implement the `ESListener-Intf`. Then the subscriber invokes `subscribe()` to register with the existing distributors in the community. The community is set using `setCommunity`() call in `ESServiceContext`. The `ResultSet` obtained as a result of `subscribe`() call contains success or failure of subscription with different distributors. The subscriber can subscribe to e-speak service events in a similar manner. The subscriber can also subscribe to e-speak core events using `ESCoreSubscriber` class. The list of e-speak service and core events are mentioned towards the end of this section.

The following code gives a simple example of how to subscribe to service events, for e.g., service.create. This subscribes the user to any service creation events in the community.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
....
....
ESSubscriber serviceCreateSubscriber =
   new ESSubscriber(coreConnection);
serviceCreateSubscriber.addEvent
   ("service.create");
serviceCreateSubscriber.setImplementation
   (new PrinterEventHandler());
ResultSet rs = serviceCreateSubscriber.subscribe();
```

Publishing Events

Publishing events is done using the `ESPublisher` class. The usage is similar to that of subscribing to events.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
....
....
ESPublisher printerEventPublisher = new
   ESPublisher(coreConnection);
printerEventPublisher.addEvent
   ("hp.headoffice.firstfloor.printer.outofpaper");
printerEventPublisher.addEvent(
   ("hp.headoffice.firstfloor.printer.paperjam");
printerEventPublisher.publish();
....
....
Event outofpaperEvent =
   new Event("hp.headoffice.firstfloor.printer.outofpaper");
outofpaperEvent.setPayload("printer foo out of A4 paper");
printerEventPublisher.sendNotify(outofpaperEvent);
```

Just as the subscriber expresses interest in receiving events, the publisher expresses interest in sending events. The publisher does this by instantiating the `ESPublisher` class, adding events of interest using `addEvent` and then calling `publish().` `publish`() does not send an event to the consumer, rather it just expresses intent to publish those events at a later point of time. Actual publishing of events takes place when the publisher calls `sendNotify()` after constructing an `Event` object. There is a default publisher available with the `ESConnection` that can be obtained using the `getDefaultPublisher()` call. This is used for publishing only the e-speak service events. The list of service events is given at the end of this section.

Distributing Events

So far we have talked about a simple subscriber and publisher assuming that there is already a distributor is available for the outofpaper and paperjam events. In case no such distributor is available, the publisher of the events can write a simple Event distributor as follows.

```
String propertyFileName = new String("/users/connection.prop");
ESConnection coreConnection = new ESConnection(propertyFileName);
....
....
ESDistributor printerEventDistributor = new
   ESDistributor(coreConnection);
printerEventDistributor.addEvent
   ("hp.headoffice.firstfloor.printer.outofpaper");
printerEventDistributor.addEvent(
   ("hp.headoffice.firstfloor.printer.paperjam");
printerEventDistributor.start();
....
....
printerEventDistributor.shutdown();
```

The distributor creates an instance of `ESDistributor` and adds events which the distributor intends to distribute using the `addEvent()` call. The distributor then starts using the `start()` call. The distributor can be stopped using `shutdown()` call. There are some pre-existing Event distributors bundled with the e-speak core. These are started along with the e-speak core. They are the service distributor and the core distributor. These distribute the service and core events listed at the end of this section. It is possible to subscribe to core events distributed by the `ESCore-Distributor`. It is possible also to subscribe and to publish service events that are distributed by the `ESServiceDistributor`.

List of Service Events

This section lists the Service Events generated by e-speak.

`service.create`	This Event is generated by e-speak Service interface when a Service is created.
`service.mutate`	This Event is generated when a Service's attributes are mutated.
`service.delete`	This Event is generated on deletion of a Service in the e-speak Service interface.
`service.access`	This Event is generated whenever the `ESAccessor` of a Service is used.
`service.pause`	A Service can voluntarily generate this Event for temporary pause of its Services.
`service.resume`	A Service can voluntarily generate this Event on resumption of its Services.
`service.genericinfo`	A Service sends out generic information about itself through this Event type.

List of Core-Generated Events

This section lists the Events generated by the e-speak Core.

`core.mutate.NameFrameInterface.3`	Bind a Resource to a new name in an existing Name Frame.
`core.mutate.NameFrameInterface.4`	Rebind an existing name in a Name Frame to a new Resource.
`core.mutate.NameFrameInterface.5`	Unbind a name from a Resource.
`core.mutate.NameFrameInterface.6`	Copy a binding from one Name Frame to another.
`core.mutate.NameFrameInterface.7`	Add a binding to a Resource to an existing name.

`core.mutate.NameFrameInterface.8`	Remove a binding to a Resource from an existing name.
`core.mutate.VocabularyInterface.3`	Modify the attribute set of a Vocabulary.
`core.mutate.ResourceFactoryInterface.1`	Register a new Resource.
`core.mutate.ImporterExporterInterface.1`	Import a new Resource.
`core.mutate.ImporterExporterInterface.5`	Update an imported Resource.
`core.mutate.ResourceManipulationInterface.1`	Unregister a Resource.
`core.mutate.ResourceManipulationInterface.3`	Modify the owner of a Resource.
`core.mutate.ResourceManipulationInterface.5`	Modify the handler of a Resource.
`core.mutate.ResourceManipulationInterface.9`	Modify a public RSD of the Resource.
`core.mutate.ResourceManipulationInterface.11`	Modify a private RSD of the Resource.
`core.mutate.ResourceManipulationInterface.13`	Modify the attributes of the Resource.
`core.mutate.ResourceManipulationInterface.23`	Modify the export-type of the Resource.
`core.failure.invalid_parameter`	Pass an invalid parameter.
`core.failure.null_parameter`	Pass a null parameter.
`core.failure.invalid_value`	Receive an invalid value.
`core.failure.invalid_type`	Receive an invalid type.
`core.failure.out_of_order`	Out of order.
`core.failure.Core_panic`	Core has an irretrievable exception.
`core.failure.service_panic`	Service has a critical exception.
`core.failure.recoverable_Core`	Received exception recoverable.
`core.failure.repository_full`	Repository is overflowing.
`core.failure.partial_status_update`	Status update is partial.
`core.failure.request_not_delivered`	Request not delivered.
`core.failure.permission_denied`	Permission denied for this operation.
`core.failure.undeliverable_request`	Request is not deliverable.

`core.failure.unrecoverable_delivery`	Unrecoverable delivery exception.
`core.failure.recoverable_delivery`	Recoverable delivery exception.
`core.failure.quota_exhausted`	Resource quota exhausted.
`core.failure.naming`	Naming exception received.
`core.failure.empty_mapping`	Mapping empty exception.
`core.failure.unresolved_binding`	Exception in binding resolution occurred.
`core.failure.multiple_resolved_binding`	Multiple resolved binding failures occurred.
`core.failure.name_not_found`	Name does not exist.
`core.failure.stale_entry_access`	Stale repository entry accessed.
`core.failure.name_collision`	Name collision exception occurred.
`core.failure.lookup_failed`	Lookup attempt failed.
`core.failure.service_failed`	Service failed.
`core.failure.nameframe_failed`	Name Frame failed.
`core.failure.invocation_failed`	Invocation failure occurred.
`core.failure.invalid_name`	Invalid name detected.
`core.failure.remote`	Remote failure occurred.
`core.failure.exporter`	Exporter failed.
`core.failure.exception`	All other exceptions.

Appendix A Thread-Safe Programming: ESThread

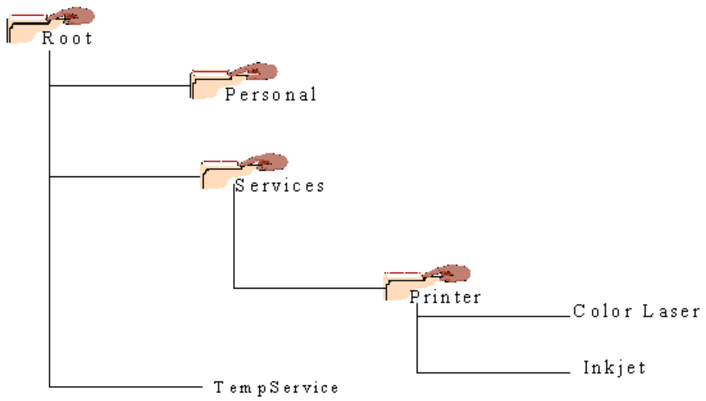
There are two aspects to threading when programming with J-SEI.

- The threads that the service handler uses to handle requests to the service
- The threads in the client program that may share a connection to the core.

When service providers create services, they create service handlers. These service handlers can be set up so that there are many threads that handle requests to this service. Essentially, each service handler comes with a ESThreadPoolManager that manages the pool of threads for servicing requests to this service. Furthermore, there are two main policies that can be used by the service provider:

- Thread per request: When the configuration indicates that the service provider wants to spawn off a thread per request, the ESThreadPoolManager creates a new thread to handle every request for the service. This thread runs to completion at the end of the request.
- Fixed thread pool: In this case, the thread pool manager has a fixed pool of threads that depends on the number of threads that the client has indicated in the ESServiceHandler. The client can set the number of threads she wants by using the setNumThreads(int num) in the ESServiceHandler.

Client applications in can be multithreaded. A Client can choose to create a connec-



tion to the Core and then create multiple threads (using the native Java threads) that use this connection simultaneously.

Because these threads share the same connection, any state stored in `ESConnection` is shared across all these threads. In certain situations, this is undesirable when the threads need to operate independently of each other. `EThread` is used in these situations.

An example of where this problem can occur is the use of the `setCurrentFolder(..)` API, to set the current folder in which new bindings are created. If multiple threads need to change the working directory without affecting other threads, then these threads should be created using an `ESThread()` call, which clones the `ESConnection` state on creation of the thread.

Just like Java Threads, there are two ways of using the J-ESI threads.

- Application developer creates a Java file which implements `ESRunnable` (counterpart of Java `Runnable`) and passes this to the constructor of `ESThread` (counterpart of Java `Thread`). The application developer does this if his Java file already extends some other Java class and hence could not extend `ESThread` directly. This is the suggested and preferred way of using ESThreads.

```
ESThread thread1 = new ESThread(esrunnable);
thread1.start();
```

Application developers can also extend ESThread directly. An example is as shown below.

```
public ApplThread extends ESThread
{
   public ApplThread(ESConnection coreConnection)
   {
      super(coreConnection);
   }

   public void run()
   {
         ESConnection clonedConnection =
            super.getConnection();
         ....
         ....
         clonedConnection.getServiceContext().
            setCurrentFolder(...);
         ....
         ....
   }
}
```

For example, the following code fragment creates two threads that operate on the same connection:

```
public class ThreadTest
{
   public static void main(String args[])
      throws    ESInvocationException, ESLibException,
      InterruptedException, IOException
   {
      ESConnection coreConnection = new ESConnection();
      ApplThread thread1 = new ApplThread(coreConnection);
      ApplThread thread2 = new ApplThread(coreConnection);
      thread1.start();
      thread2.start();
   }
}
```

Appendix B Messaging Classes

E-speak Clients discover Services and interoperate with them using a remote object model or using Events. In addition to these approaches, e-speak supports a purely messaging interaction model with the `ESServiceMessenger` class.

Clients send and receive messages using synchronous or asynchronous messages to other Services represented by the `ESAccessor` associated with the Service.

```
ESServiceMessenger messenger =
   new ESServiceMessenger (coreConnection);
```

The `messenger` can be used to send a payload object synchronously or asynchronously to any other Service. To send an object, use the following code:

```
...
Object payload = new Object();
ESService myService = finder.find(query);
ESAccessor serviceAccessor = ((ESAccessorHandle) myService).
   getAccessor();
Object retValue = messenger.sendSynchronous
   (serviceAccessor, (Object) payload);
```

or (to send asynchronously):

```
ESMessage msg = messenger.sendASynchronous
   (serviceAccessor, (Object) payload);
```

When an asynchronous message is sent, a message ID is returned in the form of an ESMessage. The service sending an asynchronous message may expect a notification to be returned. It can wait for a reply using the wait method as follows:

```
messenger.wait(msg);
```

where msg is the return value of the asynchronous send.

A Service expecting to receive messages can instruct the `ESServiceMessenger` to receive messages destined for a particular service element by using the following code:

```
ESMessage msg = messenger.receive(se);
```

where `se` is an `ESServiceElement` which has been initialized and registered.

On receipt of a message, a Service can extract its payload, operate on it. To extract the payload from the ESMessage, use the following method:

```
Object obj =  msg.getPayload();
```

On operating on the payload, the service can reply to a message by invoking the reply() method on it.

```
Object repObject = new Object();
msg.reply(repObject);
```

Appendix C IDL Compiler

The purpose of ESIDL compiler is to prepare the code that enables programmer of client code to invoke services as if they were running in the same process space. Such code includes stub, registering into message registry and serialization of objects.

ESIDL compiler can create new code and check existing code for conformance. Both E-Speak serialization and Java serialization are supported.

Generating code

ESIDL compiler uses as input description of services, types, and exceptions and creates support files needed by user to invoke service through e-speak.

There are two kinds of description:

- services are described in **service description files**
- types and exceptions are described in **type description files**.

There are several kinds of support files:

- **interface files** that define service interfaces
- **stub files** that represent service on client side
- **message registry files** that are used to register classes with e-speak messaging layer
- **class files** that implement data types and exceptions

Data type description can be in an ESIDL file (extension ".esidl") or in a Java file (extension ".java'). Service description can be only in an ESIDL file. ESIDL files are checked and new code is generated from them. If files of both types exist, then the ESIDL file is used as input.

Processing Java files is new to ESIDL compiler.

Input Files

Service Description

Service description includes information about an e-speak service. For ESIDL compiler every parameter it receives on the command line denotes an e-speak service. This is a change from the previous version where all service files and data types had to be listed. Interface file, stub file, and message registry file are generated for every e-speak service.

```
Service.esidl → ServiceIntf.java, ServiceStub.java,
ServiceMessageRegistry.java
```

Type Description

Type description includes information about:

- Types of service attributes
- types returned by service methods
- parameters to service methods
- exceptions thrown by service methods

Type description files are not listed as command line parameters to the compiler. They are searched for using import statements in the same manner as javac compiler searches for the source files (CLASSPATH environment variable or -classpath option).

```
Type.esidl → generated Type.java
Type.java → checked Type.java
```

Output Files

Interface File

Interface file includes interface that the service implements. The interface extends ESService.

Stub File

Stub file includes code for:

- serialization of service stub object
- rerouting of service method invocations through core
- calling messaging registry initialization (in message registry file) for return types, parameter types and exception types used in the service

Message Registry File

Message registry file includes code for registering types (for a list, see Type Description chapter) with MessageRegistry.

Type File

Type file is generated from type description. Serialization code is generated (for .esidl files) or checked (for .java files). E-speak serialization is default if none is specified (in declaration of type). Otherwise the specified serialization is used.

Command Line Parameters

Do not list all data type and exception files as the compiler parameters as it was required by the previous version of the compiler.

```
Java net.espeak.util.esidl.IDLCompiler [-classpath|cp
<path-to-sources>] [-verbose] <service-desc> [<service-desc> …]
```

where:

`<path-to-sources>`	Semicolon separated list of directories where compiler searches for description files. "." (dot) is used for current directory. If classpath is not specified, environment variable CLASSPATH is used.
`-verbose`	Instructs the compiler to print out information about created files.

IDL Requirements

Requirements have changed from the previous version and are now much less strict.

Description in ESIDL file

Service description is admissible if:

- it is declared `public` AND
- all methods it declares are admissible

A **user-defined abstract class** or a **user-defined exception** is admissible if:

- it is declared `public` AND
- its parent type is either `java.lang.Object` or another admissible user-defined data type

A **service method** is admissible if:

- return type and parameter types are admissible
- exceptions thrown are admissible

Description in Java file

A **user-defined data type** (a Java class) must conform to these additional restrictions:

- implements `ESSerializable` or `java.io.Serializable` interface

A **user-defined exception** (a Java class) must conform to these additional restrictions

- it must extend `ESServiceException`

A **service method** is admissible if it throws `ESInvocationException`

Appendix D Interceptors

J-ESI provides simple mechanisms for service providers and clients to monitor accesses to their services.Service providers can use these mechanisms to either generate management events such as billing events, create an access log, or provide mechanisms for performing load balancing, redirecting requests should a particular server be unavailable, etc. Clients can use interceptors for adding or removing parameters from requests or even implement a secure invocation interceptor that finds an available service with the required method and invoke it.

In essence, each ESServiceElement object contains an ESIceptorControl object that controls the list of interceptors associated with the service element. The interceptor objects are instances of classes that extend the abstract ESIceptor class. The ESIceptorControl object therefore provides methods provides methods to add and remove interceptors from the ESServiceElement. When a message arrives for the service represented by the ESServiceElement, the message is passed through each interceptor that the service provider has associated with the ESServiceElement.

Interceptors are classified based on whether they are terminal or not. A terminal interceptor is an instance of ESTerminalIceptor, and represents an actual invocation to the service. There is a single terminal Therefore, the ESIceptorControl that is associated with any ESServiceElement, comes with a default ESTerminalIceptor. However, the service provider can change that terminal interceptor with any other terminal interceptor that she writes.

The service provider must go through the following steps in order to set up an interceptor:

- Define an interceptor class that extends ESIceptor or ESTerminalIceptor. The service provider must implement the `invoke(ESRequest req)` method in the extended class. Furthermore, if the interceptor being defined is an extension of

ESIceptor, she has to make a `invokeNext()` call in the implementation of the invoke method. If this is not done, the interceptors that are added after this interceptor are not invoked.

- The service provider also must implement the `initialize(Object params)` method in the interceptor class. This method is invoked when the interceptor is added to the `ESIceptorControl` and can be used to initialize the state of the interceptor.
- When the service provider creates a `ESServiceElement` that represents the service, she makes an `addIceptor(ESIceptor icp, Object params)` that adds the interceptor to the interceptor control object associated with the service element.

The service provider is free to override other methods in EsIceptor, but the invoke, and initialize methods must be implemented.

We now present a simple example that shows using the interceptors. Consider the print service example from the previous sections. Suppose the service provider wants to:

- balance the load among multiple print service implementations
- generate information about each access to the print service

This is accomplished by the following interceptors: the LoadBalancingInterceptor, and the PrintingInterceptor

```
public class LoadBalancingInterceptor extends ESIceptor {
    // Array of backup services
    private ESService[] services;
    private ESService thisService;

    // If the service reaches this load backup services are invoked
    private int LIMIT_LOAD = 5;
    private String PRINT = "print";

public boolean invoke(ESRequest req){
    System.out.println("LoadBalancingInterceptor enter invoke");
    if(req.getMethodName().equals(PRINT)) {
         try{
        ESRequest request = new ESRequest();
        // check services load
        int load = ((PrintServiceIntf)thisService).getLoad();
        if (load > LIMIT_LOAD){
             // the load is above limit load, invoke backup service
```

```
            int minLoadService = getLowestLoadService();
        PrintServiceIntf ps = (PrintServiceIntf)services[minLoadService];
req.setReturnValue(ps.print((String)req.getParamValue("arg0")));
            invokeNext(req);
    }else{
         invokeNext(req);
    }
        }catch(Exception e){
        return false;
         }
    }else{
         try{
            invokeNext(req);
         }catch(Exception e){

         }
    }
    return true;
     }


public void initialize(Object param){
    // .. implementation that initializes the services array, etc.
    // from information in param
}

private int getLowestLoadService(){
        //… implementation
     }
}

//This is how the interceptor code looks like

public class PrintingInterceptor extends ESIceptor {
    int cnt=0;
public boolean invoke(ESRequest req) throws ESInvocationException{
        if(req.getMethodName().toString().equals("print")){
            cnt++;
            writeCountToLogFile();
        }
        try{
            invokeNext(req);
        }catch(Exception e){
        }
        return true;
    }
    private void writeCountToLogfile(){
        // implementation
}

    public void initialize (Object param){
    }
}
```

```
Now that we have defined the two interceptors, the print server, looks as follows:
public class PrintServer
{
   public static void main(String [] args)
   {
      try {
         ESConnection coreConnection = new ESConnection();
         ESServiceDescription printDescription =
         new EServiceDescription();
         printDescription.addAttribute(“Name”,”printer”);
         ESServiceElement printElement =
         new ESServiceElement(coreConnection, printDescription);
         printElement.setImplementation(new PrinterServiceImpl());
         printElement.register();
         printElement.addIceptor(new PrintingInterceptor(),”HP Lobby 49 U
printer”);
         LBIParams lbiParams = ..// initialize object expected by
         //initialize method of LoadBalancingInterceptor
         printElement.addIceptor(new LoadBalancingInterceptor(),
               lbiParams);
         printElement.start();
         System.out.println("Started printer Service ");
      }
      catch (Exception e){
      // handle the exception
      }
   }
}
```

In the above example, the printing interceptor is invoked before the loadbalancing interceptor on any request. If the service provider wants the order to be reversed, reverse the order of adding the interceptors to the service element.

Appendix E Account Manager

The Account Manager allows administrators to create and manage accounts in e-speak. This means that administrators can create accounts, grant or change permissions for accounts, and occasionally delete accounts. In the current release, the Account Manager cannot grant, change or revoke account permissions, but this will be added in a future release.

Programming Model

The Account Manager module provides three basic abstractions, the Account Manager, the Account Profile, and the Account Description. The Account Manager allows basic account management functions, including creating and deleting accounts, authenticating users, and retrieving lists of accounts. Each account created by the account manager has an account profile associated with it, which is required for authentication of the account. To change the description associated with the account profile, a profile description with a valid (non-null) vocabulary must be supplied, together with the attributes to be added to the description.

Profile Description

The Profile Description describes attributes of the Account Manager. Since the default Account Manager vocabulary does not define a set of attributes that are expected, the user must provide a vocabulary that describes the attributes which will be included in this profile description, so that it can be added to the existing description of the account profile. In the account manager module, this abstraction is represented by the ESProfileDescription class.

Account Profile

The Account Profile contains the authentication information for the account. It is initialized with just the account name and password phrase and can be modified by adding a Profile Description to it. In the account manager module, this abstraction is represented by the ESAccountProfile class.

Account Manager

The Account Manager allows the service administrator to perform administrative tasks such as creating and deleting accounts, retrieving lists of all accounts, and adding descriptions to existing accounts. The service administrator's main interaction to the Account Manager module is through the Account Manager abstraction itself. In the account manager module, this abstraction is represented by the ESAccountManager class.

In this release, there are two default accounts defined in e-speak's core Account Manager: the admin account, and the guest account. The username, passphrase pair for the admin account is "admin, admin", and the corresponding pair for the guest account is "guest, guest".

A Simple Example

Consider an example where a service administrator decides to create an account temporarily, change its description, and then delete the account. At present, only an administrator or someone presenting the credentials of the account itself can delete an account. In a future release, there will be functionality allowing the granting of permissions to new accounts, and it is probable that some new accounts will be granted permissions to delete at least some subset of the existing accounts.

First, to create the account, the service administrator should open a connection to an e-speak core. (The core is already running). After a connection has been established, its `getAccountManager()` method can be used to retrieve the active `ESAccountManager`. The `ESAccountManager` can now be used by the administrator to perform normal account maintenance activities.

```
ESConnection esconn = new ESConnection();
ESAccountManager acctMgr = esconn.getAccountManager();
```

Now that we have retrieved the ESAccountManager, the administrator can create an account based on an account name. First, an ESAccountProfile must be created using that account name. The newly created ESAccountProfile is passed in as the associated profile for the account to be created.

```
ESAccountProfile acctProf = new ESAccountProfile("testAccount");
String acct = acctMgr.createAccount(acctProf);
```

The account just created can be referred to by the `java.lang.String` returned from the creation. The administrator can now add a description to the account. To do this, he/she must first ensure that there is a valid vocabulary associated with it, then must create an ESProfileDescription and add attributes to it that describe this ESAccountProfile.

```
ESVocabularyDescription vocDesc = new ESVocabularyDescription();
vocDesc.addAttribute(ESConstants.SERVICE_NAME, "testVocab");
vocDesc.addStringProperty( "manufacturer" );
vocDesc.addStringProperty( "model" );
vocDesc.addStringProperty( "year" );
ESVocabularyElement vc = new ESVocabularyElement(esconn,
                                                 vocDesc );
ESProfileDescription desc = null;
ESAttribute [] atts = null;
ESVocabulary vocab = null;

try {
    vocab = vc.register();
} catch (NameCollisionException nce) {
    nce.printStackTrace();
} catch (ESLibException esle) {
    esle.printStackTrace();
} catch (ESInvocationException esie) {
    esie.printStackTrace();
}

desc = new ESProfileDescription(vocab);
desc.addAttribute("manufacturer", "Honda");
desc.addAttribute("model", "Civic");
desc.addAttribute("year", "2000");

try{
    acctMgr.addDescription(acctProf, acct, desc);
} catch (ESInvocationException esie) {
    esie.printStackTrace();
}
```

```
Now that there is an active account, with an account description, the properties
just described in the  ESProfileDescription can be used to find this account again.
To do this, an ESQuery is constructed, describing the value desired for the
attribute in question, and this query is used as the argument to a find() operation
in an ESServiceFinder. This retrieves the list of services (accounts, in this case)
that satisfy the requirements of the ESQuery, and then the service administrator
can switch to this account using the switchAccount() method of the active
ESConnection.
    ESServiceFinder finder = new ESServiceFinder(esconn);
    try {
        ESQuery query =
            new ESQuery(vocab,
                   "(manufacturer == 'Honda') or (year == '2000')");
        ESService[] serviceList = finder.find(query);

        ESAccessor accessor = ((ESAccessorHandle) serviceList[i]).
                                                getAccessor();
        esconn.switchAccount(accessor);
           esconn.init();
    } catch (LookupFailedException lfe) {
        lfe.printStackTrace();
    } catch (ESInvocationException esie) {
        esie.printStackTrace();
    }
```

The service administrator can also change to the account simply by creating a property file that describes the attribute and its value, and using that property file to create a new ESConnection. The following code snippet shows this method of changing to an account that has had an ESProfileDescription added to it.

```
    Properties prop = new Properties(System.getProperties);
    prop.put("manufacturer", "Honda");
    try {
        ESConnection testConn = new ESConnection(prop);
    } catch (ESInvocationException esie) {
        esie.printStackTrace();
    } catch (ESLibException esle) {
        esle.printStackTrace();
    }
```

Multiple queries can be included in the property file, one on each line, to create a complex constraint for the desired account.

If there is a need to retrieve the `ESAccountProfile` of an account, for example, when an account has been retrieved using one of the methods described above, the `getAccountProfile()` method of the `ESAccountManager` can be used.

```
    try {
        ESAccountProfile retProf = acctMgr.getAccountProfile(
                                          acctProf, acct);
    } catch (ESInvocationException esie) {
        esie.printStackTrace();
    }
```

A valid `ESAccountProfile` is required before an `ESAccountProfile` can be retrieved. This is to prevent unauthorized access to the profiles of accounts in e-speak. There are currently two valid `ESAccountProfile`s, the profile of the "admin" account, and that of the account itself. No other account profiles are authenticated to retrieve a third account profile.

The administrator can also set the `ESAccountProfile` for a given account, to allow, for example, changing of the password for the account. As in the previous example, a valid account profile is required in order to call the `setAccountProfile()` method on an account.

```
ESAccountProfile newProf = new ESAccountProfile(name, newPwd);
    try {
        ESAccountProfile retProf = acctMgr.setAccountProfile(
                                        acctProf, newProf);
    } catch (ESInvocationException esie) {
        esie.printStackTrace();
    }
```

The `getAllAccounts()` method can be used to retrieve a list of all the existing accounts. This list can, for example, be useful for checking whether a particular account name is already used. This method can be called with no authentication required for the caller.

```
    try {
        String[] accounts = acctMgr.getAllAccounts();
    } catch (ESInvocationException esie) {
        esie.printStackTrace();
    }
```

Finally, to delete an account, the caller must again provide a valid ESAccountProfile, as well as the name of the account to be deleted. The account is only deleted if the provided ESAccountProfile can be successfully authenticated.

```
    try {
        acctMgr.deleteAccount(acctProf, acct);
    } catch (ESInvocationException esie) {
            esie.printStackTrace();
    }
```

Appendix F E-speak Security

Introduction

This document describes the current basic setup for using e-speak securely. Security is subtle and it is dangerous to treat it as just another thing to “select the check box”.

This is the first release of e-speak with security and inevitably there are many practical issues relating to how e-speak enabled services and clients make effective use of security. These deployment issues will be discussed and decisions taken over the coming months. Hence, this initial release of secure e-speak is mostly concerned with basic deployment of the security architecture itself. After the architecture is deployed, the security is ready to be activated to enforce particular security properties as necessary.

There are three main sections to this appendix. The first deals with the basic aspects of the security model, and in particular PSE’s and certificates. The second section then discusses a bootstrap process used for testing purposes. The third and final section discusses security configuration files.

The Basic Security Model

A thumbnail sketch of the security model goes as follows:

Everyone (and everything) has a set of public/private keys. Entities are distributed and interact with one another by means of secure sessions using the SLS protocol—this includes firewall traversal technology. All entities can both *use* services offered by others and also *provide* services to others. This means that all parties in secure sessions have to be authenticated to each other. In particular, SLS secure sessions authenticate both parties involved by using challenge-response negotiations based on public-key cryptography.

Access control to services is done by exchanging digitally signed certificates as a part of the SLS protocol providing secure sessions. These certificates act like “tickets” that grant entities with authorisation to access and make use of services. Certificates are signed by issuing entities (or Principals) and are issued to subject principals who may use them. These certificates can also be chained together (using *delegation*) to give composite authorizations.

Refer to the E-speak Architecture Specification chapter on “Access Control” for further details concerning the security model.

PSE’s and Certificates

A Private Secure Environment (PSE) represents a keystore containing public/private key pairs. Each principal e-speak entity needs to have their own set of keys and thus needs to store them securely within a PSE. The PSE itself can be stored as a binary file in your local file system. This data is encrypted and a passphrase is required to lock/unlock the data it contains.

The PSE is responsible for generating it’s own key pairs—in particular, it has been designed so that private keys should never be exposed[1].

The other main function of a PSE involves validating and signing certificates. Validating a certificate involves checking the signature of the certificate using the issuers public-key (embedded within the certificate). Signing a certificate involves using a private key held within a PSE to create a digital signature, based upon a message digest of the certificate data.

PSE Manager

The PSE Manager is a GUI tool that supports these basic tasks:

1 Creating a new PSE and saving it as a binary file.
This involves selecting a passphrase that is used as an encryption/decryption key. It is important to keep this passphrase information secure—anyone capturing your PSE can *perfectly* masquarade as you and access everything that

1 There is currently a method that can expose private keys – but this is only temporarily present to accommodate a deprecated internal interface that soon it is unnecessary to support.

you can access. Also, losing or forgetting your passphrase means that you are unable to unlock or access your own PSE. For automatic operation, the PSE passphrase can be kept in a pass file stored on a floppy disk etc. There is a configuration option for this.

2 Creating new keys-pairs.
The PSE Manager can create new key-pairs, each of which are given a symbol label. These labels can then be used when constructing certificates. Note that this labels are refered to as roles in config.cfg and some of the security code.

3 Creating and editing certificates.
Attribute certificates typically contain information about the issuer, the subject, what is being authorised and the validity period. For convenience, the PSE's symbolic labels (or *roles*) for keys can be used to refer to known keys when constructing certificates—thus avoiding tedious and error-prone data entry of key information.

4 Validating and signing certificates as described above.

PSE data can be saved to binary files (using a passphrase for the encryption key) and certificates can be saved to text files etc.

For further information on the PSE Manager, refer to the PSE Manager user documentation.

Bootstrap process for testing

The bootstrap process *for testing purposes* is as below. When writing and deploying secure applications, refer to the J-ESI doumentation and the E-speak Architecture Specification.

1 Use the PSE Manager GUI tool to do the following:

   a Generate a keystore object (i.e. a Private Secure Environment) and is typically called `securestore.txt`. This is presently shared by all participants—the core, the client and the service. Therefore, this configuration is *not* distributed.

b Each participant has their own key-pairs. The current simple approach is to generate three different key-pairs, one for each participant, with the following labels : `client`, `core`, and `service`, all within the same PSE.

c Generate a basic attribute certificate, one for each pair of distinct participants (i.e. client as issuer, core as subject and so on for all distinct combinations) which gives each participant *arbitrary* permission to perform operations. The PSE Manager can be used to conveniently generate these attribute certificates—it has access to all the keys that were generated. The PSE labels associated with the key-pairs can be used to refer to the keys within the certificates for convenience.

d After it is generated, the important thing is that the certificate must be *issued*—this means it is *signed* by the Core itself. Thus, the certificate is issued by the Core, and having the Core again as its subject, with an all-powerful e-speak tag attribute:

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

Again, the PSE Manager can perform this function of signing these certificate by any one of the participants.

Significantly, the PSE Manager GUI tool is generally standalone and does not need any prior configuration, i.e., it does not require any configuration before it can be used.

2 To operate the core with security turned on, a security configuration file needs to be correctly loaded containing the appropriate attributes. The configuration file is more fully explained in the following sections. But a high-level snapshot goes as follows:

a The configuration file is like a Java properties file and is typically named `config.cfg`. It is searched for in the current directory, the user's home directory or on the Java CLASSPATH.

b A very simple config.cfg file is shown on the next page.

```
!=================================================================
! $Id: config.cfg,v 1.1.2.1.4.10 2000/05/16 07:17:42 ks Exp $
!=================================================================
! E-speak security properties file.
!=================================================================

!-----------------------------------------------------------------
! Security properties.
!-----------------------------------------------------------------
! Master flag controlling whether security is on or off.
net.espeak.security.activate=on

! Set a property prefix.
@prefix=net.espeak.security

! Default name of the keystore file
.pse.storefile=securestore.txt

! Gui mode runs a dialog for the passphrase.
!.pse.mode=gui
! Passphrase mode looks for the passphrase property.
.pse.mode  = passphrase
! Passfile mode looks for a file containing the passphrase property.
!.pse.mode = passfile

! Define the passphrase.
.pse.passphrase = default passphrase

! Define the default role (i.e. the default PSE key label).
!.pse.role = client
```

The following section discusses configuration files in far more detail.

Configuration files

The default configuration file is config.cfg. The file is looked for in the following places:

- config directory under e-speak home as defined by property 'espeak_home' directory specified by property 'net.espeak.util.config.file', or current directory (from system property 'user.dir') if the property is not set directory specified by 'user.home' system property as a system resource from the classpath Java system properties can be set on the java command line using the syntax -Dprop-erty=value.

- The name of the file to look for can be specified using the 'net.espeak.util.config.file' property. The file defined by the property 'net.espeak.util.config.master'is always loaded on top of all other files, if specified. The default for this property is null.
- All files found are loaded, in reverse order, with files found earlier being merged on top of properties from files found later. The format of the files is java properties file format, with the following additions.

```
@prefix=<prefix>
```

This sets a property prefix to apply to properties starting with a dot. For example:

```
@prefix=net.espeak.security
.pse.mode  = passphrase
```

results in `net.espeak.security.pse.passphrase` being set to ‘passphrase’. After it is set, a prefix remains in force until changed or set null.

```
@mode=<mode>
```

If the <mode> is “override” (default) the values found in this file is used and all previous values are ignored. After the config.cfg parser encounters a file with mode set to “override”, no more files are parsed. If the mode is “merge” config.cfg files are combined, if two files specify values for the same property, the value in the last file to be parsed is used..

The name of the configuration file to look for can be set using the system property

```
net.espeak.util.config.file
```

which has the default value `config.cfg`.

If the system property

```
net.espeak.util.config.master
```

is set, the file of that name is loaded on top of all other files found.

The configuration is got by calling

```
   ConfigIntf Config.getInstance()
```

which returns a reference to a static instance of the default configuration. Other files can be loaded directly if wanted, see util.Config for the API. Single property files can be loaded using ConfigProps.

Property file syntax

A java properties file contains property names and definitions. The name is separated from the definition by '='. Spaces before the property name and around the = are ignored. The value of the property extends to the end of line, and includes trailing spaces. Long property values can be broken across lines using \ to escape new lines. The characters ! and # introduce end-of line comments. The character : may be used as an alternative to =.

Property conversion

The class util.Convert provides methods to convert property strings to and from common types. The types int, boolean, and long are supported. The duration converters accept times in the format 12h3m1.001s and convert them to longs in milliseconds. Any zero component of a time can be omitted, and spaces may be included. A zero time can be stated as 0s.

The boolean converter accepts on, true, yes for true and off, false, no for false, regardless of case.

Argument specifications

The mapping or argument switches to properties can be defined using util.ArgSpec. This provides methods to process command-line arguments and map them onto properties in a configuration.

Security properties

The following are the properties supported by the security code.

- Master flag controlling security: `net.espeak.security.activate`, boolean, default off. If this property converts to true, security is activated.

- Property `net.espeak.security.connectOnContact`, default off, controls whether secure sessions are established with newly encountered resources. When it is off, sessions are not established unless required (by SessionRequiredException) or created explicitly.
- PSE mode: `net.espeak.security.pse.mode`. Values: gui, passphrase, passfile. Default gui. If the mode is gui, a dialog is used to get the PSE passphrase. If the mode is passphrase the property `net.espeak.security.pse.passphrase` is used to get the passphrase (default null). If the mode is passfile the property `net.espeak.security.pse.passfile` (default passfile.txt) is used to get the name of a file which must contain a `net.espeak.security.pse.passphrase` property defining the passphrase.
- PSE key file: `net.espeak.security.pse.storefile`, default securestore.txt. Defines the name of the file containing public-private key pairs.
- PSE role: `net.espeak.security.pse.role`, default client. Define the default role (symbolic PSE key name).
- PSE file protection mode: `net.espeak.security.pse.OSfileprotection`, default true. This property specifies whether local OS file protection should be applied and is supplied purely as an aid for testing purposes. For full security protection, this option should not be false.
- Certificate file suffix: `net.espeak.security.pse.certfile`, default certs.adr. The value of this property is appended to the role name to get the name of the certificate file to load. If the role is client the certificate file is clientcerts.adr for example.
- ACL file suffix: `net.espeak.security.pse.aclfile`, default acl.adr. The value of this property is appended to the role name to get the name of the ACL file (trust assumptions) to load. If the role is client the ACL file is clientacl.adr for example.
- Cipher suites: `net.espeak.security.cipherSuites`. The value is a list of cipher suites in ADR syntax. The default is to use hmac, sha-1, and 128-bit blowfish.

- Public key: net.espeak.security.pse.publicKeyAlgorithm using the values ELGAMAL and RSA. The default is ELGAMAL. Which one is being used can be found using net.espeak.security.util.PublicKeyLib. Call public static `String getPublicKeyAlgorithm()` to find out.

We support two public key algorithms: RSA and El Gamal. The entire system must use one or the other, mixing is not supported. Public key classes are not loaded statically—they are loaded dynamically based on the configured algorithm.

When El Gamal is configured, support for RSA is not be loaded, and if RSA data is encountered a NoSuchAlgorithmException is thrown. Which algorithm is used is defined by the property.

Example config.cfg file

```
!================================================================
! E-speak security properties file.
!================================================================
! Master flag controlling security.

net.espeak.security.activate=on
user.name="John Doe"


! Example time value.

foo.timeout = 12h 3m .0001s
! Set a property prefix.

@prefix=net.espeak.security
! Gui mode runs a dialog for the passphrase.

!.pse.mode=gui
! Passphrase mode looks for the passphrase property.

.pse.mode  = passphrase
! Passfile mode looks for a file containing the passphrase property.

!.pse.mode = passfile
! Define the passphrase.

.pse.passphrase = default passphrase
! Define the default role (PSE key name).
!.pse.role = foo
```

Appendix G Firewall Traversal

This chapter discusses firewall traversal using e-speak. We present how services can be accessed using e-speak while sitting behind a firewall. The proposed solution does not require modification of the existing security infrastructure.This offers a fast deployment but does not provide maximum security at the boundary.

Architecture

The architecture of the security system is presented in Figure 11. This architecture is very classic, the service is connected to the Engine Inside. The Engine Outside act as a proxy in the DMZ for the Engine Inside. Connections are established to the Engine Outside and then messages are routed to the Engine Inside.

E-speak offers end-to-end security so the number of engines relaying messages is irrelevant. The security (Confidentiality, Integrity and Authentication) is established between the services and the clients.

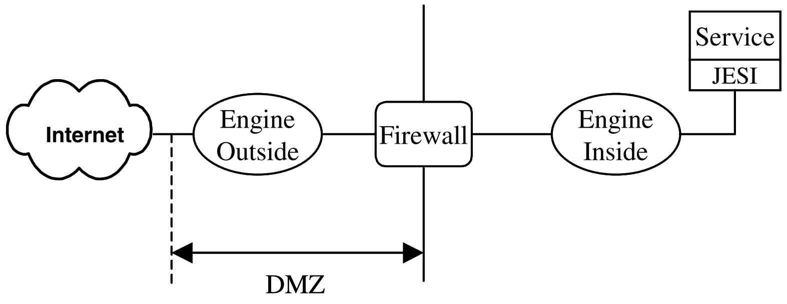


Figure 11 Security Architecture

When the service connects to the Engine Inside, it registers its metadata with the engine and establishes an outbound connection to the Engine Outside and exports this metadata.

Firewalls usually offer means to establish outbound connections. HTTP proxies are one of them. They are widely deployed and offer through the HTTP Connect method a way to establish TCP connection to external systems. SOCKS V4 servers offer the same functionality but support stronger authentication mechanism.

In order to achieve fast deployment of services with minimal architectural changes on the boundary, e-speak can be configured to use HTTP proxies or SOCKS servers.

The requirements on the service provider side are as follows:

- An HTTP proxy or SOCKS server must be present in the firewall.
- A system in the DMZ must host an e-speak engine (Engine Outside).

The requirements on the client side are as follows:

- An HTTP proxy or SOCKS server must be present in the firewall.

The HTTP proxy approach is possible only if the proxy allows for the HTTP Connect method on the Engine Outside port.

These approaches uses either SOCKS V4 or HTTP Proxies to allow inbound connection. SOCKS V4 and HTTP Proxies have been created explicitly for outbound connections. Therefore these approaches have some drawbacks. Firstly, the service is available from the outside as long as the connection to the Engine Outside is up. This requires the connections to be long-lasting. Secondly, there is no control at the boundary because the Engine Outside is not capable of such functionality. The authentication and authorization is done at the service end-point, this means that unauthorized messages can potentially be sent to the inside services.

Deployment using HTTP proxies

Figure 12 and Figure 13 presents classic deployment configuration using e-speak.

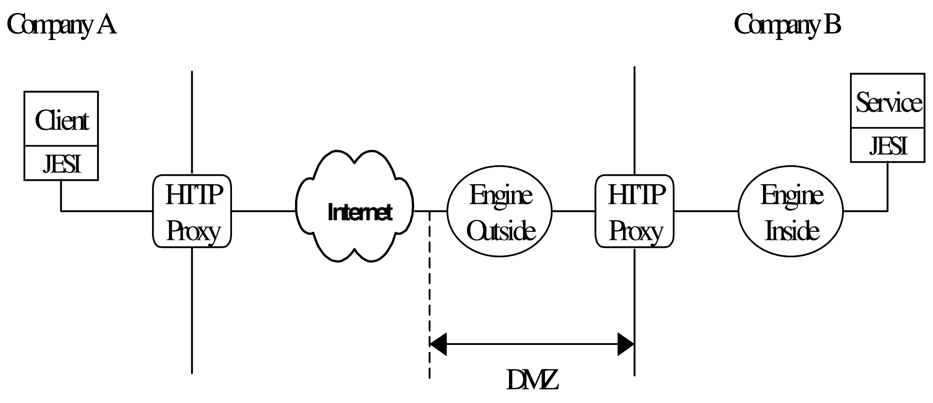


Figure 12 Client connecting directly to the Engine Outside

In this scenario, the client connects directly to the Engine Outside through J-ESI.

Follow these steps:

1. Setup the client espeak.cfg file with the following properties:
   net.espeak.infra.cci.messaging.webproxyname=<proxy name>
   net.espeak.infra.cci.messaging.webproxyport=<proxy port>
   This explicitly tells the client to open a connection through the web proxy.
2. Setup the Engine Inside espeak.cfg file with the following properties:
   net.espeak.infra.core.connector.webproxyname=<proxy name>
   net.espeak.infra.core.connector.webproxyport=<proxy port>
3. The service needs to explicitly export itself to the Engine Outside using its RemoteServiceManager.

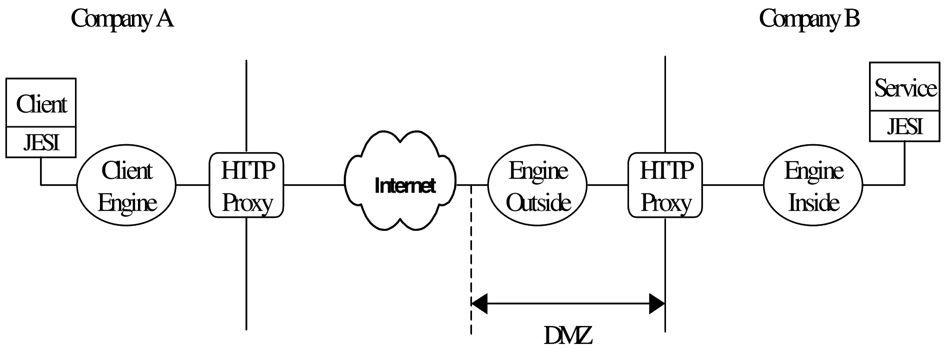


Figure 13 Client connecting to the Engine Outside using a local engine

In this scenario, the client connects to a local engine, which in turn connects to the Engine Outside.

Follow these steps:

1 Setup the Client Engine espeak.cfg file with the following properties:
net.espeak.infra.core.connector.webproxyname=<proxy name>
net.espeak.infra.core.connector.webproxyport=<proxy port>

2 Setup the Engine Inside espeak.cfg file with the following properties:
net.espeak.infra.core.connector.webproxyname=<proxy name>
net.espeak.infra.core.connector.webproxyport=<proxy port>

3 The service explicitly exports itself to the Engine Outside using its RemoteServiceManager.

4 The client explicitly imports the service from the Engine outside using its RemoteServiceManager.

NOTE: As explicit import and export are awkward, another alternative is to run an advertising service in the DMZ attached to the Engine Outside and use it to do the import and export implicitly. With this modification, the above scenario changes as follows:

1 setup the Client Engine espeak.cfg file as above.

2 setup the Engine Inside espeak.cfg file as above.

3 The service is advertised in the advertising service in the DMZ. This leads to an implicit export from the Engine Inside to the Engine Outside.

4 The client finds the service by looking up in the DMZ advertising service. This leads to an implicit import from the Engine Outside to the Client Engine.

The scenario in Figure 13 can be modified similarly.

Deployment using SOCKS Server

The SOCKS protocol is supported internally by the VMs. No specific configuration is needed except for the VM configuration itself.

Figure 14 shows one scenario where the SOCKS servers are used by the client and the service provider.

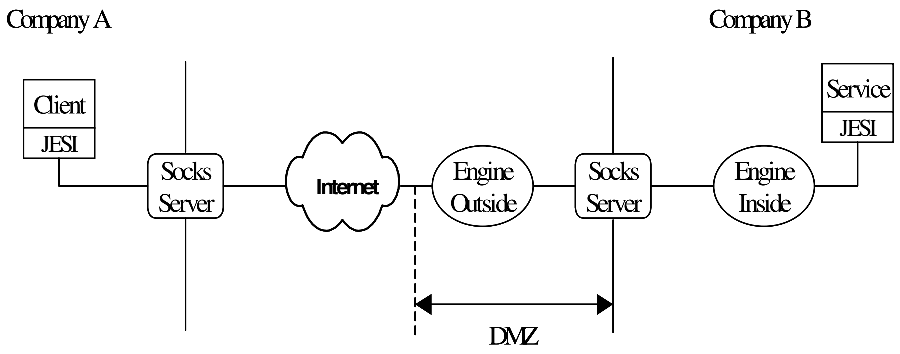


Figure 14 Using SOCKS servers

Connector

In some deployment cases, it is possible that installing the Engine Outside in the DMZ is perceived as too complicated or impossible.

E-speak introduces a new way for service providers to reach service consumers. The connector is the central point of the system. The connector is the Engine Outside moved in a DMZ of a trusted party.

Figure 15 describe the deployment scenario for the connector.

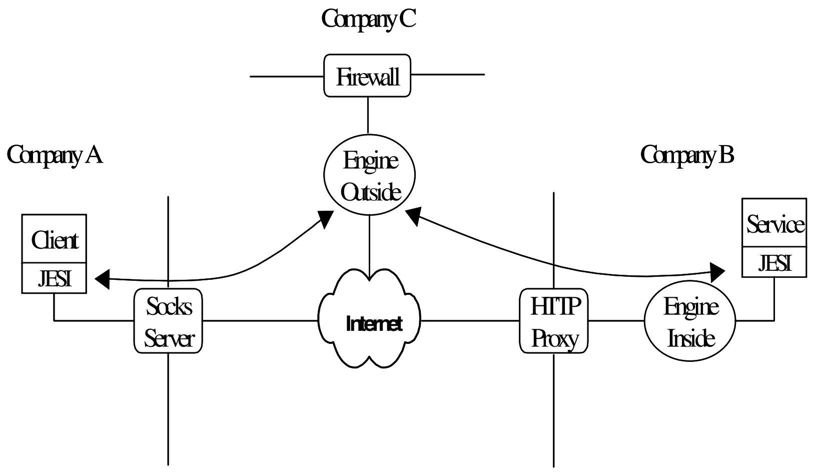


Figure 15 Deployment of a connector in a trusted party's DMZ

Company C is a trusted by Company B to host its services' metadata. When Company B wants to expose its services to the outside, it connects the Engine Inside to the Engine Outside (Connector) and exports the description of the available services.

When the clients wants to access a service, its connect to the connector and search for the given service and interact with it transparently.

Clearly Company C needs to be trusted by the service provider because it exposes the service to the outside world.

An example of the connector is the e-service village where service providers and services consumers can meet and establish relationship.

Appendix H Management

Currently system management in e-speak is undergoing a transition from a traditional network object model to a document exchange model. Infrastructure and support for this model is still being developed and collaboration between parties on common schemas and mechanisms is in the early phases. This document therefore is very much work in progress and will remain as such until a variety of inter-related elements within e-speak become available and reach stability.

Introduction

This appendix describes the infrastructure and general philosophy of how web based management tools and an XML management interaction framework are provided in e-speak.

System Management in e-speak currently consists of four things:

- A managed service model.
- XML Schemas and dialogs that enable management.
- Client support for making services manageable.
- A set of tools that provide access to the management services from the web.

This document describes the first, third and fourth of the above with code examples.

Managed Service Model

The managed service model is simply two concepts that underpin the manageability of e-speak services 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.

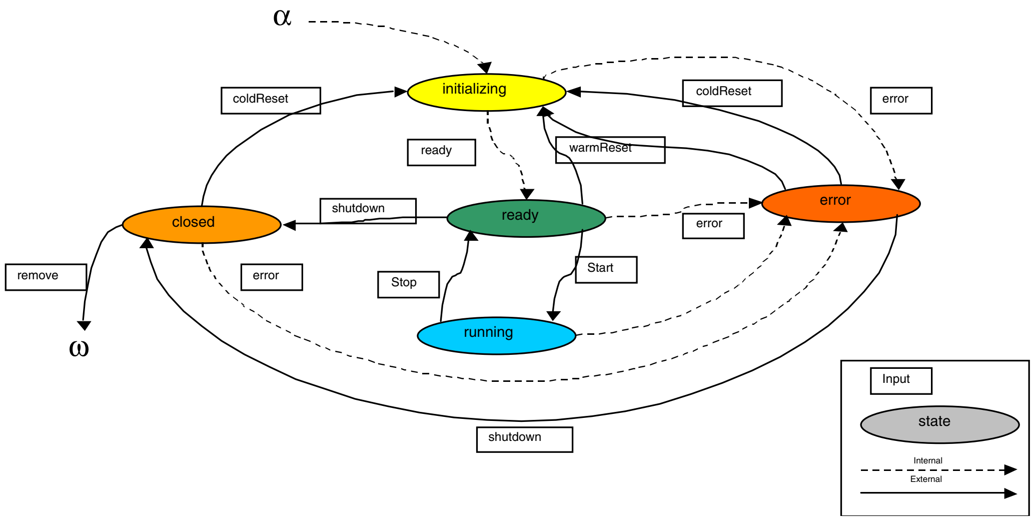
For this generic model to be useful, a client must map its service specific behavior onto this model and expose this model to a management agent. As we see later, the life cycle in the managed service model has many states, including *initializing*, *ready*, and *running*. It is entirely up to the service writer how these states are related to the service specific behavior.

For example a service might have no need of an initialization phase but must instantaneously pass this state to conform to the model. A service may also be in various operational states while running, but *running* is the only way to express its condition from a management point of view.

In summary then, the managed service model embodies a view of service behavior that is potentially common to many services. Therefore some custom management agents must deal with issues of service behavior that fall outside this model.

Managed Life Cycle

The full state transition diagram is as follows:



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 that implies that the service cannot continue to run, it should move into the error state.

Error: The service has a 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—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) but 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 can reasonably request that a client perform a warm reset, but not to become ready, the client alone can provide this input— 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—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.

The most simple usage case for a managed variable would be for a variable which the management agent has only read access but which the client varies as necessary. The manager can monitor the changes (see events later) such that an operator who understands the meaning of this variable (e.g. `secondsToDetonation`) might gain some information.

The next level of sophistication is to enable the operator to affect the services behavior by modifying the value of the variable. However, some service configuration is only of practical use during initialization (e.g. some resource allocation parameter). Knowing the instantaneous value of such a variable does not explain the clients behavior (if it had been changed) and modifying it has no effect.

To account for this, a managed variable can have one or more values. The compulsory value is its “live” value i.e. the value that is contributing to its current behavior. In the simplest case this is the only value that a variable has.

However, a variable can have other values, which are in effect “scheduled” values and associated with the various reset operations in the client's life cycle.

For example, consider the following managed variable found in some unreasonably hazardous device:

Variable Name	Values		
	Value Condition	Remote Access	Value
secondsToDetonation	Live Value	Read Only	42
	On Cold Reset	Read Only	60
	On Warm Reset	Read/Write	60

Let's assume it takes some operator three minutes to reach safety or the devices off switch. The operator brings up his management console and notices that the current value of secondsToDetonation is 42, and dropping. Clearly the circumstances are undesirable: the current value does not provide long enough to escape without injury. Increasing this value to three minutes or more and running away is ideal but the value is read only.

Glancing down the list the operator notices that there are two reset values and performs a cold reset. After the device has re-booted, and re-loaded the counters default initial value (perhaps in ROM) it promptly starts counting down from 60 seconds. Still not good but at least the operator can try and stay up all night performing cold resets every 59 seconds or less.

Having bought some time the operator looks a little more closely at the variable. The warm reset value is writable. Now the operator can set this to thirty minutes and go and disconnect the power supply.

Clearly a contrived example but it does demonstrate how relatively complex interactions between initialization, configuration value and behavior can be represented.

A more likely scenario is where a client has a set of variables that configure its behavior that must all be changed synchronously. A management agent can modify its warm reset values at its leisure (being denied access to the live values) and then perform a warm reset.

There is a restriction on variable table usage:

Uniqueness: names 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 URL of the actual service.

System Management Events

Events can be considered as notifications of some significant occurrence. In the context of system management, events must exist to convey notification of some change of state with regards the Managed Service Model. The following events are therefore defined:

ManagedServiceStateChange: This event conveys information regarding some managed service state transition to any interested management agent. In this case, the event is the name of the input provided to the management life cycle FSM. Clearly, the management agent must have known what the previous state was in order that this information be meaningful.

ManagedVariableValueChange: This event conveys information regarding a value modification in a clients configuration or resource table.

Managed Service Programmers Guide

Writing a Simple Managed Service

In this section we create a simple managed service[1].

First, to get familiar with the tools we look at the most minimal Java program that is "manageable" through e-speak.

A Minimal Service

Consider the following program:

```
import
net.espeak.management.managedservice.simplemanagedservice.SimpleXMLManagedService
;
import net.espeak.infra.cci.exception.ESException;
import net.espeak.jesi.management.ServiceContext;

public class VerySimpleExample extends SimpleXMLManagedService {
    public static final String SERVICE_NAME = "VerySimpleService";
    public static final String LOCALHOST = "127.0.0.1";
    public static final int COREPORT = 12346;

    public VerySimpleExample() throws ESException {
        super(new ServiceContext(LOCALHOST, COREPORT, "tcp"), SERVICE_NAME);
        makeManageable();
    }

    protected void stateChangeOccurred(String transition, String oldState,
        String newState) {}

    protected void resourceChangeOccurred(String resource, String oldValue,
        String newValue) {}

    protected void variableChangeOccurred(String variable, String oldValue,
        String newValue) {}
```

1 For reference, a fully working example of a managed service is supplied in: net.espeak.management.managedservice.simplemanagedservice.ExampleService

```
    protected boolean acceptStateChange(String transition, String oldState,
        String newState) {
        return true;
    }

    protected boolean acceptVariableChange(String variable, String oldValue,
        String newValue) {
        return true;
    }


    protected boolean acceptResourceChange(String resource, String oldValue,
        String newValue) {
        return true;
    }

    public static void main(String[] args) {
        try {
            new VerySimpleExample();
        } catch (ESException e) {
            e.printStackTrace();
        }
    }
}
```

VerySimpleExample (above) subclasses SimpleXMLManagedService, a utility class that in most circumstances provides adequate manageability support. We consider more sophisticated examples later.

For now let’s just consider the constructor:

```
public VerySimpleExample() throws ESException {
          super(new ServiceContext(LOCALHOST, COREPORT, "tcp"),
SERVICE_NAME);
          makeManageable ();
     }
```

First, the super class is initialized with a connection to the local core[2], second the **makeManageable()** method makes the service visible to any management agents.

2 For simplicity, the host and port are hardwired into the example and assume the core is running on the local host and uses port 12346, which is the value in the default version of xmlmanagement.ini. These values can be changed but the host must be the host name/IP address of a host running a core and the port number must match that used by the core.

Managing the Service

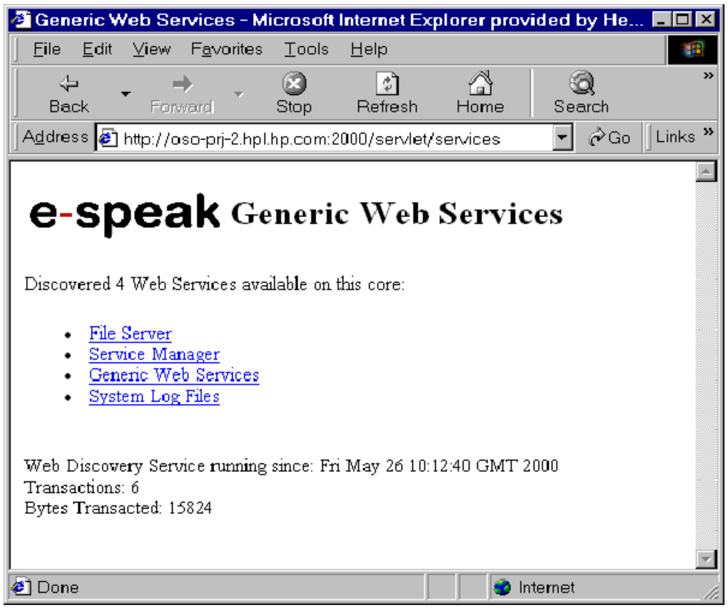
We are now ready to try the program out, but first we must have a core running. For convenience, there is a configuration file which starts all the services required for management. Go to your e-speak installation directory and type:

```
espeak -i config\management\xmlmanagement.ini
```

After a few moments, run a web browser and go to:

http://127.0.0.1:2000/servlet/services

You see a screen similar to the following:


Generic Web Services - Microsoft Internet Explorer provided by He...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address http://oso-prj-2.hpl.hp.com:2000/servlet/services
Go Links
Done
Internet

e-speak Generic Web Services

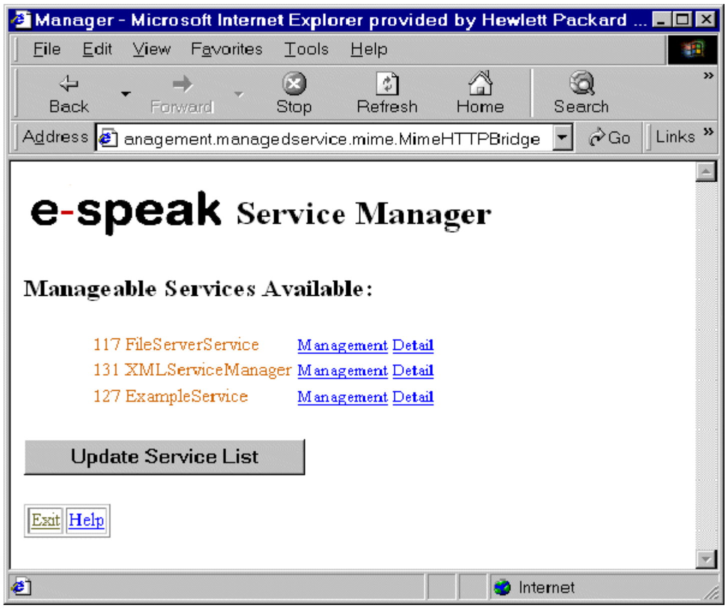
Discovered 4 Web Services available on this core:

- File Server
- Service Manager
- Generic Web Services
- System Log Files

Web Discovery Service running since: Fri May 26 10:12:40 GMT 2000
Transactions: 6
Bytes Transacted: 15824

This is a list of the visible “web enabled” services on your local core, in this case the services started by xmlmanagement.ini.

Click the Service Manager link and you see the following page:


Manager - Microsoft Internet Explorer provided by Hewlett Packard ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address anagement.managedservice.mime.MimeHTTPBridge
Go Links
Internet

e-speak Service Manager

Manageable Services Available:

117 FileServerService	Management Detail
131 XMLServiceManager	Management Detail
127 ExampleService	Management Detail

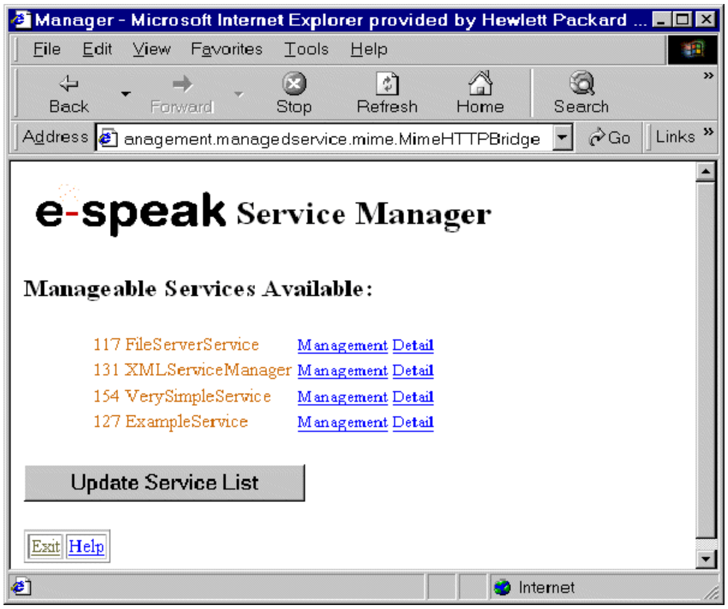
Update Service List

Exit Help

This is the main service manager screen and displays a list of all the manageable services to be found on your local core.

Note that while we call this the Service Manager, it should be remembered that it is clearly a web based interface onto the actual Service Manager. While the service manager is aware of the status of services in real time, the browser is not; so you should use the “Update” buttons provided to keep the display current. For related reasons, navigate between Service Manager screens using the links provided and avoid the use of the back button, particularly if you actively interacting with the services through the service manager.

Now we are ready to run **VerySimpleExample**. Compile the program and run it. After a few moments go back to the service manager screen and click the Update button. Now you see your program available for management:

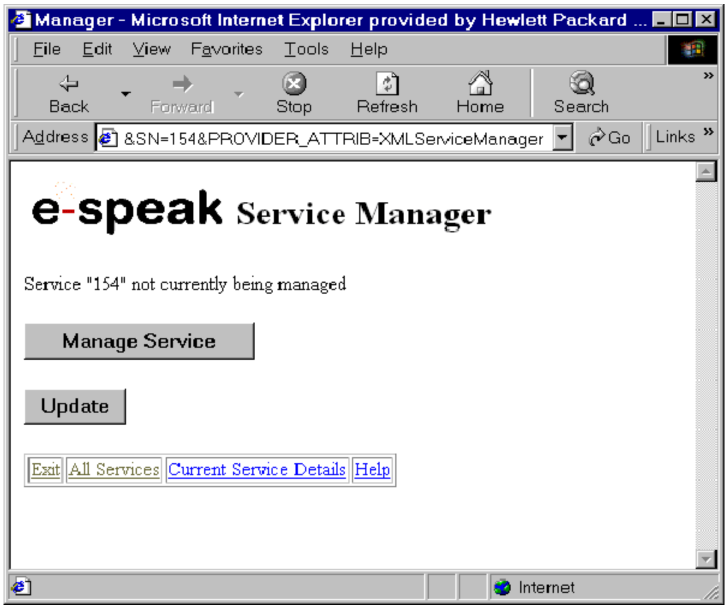

Manager - Microsoft Internet Explorer provided by Hewlett Packard ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address anagement.managedservice.mime.MimeHTTPBridge Go Links
Update Service List
Exit Help
Internet

e-speak Service Manager

Manageable Services Available:

117 FileServerService Management Detail
131 XMLServiceManager Management Detail
154 VerySimpleService Management Detail
127 ExampleService Management Detail

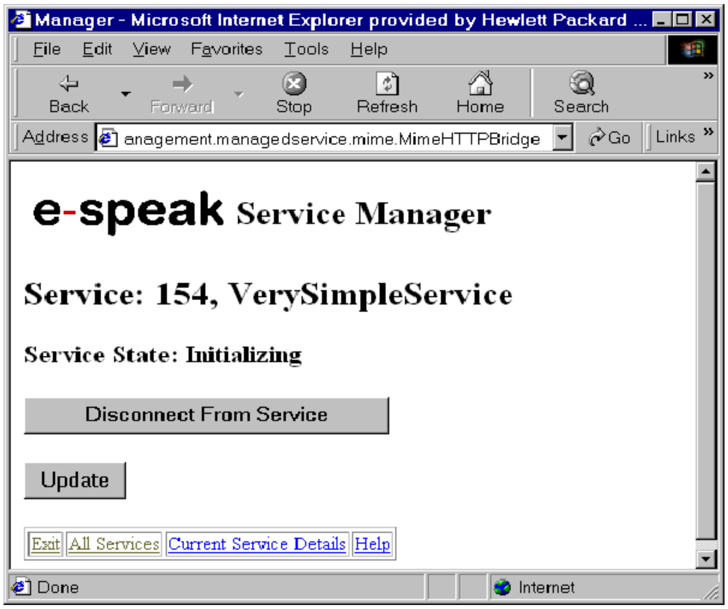
Click the Management link next to **VerySimpleService**:


Manager - Microsoft Internet Explorer provided by Hewlett Packard ...
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Address
&SN=154&PROVIDER_ATTRIB=XMLServiceManager
Go
Links
Manage Service
Update
Exit
All Services
Current Service Details
Help
Internet

e-speak Service Manager

Service "154" not currently being managed

This is the management page for your service. At the moment, your service is not being managed so click the Manage Service button:


Manager - Microsoft Internet Explorer provided by Hewlett Packard ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address anagement.managedservice.mime.MimeHTTPBridge Go Links
Done Internet

e-speak Service Manager

Service: 154, VerySimpleService

Service State: Initializing

Disconnect From Service
Update
Exit All Services Current Service Details Help

So what does this mean? Earlier in the document we discussed the concept of the Managed Life Cycle of a service. If you look back at the state diagram, you see that the start state was *initializing*. Because we have not changed service state in our program, we remain in this state forever.

Changing Service State

Now let's try something slightly more interesting. Consider this next program, very similar to the last:

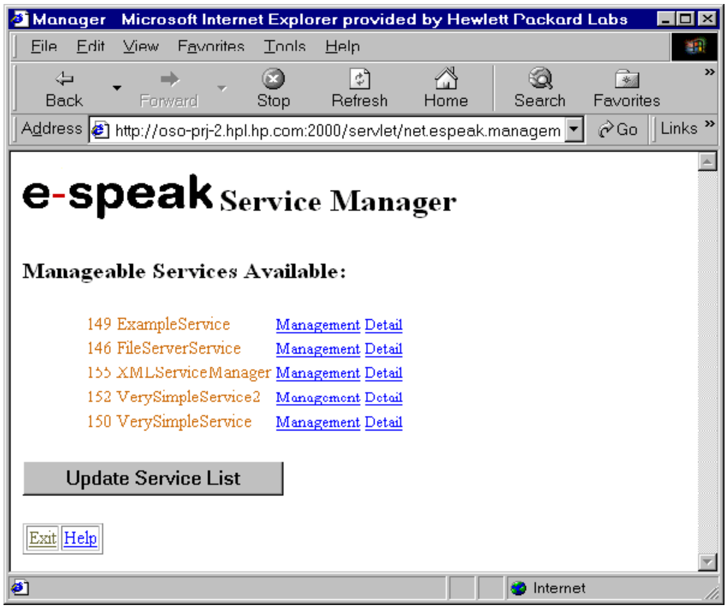
```
import
net.espeak.management.managedservice.simplemanagedservice.SimpleXMLManagedService
;
import net.espeak.infra.cci.exception.ESException;
import net.espeak.jesi.management.ServiceContext;
import
net.espeak.management.managedservice.managedstate.ManagedServiceStateMachine;

public class VerySimpleExample2 extends SimpleXMLManagedService {
    public static final String SERVICE_NAME = "VerySimpleService2";
    public static final String LOCALHOST = "127.0.0.1";
    public static final int COREPORT = 12346;

    public VerySimpleExample2() throws ESException {
        super(new ServiceContext(LOCALHOST, COREPORT, "tcp"), SERVICE_NAME);
        makeManageable();
        performTransition(ManagedServiceStateMachine.TO_READY_TRANSITION);
    }

    protected void stateChangeOccurred(String transition, String oldState,
        String newState) {}

    protected void resourceChangeOccurred(String resource, String oldValue,
        String newValue) {}

    protected void variableChangeOccurred(String variable, String oldValue,
        String newValue) {}

    protected boolean acceptStateChange(String transition, String oldState,
        String newState) {
        return true;
    }

    protected boolean acceptVariableChange(String variable, String oldValue,
        String newValue) {
        return true;
    }


    protected boolean acceptResourceChange(String resource, String oldValue,
        String newValue) {
        return true;
    }

    public static void main(String[] args) {
        try {
            new VerySimpleExample2();
        } catch (ESException e) {
            e.printStackTrace();
        }
    }
}
```

Here, the only real difference is in the constructor, where we have added the line:

```
performTransition(ManagedServiceStateMachine.TO_READY_TRANSITION);
```

Have another look at the service state diagram and you see an internal transition taking the service to the *Ready* state. The line of code, above, causes this transition to occur. Lets compile and run this program to see what happens when viewed through the Service Manager.

After the VerySimpleService2 is running, go to back to the Service Manager and click the *All Services* link and click the *Update* button. Your new program is available for management:

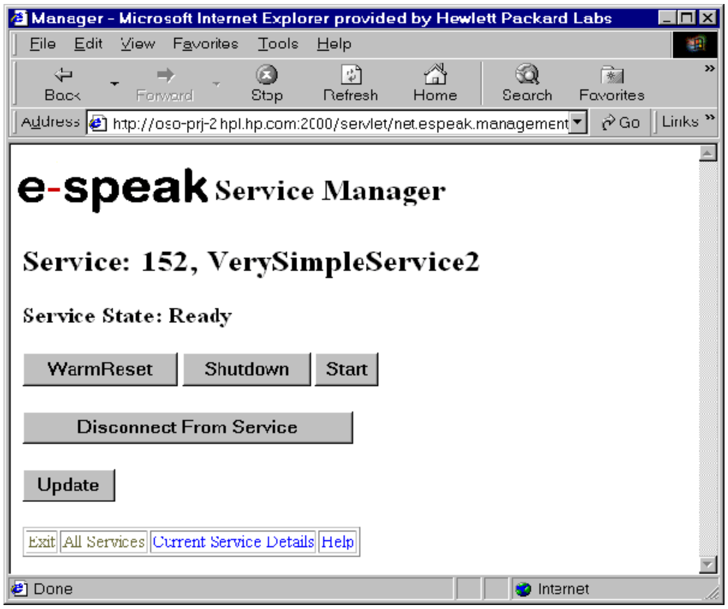

Manager Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://oso-prj-2.hpl.hp.com:2000/servlet/net.espeak.managem
Go Links
Update Service List
Exit Help
Internet

e-speak Service Manager

Manageable Services Available:

149 ExampleService	Management Detail
146 FileServerService	Management Detail
155 XMLServiceManager	Management Detail
152 VerySimpleService2	Management Detail
150 VerySimpleService	Management Detail

As before, go to the Services Management screen and click the *Manage Service* button. You see the following:

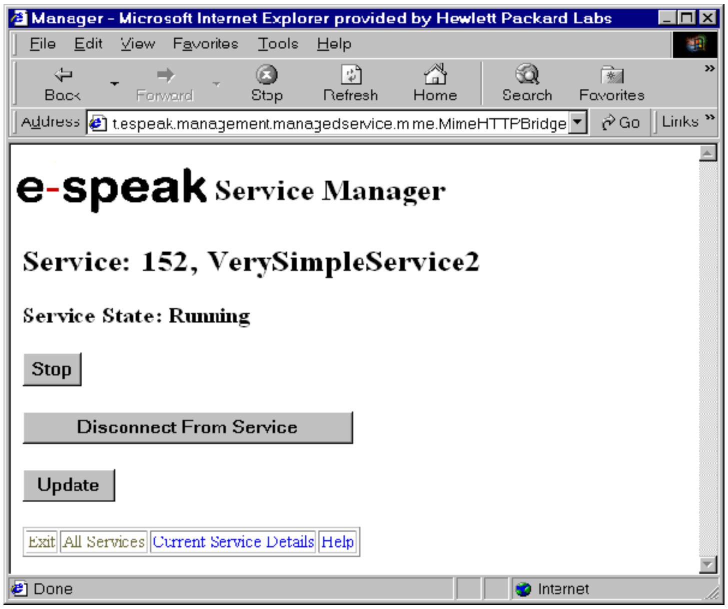

Manager - Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://oso-prj-2.hpl.hp.com:2000/servlet/net.espeak.management
Go Links
WarmReset Shutdown Start
Disconnect From Service
Update
Exit All Services Current Service Details Help
Done Internet

e-speak Service Manager

Service: 152, VerySimpleService2

Service State: Ready

The state of your service is now listed as ready and three new buttons have appeared. Looking back at the service state diagram, you can see that when a service is in the *Ready* state a manager can request that it can start, stop or perform a warm reset. Click the Start and the following screen appears:


Manager - Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address t.espeak.management.managedservice.mime.MimeHTTPBridge Go Links
e-speak Service Manager
Stop
Disconnect From Service
Update
Exit All Services Current Service Details Help
Done Internet

Service: 152, VerySimpleService2

Service State: Running

Clicking the Stop button takes you back to the previous screen. All very interesting, but how does this relate to the program we're running? At the moment the program isn't affected at all by these management changes.

Adding Custom Service Behavior

So far the service doesn't do anything interesting, so let's take a look at how we can improve the situation:

```
import
net.espeak.management.managedservice.simplemanagedservice.SimpleXMLManagedService
;
import net.espeak.infra.cci.exception.ESException;
import net.espeak.jesi.management.ServiceContext;
import
net.espeak.management.managedservice.managedstate.ManagedServiceStateMachine;

public class VerySimpleExample3 extends SimpleXMLManagedService
    implements Runnable {
```

```
public static final String SERVICE_NAME = "VerySimpleService3";
public static final String LOCALHOST = "127.0.0.1";
public static final int COREPORT = 12346;
protected boolean running = false;
protected Thread runThread = null;

public VerySimpleExample3() throws ESException {
    super(new ServiceContext(LOCALHOST, COREPORT, "tcp"), SERVICE_NAME);
    makeManageable();
    performTransition(ManagedServiceStateMachine.TO_READY_TRANSITION);
}

public void run() {
    while (true) {
        synchronized (runThread) {
            if (!running) {
                return;
            }
        }
        System.out.println("Running " + System.currentTimeMillis());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}

protected synchronized void stateChangeOccurred(String transition,
    String oldState, String newState) {
    if (newState.equals(ManagedServiceStateMachine.RUNNING_STATE)) {
        running = true;
        runThread = new Thread(this);
        runThread.start();
    } else if (runThread != null) {
        synchronized (runThread) {
            running = false;
        }
        try {
            runThread.join();
        } catch (InterruptedException e) {}
        runThread = null;
    }
}

protected void resourceChangeOccurred(String resource, String oldValue,
    String newValue) {}

protected void variableChangeOccurred(String variable, String oldValue,
    String newValue) {}

protected boolean acceptStateChange(String transition, String oldState,
    String newState) {
    return true;
}
```

```
    protected boolean acceptVariableChange(String variable, String oldValue,
        String newValue) {
        return true;
    }

    protected boolean acceptResourceChange(String resource, String oldValue,
        String newValue) {
        return true;
    }

    public static void main(String[] args) {
        try {
            new VerySimpleExample3();
        } catch (ESException e) {
            e.printStackTrace();
        }
    }
}
```

Here we have given the program a runnable thread and added some code to the **stateChangeOccurred()** method. This method is called after any change of service state. If you compile and run this program, then when you start and stop it from the Service Manager you also see the thread started and stopped.

Managing Variables

It is also possible to manage the data within your service. Consider the next derivative of the service:

```
import
net.espeak.management.managedservice.simplemanagedservice.SimpleXMLManagedService
;
import net.espeak.infra.cci.exception.ESException;
import net.espeak.jesi.management.ServiceContext;
import
net.espeak.management.managedservice.managedstate.ManagedServiceStateMachine;

public class VerySimpleExample4 extends SimpleXMLManagedService
    implements Runnable {
    public static final String SERVICE_NAME = "VerySimpleService4";
    public static final String LOCALHOST = "127.0.0.1";
    public static final int COREPORT = 12346;
    public static final String MESSAGE_VAR = "Message";
    protected boolean running = false;
    protected Thread runThread = null;
    protected String message = "running";

    public VerySimpleExample4() throws ESException {
        super(new ServiceContext(LOCALHOST, COREPORT, "tcp"), SERVICE_NAME);
        createVariable(MESSAGE_VAR, message, true);
        makeManageable();
        performTransition(ManagedServiceStateMachine.TO_READY_TRANSITION);
    }

    public void run() {
        while (true) {
            synchronized (this) {
                if (!running) {
                    return;
                }
                System.out.println (message + " " + System.currentTimeMillis());
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }

    protected synchronized void stateChangeOccurred(String transition,
        String oldState, String newState) {
        if (newState.equals(ManagedServiceStateMachine.RUNNING_STATE)) {
            running = true;
            runThread = new Thread(this);
            runThread.start();
        } else if (runThread != null) {
                running = false;
            try {
                runThread.join();
            } catch (InterruptedException e) {}
            runThread = null;
        }
    }

    protected void resourceChangeOccurred(String resource, String oldValue,
        String newValue) {}

    protected synchronized void variableChangeOccurred(String variable, String
oldValue,
        String newValue) {
        if (variable.equals(MESSAGE_VAR))
        {
          message = newValue;
        }
    }

    protected boolean acceptStateChange(String transition, String oldState,
        String newState) {
        return true;
    }

    protected boolean acceptVariableChange(String variable, String oldValue,
        String newValue) {
```

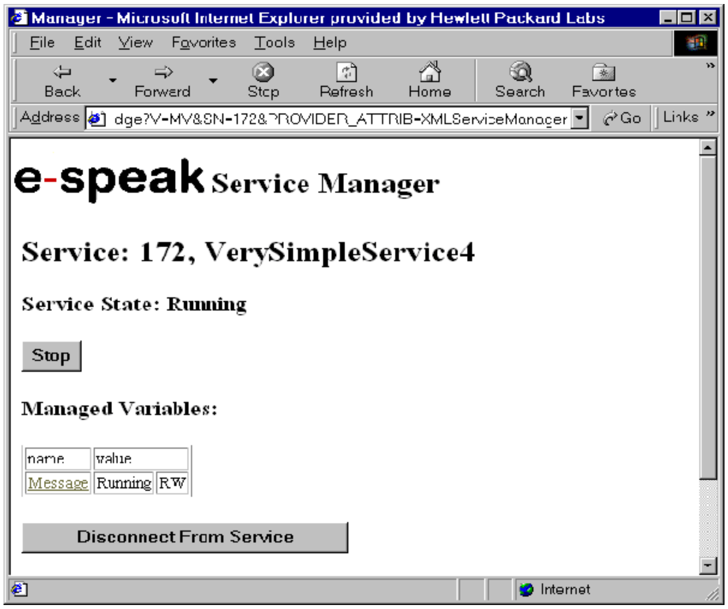
```
        return true;
    }

    protected boolean acceptResourceChange(String resource, String oldValue,
        String newValue) {
        return true;
    }

    public static void main(String[] args) {
        try {
            new VerySimpleExample4();
        } catch (ESException e) {
            e.printStackTrace();
        }
    }
}
```

Here we have added a normal class variable called “message” which we print in the run method. What we have also done in the constructor is create a *Managed Variable* that informs the system manager that such a variable exists, i.e. what it’s name is, what it’s initial value is and whether it can be remotely modified:

```
createVariable(MESSAGE_VAR, message, true);
```

This is sufficient to make the variable visible to the Service Manager, but we also want the our program to reflect changes to the value of this variable made by the service manager. To achieve this we have added some code to the **variableChangeOccurred()** method. This code simply verifies which *Managed Variable* has changed and ensures that the appropriate action is taken. In this case, it simply updates the local message variable.

Viewing this service through the Service Manager looks as follows:


Manager - Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address dge?V-MV&SN-172&PROVIDER_ATTRIB-XMLServiceManager
Go Links
Internet

e-speak Service Manager

Service: 172, VerySimpleService4

Service State: Running

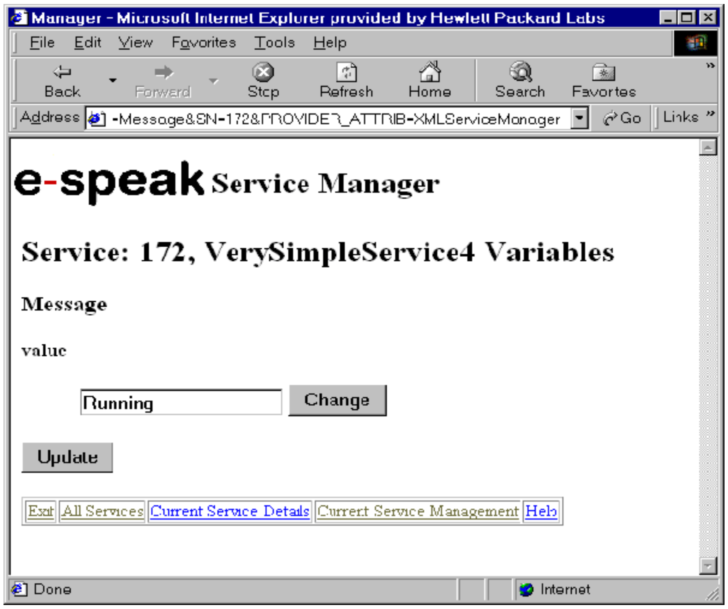
Stop

Managed Variables:

name	value	
Message	Running	RW

Disconnect From Service

You can see that in addition to the service state, our managed variable is visible. Clicking the variable name opens the following screen:


Manager - Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address -Message&SN=172&PROVIDER_ATTRIB=XMLServiceManager Go Links
Change
Update
Exit All Services Current Service Details Current Service Management Help
Done Internet

e-speak Service Manager

Service: 172, VerySimpleService4 Variables

Message

value

Running

Here you can change the value of the variable. Try changing the value and observe the output of your service change while it is running.

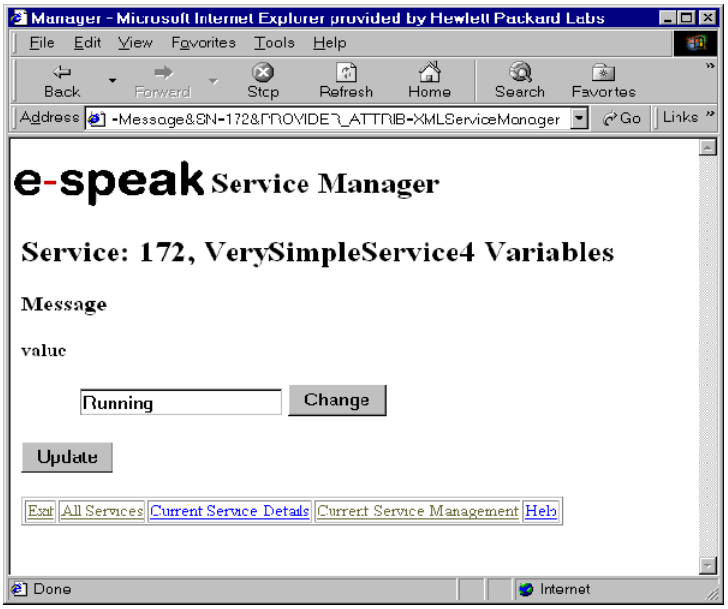
Dynamic Variables

So now we can change the value of our variables remotely, but what would happen if we changed the value locally? If the management system is to know the state of your service from second to second then obviously it needs to be told. This next example uses an additional managed variable, which is updated locally:

```
import
net.espeak.management.managedservice.simplemanagedservice.SimpleXMLManagedService
;
import net.espeak.infra.cci.exception.ESException;
import net.espeak.jesi.management.ServiceContext;
import
net.espeak.management.managedservice.managedstate.ManagedServiceStateMachine;
import java.util.Date;
```

```
public class VerySimpleExample5 extends SimpleXMLManagedService
    implements Runnable {
    public static final String SERVICE_NAME = "VerySimpleService5";
    public static final String LOCALHOST = "127.0.0.1";
    public static final int COREPORT = 12346;
    public static final String MESSAGE_VAR = "Message";
    public static final String TIME_VAR = "Time";
    protected boolean running = false;
    protected Thread runThread = null;
    protected String message = "running";

    public VerySimpleExample5() throws ESException {
        super(new ServiceContext(LOCALHOST, COREPORT, "tcp"), SERVICE_NAME);
        createVariable(MESSAGE_VAR, message, true);
        createVariable(TIME_VAR, new Date ().toString (), false);
        makeManageable();
        performTransition(ManagedServiceStateMachine.TO_READY_TRANSITION);
    }

    public void run() {
        while (true) {
            synchronized (this) {
                if (!running) {
                    return;
                }
                System.out.println (message + " " + System.currentTimeMillis());
                super.setVariable(TIME_VAR, new Date ().toString ());
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }

    protected synchronized void stateChangeOccurred(String transition,
        String oldState, String newState) {
        if (newState.equals(ManagedServiceStateMachine.RUNNING_STATE)) {
            running = true;
            runThread = new Thread(this);
            runThread.start();
        } else if (runThread != null) {
                running = false;
            try {
                runThread.join();
            } catch (InterruptedException e) {}
            runThread = null;
        }
    }

    protected void resourceChangeOccurred(String resource, String oldValue,
        String newValue) {}
    protected synchronized void variableChangeOccurred(String variable, String
oldValue,
        String newValue) {
        if (variable.equals(MESSAGE_VAR))
        {
          message = newValue;
        }
    }

    protected boolean acceptStateChange(String transition, String oldState,
        String newState) {
        return true;
    }

    protected boolean acceptVariableChange(String variable, String oldValue,
        String newValue) {
        return true;
    }

    protected boolean acceptResourceChange(String resource, String oldValue,
        String newValue) {
        return true;
    }

    public static void main(String[] args) {
        try {
            new VerySimpleExample5();
        } catch (ESException e) {
            e.printStackTrace();
        }
    }
}
```

Here we have added a new *Managed Variable* that represents the time, which we update in the run method of our service. Viewing our service through the Service Manager now looks something like this:


Manager - Microsoft Internet Explorer provided by Hewlett Packard Labs
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address -Message&SN=172&PROVIDER_ATTRIB=XMLServiceManager Go Links

e-speak Service Manager

Service: 172, VerySimpleService4 Variables

Message

value

Running

Change
Update
Exit All Services Current Service Details Current Service Management Help
Done Internet

Here we can see our new (read only) Time variable. Now when you start the service, each time you click the Update button, you see the most recent value of the time variable.