TAMPERE UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF AUTOMATION

E-SPEAK IN ENTERPRISE SCALE CONDITION MONITORING NETWORK

Master of Science thesis

Mikko Salmenperä

SUBJECT APPROVED BY DEPARTMENT

COUNCIL ON NOVEMBER 8TH 2000.

EXAMINER: PROFESSOR HANNU KOIVISTO

Acknowledgements

This work has been carried out in Tampere University of Technology, Department of Automation. It is a part of the project *Remote technologies in automation*.

Project is funded by Neles Automation Field Controls Division, Neles Automation Control Systems, Timberjack Plustech Oy, Nokia Communication Systems and TEKES Technology Development Centre.

I wish to express my sincere appreciation to supervisor of this work, professor Hannu Koivisto for commenting and improving this thesis. I would also like to thank Jari Kero and Joona Nikunen, whose comments have been invaluable aid in writing of the thesis. Furthermore I wish to thank all economical supporters of the thesis.

Tampere 1.12.2000

Mikko Salmenperä

Sammonkatu 35B46

33720 TAMPERE

puh. +358 3 2617121

Abbreviations

ACL	Access Control List.
CA	Certificate Authority.
CLR	Common Language Runtime (in .NET framework).
CRC	Certificate Result Certificate.
CRL	Certificate Revocation List.
GUI	Graphical User Interface
HTML	HyperText Mark-up Language.
HTTP	HyperText Transfer Protocol.
LDAP	Lightweight Directory Access Protocol.
MAC	Message Authentication Code.
PDU	Protocol Data Unit.
PSE	Users Private Security Environment in e-speak. Stores private and public key pairs belonging to the user.
SDSI	Simple Distributed Security Infrastructure.
SLS	Session Layer Security. E-speaks implementation of TLS.
SPKI	Simple Public Key Infrastructure.
SSH	Secure Shell. A UNIX shell program for logging into and executing commands on a remote computer.
SSL	Secure Sockets Layer. A protocol designed for providing encrypted communications on the Internet.
TLS	Transport Layer Security. Improves weak spots of SSL.
WAP	Wireless Access Protocol.

WML	Wireless Mark-up Language. Subset of XML.
WWW	World Wide Web.
X.509	Standard for constructing a multi-purpose distributed directory service.
XML	eXtensible Mark-up Language.
XSL	eXtensible Stylesheet Language.
XSLT	eXtensible Stylesheet Language Transformations.

Table of Contents

1 INTRODUCTION

An enterprise network refers to a large scale network that ties companies and their services together. In the past enterprise networks were mainly SNA networks, formed in a tree like patterns. Today, as TCI/IP is becoming a dominant connection protocol, the structure of an enterprise network is changing. TCP/IP, Internet and advanced security protocols make more diverse and open structures possible.

Virtual private networking (VPN) integrated with a strong encryption makes it possible to create enterprise network connections over an existing network infrastructure. The ability to use the existing infrastructure lowers overall costs of the network connections substantially. It also makes the upkeep of the network easier and more flexible as the majority of the network connections can be created using software only. VPN technique also offers a possibility to out-source the network management to an external service provider.

A general structure of an enterprise scale condition monitoring network is designed in this thesis. The structure of the network designed is not tied to any operating system or programming language. It is designed to be modular and layered, in order to provide a usable platform for software agents. The modular structure makes easier to re-use old components in the future applications.

Security is one of the most difficult problems the enterprise scale network must deal with. Threats cannot be eliminated by isolating the system from Internet, because Internet is designed to be a part of the network infrastructure. The system is also accessed by mobile users from various outside locations and IP addresses. Security must be solved strictly, yet in a way that does not burden administration or cause unnecessarily complex solutions in the services deployed.

There are several new technologies that are suitable for implementing the distributed network mentioned above. Enterprise Java Beans (EJB) [3] is a new technology designed to be used in creating distributed systems based on application servers. EJB technology is studied, in a thesis currently under writing by Jari Kero at Tampere University of Technology, Automation and Control Institute.

Microsoft has also presented a whole new family of products called .NET. It is a collection of Microsoft's existing technologies updated to support XML based communication. In addition to upgrading old technologies, Microsoft has added several new ones, not available previously. It is possible to use several programming languages with .NET framework. Though the whole system is limited to Microsoft's operating systems. All .NET components have not reached even beta testing phase yet, and are thus unavailable for any testing purposes.

Hewlett-Packard has developed an interesting new technique called e-speak. E-speak is a very open platform providing secure connections between clients and resources. It is an open source distribution, and the whole platform specification is available freely to the public. E-speak is designed in such a way that any programming language or communications protocol can be utilised. It has also been designed from the very begin, to operate in a hostile environment, such as Internet.

E-speak is selected as the framework for the system, because of its open structure and ability to provide secure and authenticated services. E-speak provides a framework for the network, allowing clients and services to connect to and disconnect from the on-line system. In a distributed environment e-speak handles internally all message routing and exchange of metadata between the e-speak cores.

In order to test e-speaks ability to operate in different environments, a test system is created. The test system encrypts all communication between client – core – service connections and is also used to verify cross-platform operation. Figure 1 describes the general structure of the test system.

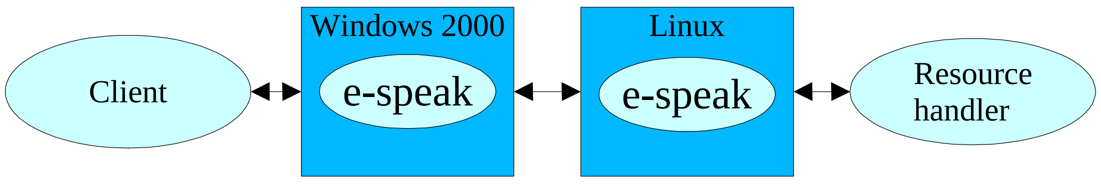

Figure 1   E-speak test system.

2 E-SPEAK

This chapter is based on references [4], [5] and [6]. E-speak is an open software platform designed specifically for the development, deployment, and intelligent interaction of e-services. E-services can mean any kind of a service available from LAN or Internet. E-speak doesn't define contents of services that are linked to it. It simply defines interfaces for these services to use .

E-speak tries to solve a large set of problems in one package. It provides a secure access to resources and services, usage monitoring, methods to search for a best matching service and ability to advertise services. E-speaks architecture also makes it scalable and although Hewlett-Packard seems to market it for business-to-consumer kind of e-services it really is a suitable platform for business-to-business e-services. Figure 2 describes standard steps when, for example, making an order of some product. Functionality e-speak provides for each item is listed on the right side.

Currently e-speak is composed of two major parts: an e-speak engine and an e-speak services framework specification. The e-speak engine is a run-time e-services engine installed on computers that connect to an e-services environment. The e-speak services framework specification defines protocols and special interfaces (APIs) that are used by the e-services.

E-speak is still under development by Hewlett-Packard. Some of the core properties are still missing from it, others might change. A core-manager resource seems to be the most volatile part of e-speak specification. So far each new release has introduced several new core-managed resources. This document is based on e-speak version 3.01.

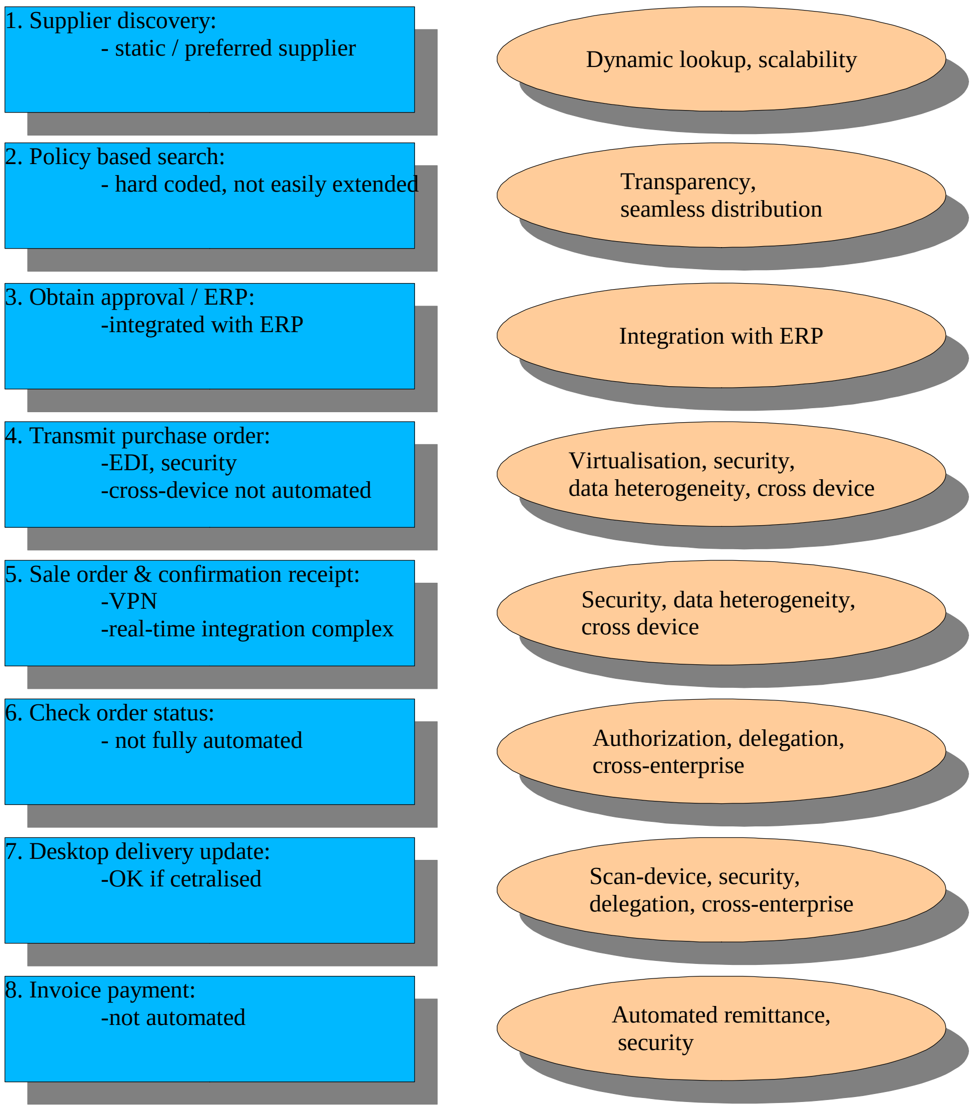
1. Supplier discovery:
- static / preferred supplier
Dynamic lookup, scalability
2. Policy based search:
- hard coded, not easily extended
Transparency,
seamless distribution
3. Obtain approval / ERP:
-integrated with ERP
Integration with ERP
4. Transmit purchase order:
-EDI, security
-cross-device not automated
Virtualisation, security,
data heterogeneity, cross device
5. Sale order & confirmation receipt:
-VPN
-real-time integration complex
Security, data heterogeneity,
cross device
6. Check order status:
- not fully automated
Authorization, delegation,
cross-enterprise
7. Desktop delivery update:
-OK if cetralised
Scan-device, security,
delegation, cross-enterprise
8. Invoice payment:
-not automated
Automated remittance,
security

Figure 2 The answer of the e-speak to business-to-business scenario.

2.1 E-speaks architectural philosophy

All system functionality and e-speak abstractions build on top of a resource. The resource is an uniform description of active entities such as a service or passive entities such as a hardware device. Unlike most platforms e-speak deals only with data about resources, metadata, and not resource specific semantics.

E-speak does not access the resources directly. A resource specific handler is always associated with the resource. The handler receives messages from e-speak and then directly accesses the resource.

In e-speak environment a client can be an agent, a GUI program operated by human or even a resource accessing another resource. The clients are in fact just an other resource from the cores point of view. Only difference is that a client does not register metadata. Thus it cannot be queried from the repository. Resources can act like clients. They can query the repository to locate suitable resources and then send to the located resources messages requesting service.

E-speaks four key terms are:

- **Client:** An active entity that requests access to the resources or responds to such requests.
- **Resource:** A resource means any computational service which is virtualised by e-speak.
- **Protection domain:** Part of a e-speak repository visible to a client.
- **Logical machine:** An active entity that performs operations needed to implement the e-speak system.

2.1.1 Resource

Resources in e-speak means all services and devices connected to the e-speak system. Services provide some functionality for the rest of the system. All the resources are treated likewise in e-speak. Even the core-managed resources, used for running the system, are treated like the rest of the resources. Figure 3 describes a general resource access.

There are two ways a client can connect to a resource. If the client already has a resource handle to the certain resource, it can then use this handle to access the resource. The resource handle is always specific to a single resource and provides way to connect to this exact resource. Another way to connect to a resource is by using the core to perform a look-up based on attributes defined in a resource description contained in each resources metadata. Most appropriate resource or resources are then returned to the client.

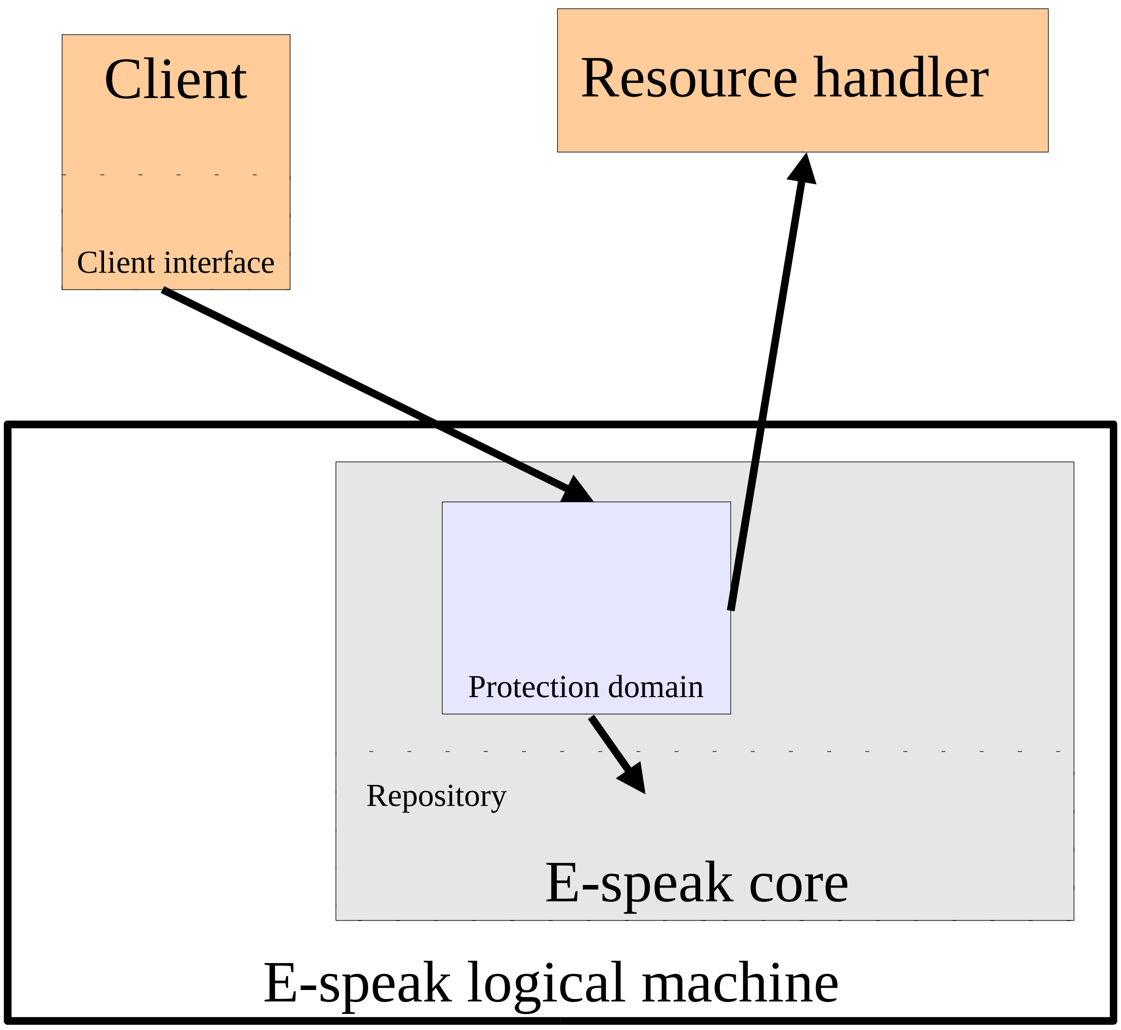

Figure 3 Resource access in e-speak.

The resource locating method of e-speak provides a transparent access to remote resources, making it unnecessary, for the client, to know exactly what resource it is connected to. There is no visible difference in accessing resources physically located on the same computer as the client from accessing resources located on some remote computer.

Connections to the resources are always mediated by the e-speak core. Thus all the resources and the clients are connected directly only to the e-speak core. This connection is implemented using a E-speak Service Interface (ESI). This interface is provided in a library, which is linked to the clients and the resources. ESI communicates with the e-speak platform through an Application Binary Interface (ABI). The communication occurs over a Session Layer Security protocol (SLS).

A list of available resources is managed dynamically. New resources can be added and existing removed on the fly. This makes management of the resources in the e-speak system easy. This modular design makes e-speak fault tolerant. If some resource handler computer crashes, only the resources that were on its responsibility become temporarily unavailable. Rest of the system continues uninterrupted. When the resource handler computer reboots, the resources in question can restore their original mailboxes and any messages stored in them.

Since clients don't necessarily know the exact handles of the resources they use, requests to a certain resource can be easily redirected to other similar resources. For example, if current resource becomes unavailable, other similar resources present in the system can take over. The resources can also move clients connected to them to other resources, if for instance load increases to too high level. This requires that there are resources present in the system, that can perform same actions as the original resource.

2.1.2 Logical machine

A logical machine means a single instance of e-speak platform. There is always one instance of the e-speak core per logical machine. Rest of the components that make up logical machine can be distributed over several other computers. There can be multiple logical machines located on a single computer.

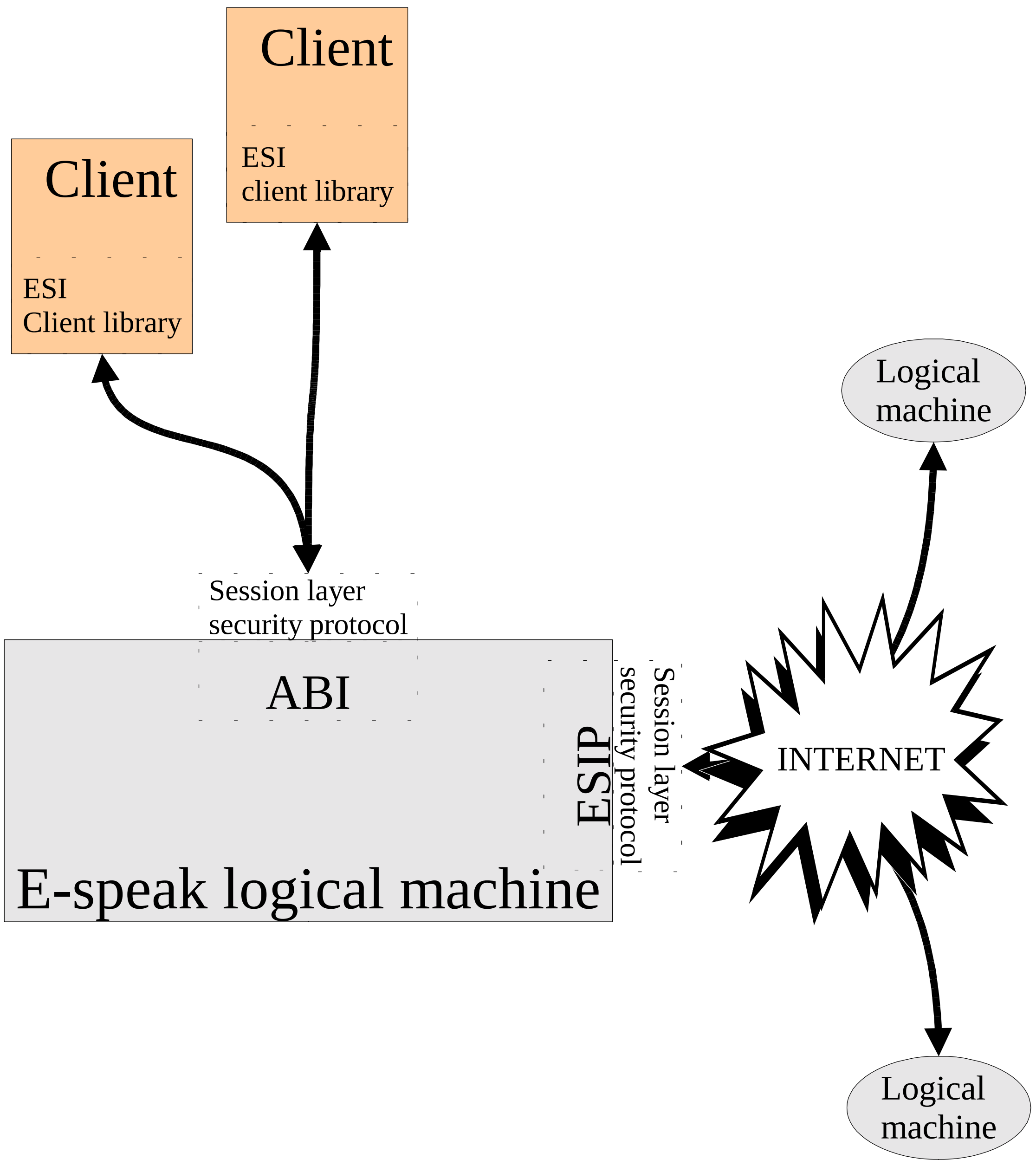

Figure 4 Communication between e-speak logical machines.

The logical machine is responsible for all the operations provided by the e-speak platform. It manages all the core-managed resources and provides the mechanisms for messaging and events. The logical machines are independent entities that can discuss with each other using a e-speak service interchange protocol (ESIP). In figure 4 are described ESI and ESIP.

2.2 Resources in e-speak

The resources are one of the central concepts in e-speak. They can be thought as services, connected to the core through libraries provided by the e-speak services framework. E-speak does not in any way define their contents. Resource handler means a wrapper that controls resources connection to the core.

2.2.1 Resource model

A resource represents a service inside e-speak. Metadata of the resource is registered with the core. Metadata describes the resource and provides a storage for important information. Registered metadata defines following items:

- **Description**: An attribute-based specification of the resource.
- **Vocabulary**: A definition of attributes and their types. The core provides a default vocabulary.
- **Resource handler mailbox**: A process/thread/task that handles the resource.
- **Contract**: Denotes the application programming interface (API). Includes version information.
- **Resource mask, owner public key and service ID**: Access control information.
- **Private resource-specific data**: Important data to the provider of the resource, such as an internal name of the resource. Not interpreted by e-speak.
- **Public resource specific data**: Important data to the user of the resource, such as a stub. Not interpreted by e-speak.

E-speaks resource names are universal resource locators (URL). Name spaces are hierarchal. Each core has a name space that begins from a root name. A client specifies the resources name starting from this root node. If the resource is located on an other core the client must also specify a host address of this core. For example a resource named foobar, located on Core_A would be named as:

es://<host_address_for_core_A>/resource/foobar.

Only valid name for a resource from a clients point of view is URL. The physical address of the service is not a valid reference. There are two ways the client can acquire resources name. First way is that the client gets the resources name from an other resource. Other way is by making a bind call that requires a search recipe as a parameter. The core then looks up the name from a local or a remote repository.

2.2.2 Resource descriptions and resource specifications

There are two categories of resources in e-speak. The core manages directly resources called core-managed resource. All the rest of the resources form a non-core-managed resources group. Only difference between these two is that the core-managed resources are managed by and run inside the core. Clients interact with the core by sending messages to these resources. The non-core-managed resources are created by service providers and are then registered to the e-speak core.

A resource is described by its metadata. This metadata consists of a resource description and a resource specification. A client registers a resource by sending a message to the core-managed resource called resource factory. This message contains both the resource description and the resource specification.

The resource description contains vocabularies and attributes associated with each other. It is used to provide information which the core uses when a client searches for suitable resources. The resource description class in e-speaks version 3.01 is defined as follows.

```
public class ResourceDescription

{
                AttributeSet[] attribSets;
}
```

AttributeSet consists of ESName of a vocabulary and ESMap of name-attribute pairs. ESMap is a general structure in e-speak used to create lists that contain named attributes. It is also used in several other classes.

The resource specification on the other hand is a collection of several different things. It contains an inbox, connected to the resource handler responsible for managing the resource. The resource specification also contains any security restrictions that apply to the resource. Variety of other fields are also included. The resource specification class in e-speaks version 3.01 is defined as follows.

```
public class ResourceSpecification

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

2.2.3 Managed resources life cycle

A managed resource means a resource that implements interface ManagedServiceIntf, defined by the e-speak services framework. This interface contains definitions to methods used in managed resources life cycle figure. A Managed resources life cycle is shown in a figure 5.

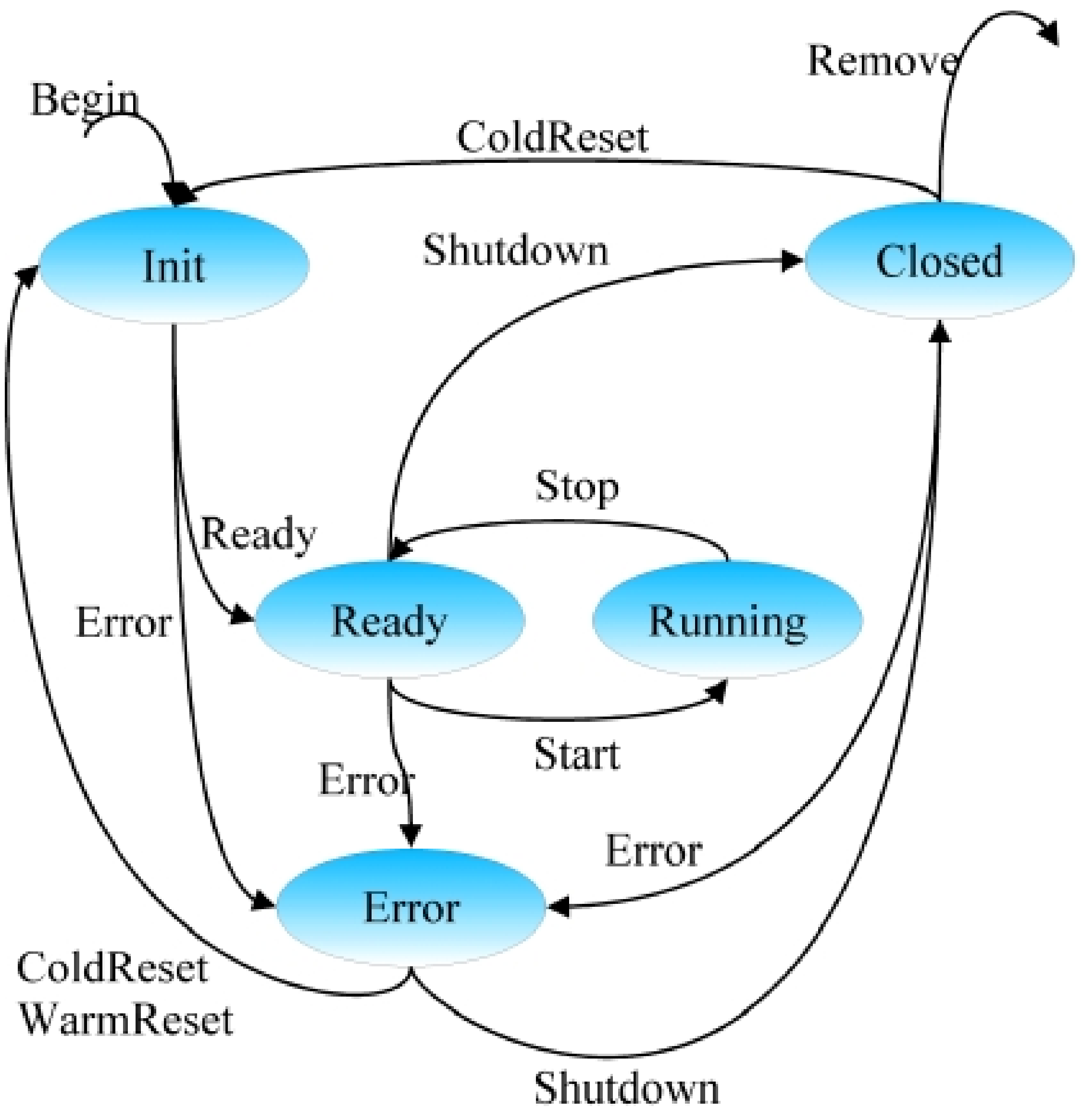

Figure 5   E-speak resource life cycle.

Descriptions of states shown in figure 5 are:

Initialising

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 state.

Ready

The service is ready to run. This state is also equivalent to Stopped or Paused state.

Running

The service is running. If an error occurs which implies that the service cannot continue to run it should move into Error state.

Error

The service has some problem and is waiting for management action.

Closed

The service has removed/deleted much of its internal state and waits for either ColdReset or Remove transitions.

Input used are following:

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 initialisation.

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 complete reinitialisation of the service and move into the initialising state. The only exemption is that resources that are already registered should not be reregistered.

WarmReset

Cause a partial reinitialisation of the service i.e. retaining some of the existing service state. Moves service into the initialising state.

Remove

Cause the service to remove itself from existence. Any non-persistent resources should be deregistered.

2.2.4 Core-managed resources

The core-managed resources are always running, while the core is running. However not all clients have necessary privileges to access them. The core-managed resources implement much of the core functionality. They also provide a way for the clients to interact with the e-speak system. The core-managed resources in e-speak version 3.01 include:

- **Connection manager**: Provides core-to-core connection management.
- **Core management**: Not supported in 3.01 version. Provides a way for a client to manage its own and other cores.
- **Remote resource manager**: Provides capabilities to import and export resources.
- **Mailbox**: E-speaks message transfer queue.
- **Name frame**: Binds ESNames to resources.
- **Finder**: A resource for finding out services provided by core.
- **Protection domain**: Encapsulates clients environment.
- **Repository view**: Contains references to set of resources.

- **Resource contract**: API that is understood by resource.
- **Resource factory**: Client uses resource factory to register resource.
- **Resource manipulation**: Once a resource has been registered only way to access its metadata.
- **Vocabulary**: Used to describe resources and to specify look-up requests.
- **Account manager**: A resource for managing user account on an e-speak core.
- **User Interface**: Not implemented in 3.01 version. Provides users a way to customise their clients.

A connection manager creates initial connection between two cores, manages it and finally closes it down. Core-to-core can be secured by using SLS messages.

A remote resource manager provides a capability to import or export resources between e-speak cores. The remote resource manager forwards (routes) messages between cores and handles import / export of metadata.

A core management resource provides clients a way to manage the core remotely. This resource has not been implemented yet, but can be expected in next releases. It will contains following methods:

- Method **ping** checks that the core is up and returns the value specified.
- Method **getClientConnections** returns a list of protection domains that are currently being used.
- Method **stopServingOutbox** and **startServingOutbox** tell the e-speak core to stop or start serving the outbox associated with the protection domain specified.
- Methods **stopServingInbox** and **startServingInbox** tell the e-speak core to stop or start serving messages to the Inbox specified.
- Method **removeProtectionDomain** removes the Protection Domain specified. Any client using the protection domain are disconnected and any resources contained in the protection domain are automatically deregistered.

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

A mailbox can be an inbox or an outbox. Only inboxes are exposed to clients as core-managed resources. The inboxes are a resource from where a client gets its messages. Single client can have more than one inbox, but all the inboxes must be connected to the client before they can be used.

A name frame manages bindings from ESNames to resources. A client has a default name frame as a part of its protection domain.

A finder resource is for locating core-managed resources. It takes a search recipe as a parameter.

A protection domain is maintained for each individual client. It contains the clients environment, fixing a set of resources available to the client at any given moment. Only the resources visible in the clients protection domain are available. A client can have more than one protection domain, but only one will be active. Messages are interpreted in this active domain.

A repository view contains a set of resources. Look-up in resource view is limited to the resources visible in this view. Clients can add and remove resources from the repository view.

A resource contract defines payloads of messages. It also includes secondary resources required by the resource and necessary access permissions. Resource denotes application programming interface (API) understood by the resource handler.

A resource factory is responsible for creating new resources. Clients create a new resource by sending a message containing resource to the resource factory. The resource creates core-managed resources as well as non-core-managed resources.

A resource manipulation resource is the only way to access and change a resources metadata after the registration.

A vocabulary is used to describe resources and to specify look-up requests. The core provides a default vocabulary, but clients define their own, if necessary.

An account manager is a resource for managing the user accounts on an e-speak core. A user account contains various information about the user including its PSE (Private Security Environment). This enables a user to authenticate to the Account Manager (via userid, password) and to retrieve their PSE. In the current implementation they need a passphrase to unlock their PSE to access their key material. In the default implementation the users private key is encapsulated inside a PSE object.

2.2.5 Access control and authentication in e-speak

E-speak security is based on a simple public key infrastructure (SPKI) [7], [8]. Each client, resource and core has an unique key pair. These keys are used in authentication only and provide no access power in the system. The access control decisions are based on access control tags contained within the certificate. Each certificate contains tags specifying the access rights issued to it. A private key is used to authenticate the sender of the message. Default implementation stores users private key in PSE, inside the core. PSE is unlocked by a passphrase. SPKI certificates are described in more detail in paragraph SPKI certificates.

SPKI certificates are different from more commonly used X.509 [2, chapter 4.2] certificates. Most common use of X.509 links a distinguished name to a public key. This leads usually to a need to search an authorisation database in order to get access rights granted to certain certificate. In e-speak the certificates are more general. They are signed statements linking a public key to a name or a tag. The tag is similar to X.509 attribute and typically states access right.

When checking access rights granted by an SPKI certificate, a service does following:

1. Examines the tag in the certificate checking access rights it grants.
2. Authenticates the entity making request using a key pair associated with the certificate.
3. Checks that the certificate has been issued by an entity the service trusts.

E-speak also implements a split trust. This means that not all services connected to a single core, has to trust the same certificate issuers equally. A service can trust one issuer to grant only a subset of access right to its operations, while an other issuer has power to grant all access rights. Each service defines itself to whom it trusts, and to what degree.

SPKI certificates used in e-speak do not have to be issued by any official certificate issuer. Anybody can issue SPKI certificate. However a certificate is valid only if it's issuer is trusted.

Access control decisions are based on a valid certificates authorisation tags. All e-speak tags are valid SPKI tags. For core-managed resources the e-speak core will check that a valid certificate is presented containing a tag that authorises the operation. However for non-core-managed resources the resource handler must check that the certificate is valid and contains a required tag. The source handler responsible for the resource is also responsible for its security, since e-speak core cannot enforce this.

2.2.6 Repository

The repository is not directly visible to clients. It holds all data needed by the e-speak core. An internal state of the core-managed resources and all resource metadata is stored here. The repository is divided into two components: a repository database and a repository access table. This division is done to allow efficient low latency access to the metadata and core-managed resources. Look-up requests on the other hand are less sensitive to latency but must be completed relatively quickly.

The repository database is a persistent storage. It can be connected to wide range of databases. Appropriate database interface is needed, of course. The core uses the repository database to:

- Register and unregistered resources.
- Access the metadata corresponding to a given repository handle.
- Modify the metadata corresponding to a given repository handle.
- Look up resources that match a search recipe.

The repository access table is fully resident in memory. This access table is rebuild from the data stored in the repository database as part of a system restart. The repository access tables main function are fast associative look-ups based on resource handles. It can be configured to act as a cache for the resource database or to hold all data needed for any resource access.

2.3 Communication in e-speak

All access to resources occurs with messages send through a core. Thus the core mediates all access between clients and resource handlers. Mediation means that the e-speak core determines to which resource handler or handlers to route the message. The e-speak core also determines how to route the message. The core can process and transform the message header and contents. Security limits processing of the message, if enabled.

2.3.1 Message flow

The e-speak core does not keep any information about replies to the messages. All messaging is asynchronous, but if a client wishes a reply, it can wait. The messages have identifier set by a sender. This identifier can be used as a reference in the reply.

The mediation architecture is invisible to both the client and the resource handler. Figure 6 shows message flow between client and resource. Message cannot be delivered if any one of the following condition occur:

- The “To” field of the message is not a valid resource.
- The Inbox specified in the destination resource metadata is not connected to a resource handler.
- The resource handler’s inbox is full.

If the message cannot be delivered an error is returned to the sender. A valid message is delivered to the receiver's inbox.

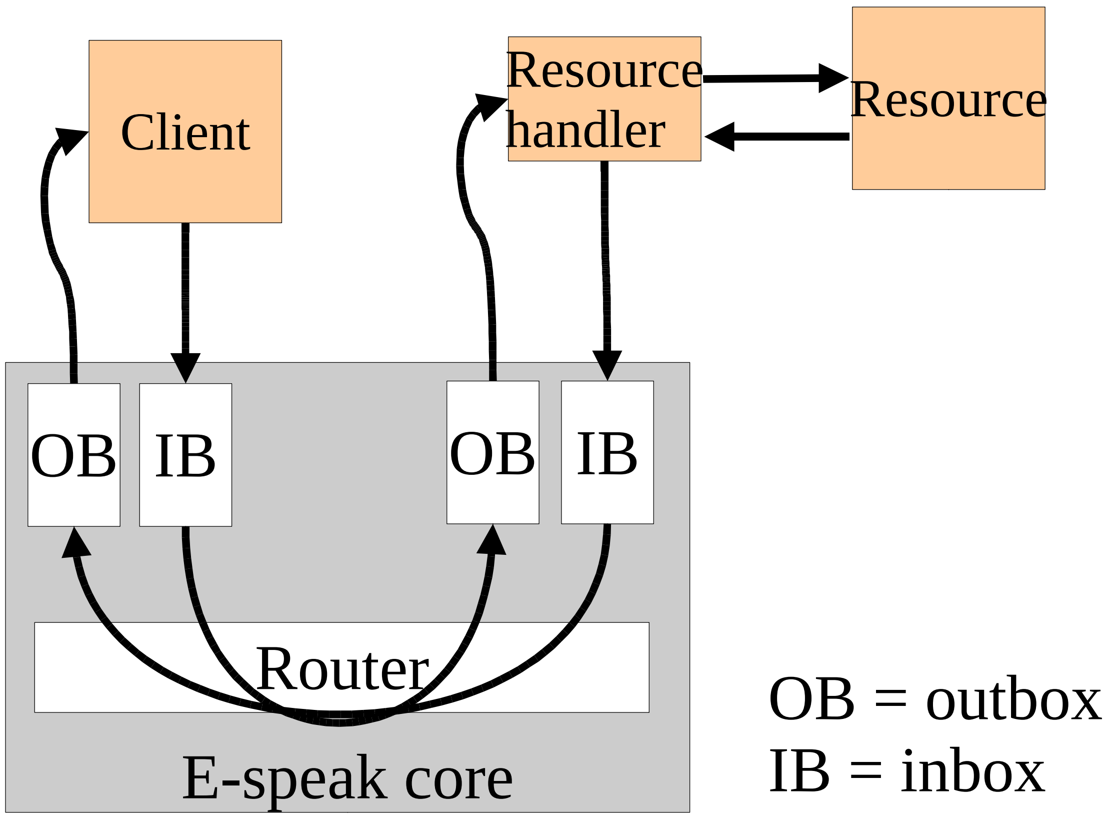

Figure 6 Message flow in e-speak.

2.3.2 Remote resource access

The message flow remains fundamentally the same, when accessing a resource connected to a different core. E-speak routes messages automatically between the cores. This routing is invisible to both the client and the resource handler. Figure 7 describes message flow when accessing a remote resource.

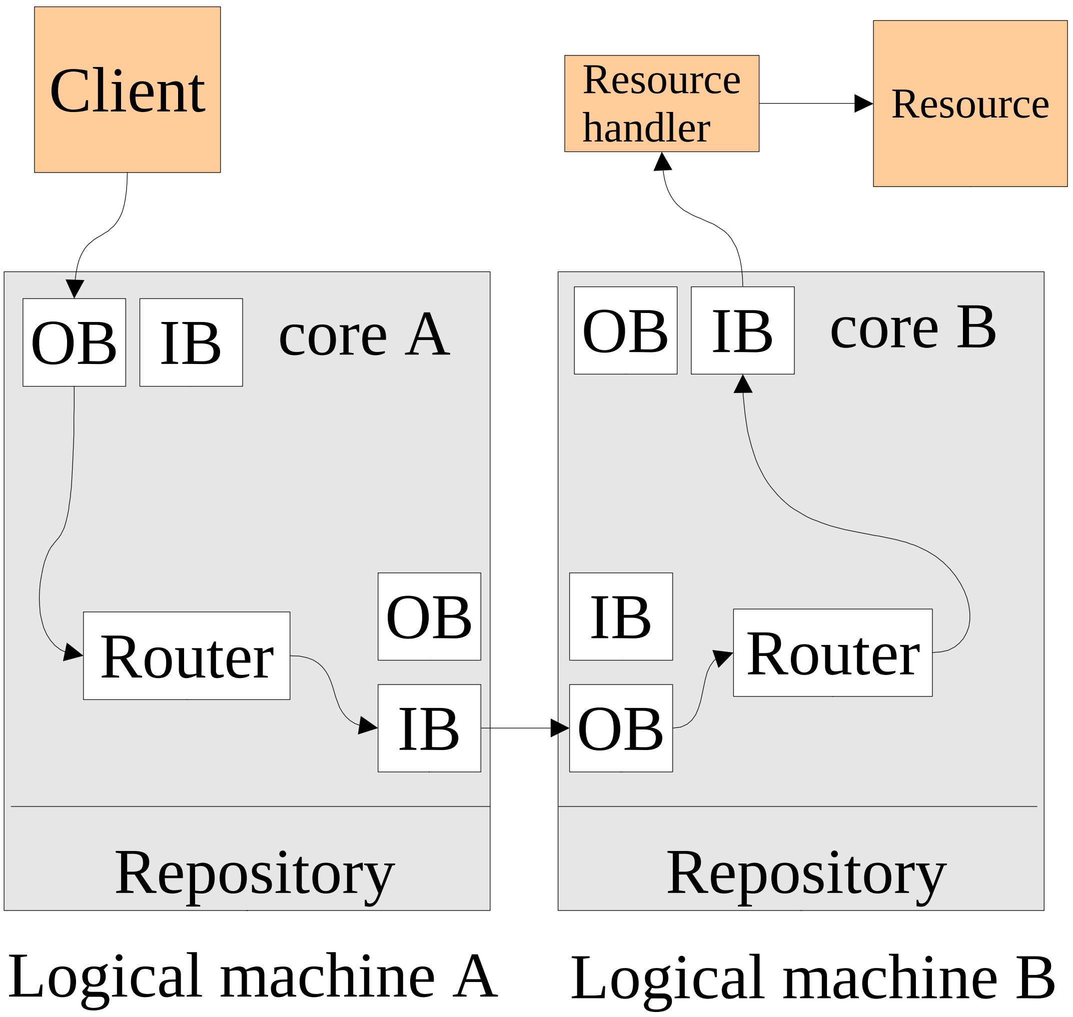

Figure 7 Message flow in remote resource access.

2.3.3 Event handling

Event mechanism in E-speak is simple. It contains some advancements over basic event model, that only contains publisher, subscriber and event queue. Four entities interact in event model that is described in figure 8.

Event publisher is the entity that generates events. Event distributor receives events from the publisher and then forwards it to subscribed listeners. The event publisher and the event distributor are typically one and the same entity. The event subscriber registers interest with the event distributor. The event listener receives the event from the event distributor. The event subscriber and the event listener are also the one and the same entity.

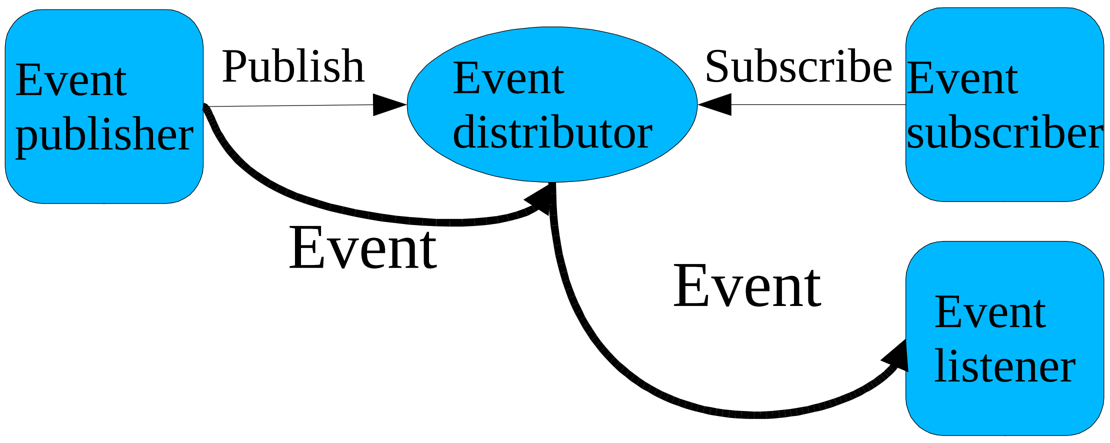

Figure 8 Events in e-speak.

The e-speak events are in fact just a subtype of the application messages. The events are handled by registering a callback with the messaging layer. This method provides a way for the subscriber to control who can send messages to it. Message is considered to be an event if a permission associated with this callback is also included in the message. Figure 9 describes the event handling mechanism of e-speak and a callback associated with it.

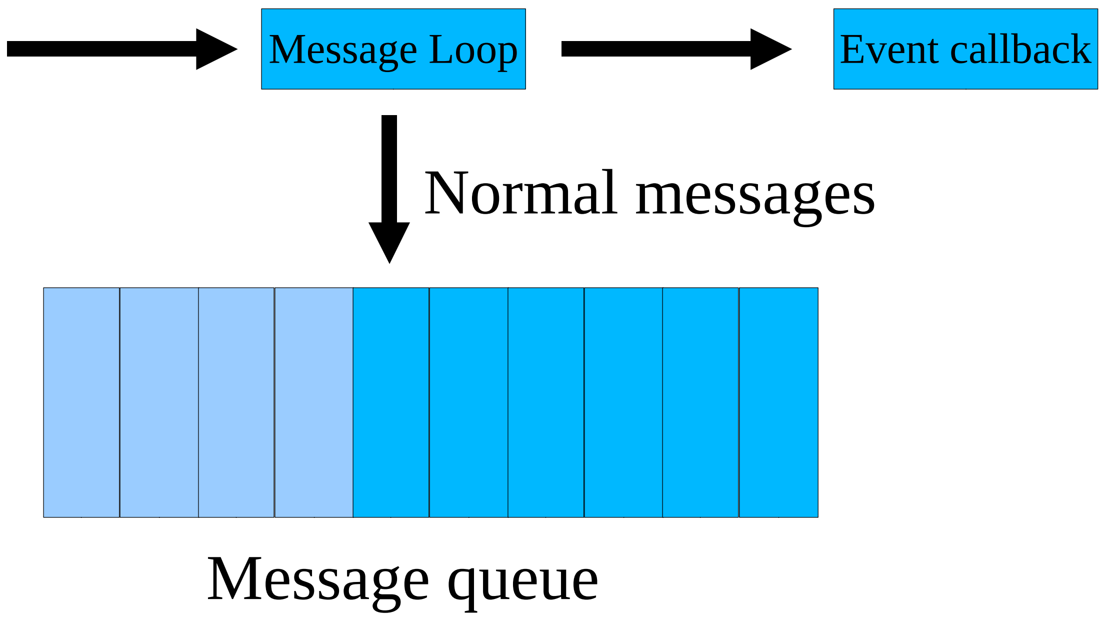

Figure 9 Event handling mechanism.

2.3.4 Protocol data unit

All messages exchanged between e-speak cores and between an e-speak core and a client are protocol data units (PDU). A single PDU corresponds to a single session layer security (SLS) message. E-speak framework defines PDU class as follows:

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

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

SPI stands for Session Parameter Index. This is used by the two endpoints in the session layer security protocol to indicate which session the message is being sent on. Since the sender and the receiver may identify the SPI separately we have two fields: spi denotes the recipient's SPI; spiSender denotes the sender's SPI.

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

The following values for message type are defined: alert, handshake, application message, tunnel and ping. Alert, handshake, tunnel and ping are used in SLS to manage sessions.

The following encoding types are defined for a PDU: clear data, protected data and secure data. Protected data is authenticated and protected from tampering by a message authentication code (MAC). Secure data is protected by a MAC and also encrypted for confidentiality.

Address fields are absolute ESNames. The core tries to resolve these names in a root name frame by finding corresponding mapping object. If the ESname can not be unambiguously resolved, exception is sent to sender.

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

The format of data member byte-array is determined by the encoding type. If the encoding is clear data, the byte array is a message body, of contents indicated by the message type.

Protected data begins with the MAC in network byte order. The length of the MAC depends on the MAC algorithm negotiated in the SLS session setup. The remainder of the contents of data is the body of the message.

Secure data has been encrypted according to the cipher negotiated in the SLS session set-up. Once it has been decrypted, it will have the same format as protected data.

Data member contains one of the following objects, according to message type:

- Alert: Used for session management.

- Handshake: Used for session management.
- Application data: Used for transferring application data
- Ping: used for session management.

2.4 Session layer security protocol

Session layer security protocol (SLS) tries to extend properties of SSL [15]. SLS properties are following:

- Transport independence: SSL links a security session with a TCP socket. If the socket dies the security session dies with it: something undesirable when the life expectation of a security session is very different from the life expectation of the transport. Also, we cannot multiplex several security sessions onto the same socket, or perform dynamic load balancing of the end-point without starting the session from scratch Finally, in some cases we might want to use a different transport for sending and receiving messages (i.e., outgoing messages use a different firewall needing two sockets). SLS tries to make the minimum number of assumptions on the communication transport solving most of the issues above.
- Tunnelling support: During firewall traversal we might want the firewall to control the client access rights to the internal LAN for every packet. However, we might not want the firewall to see all the traffic in clear (therefore, losing the end-to-end security property). This is difficult to achieve with SSL because either we let the client open a direct socket to the service or the firewall will see all the traffic in clear. On the other hand, with SLS we can nest a secure session inside another one, possibly with different end-points, allowing to achieve both goals simultaneously.
- Elliptic cryptography: Most implementations of SSL only support Diffie-Hellman key agreement algorithms based on exponentiation [2, p.190]. SLS uses a faster algorithm based on Elliptic Curve Cryptography (ECC) [2, p,193].
- Attribute certificates using SPKI. SSL only supports X.509 name certificates,

mainly to authenticate that the end-point owns , according to a configured trusted CA , the web address that we wanted to reach. Only one certificate by each party can be used, and in most cases only server authentication is performed. On the other hand, SLS performs a negotiation of tags that need to be proven represented by multiple SPKI certificates. This allows a fine grained control of security by mapping tags to actual permissions, raising the level of abstraction from a stream of bytes in SSL to a particular operation on service X in SLS, making it easier to integrate with application level security.

- Latency minimisation: SLS allows the client to send application data after a round-trip negotiation has succeeded. In SSL two round-trips are needed before the application data is sent. This can have important performance implications when network delays are large and we need a quick response from the server.

2.4.1 High level protocol state machine in SLS

Figure 10 shows possible states in SLS session. The states are defined as follows:

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

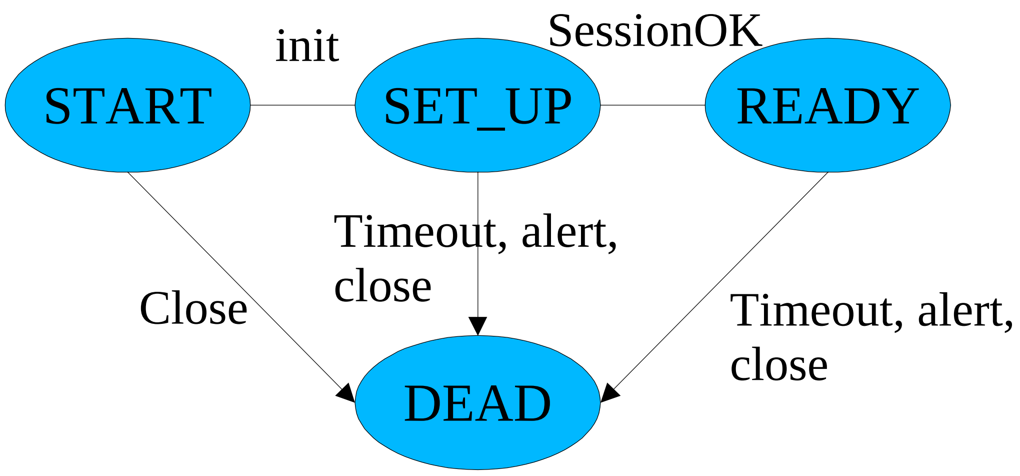

Figure 10   High level state transitions in SLS.

The events that trigger state transitions are:

- init: complete the initialisation of the session object and send a message to the other party to begin the key exchange.
- sessionOK: The key agreement has finished successfully.
- time-out: a time-out expired.
- alert: An fatal alert was handled/ sent forcing a shutdown of the session.
- close: a client forced termination of a session.

2.4.2 Key exchange

Key exchange is based on Diffie-Hellman key exchange protocol. First session key is agreed by this method. Further keys are then derived from this key. Properties of key exchange protocol used are in general as follows:

- Elliptic Diffie-Hellman key exchange is used instead of modulus exponentiation. Instead of choosing a group and checking its validity at the other end, we pick one of a pre-determined family of elliptic curves. Each party uses a random point on the curve as a private key, generates a second point from the first to send as a public key, and checks that the point he receives belongs to the same curve.

- There is no current support for multiple public keys of the same principal. This extension should be trivial by adding more than one signature in the handshake.
- Added tunnelling support: To allow the responder to notify in its first handshake message that it wants to relay the session.
- Randomised Session Parameter Indices (SPIs). To make SPIs to be hard to guess, to avert denial-of-service attacks. If SPIs are predictable, it is too easy to flood the client/server with fatal alert messages. At the same time, attention must also be given to non-authenticated alerts during the handshake.

Key exchange states are shown in figure 11. Messages sent are following:

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

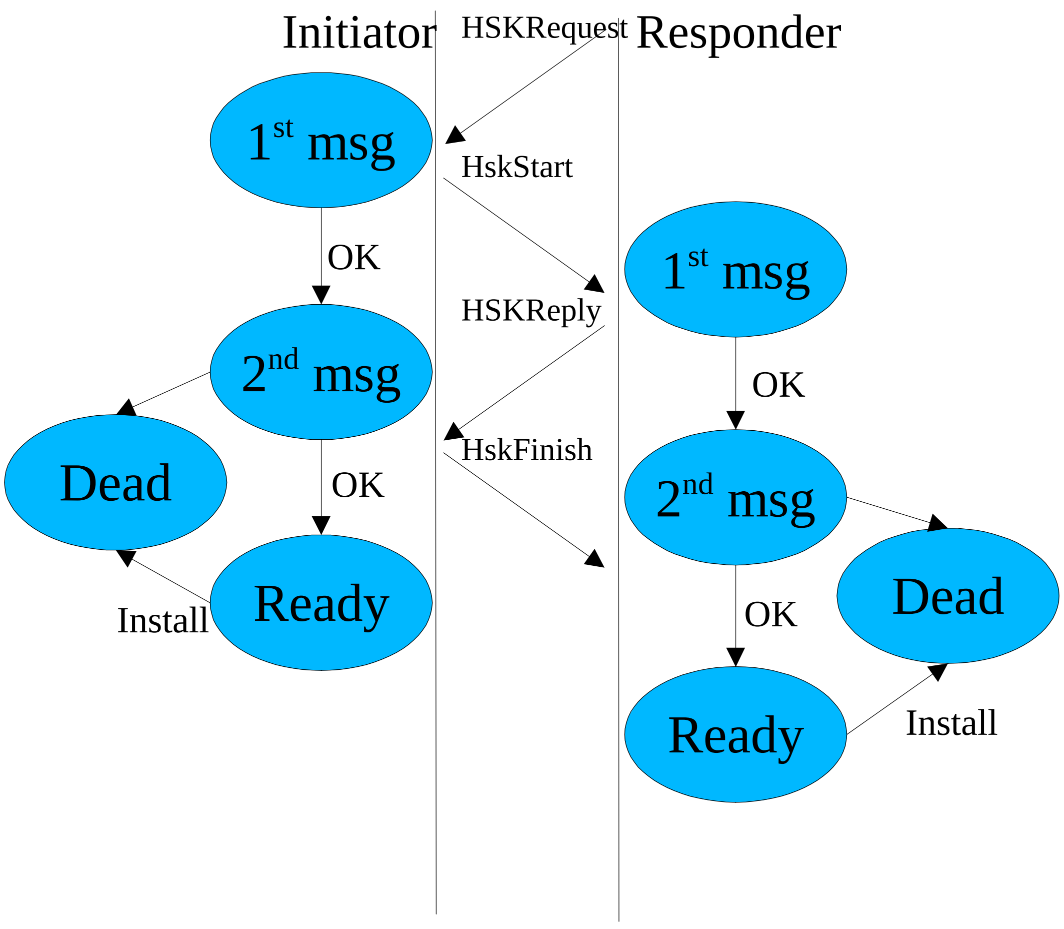

Figure 11 Key exchange states and messages.

3 CERTIFICATES USED IN E-SPEAK

Diffie and Hellman wrote in 1976 a paper [12] introducing public key infrastructure. In this paper certificates main function was to bind a name to keys or vice versa. Modern public key methods are described in book [2].

Diffie and Hellman thought they had radically solved a key distribution problem. "Given a system of this kind, the problem of key distribution is vastly simplified. Each user generates a pair of inverse transformations, E and D, at his terminal. The deciphering transformation, D, must be kept secret but need never be communicated on any channel. The enciphering key, E, can be made public by placing it in a public directory along with the user's name and address. Anyone can then encrypt messages and send them to the user, but no one else can decipher messages intended for him."

In 1988 X.509 recommendation was published as part of X.500. X.500 was planned to be global, distributed database of named entities. X.509 certificates were meant to bind public keys to distinguished names. X.500 distinguished names were globally unique names that could be used to reference entity. Originally certificates were meant to control who could edit certain X.500 tree branches. However X.500 will probably never be implemented as it was planned.

3.1 SPKI certificates

Two RFC papers [7] and [8] define SPKI certificates. They are developed by IEFT security area working group SPKI, in order to create standard certificates whose main purpose is authorisation. SPKI certificates bind either names or authentications to public/private key pairs. Requirement made to SPKI keys are:

1. Freedom to generate. Everyone can freely issue certificates and delegate access rights to them.
2. Issued rights can be transferred.
3. Authentication's can be freely defined and distributed.
4. Validity dates are clearly written on certificates.

5. Usage of public keys instead of names.
6. Provide a better mechanism for solving public key finding problem.

SPKI certificates use internally so called s-expressions as the standard format. However there is a draft [9] that proposes a way encode SKPI certificates in XML [14]. This might be more appropriate way since all messaging in the designed system is based on XML. The current implementation of e-speak supports only a s-expression format. Other reasons for using the XML encoding are:

1. XML has been quickly adopted.
2. Almost every device will have XML parser build in, in the future.
3. XML certificates can be easily verified against a schema sheet.
4. XML is both human and machine readable.
5. XML document can be easily converted using XSL.
6. There are two different commonly used APIs Document Object Model (DOM) and Simple API for XML (SAX), providing interface to XML processor. Developers can freely choose either one.

3.1.1 SPKI certificate format

Figure 12 describes SPKI certificate structure. Keys marked in the figure mean public keys. The public key 1 belongs to issuer of the certificate and can be used to open signature that verifies certificates integrity. Public key 2 belongs to subject or entity that uses the certificate. Subjects private key must be kept safe and is used to authenticate entity.

Delegation means that the subject can or can not delegate privileges granted by the certificate to someone else. Delegation is a boolean value. There is no control over the length of delegation chain. If delegation is allowed, then the subject who creates the certificates containing delegated access rights decides if these delegated certificates allow further delegation or not.

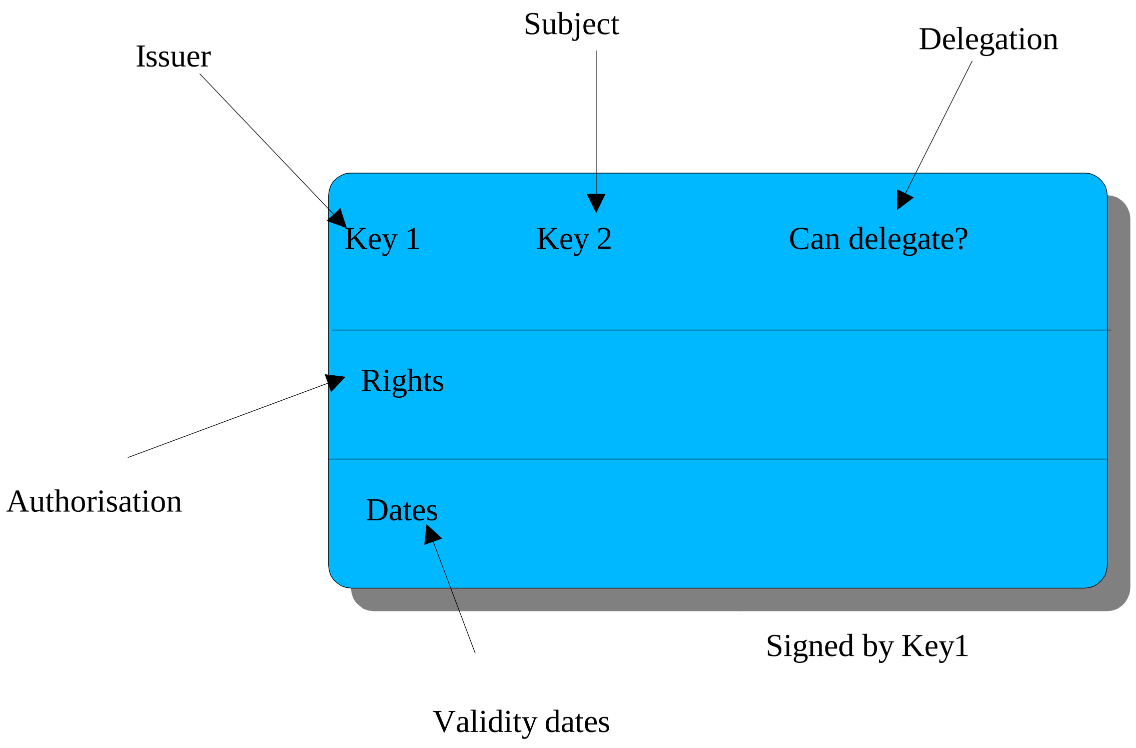

Figure 12   SPKI certificate structure.

Authorisations block contains all tags, defining privileges the certificate grants. Validity dates limit time that the certificate is valid and can be used. Other validity conditions can be defined. These include any on-line validity tests. In fact, if any on-line test are wanted they must be defined at the time of granting the certificate.

For example, if certificate is tested against some CRL (Certificate Revocation List), it must be declared here. The certificate is valid only, if all the defined on-line tests can be successfully completed. If for some reason the CRL cannot be read, the certificate would be assumed invalid. The certificate could be considered valid only after it has been successfully verified against a valid CLR.

3.1.2 SPKI compared to X.509 certificates

SPKI certificates use local names instead global ones. This means that certificates from different namespaces can have same name. A name therefore cannot be used as an identifier for the certificate. Theory of local names is shown in SDSI 1.0.

However an unique identifier is needed, if certificates are to be stored in databases. Public key is an ideal identifier. Whole certificate theory is based on an assumption that two different certificates will never have same public key.

An attribute certificate defined in X9.57 uses following mapping:

authorisation -> name -> key

This mapping probably requires two different certificate issuers. One is for name and the other is for authorisation. If either is subverted authorisation could be acquired improperly.

A basic SPKI certificate defines a straight authorisation mapping:

authorisation -> key

If someone wants to access a key holders name, for logging purposes or even for punishment after wrong-doing, then one can map from key to location information (name, address, phone, ...) to get:

authorisation -> key -> name

This mapping has an apparent security advantage over the attribute certificate mapping. In the mapping above, only the

authorisation -> key

mapping needs to be secure at the level required for the access control mechanism. The

key -> name

mapping (and the issuer of any certificates involved) needs to be secure enough to satisfy lawyers or private investigators, but a subversion of this mapping does not permit the attacker to defeat the access control. Presumably, therefore, the care with which these certificates (or database entries) are created is less critical than the care with which the authorisation certificate is issued. It is also possible that the mapping to name need not be on-line or protected as certificates, since it would be used by human investigators only in unusual circumstances.

Figure 13 show X.509 name certificate structure. It is clearly a lot more complex than SKPI certificate. An other advantage of SPKI is that is has not yet been burdened by compatibility issues caused by different versions.

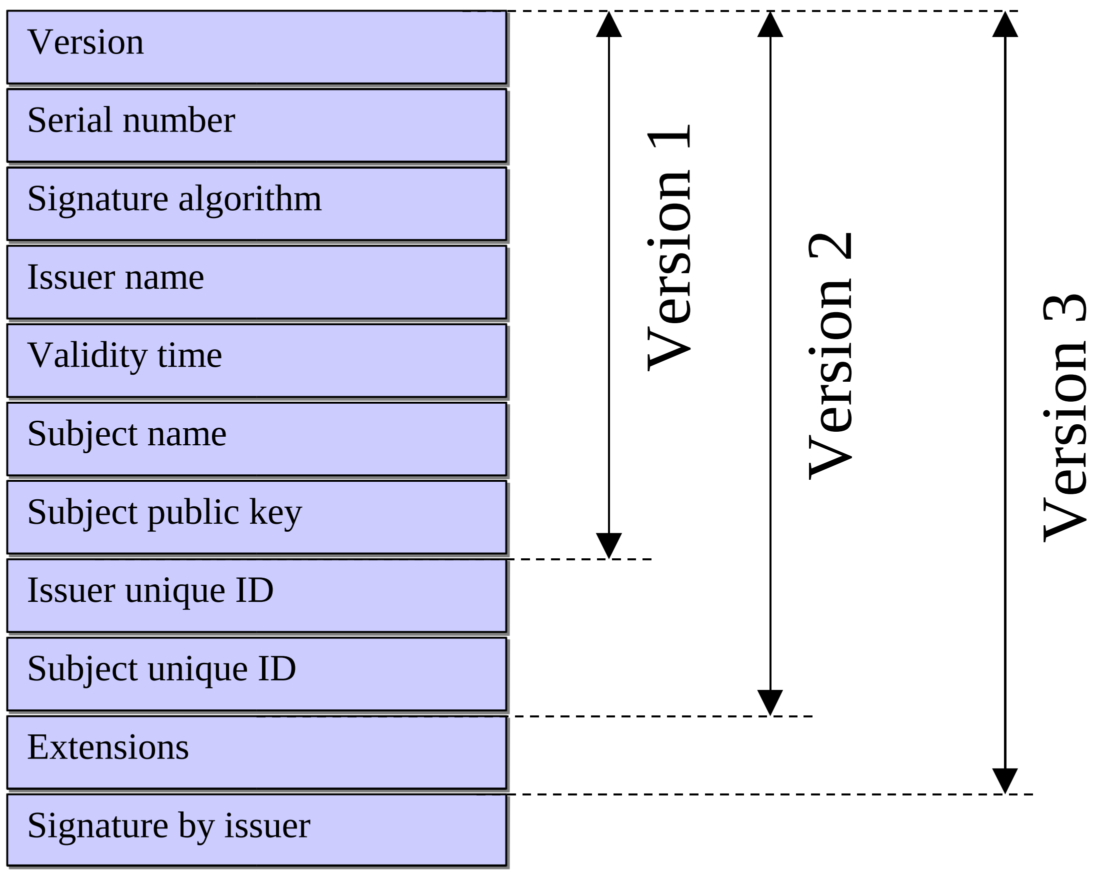

Figure 13   X.509 name certificate.

3.1.3 Delegation of authorisation in SPKI

SPKI defines a way to delegate certificates authorisation to other entities. Delegation requires no attention from original issuer of the certificate. Depth of delegation is controlled by a boolean value. Each delegator then must decide if delegated certificate can be delegated even further.

Access control lists become a lot simpler if delegation is properly used. ACL rely on <name,key> ID certificates to define a group of certificates and grant them access rights.

For example, consider firewall that uses certificates to grant authorisation. Without SPKI certificates each user would have his/her own certificate to identify user. ACL list of the firewall would contain all these certificates.

However using delegation firewalls ACL could contain only one entry, granting a single key access right. This access right could then be delegated to the users. User, of course, could no longer delegate this access right.

3.1.4 Validity conditions

SPKI requires all authorisation computation to be deterministic. Ordinary certificate revocation list (CRL) is not deterministic and thus not allowed in SPKI. However SKPI permits use of timed CRL. This means that when a certificate is referenced in CRL the following conditions must be fulfilled:

1. The certificate must list the key that will sign the CRL. The certificate may also list locations where CRL might be fetched.
2. The CRL must carry validity dates.
3. CRL validity dates must not intersect.

The certificate that uses a CRL must always be checked against a valid CRL. No CRL can show up as a surprise, meaning whether certificate uses or does not use a CRL must be decided when issuing the certificate.

Opposite to a CRL is called a revalidation. SPKI demands deterministic behaviour, so all conditions required from the CRL also affect a revalidation.

Optional validity conditions include not-before and not-after dates. On-line tests are considered to be validity conditions. In SPKI the issuer of the original certificate must specify the issuer of validity conditions. These two need not to be the same entity.

Every certificate revocation and revalidation list has certain non-zero time of validity, called a validity interval. It is not possible replace the list during this interval. No list can have zero validity interval, because it would mean that list is already out of date, when it is distributed. SPKI defines additional one time revalidation, if zero validation interval is required.

3.1.5 Tuple reduction

Tuple reduction means processing of certificates and other relevant information to acquire an authorisation result. Tuple means intermediate form of certificate that is no longer protected by a signature against changes. The tuple is only stored in computers memory, while certificate is checked. Three different entities participate in the authorisation computation.

1. A certificate: <name, key>
2. An attribute certificate of ACL entry: <authorisation, name>
3. An authorisation certificate or ACL entry: <authorisation, key>

Processing occurs in three stages. First certificates are verified. This includes checking certificate signatures and making all on-line validity tests. In the second stage all names are replaced with keys. The third stage is a final reduction to an authorisation result.

SPKI certificates map to so called 5-tuples. Components of 5-tuple are more or less the same as the components of certificate. However 5-tuple can be derived from other sources, like ACL entry or an other type of certificate. Components of 5-tuple are:

1. Issuer
2. Subject
3. Delegation
4. Authorisation
5. Validity

3.2 SDSI name certificates

SDSI name certificates [10], [11] are a way to link names to public keys. SPKI uses SDSI name certificates to find public keys based on names or other attributes. SDSI names consist of two parts. First is a chain of names, second an entity that defines them. The SDSI certificates map to an intermediate form called 4-tuples. Components in 4-tuples are:

1. Issuer
2. Subject
3. Name
4. Validity

SDSI certificate structure is described in figure 14.

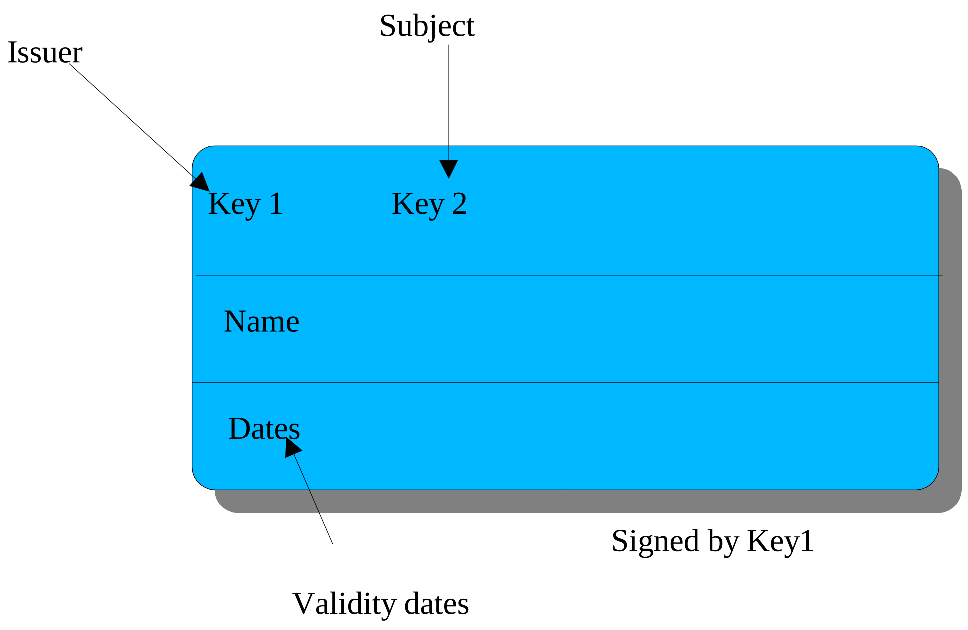

Figure 14 SDSI name certificate structure.

For example a name chain could be 'Mikko's brother's girlfriend'. This chain would then be defined by Mikko and consist of two certificates. A first certificate would be issued and signed by Mikko (key 1). Name defined, in this first certificate, is brother. The second public key present is associated with 'Mikko's brother' (key 2). The second certificate would then have to be issued by the owner of key 2 (Mikko's brother). Again it would define a name, this time the name girlfriend. A Public key associated with 'Mikko's brother's girlfriend' would be present as a second public key (key 3). Figure 15 describes certificates needed in this name chain.

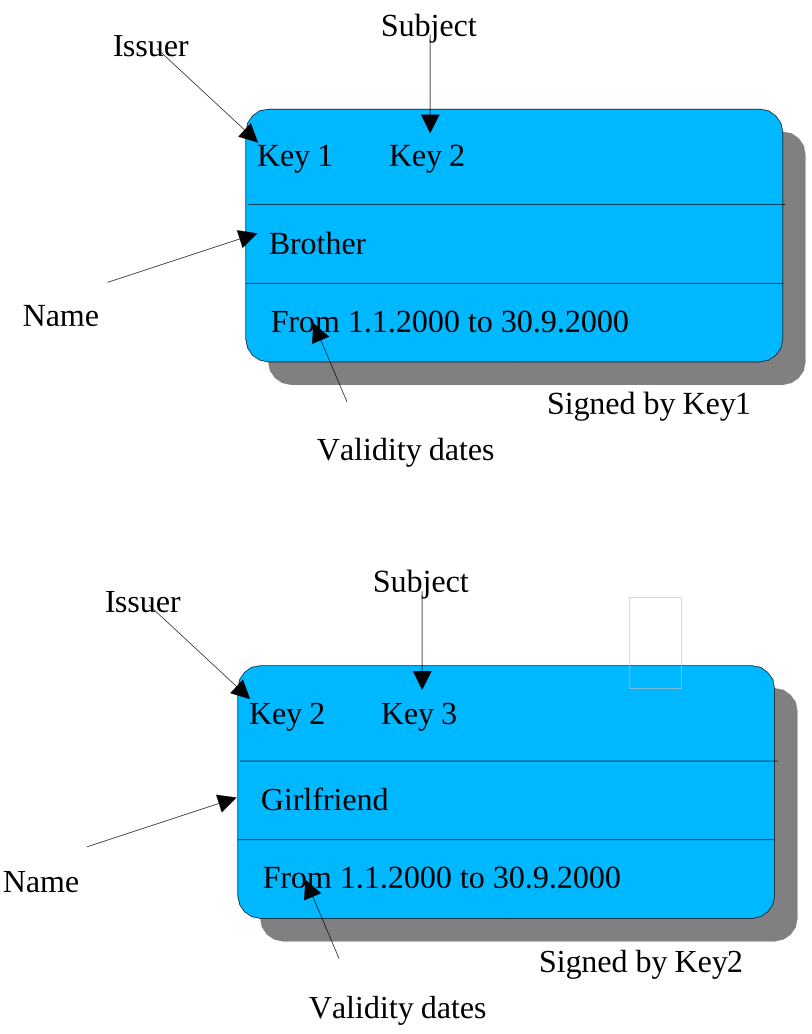

Figure 15   SDSI certificate chain.

As clearly can be seen names in SDSI certificates are not unique. A same name can exist as many times as necessary. The defining entity anchors name chain's end. Thus names are only valid for the issuer of name certificates. Names should also mean something concrete to their issuer. Same entity can even issue two or more SDSI certificates to the same name (different public keys). For example 'Mikko' can issue two certificates named brother, if he wants.

4 MICROSOFT'S .NET FRAMEWORK COMPARED TO E-SPEAK

Microsoft is developing a new WWW service framework called .NET. It contains all the current servers Microsoft has with enhancements to their functionality. .NET also provides a large set of new tools, along with the updated old tools.

.NET framework is currently under development and not even in beta testing phase. First beta releases can be expected near the end of year 2000. Most of the .NET enterprise servers are available as beta releases. However .NET has been designed to be integrated with next version of Windows 2000 called Whistler. At first the .NET framework only works with Whistler, but Microsoft has promised upgrades to at least Windows 2000 operating system.

One of the most clear advances in .NET is a Common Language Runtime (CLR). This enables easier interaction between components written with different Microsoft languages. Unified class library provides the same libraries into all used programming languages. Components written in C# (C sharp) can be used in VB.NET (.NET version of Visual Basic) program, as they would have been written in Visual Basic.NET, provided that only unified class library components are used.

4.1 Components of .NET

XML is strongly present in .NET framework. Microsoft's initiative SOAP [16] is a XML based remote method invocation language. XLM is also used widely elsewhere to transfer information between .NET components, enterprise servers and to import/export information to/from other manufacturers systems.

Major components in .NET framework are .NET enterprise servers. These are:

1. BizTalk Server 2000: Business orchestration and document handling.
2. SQL Server 2000: Database server.
3. Exchange 2000: Message transfer and collaboration.

4. Host Integration Server (HIS) 2000: Gateway to other non-Microsoft systems.
5. Commerce Server 2000: Web site design and management.
6. Application Server 2000: Tool for managing server clusters.
7. Internet Security and Acceleration Server 2000: Firewall and proxy.

Other part of .NET is a .NET framework. It provides application development tools, unified libraries and compiler/debugger tools needed to test applications.

ADO+ (Access Data Object) extends current ADO. It provides generalised object model for data. Business logic is then designed to use these objects. No direct writing of SQL is necessary. Older version of ADO was meant only to be used with relational databases. ADO+ has been developed to understand much wider variety of data sources. In fact any data source that can produce XML can be used in conjunction with ADO+

Framework also contains Visual Studio.NET development platform. Main languages to be used are C# and VB.NET. C++ and JScript are also present. Both VB.NET and C# use unified class libraries and are in fact two different syntax's of one programming language. This makes integration of programs written with these two languages possible.

Biztalk server 2000

Biztalk server is some sort of glue that is used to control and design operation of the system. It is also used to orchestrate long-term business transactions. This means that a flow of events when for instance an order form is processed is designed visually in BizTalk server. Server then oversees this process, making sure thing happen in right order and properly.

Biztalk server supports long lasting transactions, that can take days to complete. Ordinary transaction servers would easily be overloaded, since they keep all open transaction state in memory. BizTalk on the other hand stores transactions current state on disk, while waiting next step to complete.

XML is widely supported in Biztalk. All handled documents can be XML based, though other typed of documents can also be handled. Document delivery is embedded into Biztalk. It can send, receive and queue messages into variety of sources. These sources include at least MSMQ, e-mail and flat file on disk. It is also possible to write own message handling COM object.

Documents can be tracked as they are processed in server. It is always possible know in what stage document is in. If questions arise after document has been handled log files can be used to track, what actions where taken while processing the document. BizTalk also provides GUI tools needed to plan, implement and monitor the whole business system.

4.2 Comparison of .NET to e-speak based platform

When comparing .NET to e-speak one must remember that both are still under development, and changes to their architecture may yet occur. This comparison is written on basis of current version of these two platforms.

Comparison only lists things that are present in both systems. .NET tries to solve a whole lot more issues than e-speak is aimed for. For example, e-speak does not contain any development tools and .NET includes a whole IDE environment for application development.

Table 1 summarises features each system provides. This is general list, not trying to list every detail of each system. Only major features have been listed.

Table 1 E-speak compared to .NET.

Feature list	***e-speak***	***.NET***
Loosely coupled app integration	Yes	Yes
Document translation/transformation	Yes	Yes
XML support	Yes	Yes
Multi-transport	Yes	Yes
Fine grained access control	Yes	**No**
Dynamic firewall traversal	Yes	Yes
Cross-platform resource management	Yes	**No**
System management support	Yes	**No**
Security: Encryption & certificate support	Yes	**No**
Support for cross-community advertisement	Yes	Yes
Open source deployment of service engine	Yes	**No**
Negotiation	Yes	**No**
Introspection	Yes	Yes
Document delivery (QOS)	**No**	Yes
Tools	**No**	Yes
Legacy integration	**No**	Yes
Developer channels	**No**	Yes
Messaging specification	SOAP 1.1	SOAP 1.1

Differences between these two systems are not that great on the paper. In real life however these two systems look quite different. .NET is tightly integrated into Microsoft's operating systems. It is impossible to implement .NET system to any other operating system than Microsoft's. E-speak is not bound to any operating system, programming language or software vendor. This gives an advantage to e-speak when designed systems have a long life cycle and must provide flexible interoperability.

Both HP and Microsoft are large companies, and seem to be committed to develop their technologies further. However since .NET is not even in beta testing phase, it is difficult to anticipate where Microsoft is going to focus development next. HP currently seems to focus E-speaks development into management and administration tools. Security is also under consideration.

Access control is one of the most difficult problems an enterprise network has to solve. If not properly taken care of, severe problems will most certainly arise. E-speak security relies on a clearly documented and rather simple SPKI certificate model. SPKI certificates are still a very new technology. First writings appeared in 1998. SPKI certificates provide e-speak a very fine grained control over users and resources available to them. .NET has answer to security and authentication is a component named Passport. It provides web services user authentication. Hoverer its capabilities are not well documented at the time of writing of this document.

User authentication can always be implemented inside services. This means that each service must implement its own authentication methods, making them more complex. It also means that unauthorised users can create significant load on service by simply trying to access it. This forces the service to perform a new authentication procedure at each access. E-speak filters out invalid messages before they reach a resource handler.

.NET provides a lot of GUI tools, that make system development and management more easy. E-speak provides currently only a minimal set of tools, needed to configure an e-speak core. This might change in the future. However e-speak will probably never have the same amount of tools as .NET has.

.NET is tightly integrated system, that claims to be an effective and fast development platform. Microsoft provides some tools to integrate other vendors applications and legacy systems into new .NET based system. XML of course in itself is very easily transferred from one system to an other. This increases usability of old applications and components.

E-speak is also a compact platform, but it is meant to provides a lot less functionality than .NET. However e-speak is even more open system making integration with existing systems a lot simpler task. Open source implementation is very helpful when integrating old services into the new framework.

Security and access control are two areas e-speak has most clear advantage over .NET. .NET on the other hand provides GUI tools and ready made templates to manage and develop programs.

5 E-SPEAK TEST SYSTEM

This chapter briefly documents the implemented e-speak test system. The goal of this test was to verify the current functionality of e-speak. The test system included computers running under different operating systems and to cross platform connections were made. The structure of the test system and the configuration of e-speak are treated in the chapters 7.1 and 7.2. The implementation of the resource handler is discussed in the chapter 7.3. Finally the implementation of the client is discussed in the chapter 7.4.

5.1 Structure of test system

The test system is composed of two e-speak cores handling the message mediation. The cores are located on different computers A and B. The core A is located on the computer A running under Redhat 7.0 Linux operating system. The core B is located on the computer B running under Windows 2000 operating system. The client is located on the the computer C running under Mandrake 7.2 Linux operating system. The resource handler is running on the computer A with the core A. All components are run inside their own Java virtual machines (VM). Figure 16 describes the test systems layout. Firewall between the computers is configured to allow net traffic to ports e-speak uses (port 12346).

Network connections between computers are made through the local ethernet. A packet filter is used to verify the encryption of the transferred data between the core B and the client.

5.2 Configuration of e-speak cores

The E-speak cores were configured to advertise registered resources in ACITest group. Advertising service enables the client connecting to the core B locate the resource registered to the core A. Both cores are configured to listen to default port 12346.

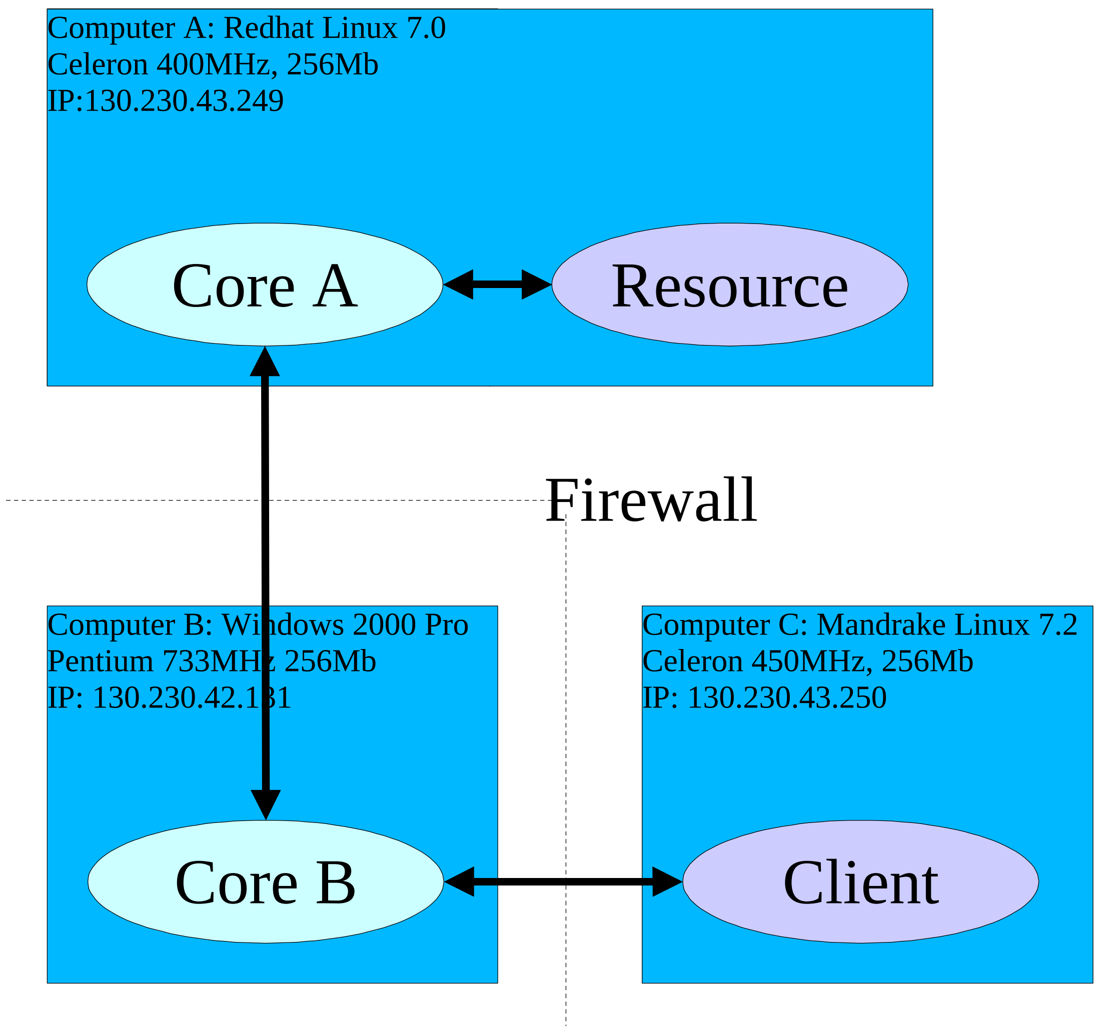

Figure 16   E-speak test system layout.

The cores are started with the following script:

```
[Tasks]
Start=Core

[Core]
Class=net.espeak.infra.core.startup.StartESCore
Args=-rep %espeak_home%/config/repository.ini -r -p TCP:12346
```

The script simply specifies the class used to start the e-speak core, the repository initialisation file and the TCP port number (12346) to listen to.

The advertising service responsible for providing the database for searches in multicore system, is started with the following script:

```
[Properties]

[Tasks]
Start=SAS

[SAS]
Class=net.espeak.services.advertise.ypserver.AdvertisingService
Args=-eshost localhost -esport 12346 -group ACItest -mport 1438 -beproto slp
```

The script specifies the class used to start the advertising service and the arguments passed to it. The arguments specify that the advertising service connects to the core running on localhost port number 12346.

Since no external directory service is specified, advertising service defaults to HP managed advertising directory service. This directory can be used to test advertise services, if a local directory is not available. The advertising service is basically a LDAP directory service. Since very little documentation of configuring an own advertising service was available at the time of making this test, HP's default advertising directory was used.

This test does not use authentication or access control. If authentication is required, each core, client and resource must have their own key pairs generated and proper SPKI certificates issued.

5.3 Test resource handler

A resource handler is a component that provides a service for a client. The resource handler can implement service's functionality internally or can connect to other service in order to acquire needed functionality.

Resource handler task in the test system is simply to return device data objects to client, when requested. The resource handler used EJB based database query, to create list of devices. Other EJB components are not part of this thesis, and will not be described.

E-speak core is used to mediate remote procedure call (RPC), and then to return results back to the client. The use of RPC requires that the called function is described in esidl files. Esidl is similar to Java interface definition. E-speak provides an esidl compiler, that provides necessary Java files, from esidl files. Passed parameter's and return values types must also be defined with esidl.

5.3.1 Implementation of resource handler

Resource handler implementation is simple. It uses libraries provided by e-speak, to handle connection to and messaging with the core. Resource handler Java code can be found in appendix 1: TestServer.java.

Test server does following things:

1. Connects to the e-speak core.
2. Creates a service element, assigns it an implementation (TestService).
3. Registers the service element with the core.
4. Registers the service element with the advertising service.
5. Starts the service element.
6. Exits.

Test server reads configuration information about the core it connects to, from a file name TestServer.prop. This file contains following definitions:

```
hostname = localhost
portnumber = 12346
community = ACItest
```

Hostname defines address of e-speak core. Portnumber defines the TCP port the core listens to. Community defines advertising service community used.

5.3.2 RPC definition

In order to make RPC, proper interface must be defined. E-speak uses esidl language, to define these interfaces. If parameters passed to or received from RPC are user defined classes a serialisation must be implemented. E-speak esidl compiler can be used to automatically generate this file. If own serialisation is required, then esidl compiler can be used to check the implementation of serialisation.

In the test system used RPC interface (getData method) is defined as is show in appendix 2. Appendix 3 contains actual server side implementation of getData method. Appendix 4 defines data type getData method return to client.

Esidl compiler generates several files from these definition. Files generated are similar to files used in Java RMI (Remote Method Invocation). RMI is described in [3].

Interface definition is generated into file TestServiceIntf.java. Stub file TestServiceStub.java created contains code for serialisation of service stub object, rerouting service method invocations through core and message registry initialisation. Message registry file TestServiceMessageRegistry.java contains code for registering types with message registry.

5.4 Test clients implementation

A client program first connects to a e-speak core. It then creates an search query, in order to locate the correct resource handler. The search query is based on a metadata the resource handler has registered with e-speak. This query is then used to locate a suitably named service. ESServiceFinder class used in the test client, returns a stub of the service to the client automatically. However the proper interface (TestServiceIntf.java) must be available to the client, when the code is compiled. It is not necessarily the same file which was generated by the esidl compiler. The interface definition must simply define the correct methods associated with RPC.

5.5 E-speak test results

The test system was simple to create. The Java interface definitions (esidl) and the corresponding Java files were easy to design and generate. The test server and client implementations were straight forward to code and e-speak libraries were correctly documented. Overall feeling was that the architecture of e-speak is suitable for both complex and simple systems.

The configuration of the cores were also easy to make. Some additional configuration would be needed if either the repository is made persistent or the advertising service local. A persistent repository requires currently Oracle database. An advertising service requires LDAP accessible directory service. Both components must be properly configured, which is a quite complex task. However these services need only be configured once. When they are operational no further changes are needed because of changes to services provided by the e-speak system.

The connection between core-client-service was proven reliable. Different operating systems caused no additional problems or configuration. Linux ngrep v1.38 program was used to view the packet traffic between the client and the core. It was clearly seen that no plain text data was transferred over this connection.

E-speak proved to be a reliable but rather slow method to make RPC. Even though no extensive performance analysis was done, it seems that e-speak generates a lot of network traffic. For example, traffic generated by a client registering to a core and making a single RPC call, resulting two devices, produced a file approximately 50kb of size. Recording was done with ngrep and the net traffic was stored in ASCII format. SLS encryption causes extra processor load. It was also found that a single RPC action took several seconds to complete. These are only approksimative results. More detailed performance analysis has been planned, but has not yet been performed.

6 BIBLIOGRAPHY

[1] Kassem. N. et al. 2000. Designing enterprise applications. SUN Microsystems Ltd.

[2] William Stallings. 1999. Cryptography and Network Security. Principles and Practice. Second edition. Prentice Hall.

[3] Ayers D. et al. 1999. Professional Java Server Programming. Wrox Press.

[4] E-speak architectural specification. 2000. Developer release 3.01. Hewlett-Packard.

[5] E-speak programmers guide. 2000. Developer release 3.01 Hewlett-Packard.

[6] Service Framework Guide. 2000. Developer release 3.01. Hewlett-Packard.

[7] C. M. Ellison et al. 1999. SPKI Requirements. RFC 2692. http://search.ietf.org/rfc/rfc2692.txt

[8] C. M. Ellison et al. 1999. SPKI Certificate Theory. RFC 2693. http://search.ietf.org/rfc/rfc2693.txt

[9] J. Paajarvi. 2000. XML Encoding of SPKI Certificates. http://search.ietf.org/internet-drafts/draft-paajarvi-xml-spki-cert-00.txt

[10] Ron Rivest and Butler Lampson. 1996. SDSI - A Simple Distributed Security Infrastructure. http://theory.lcs.mit.edu/~cis/sdsi.html

[11] Fredette, M. 1997. An Implementation of SDSI – The Simple Distributed Security Infrastructure. Master's thesis. Massachusetts Institute of Technology. http://theory.lcs.mit.edu/~cis/theses/fredette-masters.ps

[12] Whitfield Diffie and Martin Hellman. 1976. New Directions of Cryptography. IEEE Transactions of Information Theory. pp. 644-654. IEEE Press.

[13] Michael Kay. 2000. XSLT Programmer's Reference. Wrox Press.

[14] B. McLaughlin. 2000. JAVA and XML. O'Reilly.

[15] B. Schneier, D. Wagner. 1997. Analysis of the SSL 3.0 protocol. http://www.counterpane.com/ssl-revised.pdf

[16] Don Box, et al. 2000. Simple Object Access Protocol (SOAP) 1.1. W3C Note.
http://www.w3.org/TR/SOAP/

[17] JavaTM Servlet 2.3 Specification, proposed final draft. 2000. Sun Microsystems Ltd.
http://java.sun.com/aboutJava/communityprocess/first/jsr053/servlet23_PFD.pdf

Appendix 1: TestServer.java

```
package fi.tut.ad;

import net.espeak.infra.cci.exception.*;
import net.espeak.jesi.ESConnection;
import net.espeak.jesi.ESServiceElement;

public class TestServer {
    private String name = null;
    private String propFile = null;


    /**
     * Constructs an TestServer and assigns it
     * a name by which it can be discovered by clients.
     * @param pFile The property file name.
     * @param Name The name of Server.
     */
    public TestServer(String pFile, String name) {
        this.name = name;
        this.propFile = pFile;
    }

    /**
     * Entry point for the TestServer.
     * Constructs a new TestServer and then
     * starts it.
     */
    public static void main(String[] args) {
        TestServer server = null;

        if (args.length < 2 || args.length > 2) {
            System.err.println(usage());
            System.exit(-1);
        }
        server = new TestServer(args[0], args[1]);
        server.start();
    }

    /**
     * Returns a usage string indicating correct usage
     * @return A string indicating proper usage of a TestServer.
     */
    public static String usage() {
        String usage = "Usage: TestServer <PropertyFile> <ServerName>";

        return usage;
    }

    /**
     * Registers the TestServer with core and starts it.
     */
    public void start() {
```

```
try {
```

```
            System.out.println("**********************************");
            System.out.println("**  TestServer starting! ***");
            System.out.println("**********************************");
            //
            // Establish a connection to e-speak.
            // Uses default (localhost:port) values for connecting.
            //
            ESConnection connection = new ESConnection(propFile);

            //
            // A ServiceElement is a server-side representation
            // of a service.
            //
            ESServiceElement element = new ESServiceElement(connection,
                                                         this.name);

            //
            // The implementation object to which all incoming
            // method invocations are forwarded.
            //
            element.setImplementation(new TestServiceImpl());

            //
            // Register the service with the local repository.
            //
            element.register();

            //
            // Advertise the service with the Advertising service, so that
            // other cores can locate this
            //
            try {
                element.advertise();
            } catch (ESServiceException e) {
                System.out.println("Advertising service not found!");
            }

            //
            // Start servicing requests.
            //
            element.start();
            System.out.println("Started TestService succesfully!");
        } catch (ESLibException e) {
            System.err.println(e);
        } catch (ESInvocationException e) {
            System.err.println(e);
        } catch (NameCollisionException e) {
            System.err.println(e);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Appendix 2: TestServiceIntf.esidl

```
package fi.tut.ad;

import  net.espeak.jesi.ESService;
import  net.espeak.infra.cci.exception.ESInvocationException;

public interface TestServiceIntf  extends ESService {

   /**
    * The getData method
    * @exception ESInvocationException thrown when an error is detected during
    *            remote invocation
    */

    public Device[] getData() throws ESInvocationException;
}
```

Appendix 3: TestServiceImpl.java

```java
package fi.tut.ad;

import net.espeak.jesi.ESService;
import java.rmi.RemoteException;
import java.util.Enumeration;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestServiceImpl implements TestServiceIntf {

   static Context initialContext = null;
   static DeviceListHome home = null;

   /**
    * Method TODOJAVADOC
    *
    * @param
    * @return deviceList[]
    */
   public Device[] getData() {

      //Initialising EJB context
      try {
         initialContext = new InitialContext();
      } catch (NamingException e) {
         e.printStackTrace();
         System.exit(2);
      }

      // Lookup bean home
      String beanName = "DeviceListHome";
      try {
         home = (DeviceListHome) initialContext.lookup(beanName);
      } catch(Exception e) {
         e.printStackTrace();
         System.exit(2);
      }


      //Creating DeviceList
      try {
         DeviceList deviceList = home.create();
         return deviceList.getDevices();

      } catch(Exception e) {
         e.printStackTrace();
         System.exit(2);
      }

      return null;
  }
}
```

Appendix 4: Device.esidl

```
package fi.tut.ad;
import  net.espeak.infra.cci.messaging.ESSerializable;

public abstract class Device implements ESSerializable {
   private long _id = -1;
   private String _name = null;
   private String _desc = null;
   private String _locat = null;
}
```

Appendix 5: TestClient.java

```
package fi.tut.ad;

import net.espeak.infra.cci.exception.*;
import net.espeak.jesi.ESConnection;
import net.espeak.jesi.ESQuery;
import net.espeak.jesi.ESServiceFinder;
import java.util.Properties;

public class TestClient {

    private String name = null;
    private String propFile = null;


    /**
     * Constructs an Client.
     *
     * @param pFile The property file.
     * @param name The name of the Server
     */
    public TestClient(String pFile, String name) {
        this.propFile = pFile;
        this.name = name;
    }

    /**
     * Entry point for the Client.
     *
     */
    public static void main(String[] args) {
        TestClient TestClient = null;

        if (args.length < 2 || args.length > 2) {
            System.err.println(usage());
            System.exit(-1);
        }

        TestClient = new TestClient(args[0], args[1]);
        TestClient.start();
    }

    /**
     * Provides the usage string for TestClient.
     * @return A string indicating proper usage of an TestClient.
     */
    public static String usage() {
        String usage =
            "Usage: TestClient <PropertyFile> [Server]";

        return usage;
    }

    /**
```

```
     * Starts the TestClient. Finds an TestServer and sends it
     * a String and checks if the returned string matches the
     * string that was sent.
     *
     */
    public void start() {
        try {

            //
            // Establish a connection to e-speak.
            //
            ESConnection connection = new ESConnection(propFile);

            //
            // The interface of the service that the client is looking
            // for. This needs to be specified so that the finder can
            // return a stub that implements the appropriate methods.
            //
            String interfaceName = TestServiceIntf.class.getName();

            //
            // Give the name of the interface to the Finder.
            //
            ESServiceFinder finder = new ESServiceFinder(connection,
                interfaceName);

            //
            // Construct the search query
            //
            ESQuery query = new ESQuery("Name == '" + this.name
                + "'");

            //
            // Look for a server that has a matching name.
            //
            TestServiceIntf TestService = (TestServiceIntf)finder
.find(query);

            //
            // Get a DeviceList from the server.
            //

            Device[] dl = TestService.getData();
            for ( int i=0;i<dl.length;i++ ) {
                System.out.println("\nDevice number: "+ i);
                System.out.println("Device id: "+ dl[i].getId());
                System.out.println("Device name: "+ dl[i].getName());
                System.out.println("Device description: "+ dl[i].getName());
                System.out.println("Device Location: "+ dl[i].getName());
            }

            System.out.println("\nReceived DeviceList succesfully!\n");
            System.exit(1);


        } catch (ESLibException e) {
            System.err.println(e);
        } catch (LookupFailedException e) {
            System.err.println(e);
        } catch (ESInvocationException e) {
            System.err.println(e);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```