

HP Web Services

e-Speak High Availability

*Release A.03.14.00
August 2001*



ICOPYRIGHT NOTICE

© 2001 HEWLETT-PACKARD COMPANY

To anyone who acknowledges that this document is provided "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTY: permission to copy, modify, and distribute this document for any purpose is hereby granted without fee, provided that the above copyright notice and this notice appear in all copies, and that the name of Hewlett-Packard Company not be used in advertising or publicity pertaining to distribution of this document without specific, written prior permission. Hewlett-Packard Company makes no representations about the suitability of this document for any purpose.

Windows and Windows NT are registered trademarks of Microsoft Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Linux is a trademark of Linus Torvalds. All other trademarks are the properties of their respective owners.

Hewlett-Packard Company

Web Services Operation

14050 Ridgeview Court, Cupertino

California, 95014

Contents

1.0	Introduction	1
1.1	Objective.....	1
1.1.1	Background	1
1.1.2	Approach	1
1.2	Requirements	2
1.2.1	High Availability Requirements	2
1.2.2	Scalability Requirements	3
1.2.3	Factors Limiting e-Speak HA and Scalability.....	3
1.2.3.1	Resource Naming Scheme	3
1.2.3.2	Data Replication.....	3
1.2.3.3	E-Speak Configurations	3
1.2.3.4	Fault Detection	4
1.2.4	Factors Supporting e-Speak HA.....	4
1.2.4.1	Core Can be Started with Logical Server Name	4
1.2.4.2	Core Recovers from Temporary Unavailability of Database.....	4
1.2.4.3	Message Delivery Status	5
2.0	Proposed Solutions	6
2.1	Failover Cluster for Databases	6
2.2	Failover Cluster for e-Speak Core	7

2.2.1	Changes in e-Speak Core	8
2.2.2	Pros	8
2.2.3	Cons.....	8
2.3	Reconnection to Core	9
2.3.0.1	Connection Property File	9
2.3.0.2	Constructor	9
2.3.0.3	ESConnection.reconnect()	9
2.3.1	Pros	10
2.3.2	Cons.....	11
2.4	Connection Load Balancing	11
2.5	Analysis of a Core Cluster Approach.....	11
2.5.0.1	Client Context	13
2.5.0.2	Service Provider's Context.....	13
2.5.1	Changes.....	13
2.6	Scalability through configurations.....	14
2.6.1	Connection Pooling	14
2.6.2	Partitioning of Users by Core.....	14
2.6.3	Problems	15
2.6.3.1	Name resolution	15
2.7	Common Repository for Multiple Cores	15
2.7.1	Core Groups	15
2.7.2	Cache Consistency	16
2.7.3	Transaction Management	18

2.8	Multiple Service Registrations	19
2.8.1	Limitations	20
2.8.2	LocatorService Function	21
2.8.3	Interface	21
2.8.4	Enhancements	22
2.9	Group-wide Resource Registration	23
2.10	Handling Service Failure	23
2.10.1	Problems to be Solved	23
3.0	HA and Scalability study	25
3.1	Single Point of Failures (SPOF) in e-Speak Based Systems.	25
3.2	Service Provider – Core Interaction	30
3.2.1	Calls that Affect Data in Core	30
3.2.2	Method Invocation	32
3.2.3	Read Requests	32
3.3	Problems to be addressed in HA Solution	33
3.3.1	ESName Alias	33
3.3.2	Data Integrity	34
3.3.3	Fault detection	34
3.3.4	Recovery	34
3.4	What Does not Scale?	34

4.0	ServiceGuard Configuration	37
4.1	Pre-requisites/Assumptions	37
4.2	Configuration steps (using ServiceGuard commands).....	37
4.2.1	Create a package	37
4.2.2	Verify the package configuration	39
4.2.3	Copy the generated files to other nodes	40
4.2.4	Start the cluster and package	40
4.2.5	Check if the package has started	40

E-Speak High Availability

1.0 Introduction

1.1 Objective

This document identifies e-Speak high availability (HA) and scalability requirements, and proposes solutions for enhancing both. Also covered is configuration information for using ServiceGuard for e-Speak HA.

1.1.1 Background

Mission critical e-services using e-Speak are being deployed by customers around the world. E-Speak is a critical component in these configurations. Therefore, any downtime in e-Speak can cause disruptions of these e-services. E-Speak version 3.0.x based systems have several single-point of failures (SPOF) that can bring down an e-Speak based system. Recovery from these failures, in many cases, involves restarting the e-Speak core, services and clients. Section 3 describes SPOFs, faults and recovery procedures.

1.1.2 Approach

Identify availability and scalability problems in current e-Speak based configurations.

Identify single-point of failures possible in e-Speak based systems and suggest configurations and API changes to achieve high availability.

- Identify configurations and API changes to achieve high availability in e-Speak based systems.
- Identify configurations and API changes to achieve scalability in e-Speak based systems.

1.2 Requirements

1.2.1 High Availability Requirements

E-Speak clients is able to access e-Speak services even when e-Speak components fail or are temporarily not available.

E-Speak HA requirements from client and service perspectives are:

- 1** E-Speak shall provide API and configuration options such that clients will be able to use e-Speak after failure of one or more e-Speak components. These facilities can be in the form of redundancy in the e-Speak component, or a retry facility built in e-Speak, to recover from failures.
- 2** E-Speak based client will not have to stop and restart after failure of client's core.
- 3** Service providers do not have to restart their services in the case of failure of the core.
- 4** E-Speak based client will be able to use e-Speak service after failure of service's core.
- 5** E-Speak will reliably deliver messages sent from client to service and their responses.
- 6** E-Speak will ensure data integrity of data residing in e-Speak (metadata, folders, accounts, public and private RSD, etc.) if a failure occurs in one or more e-Speak components.

A commonly used technique for increasing HA is by providing redundancy in the system. Redundancy is typically provided in a system by using clustered configurations, such as HP MC/ServiceGuard on HP-UX, Microsoft Cluster server on NT, etc. These clustering solutions are of the "failover" type because they provide HA by managing redundancy in Server hardware, Network interfaces, Storage (Mirrored disks, with multiple access paths). The above-mentioned clustering solutions provide APIs for making applications HA, using cluster hardware and software facilities.

In the case of planned or unplanned outage of a server, cluster software will move the IP address and storage to a standby server, so that the affected application can continue execution on another server.

1.2.2 Scalability Requirements

E-Speak scalability requirement is that an e-Speak based system shall serve large numbers of users (numbers TBD) without performance degradation. Some of the customers will be using e-Speak with 40,000 concurrent connections.

Supporting a large number of concurrent connections means the ability to maintain thousands of concurrent, long-lived connections. Applications that require thousands of concurrent connections require thousands of concurrent threads and sockets from their JVM. Threads and sockets are system resources and, as such, are finite commodities. Moreover, they consume other system resources such as virtual memory. Once the thread, socket, or memory limit is reached, no additional connections can be established.

A single e-Speak core has a limitation on the number of connections that can be supported with acceptable performance. Current known limit is 4,000 connections (to be verified). Therefore, a service provider will have a finite limitation on the number of clients served, due to system resources.

Section 2 provides some suggestions on enhancing e-Speak scalability.

1.2.3 Factors Limiting e-Speak HA and Scalability

1.2.3.1 Resource Naming Scheme

ESName or URL identifies an e-Speak resource. ESName is in the form of `es://<host>:<port>/<id>`. ESName has information on the path to the resource, i.e., core's host name and port number. A resource identified by the ESName is managed by the core specified in the ESName. Messages for the resource get routed through the core. If a core is not available, or is not accessible from the client, the resource is not accessible.

The challenge for an HA solution is to provide an alternate path of accessing the resource, considering that the access path of a resource is built in to ESName.

1.2.3.2 Data Replication

To achieve scalability using resource replication, it may be necessary to register resources with multiple cores or replicate data to multiple cores. This means that the metadata of resources will be present in multiple cores. Maintaining data integrity between multiple repositories is a potential problem. However, resource replication puts additional load on the system, thus bringing down system throughput.

1.2.3.3 E-Speak Configurations

E-Speak provides a lot of configuration options. HA and scalability solutions are ineffective for some of the features.

- Cores can be started with persistent or in-memory repository.
- Transient resources do not live across core reboots.
- Transient resources live only during the life of a connection

HA solutions will address recovery procedures to recover from failures in e-Speak. In many cases, recovery is done by restarting components or by reestablishing connections. Transient resources may be lost during recovery.

Core and advertising services can be configured without a persistent back-end. In these configurations, core and advertising services cannot be brought to same state before failure.

1.2.3.4 Fault Detection

E-Speak does not have a built-in monitoring mechanism for detecting the status of components (cores, services). If a request times out, translating this error to an exact cause is difficult. Timeout can occur because of load on the core, network failure, service failure in the middle of work, a dropped message or load on the service. This makes recovery difficult.

1.2.4 Factors Supporting e-Speak HA

1.2.4.1 Core Can be Started with Logical Server Name

E-Speak core uses a 'hostname' call to get the server name. Host name is important for e-Speak, as it becomes part of the ESName of the resource. Failover clusters perform IP migration and use the logical server name, instead of hostname.

To solve this problem we can use the property "net.espeak.infra.core.virtualhostname" in the e-Speak configuration file, to start a core with a logical server name.

1.2.4.2 Core Recovers from Temporary Unavailability of Database

Temporary database non-availability is taken care of by e-Speak core. If the backend database is temporarily unavailable for any reason, a configured number of attempts for configured error codes are made by the core to reestablish the connection to the backend database.

The following properties can be used/modified in the e-Speak configuration file for this purpose:

- 1 `net.espeak.infra.core.repository.Repository_Params.JDBCConnectionRetrials`
- 2 `net.espeak.infra.core.repository.Repository_Params.JDBCConnectionErrorCodes`

NOTE: (For more information please see the e-Speak configuration file)

1.2.4.3 Message Delivery Status

Although e-Speak does not guarantee delivery of messages from clients to services, this can be achieved by using RMS (Reliable Messaging) feature.

2.0 Proposed Solutions

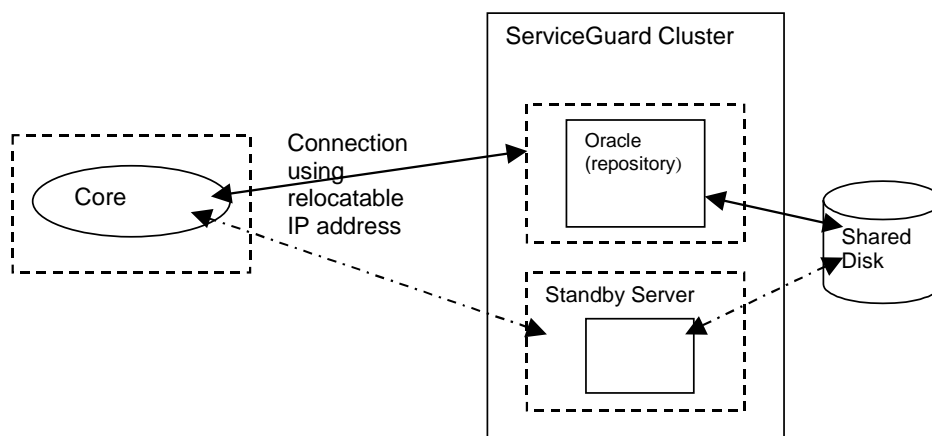
This section describes solutions to address availability and scalability problems.

One of the HA solutions is to run e-Speak on standard failover clusters for achieving high availability. Also, running multiple e-Speak cores in an administrative group can be considered as a cluster of e-Speak cores. However, proposed solutions (given below) do not need a concept of a cluster. Hence, topics of cluster formation and membership determination are not addressed in this document.

2.1 Failover Cluster for Databases

Database non-availability can result in restart in many components of an e-Speak based system, resulting in down time.

Failover cluster should be used for database HA. The Oracle server will reside on a failover cluster. Failures in server, network, storage or the Oracle software will result in restart of the Oracle server or migration of the Oracle server to the standby server. Database clients will have to reconnect to the database after database server restart.



Temporary database non-availability is taken care of by the e-Speak core. If the backend database is temporarily unavailable for any reason, the configured number of attempts for the configured error codes are made by the core to reestablish the connection to the backend database.

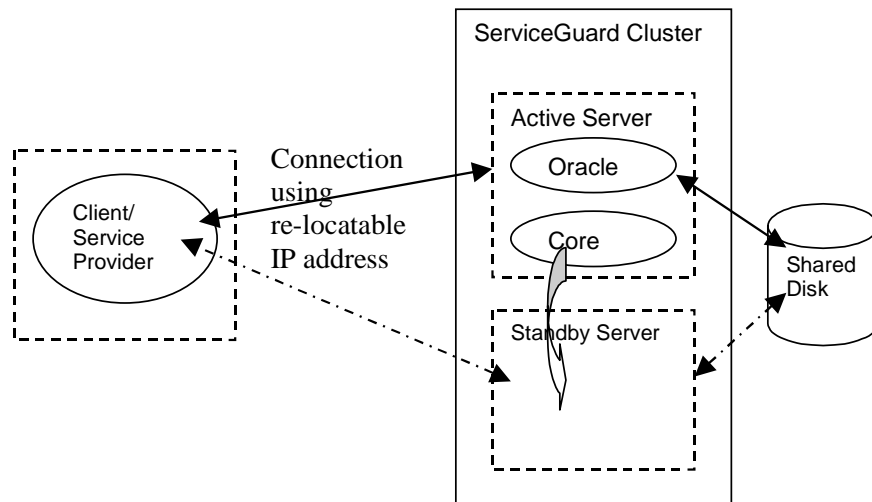
When considering this approach, some of the issues to be resolved are:

- 1 Identification of database transactions done from e-Speak and retrying incomplete database transaction
- 2 Ensuring consistency of cache and database for transient resources
- 3 Exception handling during the period database is not available
- 4 Configuration file changes for adding retry interval for database reconnection
- 5 Possibility of using Oracle Parallel Server (OPS). If Oracle parallel server is used as a persistent repository along with Level 2 JDBC driver (thick driver), e-Speak can benefit from Oracle's Transparent Application Failover feature. However, OPS installations are complex and costly. OPS can run only on a clustered system.

2.2 Failover Cluster for e-Speak Core

Using a failover cluster for running e-Speak core will reduce downtime. This solution will involve some downtime, and will involve restarting core and services; however, this approach provides one way of reducing complete system outage, without modifying e-Speak APIs and services.

In this solution, e-Speak is made cluster-aware by writing cluster specific agents. Cluster software will detect hardware; software and network problems and it will migrate IP addresses and disk ownership to standby servers. Under control of cluster software, e-Speak will be able to migrate to a standby server.



In this solution, migration time is of the order of minutes (1 to 5 minutes). Database and e-Speak can migrate to standby server independently. Clients and service providers will connect to core using logical server name (re-locatable IP in ServiceGuard terminology).

2.2.1 Changes in e-Speak Core

E-Speak shutdown: Command line program to shutdown core and other services (advertising service, distributor) will be necessary to manage e-Speak. ServiceGuard will use this command to shutdown e-Speak, before migrating it to the standby server.

E-Speak health monitor: Facility to monitor health of core: ServiceGuard depends on application monitors to monitor health of the application. It will be necessary to develop programs to monitor health of e-Speak such as hung core, failed core, resource problems, etc. Though there are facilities in e-Speak that can let the client applications (in this case, ServiceGuard) know if the core is dead or alive, a script is required to inform ServiceGuard about these facilities.

2.2.2 Pros

- Increased availability of e-Speak core using off-the-shelf components
- Mission critical applications are likely to use clusters with HA databases. The same failover cluster can be used for running e-Speak core.

2.2.3 Cons

- This solution is a platform-dependent solution. Solutions on HP-UX and Windows NT will be different solutions.
- Failover clusters provide high availability by having redundant resources, thus increasing cost of the solution.
- Administration and management of a cluster is complex.
- This solution does not address scalability.

2.3 Reconnection to Core

The `ESConnection` object holds critical information about the connection from client to core. The information includes security context, core communication channels, configuration values, user's context, etc. The `ESConnection` object reference is held by the `ESAccessor`, the `ESServiceElement`, and by other objects created by clients and services. Information stored in the `ESConnection` is used when exchanging messages via core. It is desirable to recover from connection failure by reconnecting to the same or alternate core, without destroying the `ESConnection` object.

However, whenever a connection to core is closed, transient resources created using that connection are destroyed. Services or clients that create transient resources will not be able to use this facility.

To implement this scheme, changes will be required in the Connection property file, the `ESConnection` constructor, the `ESMessenger` class, and by the addition of a method to reconnect.

2.3.0.1 Connection Property File

Add the following configurable options, in the form of properties:

```
esurl <url> [ <url> ]..
Autoreconnect=True/False (default false)
Reconnect_retry_attempts=<n> // home may retry attempts
reconnect_timeout=<seconds>
```

2.3.0.2 Constructor

Currently, the `ESConnection` constructor does not use timeout when creating a client channel. The constructor can use connection timeout when creating a client channel and try to connect to an alternate core, if so provided in the connection property file. The constructor will attempt connection to alternate cores listed in `esurl` list until client channel is created successfully. The connection property file can specify a connection retry policy that can include infinite timeout.

2.3.0.3 `ESConnection.reconnect()`

A new method 'reconnect' can be added that will close the current connection to core and attempt to reopen a connection to the same core. The `reconnect()` operation will close the connection and open a new connection, using same parameters. Note that multiple threads may hold reference to `ESConnection`; it is likely that they hold some of the information that will be made obsolete by `close()`.

By trapping `socketException` (or other equivalent exceptions that indicate closing of connection between client and core) in `EServiceMessenger`, `ESThreadPoolManager` and in other classes that send messages to core, we will be able to attempt reconnect to core without shutting down service.

When reconnect is done, it is assumed that transient information that existed in core no longer exists.

Refer to the list in the section, Service Provider – Core Interaction, for classes that sent messages to core, that can potentially catch socket exceptions.

Some of the possible classes are:

`ESMessenger.sendMessage()` can catch an exception, indicating core failure, if it occurred while sending a message to core. This method can reconnect and resend the message.

`ESThreadPoolManager.run()` can catch `SocketException` and reopen `ESConnection`. This will permit services to survive across core reboot, without needing core restart.

Alternatively, the programmer can catch `SocketException` and can call `ESConnection.reconnect()`, if it is suitable in that application.

We can consider `ESConnection.reconnect(new coreURL)`, thus allowing connection to a different core than earlier.

Wherever a socket exception is caught, message delivery can be retried. The reliable messaging project will probably address the issue of when a core failed, whether the message is delivered to the destination or not.

In the case of method invocation, it is a tricky decision, whether to do retry or not; even if a message was delivered to service, client is not aware of the status or execution results. Service will not be able to deliver results to the caller, because return addresses put in the message are no longer valid. If a service is capable of handling duplicate messages or is an idempotent service, duplicate messages will not have any side effect.

2.3.1 Pros

This solution will simplify programming logic in clients and services, by providing an API for recovery from core failure.

2.3.2 Cons

Automated message retry may not work with all types of services, especially if the service is not stateless. Programmers will have to be careful when using this facility.

2.4 Connection Load Balancing

Currently JESI has the ability to discover cores using IP multicast. This feature is not documented in the manuals. This feature can be extended to perform connection load balancing. Customers who do not maintain context information in e-Speak (accountname, folders) can use the core discovery feature.

The `ESConnection` constructor will send multicast messages to the multicast address specified in connection property file. Cores that get multicast messages will send information on resource availability in their multicast reply. `ESConnection` can use information in the multicast reply to perform load balancing.

2.5 Analysis of a Core Cluster Approach

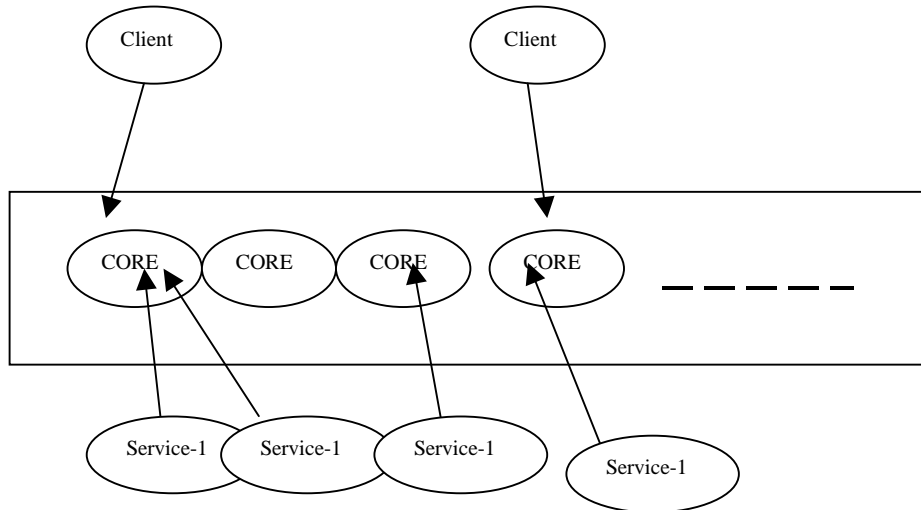
Desired behaviour of an e-Speak based system is as follows:

- A client can connect to a core out of a pool of cores to use a service that is available within same core or is available in some other core.
- Service provide is able to register service replicas with multiple cores, thus creating multiple paths to reach to its service.
- Ease of use of updating metadata of srevice replicas and managing (startup, shutdown) of services.
- Load balancing based on availability of free resources in cores.

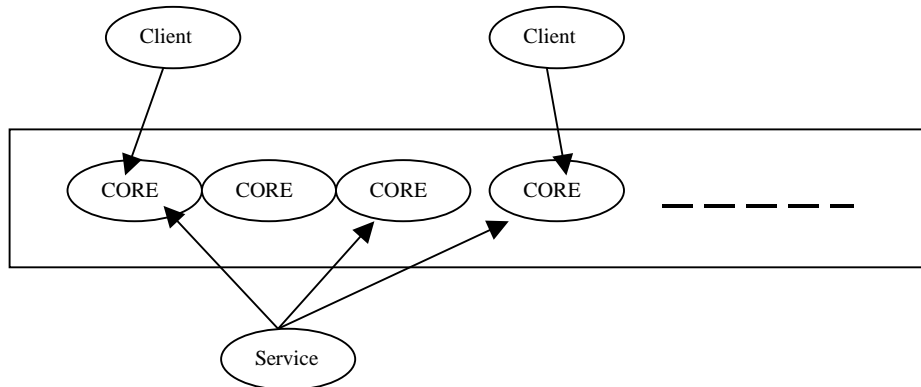
We have not proposed a solution based on this approach; however, this approach has been analyzed and high level changes have been identified in order to implement this approach. Further analysis on this approach is necessary to establish its feasibility.

Some of the use cases for the approach are:

- 1 Multiple resource instances all logically representing a single resource (e.g. Service-1 in the following figure). Instances can be present in the same core or different cores.



- 2 Single instance of a resource, which is reachable via multiple paths (i.e., cores).



To allow both of these resource representations, changes may be necessary in the resource naming scheme. To know more about core clustering, please see the document titled “**Core Clustering**” (version 1.1), authored by **Doug Myers**.

2.5.0.1 Client Context

When a client establishes a connection to a core, the following context is created for the user:

```
USER_ACCOUNT
Folder and their state
Inbox
Outbox
```

In short, a client stores context information in core, that can be used by the client in a subsequent session. The challenge is to make this information.

2.5.0.2 Service Provider's Context

- Services
- Vocabularies
- Contracts
- Resource Description - metadata
- Security context
- USER_ACCOUNT
- Folder and their state
- Inbox
- Outbox

Any resource that is created in a core is identified by an ESName that contains core and port. Context is backed up in persistent repository that is accessible by one and only one core. Core understands remote resource handlers and resource proxies, that redirect messages from one resource to another.

The challenge is to establish the relationship between multiple ESNames, such that resources can be identified as replicas.

2.5.1 Changes

This solution may require changes in several areas, namely:

- **ESName**: add concept of a global name that can represent multiple ESNames.

Resource replication: replication of a resource among cluster members.

- **Core:** Metadata replication.
- **JESI:** clusterwide registration; service replica API for creation, lookup, and management of replicas message redirection; load balancing.
- **Cluster membership management**
 - Cluster formation
 - Maintaining consistent view of membership among all the members
 - Heartbeats or similar scheme to check availability of cluster members
 - Handling cluster membership changes
- **E-Speak management:** To manage clusters
- **Reliable fault detection**

2.6 Scalability through configurations

Need to serve 40,000 concurrent users for SWATCH

Need to serve 1500 concurrent users for Citibank

2.6.1 Connection Pooling

Core has a limitation on the number of simultaneous connections.

When number of users > max. number of connections possible, use connection pooling (as done by Web Access and use **ESConnection.switchPD** to overcome problem of number of connections.

This can increase service response time, because multiple users will have to share a connection and wait for completion of the current set of requests.

2.6.2 Partitioning of Users by Core

Distribute load among multiple cores. Use core discovery and user partitioning to distribute connection load among multiple cores.

2.6.3 Problems

Environment of a client (folder, account, and vocabulary view) is not available in alternate core. Resource replication should solve this problem. Resource replication involves propagating changes from one core to other cores, whenever a resource is mutated. Core creates an event indicating resource mutation.

2.6.3.1 Name resolution

If a client application is using folders to store ESUrls, it is necessary to map ESNames of resources from the failed core to the surviving core. This problem has to be solved by some means of establishing ESName equivalence.

This solution addresses stateless services or stateful services that will manage state information. The proposed scalability solution involves multiple cores and services registration with multiple cores but using an identical Service Identifier.

2.7 Common Repository for Multiple Cores

Please see the document titled “**Core Clustering**” (version 1.1) for more details. This section analyses the requirements and issues involved in solving the common repository problems.

There are three big issues connected to the common repository for multiple cores:

- 1** how multiple cores specify that they want to share their repository in the first place
- 2** issue of cache consistency
- 3** issue of handling simultaneous updates of the same data from multiple cores in a manner that insures data consistency

2.7.1 Core Groups

Today each core has its own private set of metadata. Cores can use the same database but all the data in the database is tagged with a core id that indicates which core this data belongs to. Therefore if one service is registered in two cores which are both using the same database, there would be two sets of metadata for this service, one for each core. The existing functionality is useful because many cores, which may not have anything to do with each other, can take advantage of a single database server for persistence. Ideally, we want the customer to be able to decide which cores should share data and which should not. We can achieve this by having data within the database tagged by a core group id rather than just a core id.

Then, each core that wants to share data would be in the same group, using the same core group id. By default, a core would get a unique core group id so that all data for this core is private, which is how we operate today. Optionally, a core could be configured to use a core group id, which is also used by other cores. This would be specified either through one of the management tools, or possibly in the `repository.ini` file.

2.7.2 Cache Consistency

Since each core has its own cache, how does one core's cache gets updated after another core updates the database? There are three possible alternatives. The first is to eliminate the cache and always query the data directly from the database. This is not a good solution since it will seriously degrade performance. An only slightly better approach may be to always query the database to see if anything of interest has changed. It is probably possible to make this query cheaper than querying all the actual data required. In this case, the cache could be of some value if things don't change very often. If they do, it will be more costly since there would be two queries instead of one: one to check if anything has changed, and one to get the change. Therefore, performance would also be poor when using this second alternative. The third is to have each core which is sharing a repository notify all the other cores when it changes data. The other cores can then simply discard their cached data for the resource being changed, and re-query it from the database when necessary. This third alternative is likely the only alternative that will give us the desired performance.

The advantages of the this third alternative is that the core only needs to go to the database get data the first time, or when the data is actually changed. This takes full advantage of the cache. In a typical engagement of e-Speak, we expect resource metadata to be changed very rarely. In the case of Swatch, we have been told that only 60% of the messages that come to the core would be invoking a service, 20% would be discovering a service, and 10% would be establishing a connection to the core. All of these are basically read-only operations. Only 3% would be registrations of new services or changes to existing services. We don't need to worry about registrations of new services since that data cannot be any core's cache. Therefore, in the case of Swatch, less than 3% of the message coming to the core would cause cache invalidation.

The disadvantage of the cache invalidation approach is that it is nearly impossible to update the database and invalidate every core's cache atomically. To do so would require reliable synchronous messaging to all cores involved while the database holds locks on the data being changed. Each core would have to acknowledge the cache invalidation notification before the data changed and the locks released. This is problematic since there is no guarantee that the notification and its acknowledgement will get through, particularly if one of the cores sharing the repository goes down or is inaccessible.

Without atomicity, there will always be windows where a core may use old data in its cache, which is inconsistent with what is in the repository database. Considering that there is currently not any way to ensure consistency between repositories and the advertising service, this issue of atomicity of updates may not be big issue. The implications of this need further investigation, particularly in the area of security. In the situation where absolute consistency is required, repository sharing may simply not be an option.

Assuming that in the those engagements which require multiple cores sharing a repository can live with windows of inconsistency, then a simple asynchronous notification to each core sharing the repository is sufficient when a core changes existing metadata. The easiest way of doing this is to use a publish-and-subscribe methodology to communicate events. Each core that shares a particular repository database would subscribe to receive notification events whenever any metadata is changed. Then, when any core changes existing metadata, it would publish an event that would notify all the other cores of the change. The event would include what resource was changed, but would not need to indicate what the change was. Each core would then re-query the database to get the change. Using publish-and-subscribe methodology, each core does not need to know exactly which other cores are sharing the repository, but only that the repository is shared. Also, the issue of cores being down or inaccessible can be handled by the publication service and not by each core. Most importantly, the cores don't have to wait for each core to receive the notification. The disadvantage of the using publish-and-subscribe is that it introduces a single point of failure, namely the publication service. (Obviously, the shared repository is also a single point of failure, but all industrial strength DBMSs have high availability solutions to deal with this.) There is already an event distribution service that could possibly be used.

The other alternative is to have each core notify all the other cores directly about changes to existing metadata. This would eliminate the single point of failure in using a publication service. This can be done in a separate thread so as to allow the core to continue processing other messages. If the core supported multiple threads handling messages, this would not be an issue. The disadvantage of this approach is that each core has to keep an up-to-date list of what other cores are also sharing the same repository. When a core comes up, it would need to register with the repository it is connected to. It would also need to read from the repository what other cores are connected. It would then need to notify each of these other cores so that they know that this new core is now sharing their repository. Similarly, before a core goes down, it should unregister itself and notify all the other cores that it is no longer sharing the core. Of course, a core may come down abruptly, in which case the other cores will need to somehow determine that the core is no longer up and unregister it. Because of the complexity of connection management, we prefer the publish-and-subscribe model.

2.7.3 Transaction Management

The third major issue is how to handle simultaneous updates of the same data from multiple cores to insure data consistency. For example, if two cores both bind different services to the root nameframe, we need to make sure that neither of the binding gets lost.

This is basically an issue of transaction management. Since databases handle transaction management well, it makes sense to use the database to manage transactions between cores. The problem again is that the data in the cache may not be consistent with the data in the repository database. Currently updates are done by getting a reference to the data in the cache, modifying that data, and then calling store on the Repository to writing the data in the cache to the database. This works as long as the cache is guaranteed to be consistent with the database before the data in the cache is modified. The problem is that by the time the store method is called on the Repository, the data in the cache has already been modified by the calling code.

It is possible at store time to detect if the data to be stored was based on old data by using a version field in the metadata, which gets updated to a new value each time the metadata is updated. When doing an update in the database, the criteria can include the version number that the core expects based on the version value in the cache. If this version does not match in the database, the update will fail.

For example, the update of the specification of the resource with RepositoryHandle UID = 60 where the current cached version was version 2 would look something like,

```
update resspec set version = 3, ....
```

where rephandle = 60 and version = 2;

If the specification was already updated by the other core, then the version in the database would no longer be 2 and the update operation would return 0 rows updated. This would indicate that the cached version was out-of-date with the database version. Then an exception should be thrown to indicate to the cache was out-of-date and the update failed. At the same time, the out-of-date version would be thrown out of the cache so that a new version would be retrieved from the database next time this particular metadata is referenced. Since we do stores on three different types of resource metadata (specification, description, and state), each of these should have its own version number.

To keep the database logically consistent, whenever an update fails, all the work done for the particular call to the core should be rolled back. This implies that all work associated with a single call to the core should be done in one transaction.

It is easy enough to start a transaction within the database when starting to process a message to the core, and commit the transaction at the end of processing the message. Any failure in processing the message would result in the transaction being aborted, causing all changes to be rolled back. In this way the database is reverted back to before the message was issued, and the Client can reissue the same message or a different message.

Unfortunately, we don't have this level of transactionality today because the cache does not handle rollbacks properly. Specifically, if the cache has a transient resource state or specification which needs to be written to the repository database so that it can be removed from the cache, then if the transaction is rolled back in the database after the entry is removed from the cache, there is currently no way to rollback the cache to restore the entry in the cache. The entry is lost. It is no longer in the cache or in the database. We need to fix the cache to properly rollback in the case a transaction is aborted. In the case of an entry being swapped out to the database, it needs to be swapped back in. In the case of data which has been deleted, updated, or inserted, it is sufficient to clear the cache and reacquire the data from the database in the case of a transaction being aborted.

It is interesting to note that the root of the problem is that we are not writing through transient data to the database since transient data does not need to be persistent. The only time we need to write it out is when we need to remove it from cache to open up space in the cache. However, if multiple cores are sharing the repository, then all data needs to be written out to the database so that all cores sharing the repository can see it. This being the case, the above problem goes away.

To avoid the possibility of update failure due to out-of-date cache data, a new call to the repository could be created that would force a refresh of the cache data for a particular resource from the database. At the same time, it would lock the database data for this resource so that it cannot be updated by another core. Then the cache data would be guaranteed consistent and an update on this data would never throw an "out-of-date" exception.

2.8 Multiple Service Registrations

Stateless services that do not change their metadata frequently can use this approach.

In this approach,

- Multiple copies of services connected to the same core to share load (if application permits having multiple copies)

- Multiple copies of services connected to different cores to share load (if application permits having multiple copies), along with a **Locator Service** that will provide `ESuccessor` of appropriate service that can take up load.

This proposal uses `ServiceId` for defining service replicas. Please refer to Architecture Guide Chapter 5, section Service Identity for details on `ServiceId`. Architecture Specification writes:

The serviceId is intended for use by applications to identify services without using the Resource name or access path (ESNames).

This decouples authorization from resource naming and has several advantages:

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

None of this is possible using ESName for service identity.

Services that use matching `ServiceId` can be considered as replicated services. Replicated services need not have same metadata. If service metadata contains load balancing related data to be used in load distribution, every service in the replicated service group will have different metadata. E-Speak APIs will be used by services to create replicated services. E-Speak will provide a new Locator Service to locate replicated services. Details of the API and services are given in the following sections.

2.8.1 Limitations

- E-Speak services will have to be registered separately with multiple cores.
- Services will have to set same `ServiceId` using `ESServiceElement.setServiceId()` call.
- Replicated services will have distinct ESNames. These will be independent services; one or more services can be advertised.
- E-Speak will not attempt any synchronization of metadata of replicated services.

- This scheme will be suitable for services that do not update metadata frequently.
- E-Speak will not redirect messages for a resource to its replicas

2.8.2 LocatorService Function

Given an ESName, find out ESNames of service replicas using Service Id of input ESName.

Locator Service will perform the function of providing information on replicated service to clients. One or more copies of locator services can be started in each core.

Locator Service will be connected with all the cores. Locator Service will be able to list service replicas from these cores.

2.8.3 Interface

```
ESName[] getReplicatedServices(ESName)
```

Given an ESName, this service will query all the cores in the group and obtain ESNames of services that match ServiceId.

Logic of Locator service is given below.

Startup command

```
LocatorService <group> <list of core URLs>
```

Main:

```
Create LocatorConnectionList.
```

This object maintains list of all ESConnections maintained with all the cores and their group.

```
LocatorObject = new LocatorServiceImpl (LocatorConnections)
```

```
Loop:
```

```
Conn = ESConnection(core <n> )
```

```
LocatorConnectionList.add(Conn)
```

```
LocatorDesc = New ESServiceDescription()
```

```
LocatorElem New ESServiceElement(LocatorDesc)
```

```
LocatorElem.setImplementation( LocatorObject )
```

```
// Register

LocatorElem.register()

// Set service ID
LocatorElem.setServiceId("myuniqueId");

// Advertise and start
LocatorElem.advertise()
LocatorElem.start()
EndLoop

GetReplicatedService(ESName source)
    Get serviceId of source service represented by ESName

    If locator service is not connected to CORE:PORT of source service,
        Then open connection to CORE:PORT in source service
            Contact LocatorService in remote core
            replicaList GetReplicatedServiceList(source)
            Return replicaList

    Loop through all connections to cores
        Find ESNames of services tha have matching serviceId
        Add ESNames to replicaList (list of ESNames to be returned)
    EndLoop
    return replicaList
```

2.8.4 Enhancements

Locator Service can be used for load balancing purposes, if load-balancing policy can be defined for the service and core or service can provide information on service load.

For improving performance, Locator Service can build cache from previous queries and use Service register, unregister events to update cache.

Search options can be added so that when returning replicated service ESNames, the run status of the service can be returned to caller (started/ not started).

2.9 Group-wide Resource Registration

JESI can provide an API for group-wide registration of resources (services, account, folders, vocabularies, and contracts). If this feature is to be provided with cores using a separate repository, we will have to handle data integrity issues.

One JESI call may result in multiple messages, sent to different cores. Issues to be resolved will be:

- How to perform registration / unregistration with group members that are not available at the time of operation
- How to handle errors when operation is not successful with all the group members

We suggest providing this option, when multiple cores can share a repository.

2.10 Handling Service Failure

JESI provides an interceptor facility to intercept any messages before sending them to a service and after receiving response from a service.

We have proposed using an interceptor for handling a service failure condition, by doing retry from the interceptor. The interceptor will act on exception messages, and will use the replicated service list provided by the Locator Service to retransmit messages to service replica.

This proposal involves creation of a service interceptor and a part of JESI. E-Speak programmers will use this interceptor, when writing the e-Speak client.

2.10.1 Problems to be Solved

Exceptions received by the client side are not sufficient to determine the cause of the problem.

Assuming the client and core are connected to separate cores, error conditions are described in following table:

Error condition	Error received	Desired corrective action
Client side core failure	Socket error	Use alternate core
Network failure between client – service’s core	Timeout	Retry through same core or use alternate core
Service side core failure	Timeout	Retry through alternate service core
Service failure	Timeout	Use alternate service if available
Service loaded, delay in processing request	Timeout	Retry operation

In case there is failure of the client or the service core, the state of message will not be known. The client or the service’s core failed message delivery status can be any of the following:

- a** Message was not successfully delivered to destination (no delivery of partial delivery)
- b** Message being processed at destination
- c** Message was processed, reply sent by destination, however reply message was in failed core
- d** Reply message successfully delivered to client

In case A, B, C client does not know the status.

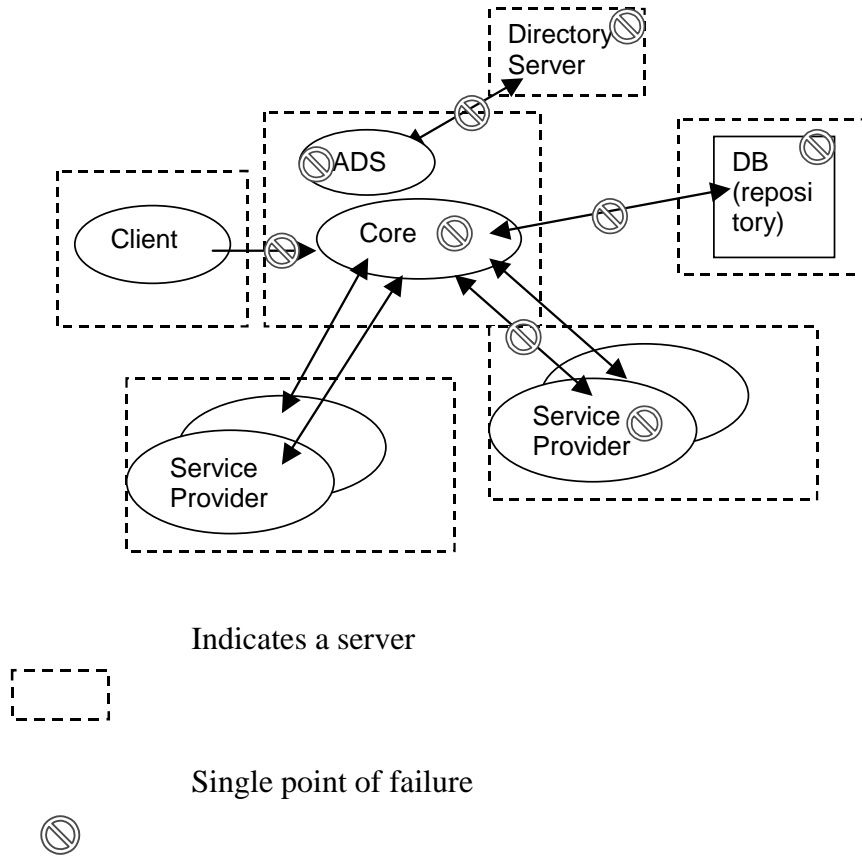
Automatic retry from JESI may not be suitable for all the services. Idempotent services can handle duplicated messages without undesirable side effect. Non-idempotent services can have side effects if the message delivered to them is a duplicate message.

3.0 HA and Scalability study

E-Speak code and configurations were studied to analyze HA and scalability problems.

3.1 Single Point of Failures (SPOF) in e-Speak Based Systems

A typical configuration of e-Speak based system is given below, along with SPOFs in the system.



SPOFs are:

- Core software
- Advertising service
- LDAP server
- Database server

- Service provider
- Network connection between Core – Repository
- Network connection between Core – Service Provider
- Network connection between ADS – directory server
- Server hardware failure – servers running core, database, service provider

Impact of failures and recovery steps are described in the table below:

Fault	Impact	Current recovery scheme	Desired recovery (Wish list)
CORE			
Core stopped Server running core stopped	Clients / service providers using this core get socket exception. Message delivery status (client to service provider, service provider to client) is not known. Operation retry can result in data integrity problems.	<ol style="list-style-type: none"> 1 Start sever (if server had failed) 2 Start core 3 Start service providers 4 Start clients <ul style="list-style-type: none"> • Service providers have to wait for core to start. • Clients have to wait for core and service provider to start before using the service. 	<p>Another e-Speak core can act as a backup of failed core.</p> <p>Services are reachable through more than one core.</p> <p>Clients can access service provider using alternate core, without needing restart of client/ service provider</p> <p>Service provider need handle duplicate messages.</p>

Resource problems in core, e.g. out of memory	Some of the messages can get dropped due to temporary resource problems. Non recoverable critical errors can result in stopping core.	No recovery scheme for dropped message Core does not exit on non recoverable errors	Well defined exceptions for recoverable errors Non recoverable critical errors will result in stopping core. Further recovery as per 'core stopped' case.
Core-Core connection failure	No attempt is made to restore connection. Remove resources cannot be accessed by client/services.	One of the core has to restart	Well defined exceptions for recoverable errors Non recoverable critical errors results in cleaning up remote core related resources Core attempts to reestablish connection.
Database			
Server that is running database stopped / Database stopped	Core not able to read/write repository. Connection to database from core is broken. Core gets exception. Core does not attempt to reconnect to database. Clients are unable to use core, service providers	<ol style="list-style-type: none"> 1 Stop core 2 Stop clients 3 Stop Service providers 4 Start database 5 Start core 6 Start service provider 7 Start clients 	Use database replication or Oracle parallel server.

<p>Temporary non availability of database (database running on a cluster)</p>	<p>Core not able to read/write repository. Connection to database from core is broken.</p> <p>Core gets exception. Core does not attempt to reconnect to database.</p> <p>Clients are unable to use core, service providers</p>	<ol style="list-style-type: none"> 1 Stop core 2 Stop clients 3 Stop Service providers 4 Start core 5 Start service provider 6 Start clients 	<p>Core attempts to reconnect to database for predefined retry interval. Stop core if database is not accessible after retry interval.</p> <p>Exceptions are returned to clients for operations that require repository access, indicating recoverable errors.</p>
<p>Service Provider</p>			
<p>Service stopped</p>	<p>Client has to recover from error e.g. Find another service and continue operation</p> <p>or</p> <p>Wait for service to be available again</p>	<p>Client unable to access service. Client gets exception.</p> <p>Application dependent logic for recovery</p>	<p>Core does not give accessor of stopped service to caller</p> <p>Core detects this condition and stops sending messages to stopped service, informs client about failure (if possible)</p> <p>Replicated service available to client through same or different core. Service can be stateful or stateless.</p> <p>Stateless services can service client transparently</p> <p>Stateful services will require client to initiate recovery. Client should be informed about service failure, so that client can start recovery.</p>

Service not reachable from core.	Client gets timeout or exception message. ? What happened to messages in service's INBOX	Retry the message	Use alternate service if available
Advertising Service			
Advertising service failure	Unable to advertise new service Unable to find services Unable to unregister services LDAP repository may become out of sync with database based repository.	Start advertising service Retry advertise operation Retry find operation	Queuing of advertise/unadvertise operations so that ADS can pick up these events when restarted and update LDAP repository or global service repository.
LDAP Server			
LDAP server failure	Unable to advertise new service Unable to find services Unable to unregister services	Start LDAP server Retry advertise operation Retry find operation Retry unregister operations	?? Does advertising service return proper error? Does it reopen connection to LDAP server for every lookup/register?
Cannot access LDAP server (network problem)	Unable to advertise new service Unable to find services Unable to unregister services	Start LDAP server Retry advertise operation Retry find operation Retry unregister operations	? any retry from ADS How many times? How long?

3.2 Service Provider – Core Interaction

This section identifies messages exchanged between a service provider/client and core. Message originator is client or service provider. For some of the messages, destination is core; core performs action on the messages and sends replies. When the message destination is a service provider or a client, the core acts as a router and routes messages to the appropriate destination. Messages have to be considered of HA solution, because some the messages affect the state and data residing in core.

3.2.1 Calls that Affect Data in Core

The following calls may modify data in the core repository, hence will have to be sent to multiple cores in a cluster, to maintain data integrity.

- 1 Service Register: `ESServiceElement.register()`
- 2 Service Update: `ESServiceElement.update()`
- 3 Service Unregister: `ESServiceElement.unregister()`
- 4 Service Restart: `ESServiceElement.restart()`
- 5 `ESAccessor.setResourceProxy()`
- 6 `ESAccessor.setDescription()`
- 7 `ESAccessor.setPublicData()`
- 8 `ESAccessor.removePublicData()`
- 9 `ESAccessor.clearPublicData()`
- 10 `ESAccessor.addPublicData()`
- 11 `ESAccessor.setPrivateData()`
- 12 `ESAccessor.removePrivateData()`
- 13 `ESAccessor.clearPrivateData()`
- 14 `ESAccessor.setPrivateData()`
- 15 `ESAccessor.getEventFlag()`
- 16 `ESAccessor.setMetadataMask()`
- 17 `ESAccessor.setResourceMask()`

- 18** ESAccessor.setOwnerPublicKey()
- 19** ESAccessor.setServiceId()
- 20** ESAccessor.setServiceOwner()
- 21** ESAccessor.setServiceProxy()
- 22** ESServiceHandler(ESAccessor esa) -> reconnect()
- 23** ESBaseServiceStub.invokeSynchronous()
- 24** ESConnection.switchAccount()
- 25** ESFolder.rename()
- 26** ESFolder.remove()
- 27** ESFolder.addNameBinding()
- 28** ESRemoteConnectionManager.openConnection()
- 29** ESRemoteConnectionManager.closeConnection()
- 30** ESViewStub.add()
- 31** ESViewStub.remove()
- 32** ESViewStub.clear()
- 33** ESAccountManager.createAccount()
- 34** ESAccountManager.deleteAccount()
- 35** ESAccountManager.setAccountProfile ()
- 36** ESAccountManager.addDescription ()
- 37** ESRemoteServiceManager.exportService() -> multiple messages, 1 per ESAccessor
- 38** ESRemoteServiceManager.importService()
- 39** ESRemoteServiceManager.unexportService()
- 40** ESRemoteServiceManager.updateExportedService()
- 41** ESRemoteServiceManager.updateImportedService()

3.2.2 Method Invocation

Invocations are done by the following classes, and the interceptor mechanism is available to process the response.

```
ESRequest.invoke
```

3.2.3 Read Requests

Following method calls result in messages to core, however, they may not affect data in core (any context maintained by find?).

- 1** ESAccessor.getUrl()
- 2** ESAccessor.getDescriptions()
- 3** ESAccessor.getVocabularies()
- 4** ESAccessor.getContracts()
- 5** ESAccessor.isPersistent()
- 6** ESAccessor.getEventFlag()
- 7** ESAccessor.getMetadataMask()
- 8** ESAccessor.getResourceMask()
- 9** ESAccessor.getOwnerPublicKey()
- 10** ESAccessor.getServiceId()
- 11** ESAccessor.getPublicData()
- 12** ESAccessor.getPrivateData()
- 13** ESAccessor.getESUID()
- 14** ESAbstractFinder -> findLocalAccessors/findServiceList
- 15** ESAbstractFinder -> abstractFindNext
- 16** ESFolder.getURL()
- 17** ESFolder.containsName()
- 18** ESFolder.listNames()

- 19** ESRemoteConnectionManager.getConnection()
- 20** ESViewStub.contains()
- 21** ESVocabularyStub.getAttributePropertySet()
- 22** ESAccountManager.authenticateUser()
- 23** ESAccountManager.getAccountProfile ()
- 24** ESAccountManager.getAllAccounts ()
- 25** ESAccountManager.getUserESURL ()
- 26** ESAccountManager.checkCredentials ()

3.3 Problems to be addressed in HA Solution

HA solution will have to solve the following problems:

- 1** Data integrity between multiple cores
- 2** ESName aliases (alternate ESName)
- 3** Reliable fault detection
- 4** Recovery after failure
- 5** Reliable message delivery

3.3.1 ESName Alias

ESName of a service provider contains core's host and port number. Messages for the service provider get routed through the core. For HA solution, we have to deliver a message to a service provider, after core failure. To achieve this, the service provider has to obtain an alternate ESName by registering the service with some other core and establishing that the two ESNames are alternate ESNames.

Most of the calls in list A above (A. Calls that affect data in core) act on the local resource, and have to be provided with a URL referring to local resource of the core. If we decide to copy messages to other cores, URLs that represent the alternate URL of the resource will have to be sent by the sender. Alternatively, core may recognize the ESName/URL aliases and substitute a local resource name in place of the remote resource.

Provide a scheme for creating, storing, and retrieving ESName alias. A `Find` call will return one ESName of the service provider to the client. If a service provider is not reachable through ESName, how do we find the alias of the ESName? Aliases must be available outside core's repository so that ESName alias can be obtained in case of core failure.

3.3.2 Data Integrity

Resource related data and state resides in core. If we establish an alternate ESName, it is necessary to keep resource metadata and state in sync, after any changes. This has to be done by either duplicating messages or replicating data from one core to the other.

3.3.3 Fault detection

If we are able to relate exception and error messages to problems, recovery is possible. Currently timeout and exception mechanisms are available that indicate faults. Timeout is a generic error and it is difficult to relate timeout error to a specific fault.

3.3.4 Recovery

If we consider a scheme where responsibility of the retry operation is given to client, we have following alternatives:

Client chooses alternate ESName and sends message. This is equivalent to making a `findAll (MAX=2)` call and remembering alternate names.

Core understands that ESName is not reachable and redirects message to alternate ESName.

3.4 What Does not Scale?

Scalability solution will not be a general-purpose solution. It will depend on the nature of the application, the number of users, the number of services, work load, the nature of services, and performance requirements.

These are the perceived bottlenecks today:

Bottlenecks	Possible solutions	Potential problems
Connections to one core	Connection pooling. Maintain less connections, use <code>ESConnection.switchAccount ()</code> for operations	Overhead in doing <code>switchAccount</code>
Messages sent via core	Use multiple cores to distribute load on core, i.e. a service can be reached via multiple paths (cores) instead of a single core.	Name resolution scheme, Implementation changes that allow service threads to read from multiple inboxes residing in different cores
Number of services	<p>This scalability can translate to memory requirements in core, database size, database operation time, other core resources such as inboxes/outboxes.</p> <p>Possible solution will be partitioning of services by core so that load on core is reduced</p>	Core to core communication will increase and may affect response time
Number of clients	<p>i.e. number of clients who want to use a service exceeds number of service threads available to process requests.</p> <p>JVM imposes limitations on number of service threads that can be created.</p> <p>TO handle this problem, services will have to either allow replicas or have to act as message router/ load balancer, and queue messages for processing logic running in separate JVMs and accept new client requests</p> <p>Alternatively, allow replicated service concept in e-Speak and use external load balancing logic to solve problem of many clients trying to use one service</p>	<p>Adding replicated service concept in e-Speak</p> <p>external load balancing service</p> <p>Replicated services to share metadata, so that metadata updates are not replicated</p>

Queries	<p>Queries can take-up core resources. If number of queries is large, then possibly services are discovered dynamically and used by clients.</p> <p>If query scalability is not a problem, then it is likely that service handles are stored in folders and used by clients.</p> <p>Both the above cases have to be considered when trying to solve e-Speak scalability problems</p>	
One copy of service unable to handle multiple clients requests?		
Remote resource invocation	<p>Clients connected to core that does not have local resources, thus requiring core-core connection.</p>	

4.0 ServiceGuard Configuration

4.1 Pre-requisites/Assumptions

- 1 Installation of ServiceGuard cluster.
- 2 Installation(s) of e-Speak on all the nodes of the cluster.

4.2 Configuration steps (using ServiceGuard commands)

4.2.1 Create a package:

- 1 Create a subdirectory for each package you are configuring in the `etc/cmcluster` directory:

```
# mkdir /etc/cmcluster/espeak
```

(You can use any directory names you wish.)

- 2 Generate a package configuration template for this package:

```
# cmmakepkg -p /etc/cmcluster/espeak/espeak.config
```

(You can use any file names you wish for the ASCII templates.)

- 3 Edit the generated file `espeak.config`, as follows:

- a Enter the name of the package (here it is, `espeak`) as against `PACKAGE_NAME` field.

- b Enter the names of the nodes configured for this package. Repeat this line as necessary for adoptive nodes. The order is relevant. Put the second adoptive node after the first one. Eg,

```
NODE_NAME esp8
```

```
NODE_NAMEesp5
```

- c Enter the complete path for the run and halt scripts. (These are generated scripts. The way they are generated is described later in the document.) In most cases, the run script and halt script will be the same script. This is the package control script generated by the `cmmakepkg` command. For example, the entries are made as follows:

```
RUN_SCRIPT/etc/cmcluster/espeak/espeak.sh
```

```
HALT_SCRIPT/etc/cmcluster/espeak/espeak.sh
```

- d** Enter the service name. All the service names should correspond to the service names used by `cmrunserv` and `cmhaltserv` commands in the `run` and `halt` scripts.

Eg., `SERVICE_NAME=espeakservice`

- e** Enter the network subnet name that is to be monitored for this package. Repeat this line as necessary for additional subnet names.

Eg., `SUBNET=15.76.100.0`

- 4** Generate a package control script for this package. The package control script contains all the information necessary to run all the services in the package, monitor them during operation, react to a failure, and halt the package when necessary. Each package must have a separate control script, which must be executable. The control script is placed in the package directory and is given the same name as specified in the `RUN_SCRIPT` and `HALT_SCRIPT` parameters in the package ASCII configuration file.

To generate a control script, use the following command:

```
# cmmakepkg -s /etc/cmcluster/espeak/espeak.sh
```

- 5** Edit the generated file `espeak.sh` as follows:

- a** Set `PATH` to reference the appropriate directories.

Eg., `PATH=/usr/bin:/usr/sbin:/etc:/bin`

- b** Specify the IP and Subnet address pairs which are used by this package. This must be set in pairs, even for IP addresses on the same subnet.

Eg., `IP[0]=15.76.101.205`

`SUBNET[0]=15.76.100.0`

- c** Specify the service name, command and restart parameters that are used by this package. For example,

```
SERVICE_NAME[0]=espeakservice
```

```
SERVICE_CMD[0]="/etc/cmcluster/espeak/espeak_ha.sh"
```

```
SERVICE_RESTART[0]="-r 4"
```

("-r 4" indicates that the service will be restarted 4 times.)

Note: `espeak_ha.sh` is a shell script that contains the following command:

```
/opt/e-speak/bin/espeak -i /etc/cmcluster/espeak/espeak.ini
```

`espeak.ini` has the following form:

```
[Properties]
```

```
[Tasks]
Start=Core

[Core]
Class=net.espeak.infra.core.startup.StartESCore
Args=-p esip:2950
```

- d** Customer defined function: (a place holder for customer defined functions). All actions that need to happen before the service is started should be defined here. For e-Speak, the actions correspond to the setting of all e-Speak related environment variables. For example:

```
function customer_defined_run_cmds
{
/opt/e-speak/espeak_cfg.sh
/opt/e-speak/start_oracle.sh
    test_return 51
}
```

Here, `espeak_cfg.sh` contains:

```
export EHOME=/opt/e-speak
export CLASSPATH=$EHOME/lib/es.jar:$EHOME/extern/cryptix/
    cryptix32.jar:$EHOME/extern/ldap/ldapjdk.jar:$EHOME/extern/
    parser/xerces.jar:/opt/java1.3/jre/lib/rt.jar
export SHLIB_PATH=$EHOME/lib
export ESPEAK_HOME=$EHOME
export PATH=/opt/java1.3/bin:$PATH
```

`start_oracle.sh`: This shell script contains all the commands that are required to start the Oracle database server.

4.2.2 Verify the package configuration

In the above case, the following command is used to verify the package configuration:

```
# cmcheckconf -k -v -P /etc/cmcluster/espeak/espeak.config
```

NOTE: Errors are displayed on the standard output. If necessary, edit the file to correct any errors, then run the command again until it completes without errors.

4.2.3 Copy the generated files to other nodes

The generated files `espeak.config` and `espeak.sh` along with `espeak`-specific scripts such as, `espeak_cfg.sh`, `espeak_ha.sh` and `espeak.ini` should be copied from the configuration node (in this case, `esp8`) to the same pathname on all nodes (in this case, `esp5`) which can possibly run the package.

4.2.4 Start the cluster and package

Command: `cmruncl`

NOTE: After starting the cluster, the command “`cmrunpkg -v espeak`” can also be run to start a specific package.

4.2.5 Check if the package has started

Command: `cmviewcl -v`