Messaging for a Dynamic World

Mobeon Internal

Approved: Per Berggren	No: 3/FD-CRH 109 581-1 Uen

Copyright Mobeon AB All rights reserved	Author: Tommy Lundemo Title: FD – Profile Manager	Version: PA5 Date: 2008-03-13	1/17

FD – Profile Manager

Content

History

Version	Date	Adjustments
PA1	2007-08-27	First version of the document. Original document was found in MAS. (EMAHAGL)
PA2	2007-09-19	Updated with attribute value control. (EMAHAGL)
PA3	2008-02-26	Minor updates for segmented CoS and 3pp (FE31)

		(QTOMMLU)
PA4	2008-03-07	Updates after internal review (QTOMMLU)
PA5	2008-03-13	TA after review (QTOMMLU)

1 Introduction

The Profile Manager (ProfM) provides subscriber data handling such as retrieving subscriber mailbox instances, and retrieving and changing: subscriber parameters, subscriber greetings, spoken name, and distribution lists.

The ProfM stores and retrieves subscriber data from the Messaging User Register (MUR) using an LDAP implementation of JNDI. Greetings and spoken name are retrieved from and stored in Message Store (MS) using an IMAP implementation of JavaMail.

Additionally, ProfM also creates and deletes subscribers using the Provision Manager (ProvM).

ProfM hides the Segmented CoS model, providing components a way of reading CoS settings without the knowledge of the underlying structure.

2 Function Structure

2.1 Dependencies

The dependency of the ProfM on other Backend components is shown in Figure 1.

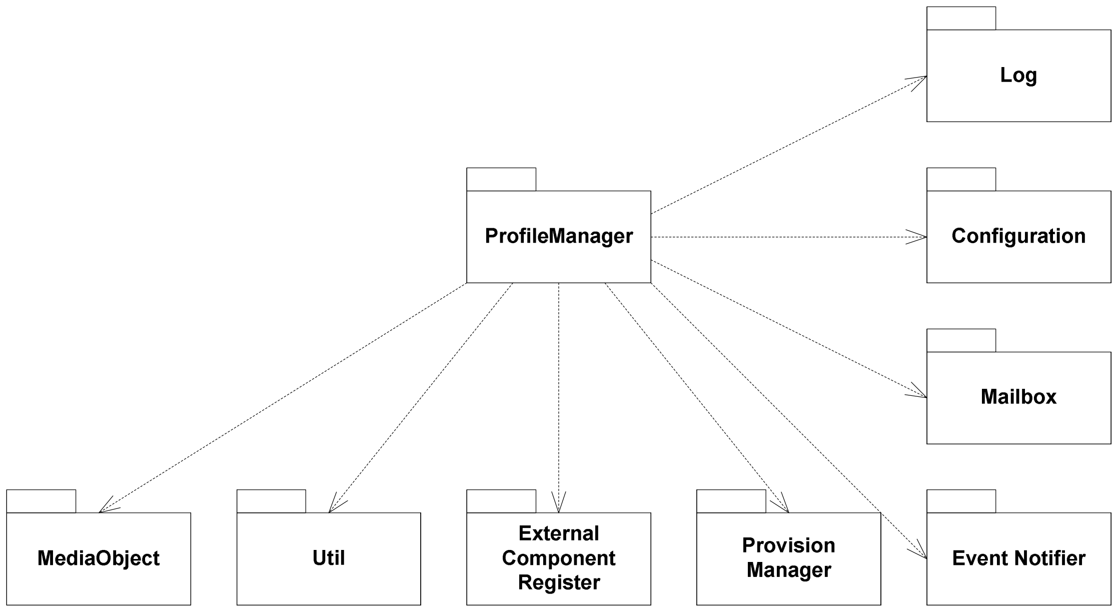

Figure 1 Components dependency

2.2 Overview

An overview of the main classes in the ProfM is shown in Figure 2.

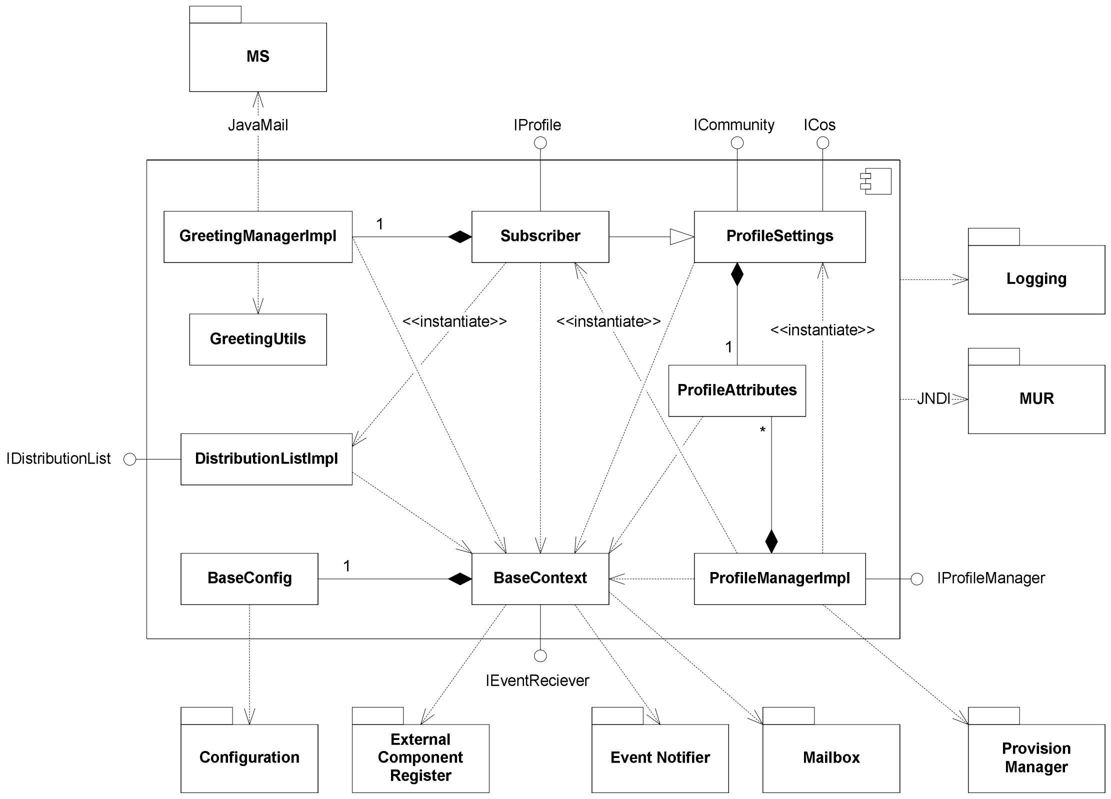

Figure 2 Overview Profile Manager

The ProfileManagerImpl class implements the IProfileManager interface and is used for searching and retrieving subscribers from MUR. JNDI is used for communicating with MUR. It is also used for creating and deleting subscribers from MUR using the ProvM.

The Subscriber class handles all data associated with a subscriber, i.e. attributes, greetings and spoken name, mailbox, and distribution lists. It caches the subscriber data located in MUR.

The ProfileSettings class handles read-only attributes. It implements the ICommunity and ICos interfaces and is used to return a subscriber's community and class of service (CoS) values.

The ProfileAttributes class encapsulates and parses the JNDI search results.

The DistributionListImpl class handles all data associated with a subscriber's distribution list.

The GreetingManager class encapsulates all greeting and spoken name handling: getting and setting greetings and spoken name as MediaObjects. The GreetingManager uses JavaMail for this. Greetings and spoken name are stored in the MS.

The GreetingUtils class contains helper methods for the GreetingManager, when creating greeting mails.

The BaseContext class is a common context class used by most of the other classes. It maintains the current configuration among other things. It also implements the IEventReceiver interface and receives the ConfigurationChanged event.

The BaseConfig class parses the configuration to a suitable format. It contains a class ProfileMetaData that holds the configuration for each attribute that can be retrieved or set. The ProfileMetaData also contains an optional class AttributeValueControl that has rules on valid values on an attribute. These rules can be checked when an attribute is set.

2.3 GreetingManager

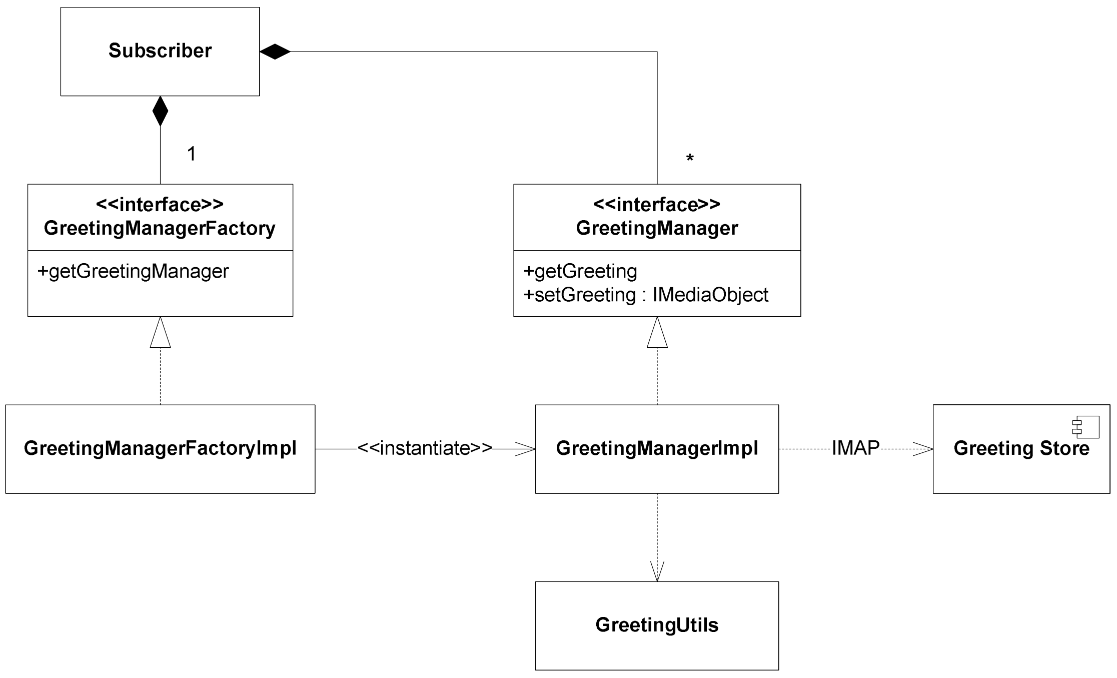

Figure 3 GreetingManager

To simplify testing, the GreetingManager is an interface implemented by the GreetingManagerImpl class. A GreetingManagerFactory injected into the Subscriber object makes unit testing of the Subscriber class independent of greeting storage solutions.

2.3.1 GreetingSpecification and SpokenNameSpecification

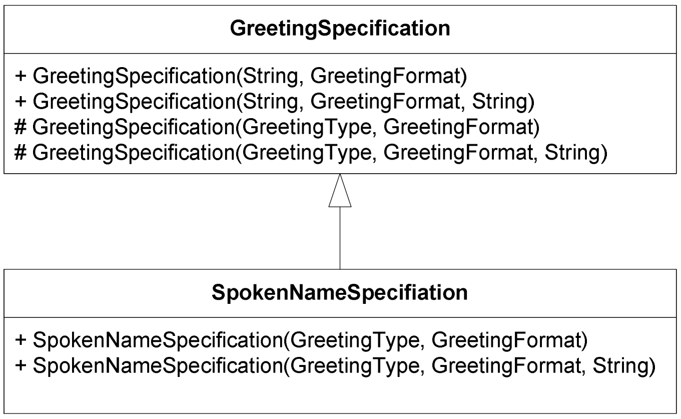
GreetingSpecification
+ GreetingSpecification(String, GreetingFormat)
+ GreetingSpecification(String, GreetingFormat, String)
# GreetingSpecification(GreetingType, GreetingFormat)
# GreetingSpecification(GreetingType, GreetingFormat, String)
SpokenNameSpecifiation
+ SpokenNameSpecification(GreetingType, GreetingFormat)
+ SpokenNameSpecification(GreetingType, GreetingFormat, String)

Figure 4 GreetingSpecification and SpokenNameSpecification

The GreetingSpecification class is used externally by ProfM clients to specify the type and format of the greeting to retrieve or store in the Subscriber class's get/setGreeting methods. The greeting type is specified by one of the predefined string values: allcalls, noanswer, busy, outoufhours, extended_absence, cdg, temporary and ownrecorded. The greeting format is specified by the GreetingFormat enum, containing the values VOICE and VIDEO.

Internally in the ProfM, subscriber and distribution list spoken names are also handled by the GreetingManager's get/setGreeting methods. To avoid external clients to retrieve spoken names through Subscriber's getGreeting method, constructors using the GreetingType enum are protected in the GreetingSpecification constructor. The SpokenNameSpecification class, which is not available externally, provides a public constructor with the GreetingType enum and is used when getting and setting spoken names.

2.4 Search criteria

Searches are performed according to a specified search criterion. The criterion is expressed as a tree of criteria. Available criteria are the logical criteria classes: ProfileAndCriteria, ProfileNotCriteria, ProfileOrCriteria; and the subscriber attributes criteria classes: ProfileStringCriteria, ProfileIntegerCriteria and ProfileBooleanCriteria.

To create an LDAP search string based on the criteria search tree the LdapFilterFactory class is used. This is a Visitor class which creates the LDAP search string by traversing (visiting) the nodes in the criteria tree.

2.5 DirContextPool

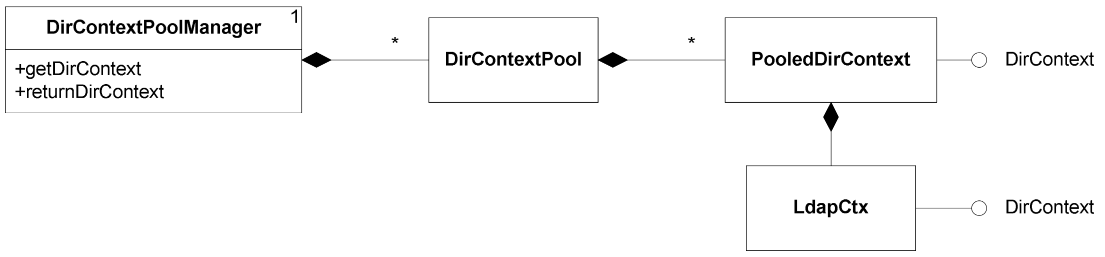

Figure 5 DirContextPool

The DirContextPool is used for caching connections to MUR. It consists of a singleton DirContextPoolManager which is used for getting and returning DirContext implementations. There is one pool for each MUR.

The PooledDirContext decorates the real DirContext implementation adding a timestamp when the DirContext was created. This is used to control the maximum lifetime of DirContext:s. When returning a DirContext to the pool, it is discarded if it is considered to old.

2.6 ProfileManagerImpl

Profile searches conducted by the ProfileManagerImpl class are not cached. If caching is required this has to be done on the client side.

When creating and deleting subscribers, the ProfileManagerImpl converts the submitted Subscription into the Subscription object used by the ProvM, see Figure 6.

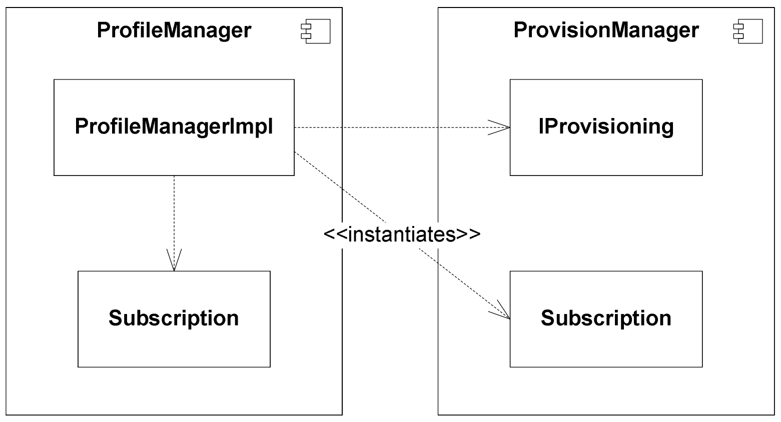

Figure 6 User provisioning

2.6.1 Segmented COS

The profileManagerImpl also keeps track of possible segmented class of service objects. Each segmented COS object consists of one or more Compound Services (Service Clusters), and each Compound Service consists of one or more End User Services (Service Instances), all identified by a service ID (see Figure 7). The idea is that MUP should be used to grow or shrink COS settings by rearranging the references in MUR. Such rearrange (of either clusters or instances) will not take effect until after next cache timeout. See 3.1.3. The Segmented COS objects are instantiated on demand and MUR lookups are only performed if there are misses in either Cluster or Instance cache.

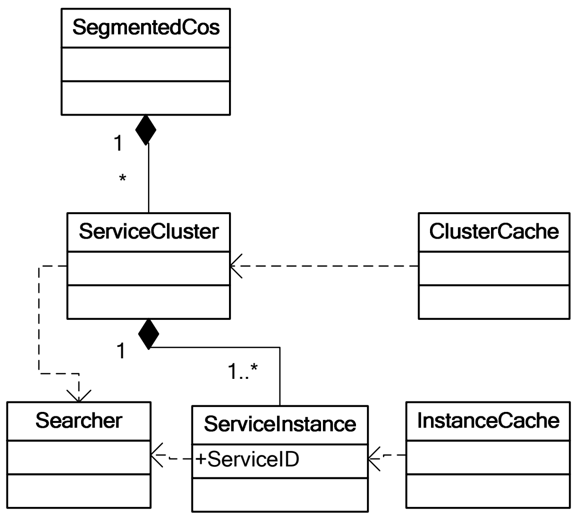

Figure 7 Segmented COS

2.6.1.1 Special Handling of Attributes for Compatibility Reasons

The rearrangements in user register requires profile manager to translate certain attributes from emServiceDn to actual attribute names and the other way around. It also needs to aggregate a special attribute (userNTD) and keep track of which attributes in the user profile that are considered basic (i.e. valid in a Segmented COS).

The following table describes the translations made in the Segmented COS:

API shall return	SegCos in MUR
emServiceDn=prerecordedgreeting_email	emViaExtProvider
emServiceDn=prerecordedgreeting_tui	emViaTelephony
ttsemailenabled	emServiceDn=ttsemail
Faxenabled	emServiceDn=fax
emServiceDn=message_recovery	emMessageDeletedRecovery
emServiceDn=forwardmessagewithoutcomment	emMessageForward
emServiceDn=forwardmessagewithcomment	emMessageForwardComment
emServiceDn=sendvoicemessage	emMessageSend

The basic attributes are listed here (they can optionally be configured in backend.xml):

"passwordminlength,passwordmaxlength,maxloginlockout,inhoursstart,inhoursend,inhoursdow,diskspaceremainingwarninglevel,emexpirationrule,emretentiontime,emnoofmailquota,mailquota,emnotretrievedvoicemsg,cosname”

2.7 MUR data

The classes connected to MUR data is shown in Figure 8.

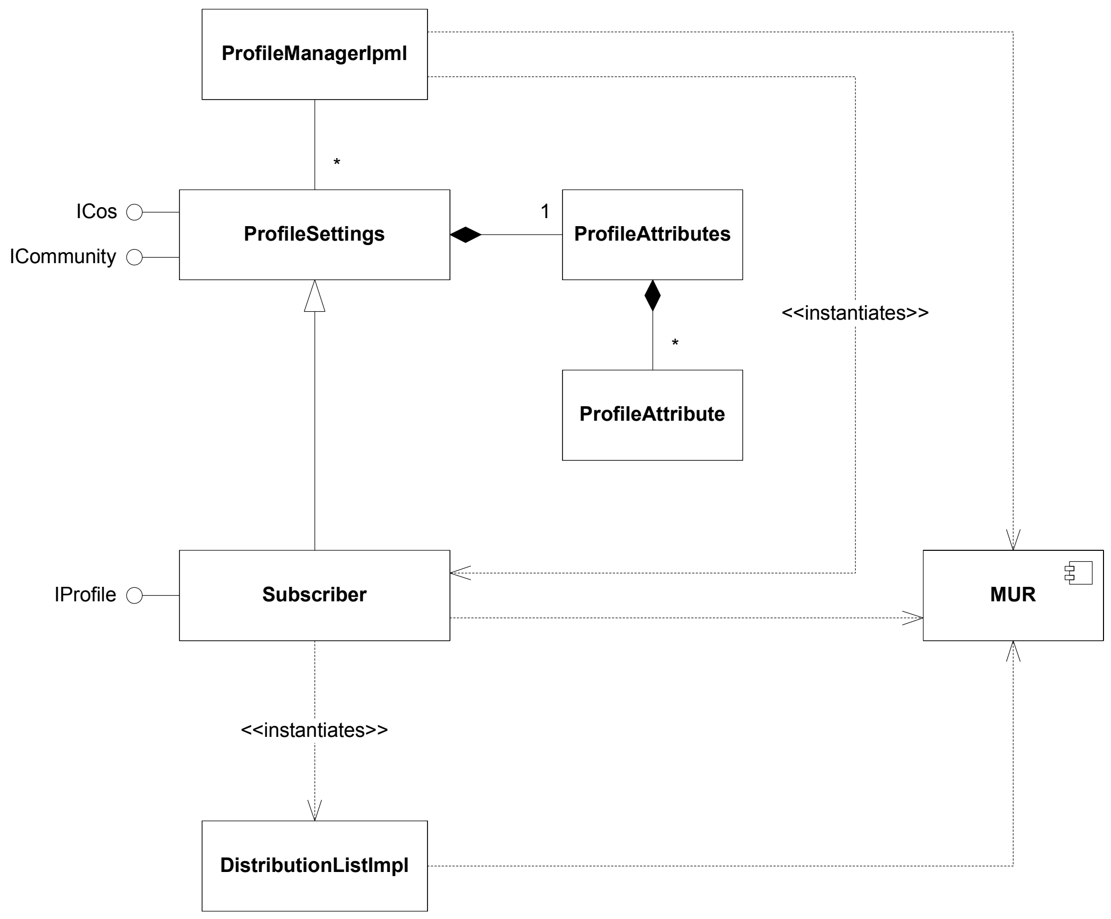

Figure 8 Classes connected to MUR data

ProfileManagerImpl keeps the community and COS caches as ProfileSettings objects. It is also responsible for creating Subscriber objects, but not for closing the resources connected to them.

The Subscriber is responsible for creating the DistributionListImpl objects reflecting the distribution lists in MUR.

3 Function Behavior

3.1 ProfileManagerImpl

3.1.1 Search

The ProfileManagerImpl searches MUR for subscribers matching the submitted criteria, expressed as a search term condition tree. For each subscriber found a new Subscriber object is created and all search results are returned to the client, see Figure 9.

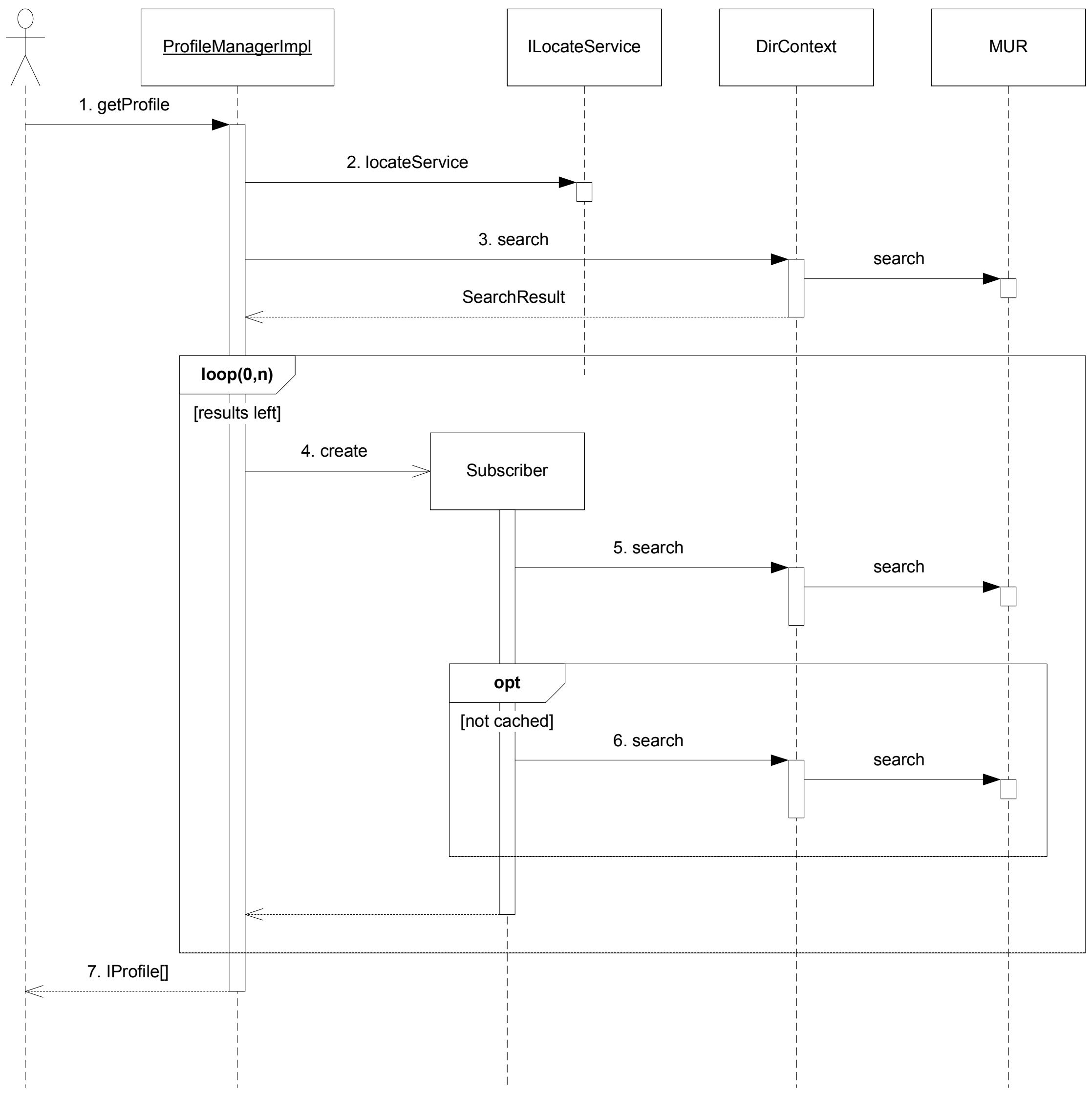

Figure 9 Subscriber search

1. A client calls getProfile.
2. The MUR to search in is located.
3. The search request is sent to MUR.
4. For each search result a new Subscriber object is created.
5. Additional searches are made to complete the Subscriber information, e.g. if the initial search returned results from the user entry, the billing entry is located and retrieved, and if the initial search returned result from the billing entry, the user entry is located and retrieved.
6. If community or CoS values are not already cached, they are also retrieved from MUR and subsequently added to the community and CoS caches. For details on Segmented CoS Model see 3.1.3.
7. The found subscribers are returned to the client.

3.1.2 Provisioning

The ProfileManagerImpl class is also responsible for creating and deleting subscribers using the ProvM, see Figure 10 and Figure 11.

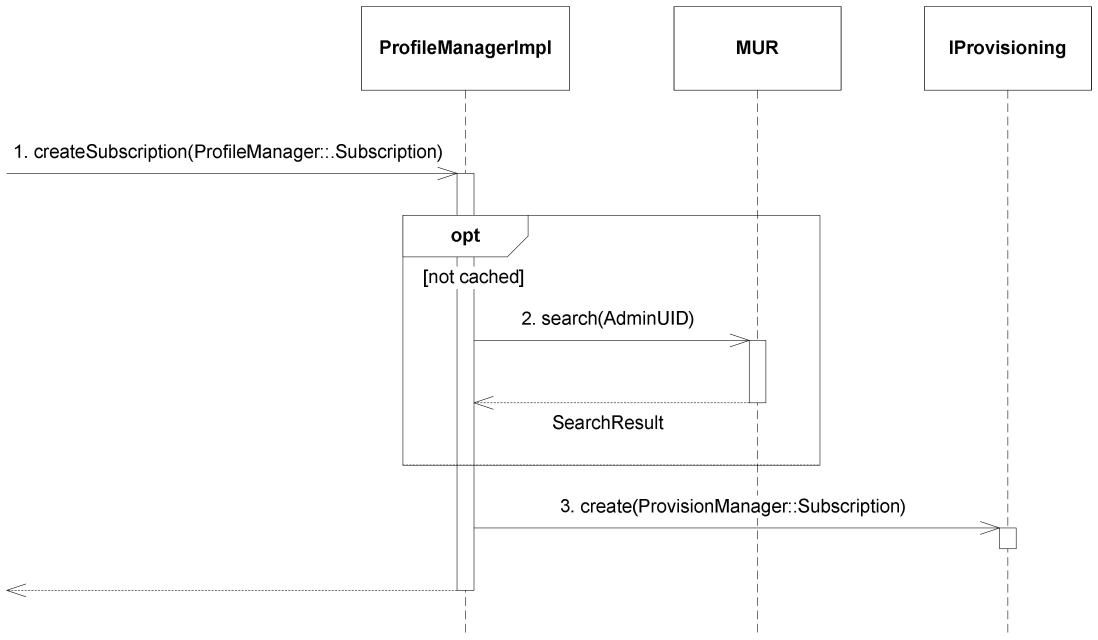
ProfileManagerImpl
MUR
IProvisioning
1. createSubscription(ProfileManager::.Subscription)
opt
[not cached]
2. search(AdminUID)
SearchResult
3. create(ProvisionManager::Subscription)

Figure 10 Creating a subscriber

1. A client calls createSubscription with a ProfM Subscription.
2. If not cached, the subscriber community administrator is retrieved from MUR.
3. The request is forwarded as a ProvM Subscription using the IProvisioning interface.

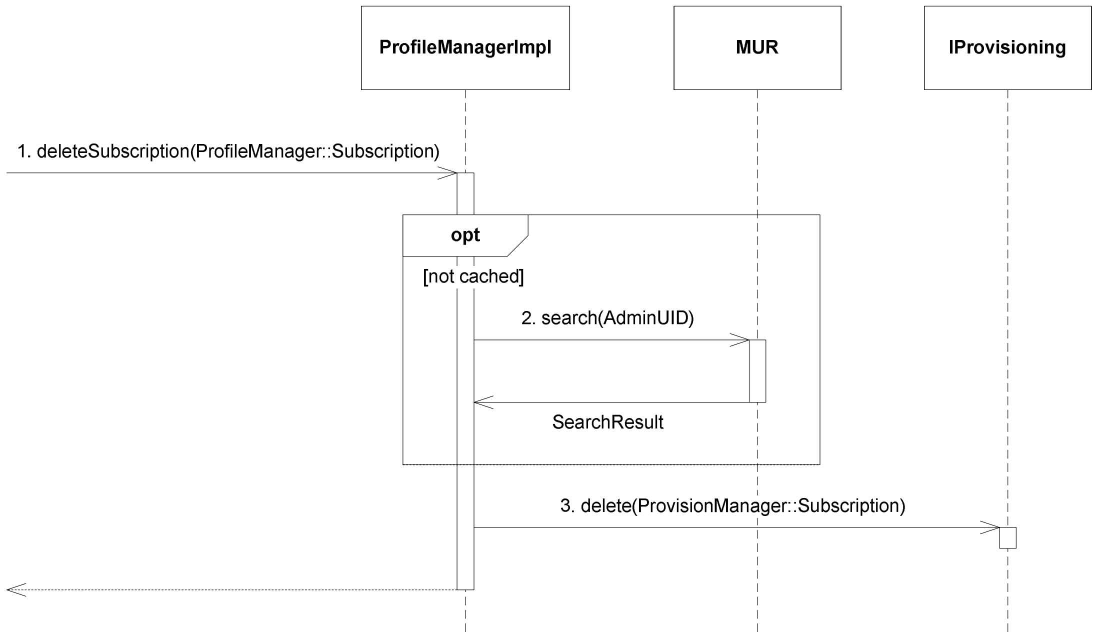
ProfileManagerImpl
MUR
IProvisioning
1. deleteSubscription(ProfileManager::Subscription)
opt
[not cached]
2. search(AdminUID)
SearchResult
3. delete(ProvisionManager::Subscription)

Figure 11 Deleting a subscriber

1. A client calls deleteSubscription with a ProfM Subscription.
2. If not cached, the subscriber community administrator is retrieved from MUR.
3. The request is forwarded as a ProvM Subscription using the IProvisioning interface.

3.1.3 Segmented CoS retrieval

The ProfM is responsible for all reading of segmented COS attributes. When a Subscriber object is created during subscriber search, an object class of emSegmentedConfCos indicates that this is a Segmented COS.

- If there is only a Compound Service reference in the COS, the attributes from the End User Services referred to by the Compound Service builds the current COS.
- If there are also references to one or more Compound Service(s) in the user profile, these settings "override" and complement the settings from the COS Compound Service(s).
- If there are conflicting End User Services, a priority (enforced during creation) is used to indicate which of the conflicting End User Service attributes to read.

3.1.4 Caches

The ProfileManagerImpl class keeps caches of frequently requested objects that does not change often, e.g.: CoS and community values, and provision user administrators. The caches are implemented using a TimedCache class which returns null if the requested object does not exist or is considered too old. For the

Segmented CoS model a cache of Compund Services (ServiceCluster) and End User Services (ServiceInstances) is kept (keyed on Directory Name). Inconsistencies due to old caches (for example if an End User Service is deleted in MUR but Compound Service is cached with references to the non-existing End User Service) do not lead to errors, corrupt Compound Services or End User Services are ignored.

3.2 Subscriber

The Subscriber class keeps a cache of requested mailboxes, so that the mailbox connection is kept alive as long as the Subscriber object exists. When finished using the Subscriber object, the close method must be called so that resources can be deallocated.

Changes in a Subscriber object are forwarded to the MUR immediately.

3.2.1 Distribution lists

The Subscriber class is responsible for searching (see Figure 12), creating new (see Figure 13) and deleting (Figure 14) existing distribution lists.

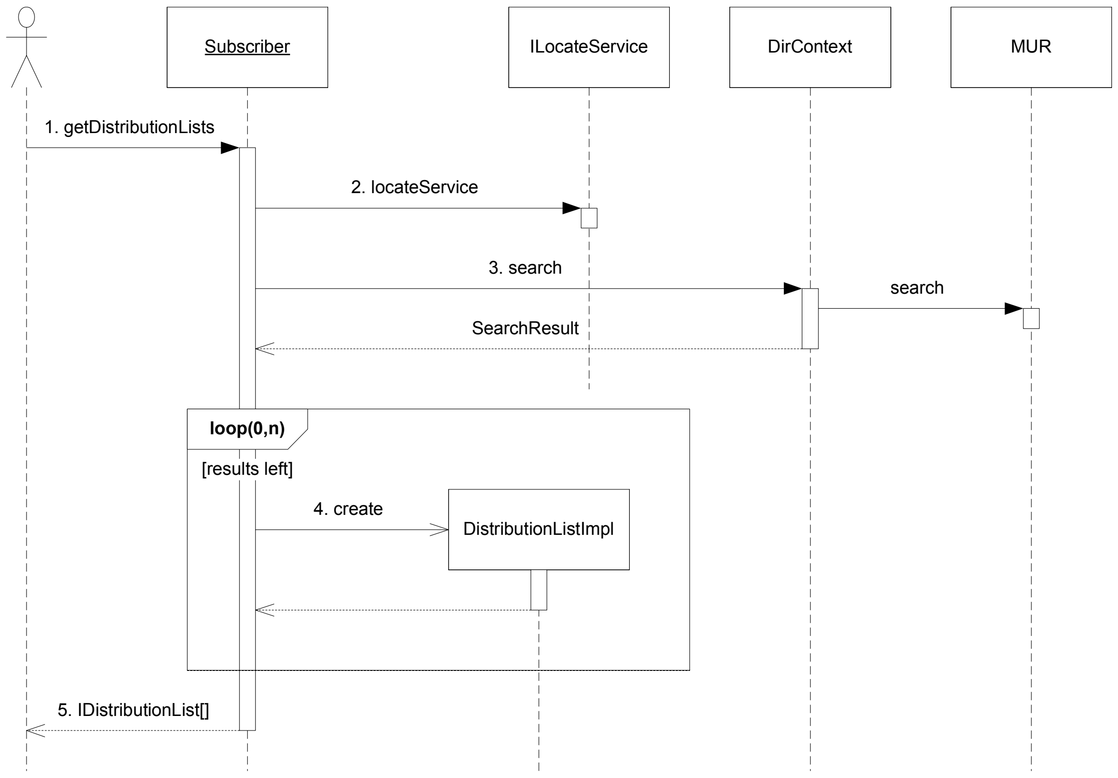

Figure 12 Searching for distribution lists

1. A client calls getDistributionLists.
2. The MUR to search in is located.
3. The search request is sent to MUR.

4. For each search result a new DistributionListImpl is created.
5. The found distribution lists are returned to the client.

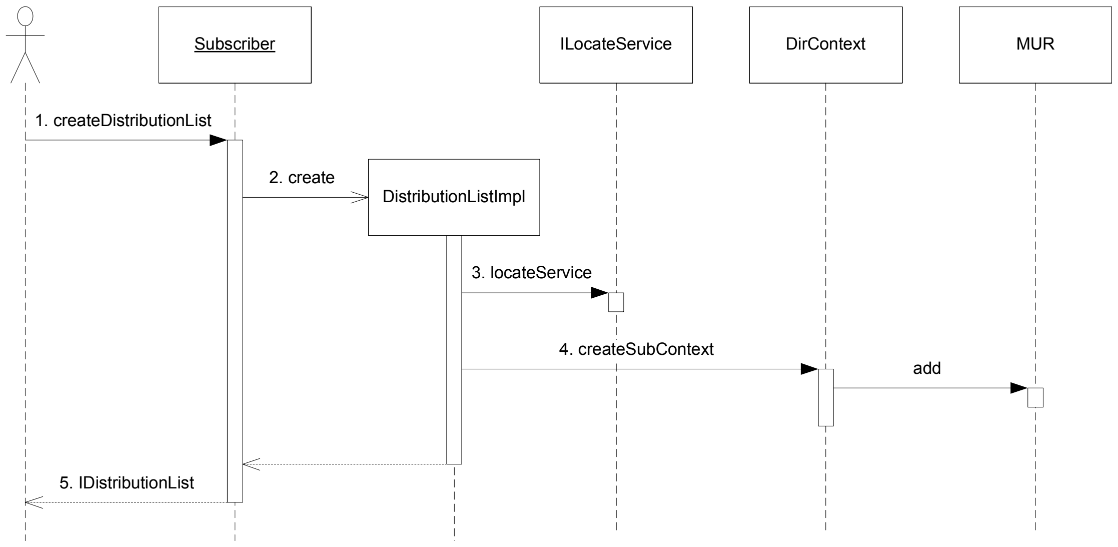

Figure 13 Creating a distribution list

1. A client calls createDistributionList.
2. A new DistributionListImpl is created.
3. The MUR to add the distribution list to is located.
4. The distribution list entry is created in the MUR.
5. The new distribution list is returned.

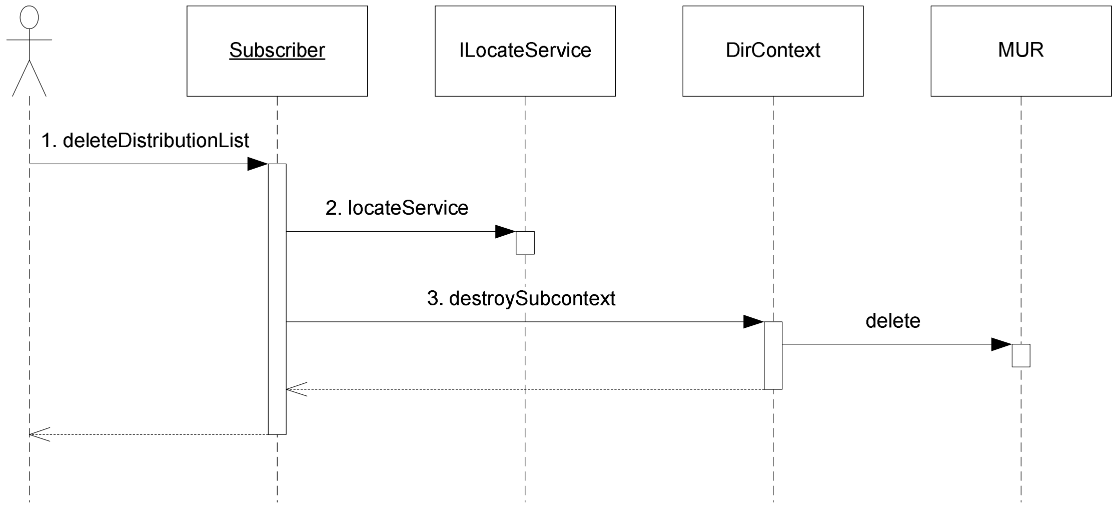

Figure 14 Deleting a distribution list

1. A client calls deleteDistributionList.
2. The MUR to remove the distribution list from is located.
3. The distribution list entry is removed from the MUR.

3.3 DistributionListImpl

Changes made to distribution lists are immediately forwarded to the MUR.

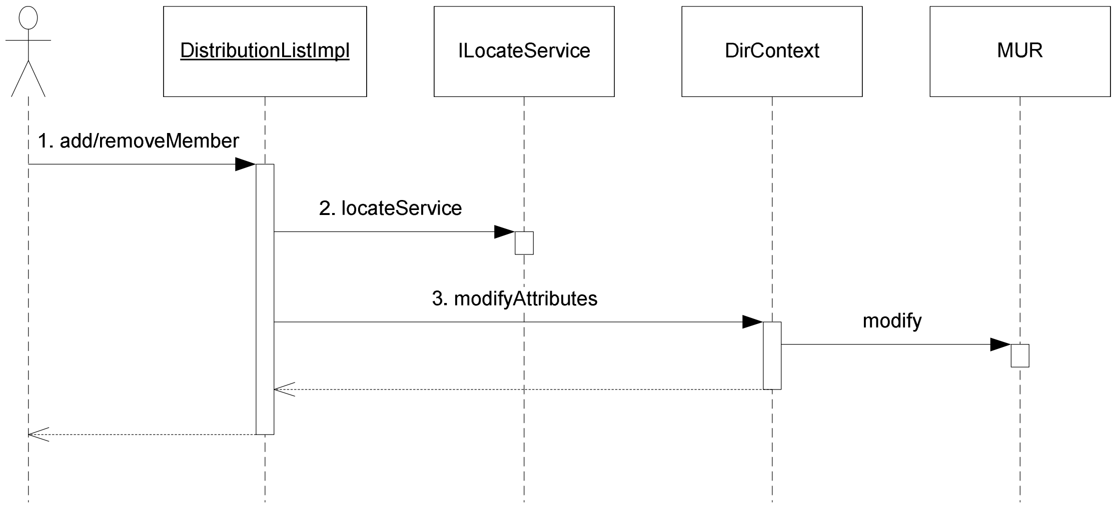

Figure 15 Adding/removing members in distribution lists

1. A client calls addMember or removeMember
2. The MUR to modify the distribution list on is located.
3. The distribution list entry is modified.

3.4 Greetings and spoken name

The retrieval and storage of greetings is handled by the GreetingManagerImpl class. The methods for getting and setting spoken name internally uses the getGreeting and setGreeting methods, therefore sequence diagrams are not shown for the getSpokenName and setSpokenName methods.

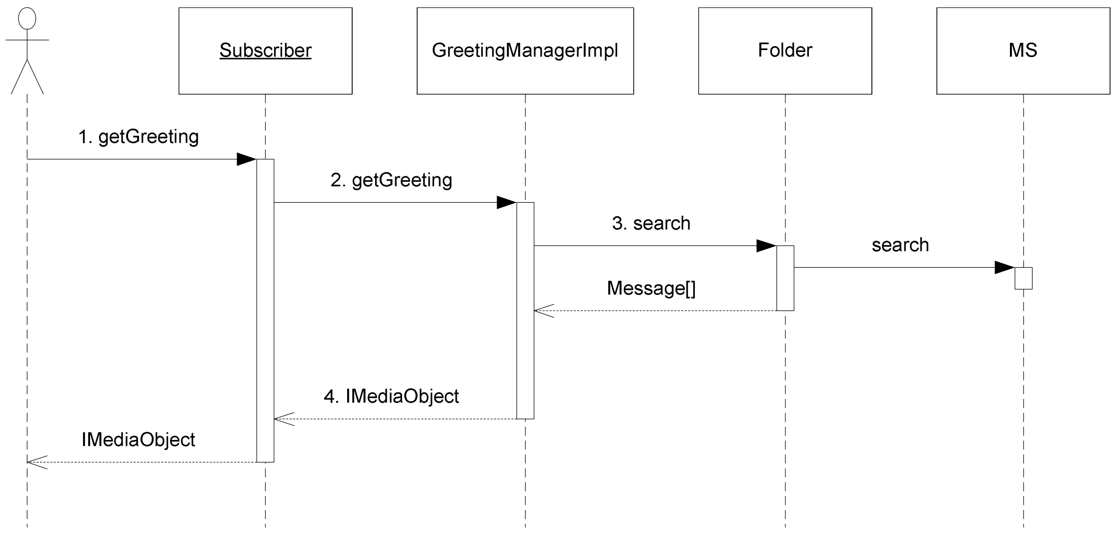

Figure 16 Get a greeting

1. A client calls getGreeting
2. The call is forwarded to the GreetingManagerImpl class.
3. The GreetingManagerImpl class opens the greeting folder on the MS and searches the greeting folder after a matching greeting.
4. The greeting is returned as an IMediaObject.

If no greeting is found, a GreetingNotFoundException is thrown.

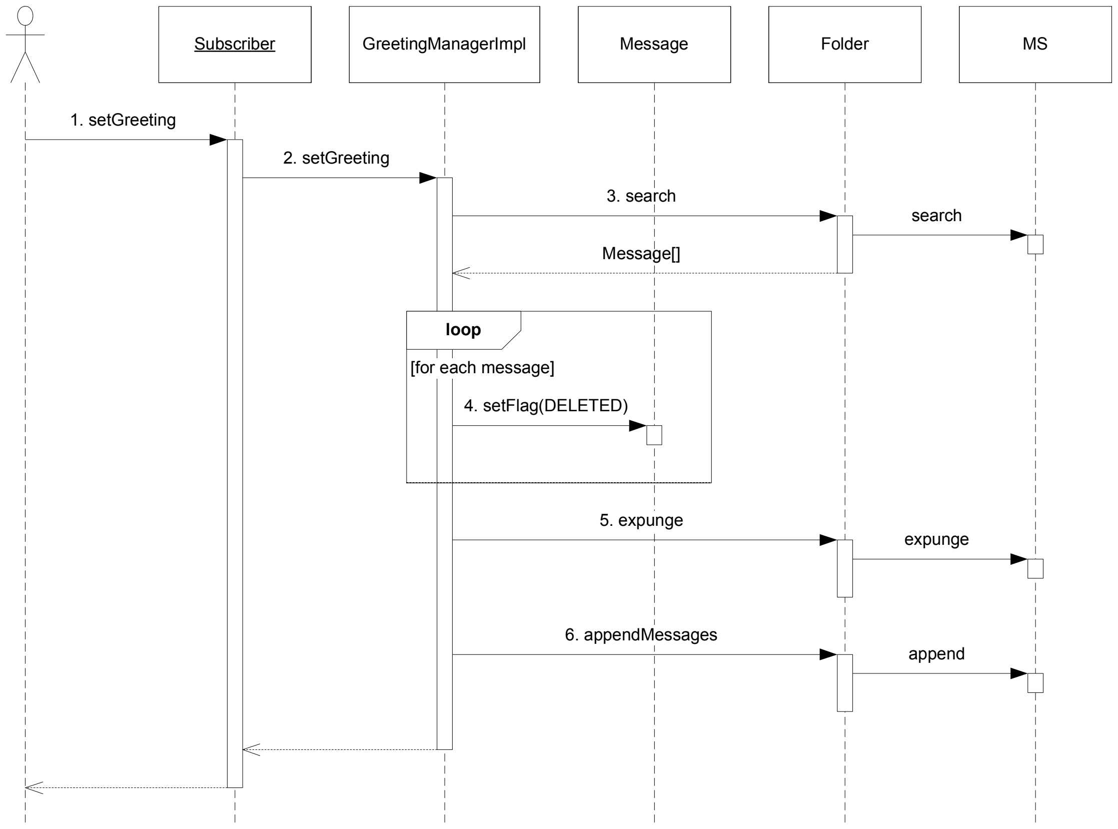

Figure 17 Set a greeting

1. A client calls setGreeting
2. The call is forwarded to GreetingManagerImpl class.
3. The GreetingManagerImpl class opens the greeting folder on the MS and searches the greeting folder after a matching greeting.
4. Found greeting messages are set to deleted and…
5. …the greeting folder is expunged.
6. The greeting to set is appended to the greeting folder.

The DistributionListImpl class uses the GreetingManager in much the same way as the Subscriber class. The only difference lies in the name of the greeting folder.

3.5 Timeouts and retries

JNDI does not support client timeout for its methods. The timeouts in JNDI are on the server side. This means that if the MUR is not able to handle a request, the call could block longer than the configured timeout. To be able to handle such problems with the MUR host, the TimeoutRetrier class from the com.mobeon.common.util package is used. It will execute a Callable in a separate thread, interrupting the thread if the timeout expires. The TimeoutRetrier will also retry executing the Callable in case of certain failures (indicated by the Callable throwing a RetryException encapsulating the real exception) a configurable number of times during a configurable time period. The timeout, try limit and try time limit are all read from the configuration.

The search and modify request tasks throw RetryException when catching a JNDI CommunicationException. In those cases the MUR host is also logged as unavailable using the HostedServiceLogger (see [2]).

3.6 ConfigurationChanged

Almost all classes use the BaseConfig class to retrieve configuration parameters. The BaseConfig is retrieved from the BaseContext class. The BaseContext class listens to ConfigurationChanged events. When received, the BaseContext class tries to create a new BaseConfig object with the new configuration. If the creation succeeds, the BaseConfig object is replaced, otherwise an error log is created and the previous BaseConfig object is used.

To be threadsafe, the BaseConfig and IConfiguration getters (and modifiers) are synchronized.

4 References

[1] FS Profile Manager
3/FS-CRH 109 581/1 Uen

[2] FS Logging
2/FS-CRH 109 581/1 Uen

5 Terminology

CoS	Class of Service
JNDI	Java Naming and Directory Interface
LDAP	Lightweight Directory Access Protocol
MAS	M3 Application Server
MS	Message Store
MUR	Messaging User Registry
ProfM	Profile Manager
ProvM	Provision Manager