FD – Log Manager

Content

1 History

Version	Date	Adjustments
PA1	2007-08-27	First version of the document. Original document was found in MAS.

2 Introduction

This document describes the implementation of the Log component. It outlines the main classes, how they relate and the public interfaces of the component.

Further, a few scenarios illustrate how the component should be used.

2.1 Glossary

Category	Grouping of source locations, from where a Log Entry can originate. In most cases a Category corresponds to a class package or a set of class packages. The Categories are configured in the Log configuration.
Exception	A Java or C++ Exception, carrying error information such as error type, error message and call stack information.
Log	A collection of Log Entries. In its simplest form, the Log is a text file where information from the Log Entries is written.
Log Entry	An object, containing information which shall be logged. In many cases, the object is simply a textual string.
Log Level	A fixed set of levels, used to classify the severity of a Log Entry. The following Log Levels exist: - ALL - DEBUG - INFO - WARN - ERROR - FATAL - OFF The levels may be seen as thresholds used to determine if a Log Entry should be forwarded to the Log or not.
Hosted Service	A service following a protocol and resides at a host available on a specific port.

3 Function Structure

3.1 Overview

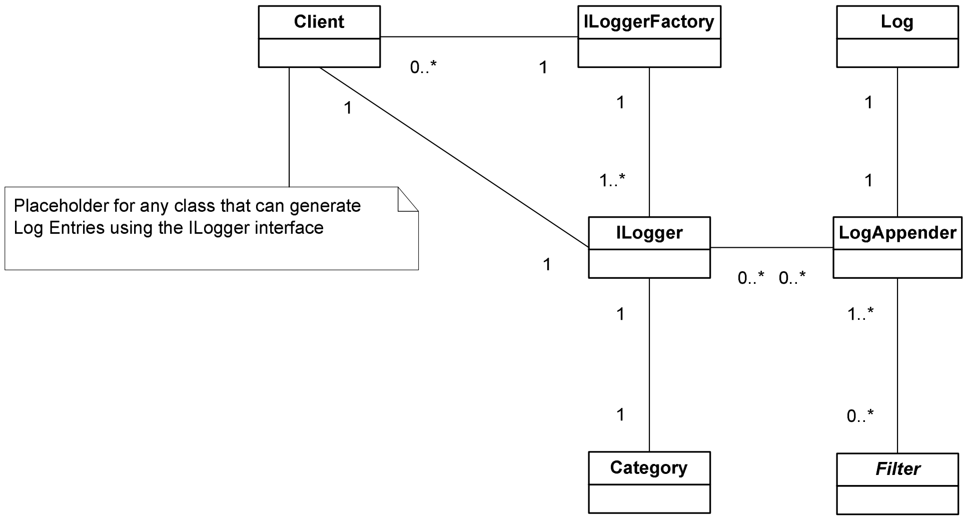

Figure 1 Overview of the Log component

The illustration above shows the main entities of the Log component. The Client in the illustration is not part of the Log component, and is a placeholder for any external user that interacts with the Log component.

The main interface in the component is the ILogger interface. It provides methods to the clients for generating Log Entries. There is only one ILogger object per Category. A Category is a grouping of source locations, from where Log Entries can be created. Any grouping algorithm may be used, but the preferred grouping is by using the package names of the class files. I.e. all classes residing in the package a.b.c share the same Category.

A LogAppender is used to forward a Log Entity to specific Log. The LogAppender to use, and therefore also the Log to use, is configurable through a configuration file. Each ILogger may use multiple LogAppenders (at least one), allowing for a Log Entity to end up in multiple Logs. Each LogAppender may be used by multiple ILoggers, allowing Log Events from multiple categories to end up in the same Log.

It is possible to add filters to the Log Appander, allowing for only specific Log Entities fulfilling some criteria to be forwarded to the Log.

3.2 Log4X

The bulk of the features and functions provided by the component are implemented using a 3rd party framework, Log4X, from the Logging Services Project at the Apache Software Foundation, see [1] In particular, the Log4J service is used for creating a Java binding and Log4Cxx is used for creating a C++ binding.

The interfaces provided by the Log4X packages are extended to encapsulate and extend some of functionality. The major part of the functionality and configuration still remains the standard Log4X implementation/configuration.

In the document it is indicated when a function, or a configurable entity, is added outside the Log4X frameworks.

For details on the structure of the Log4X services and how to configure those services, see [1]

3.3 Interfaces

3.3.1 ILoggerFactory

```
public class ILoggerFactory {
    public static ILogger getILogger(String clazz) ;
    public static ILogger getILogger(Class clazz) ;
}
```

The ILoggerFactory class provides a set of static methods allowing a client to create/retrieve an object implementing the ILogger interface. The argument supplied is used to identify a Category, and thereafter find or create object implementing the ILogger interface for that Category.

Example:

```
ILogger logger = ILoggerFactory.getILogger(MyClass.class);
```

Or

```
ILogger logger = ILoggerFactory.getILogger(“SomeOtherCategory”);
```

The latter example should be avoided, since it makes the hierarchy of the Categories to become less clear, or at least less implicit.

In the former example, the class package is used as Category, i.e. when the factory tries to locate a logger for the class `com.mobeon.event.Notifier`, it starts looking for a configured Category named `com.mobeon.event.Notifier`. If no such Category exists, it looks iteratively for `com.mobeon.event`, `com.mobeon` and `com`. It still no Category has been found, the root i.e. the default category is chosen.

3.3.2 ILogger

The ILogger interface provides the following methods (the listing shows the methods as implemented in Java):

```
public interface ILogger {
    public void debug(Object message);
    public void debug(Object message, Throwable t);

    public void info(Object message);
    public void info(Object message, Throwable t);

    public void warn(Object message);
    public void warn(Object message, Throwable t);

    public void error(Object message);
    public void error(Object message, Throwable t);

    public void fatal(Object message);
    public void fatal(Object message, Throwable t);

    public void clearSessionInfo();
    public void registerSessionInfo(String name, Object sessionInfo);

    public boolean isDebugEnabled();
    public boolean isInfoEnabled();
}
```

When a client wishes to issue a debug Log Entity, one of the debug(…) methods of the ILogger interface is called, supplying the message to log as an argument. It is possible to add an Exception as a second argument. If so, the information such as stack trace and cause of the Exception will end up in the Log as well as the log message.

In analogue, in order to generate a warn Log Entity, one of the warn(…) methods are used, error(…) for an error Log Entity etc.

3.3.3 HostedServiceLogger

This class decorates an ILogger instances with methods specialized to log hosted service circumstances. By using the HostedServiceLogger the log system is able to filter repetitive logging of hosted service communication failures.

```
public class HostedServiceLogger implements ILogger {

   public HostedServiceLogger(ILogger logger)

   public void notResponding(String protocol, String host, int port,
                             String message)

   public void notResponding(String protocol, String host, int port)

   public void notAvailable(String protocol, String host, int port,
                            String message)

   public void notAvailable(String protocol, String host, int port)

   public void available(String protocol, String host, int port)

    .
    .
    .

}
```

When a client successfully connects to a hosted service it notifies the log system that the hosted service is available. This done by calling the `available` method with the value of the used protocol, host name and port number submitted.

When a client fails to connect to a hosted service, and intends to keep trying to connect, it notifies the log system that the hosted service is not responding. This is done by calling the `notResponding` method with the value of the used protocol, host name and port number submitted. If the client has details of what was causing the hosted service not to respond they can be provided in an optional message.

When a client fails to connect to a hosted service, and not intends to keep trying to connect, it notifies the log system that the hosted service is not available. This is done by calling the `notAvailable` method with the value of the used protocol, host name and port number submitted. If the client has details of what was causing the hosted service to not be available they can be provided in an optional message.

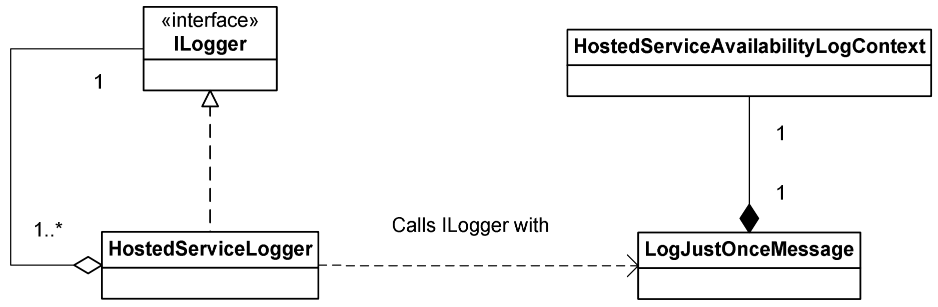

Figure 2 Hosted service logger class structure

As mentioned above the Hosted Service Logger enables filtering of repetitive logging which is described in 3.7.

3.4 Configuration

The Log component supports the standard configuration used by the Log4X framework. The configuration is in XML format. For details on configuration, see [1]

The path to the configuration file is propagated to the JVM using system properties. Such a property may be set using parameters to the JVM at startup:

```
-Dlog4j.configuration=path-to-config-file.xml
```

An example config file is illustrated below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
    <appender name="ROLLFILE"
     class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="mobeon.log"/>
        <param name="MaxFileSize" value="10MB" />
        <param name="MaxBackupIndex" value="10" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
             value="%d{ISO8601} %c %5p [SID:%X{session}]  - %m%n"/>
        </layout>
        <filter class="com.mobeon.logging.SessionFilter"/>
    </appender>

    <category name="com.mobeon.logger.sub1">
      <priority value="info" />
    </category>

    <category name="com.mobeon.logger.sub2">
      <priority value="error" />
    </category>

    <root>
        <priority value ="fatal" />
        <appender-ref ref="ROLLFILE" />
  </root>
</log4j:configuration>
```

The configuration has a few distinct areas:

3.4.1 Defining appenders

An appender is the entity responsible for formatting a Log Entity and thereafter sending it to a Log.

An appender is referenced by a name, ROLLFILE in the example. It uses a specific implementation for handling the Log, in the example a RollingFileAppender is used. It will create a specific file, mobeon.log, which will grow up until a specific size, 10MB, before "aging" the file in maximum number of backups, 10 in the example.

```
    <appender name="ROLLFILE"
     class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="mobeon.log"/>
        <param name="MaxFileSize" value="10MB" />
        <param name="MaxBackupIndex" value="10" />
```

It may use configurable patterns in order to generate a specific format for the Log Entity. In the example, the ConversionPattern has the format:

<Timestamp> <sourceRef> <severityLevel> [SID:<sessionId>] - <LogMsg>

```
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
             value="%d{ISO8601} %c %5p [SID:%X{session}]  - %m%n"/>
        </layout>
```

It is possible to add filters, allowing for only specific Log Entities fulfilling some criteria to be forwarded to the Log. In the example a custom filter is used to filter out only Log Entities related to a specific end-user session(s). The filter is a custom-made class, that will be called in order to determine if the Log Entity shall be sent to the Log or not. Any number of filters may be added, they will be called in a chain. If a Log Entity passed one filter it is passed to the next in the chain etc. The default is NOT to have any filters.

```
         <filter class="com.mobeon.logging.SessionFilter"/>
```

A root Category is defined, stating the default log level threshold to use, and what appender to use. In the example, fatal is the default level, and the ROLLFILE appender defined above. The root Category MUST be defined.

```
    <root>
        <priority value ="fatal" />
        <appender-ref ref="ROLLFILE" />
    </root>
```

Any number of categories may be added. The example has two additional Categories. They inherit the attributes from the root Category, but have the possibility to override any attribute. In the example, only the log level is overridden to info and error respectively.

```
    <category name="com.mobeon.logger.sub1">
      <priority value="info" />
    </category>

    <category name="com.mobeon.logger.sub2">
      <priority value="error" />
    </category>
```

3.5 Log4cxx

Log4cxx, which currently is only utilized in Stream, is not configured through XML. Stream logging through log4cxx is configured in a property file (stream.log.properties located in the cfg directory).

Here by follows an example of the Stream logging configuration:

```
log4j.rootCategory=INFO, STDOUT
log4j.appender.STDOUT=org.apache.log4j.ConsoleAppender
log4j.appender.STDOUT.Append=false
log4j.appender.STDOUT.layout=org.apache.log4j.PatternLayout
log4j.appender.STDOUT.layout.ConversionPattern=%d{ISO8601} %c %5p %m%n
```

Hence the logging in C++ is directed to stdout due to legacy. The first version of the logger in Stream was a set of macros using cout (stdout). The MAS start-up script is responsible for redirecting stout/stderr to the log file process.log (in the directory log).

As you can see of the configuration of the pattern layout the session ID is handled internally in Stream C++.

3.6 User Session Filtering

Session filtering (aka call trace) is handled by the filter class com.mobeon.logging.SessionFilter. The session filter contains a list of key-value-pairs. The key value pairs are matched against the key-value-pairs registered in the log4j MDC. With other words the session information.

The filter, when active, forward Log Entities to the Log according to the result of session data matching:

If the key-value-pair list of the filter is empty then all Log Entities are forwarded.

If one of the keys in the filter does match one of the keys in the MDC (Log Entity) then the filter will only forward the Log Entities which have a matching value.

This is sufficient for our application since either the MDC will contain session data or it will contain no session data and the session data will always have the same keys (which are set by the Call Manager).

The key-value-pairs are passed to the filter as parameters in the filter clause (please see how to define appenders here above). The value-key-pair is a string in following format: <key>:<value>.

Where key can be one of:

- calling
- called
- redirecting

And value is (when the key is one of above) a phone number.

The following example the session filter will forward all Log Entities related to sessions where the caller party is 1234 or the called party is 4567. Please note that Log Entities which are not related to any session will always be forwarded (and that this piece of XML belongs to the filter clause in the log appender clause in the log manager configuration shown here above):

```
<filter class="com.mobeon.logging.SessionFilter">
  <param name="TraceItem" value=”calling:1234”/>
  <param name="TraceItem" value=”called:4567”/>
</filter>
```

You should also note that the Log Entities forwarded to the filter corresponds to the current log level. With other words this is a discriminating filter. Hence in order to forward session related logging for info events the log level must be set to info.

3.7 Filtering repetitive logging

The filter class `RepetitiveLoggingFilter` is able to filter repetitive log messages. If the message of an arriving log event is of class `LogJustOnceMessage` the filter will decide if it's a repetitive message or not. If the message is of another class the filter is neutral.

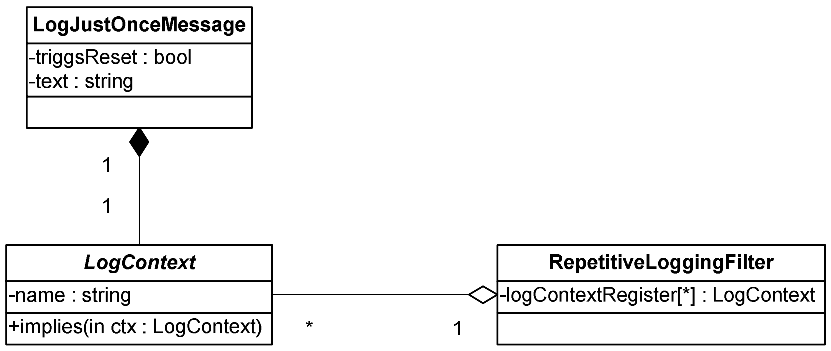

Figure 3 Repetitive logging filter class structure

The filter has a log context register to keep record of which type messages already has passed the filter. An arriving message is considered to be repetitive if the register contains a log context:

- that is equal to the log context of the arriving message
- that implies the log context of the arriving message

If the arrived message is considered repetitive it's filtered and will not be logged.

An exception to this rule is if the message's `triggReset` flag is set to <u>true</u> and the register contains a log context:

- that is equal to the log context of the arriving message
- that is <u>implied by</u> the log context of the arriving message

All log contexts in the register that matches the above criteria will be removed to reset the specific log context in the filter. The resetting message is considered informative and will be logged. Note: If no matching log context was found in the register (no reset executed) the message will not be logged.

4 Function Behavior

4.1 Use cases

4.1.1 Create a new Log object

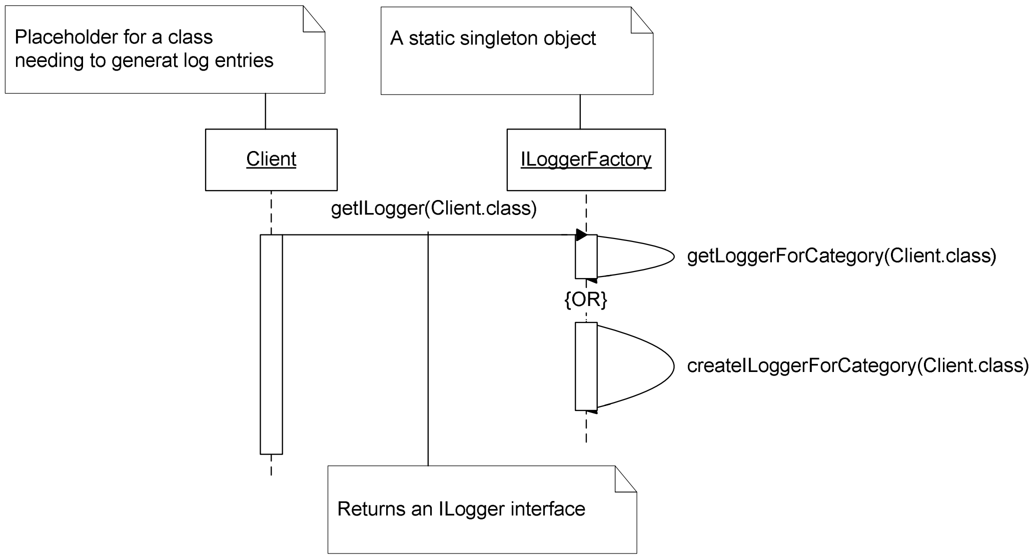
Placeholder for a class needing to generat log entries
A static singleton object
Client
ILoggerFactory
getILogger(Client.class)
getLoggerForCategory(Client.class)
{OR}
createILoggerForCategory(Client.class)
Returns an ILogger interface

Figure 4 Create new ILogger instance

The client calls the getILogger(…) method on the static factory class ILoggerFactory, supplying the class name of the client as an argument.

The factory uses the class name supplied as a Category. It tries to locate an ILogger related to the Category. If no ILogger currently have been created for that Category, a new ILogger object is instantiated.

4.1.2 Send a Debug Log Entry

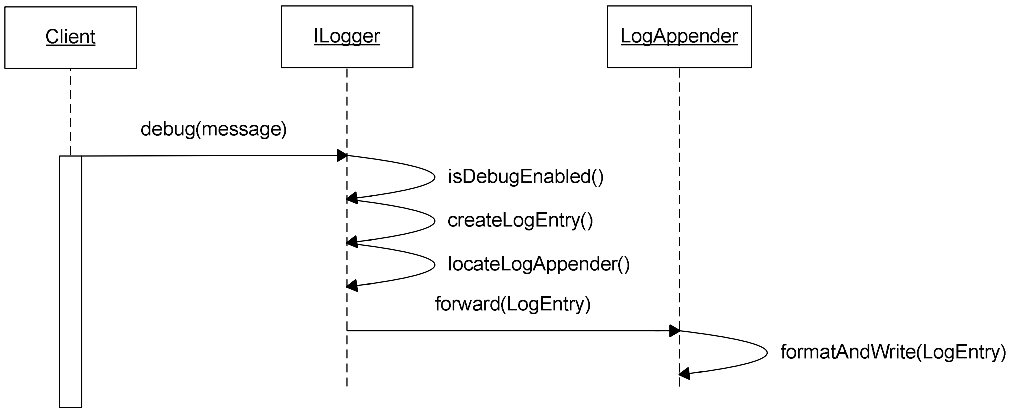

Figure 5 Send a Debug Log Entry

The client calls the debug(…) method on the ILogger object, supplying the message to log.

The ILogger verifies that the DEBUG LogLevel is enabled for this ILogger (i.e. for the Category corresponding to the ILogger object). If so, a Log Entry is created based on the log message, the date/time, the LogLevel (i.e. DEBUG in this example) etc.

The LogAppender for the specified Category and LogLevel is located, and the Log Entry is forwarded to the appender. The LogAppender format the Log Entry according to the configured rules. The formatted Log Entry is thereafter forwarded to the Log.

4.1.3 Register a new User Session with the Log component

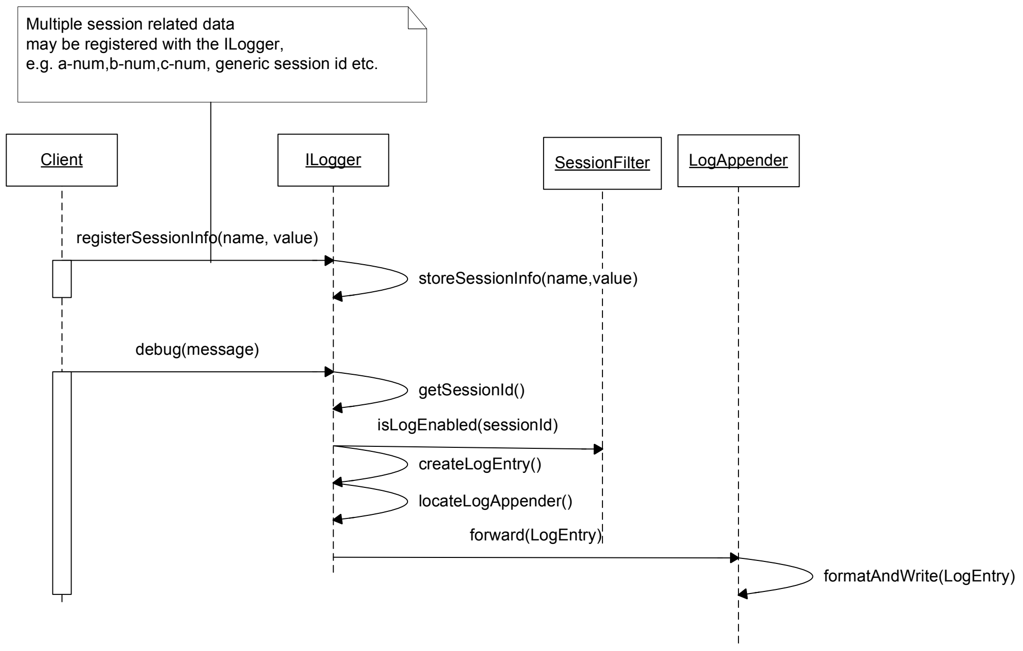
Multiple session related data may be registered with the ILogger, e.g. a-num,b-num,c-num, generic session id etc.
Client
ILogger
SessionFilter
LogAppender
registerSessionInfo(name, value)
storeSessionInfo(name,value)
debug(message)
getSessionId()
isLogEnabled(sessionId)
createLogEntry()
locateLogAppender()
forward(LogEntry)
formatAndWrite(LogEntry)

Figure 6 User Session filtering

When a new session is started, the client registers session specific information, such as a-num, b-num,c-num, session id etc. with the ILogger object.

Once the client generates a Log Entry by calling e.g. the debug method of the ILogger object, the information related to the session is used by the SessionFilter to deduce if Log Events related to this specific session should be logged or not.

If the SessionFilter acknowledge that logging is enabled for this session, the standard logging routine is followed.

4.1.4 Notifying that hosted service is available

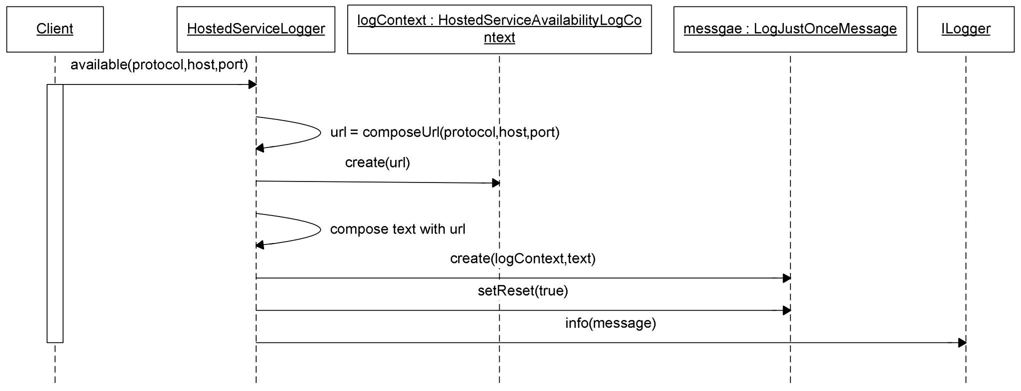
Client
HostedServiceLogger
logContext : HostedServiceAvailabilityLogContext
messgae : LogJustOnceMessage
ILogger
available(protocol,host,port)
url = composeUrl(protocol,host,port)
create(url)
compose text with url
create(logContext,text)
setReset(true)
info(message)

Figure 7 Notifying that hosted service is available.

1. The client calls the `available(…)` method on the `HostedServiceLogger` object, supplying protocol, host and port.
2. The `HostedServiceLogger` object composes an URL from protocol, host and port.
3. The `HostedServiceLogger` object creates a `HostedServiceAvailabiltyLogContext` object and composes a message text using the URL.
4. The `HostedServiceLogger` object creates a `LogJustOnceMessage` object using the log context object and the text, and sets the `triggReset` flag to true.
5. The `HostedServiceLogger` object calls the `info(…)` method on the `ILogger` object, supplying the `LogJustOnceMessage` object.

4.1.5 Notifying that hosted service is not responding/available

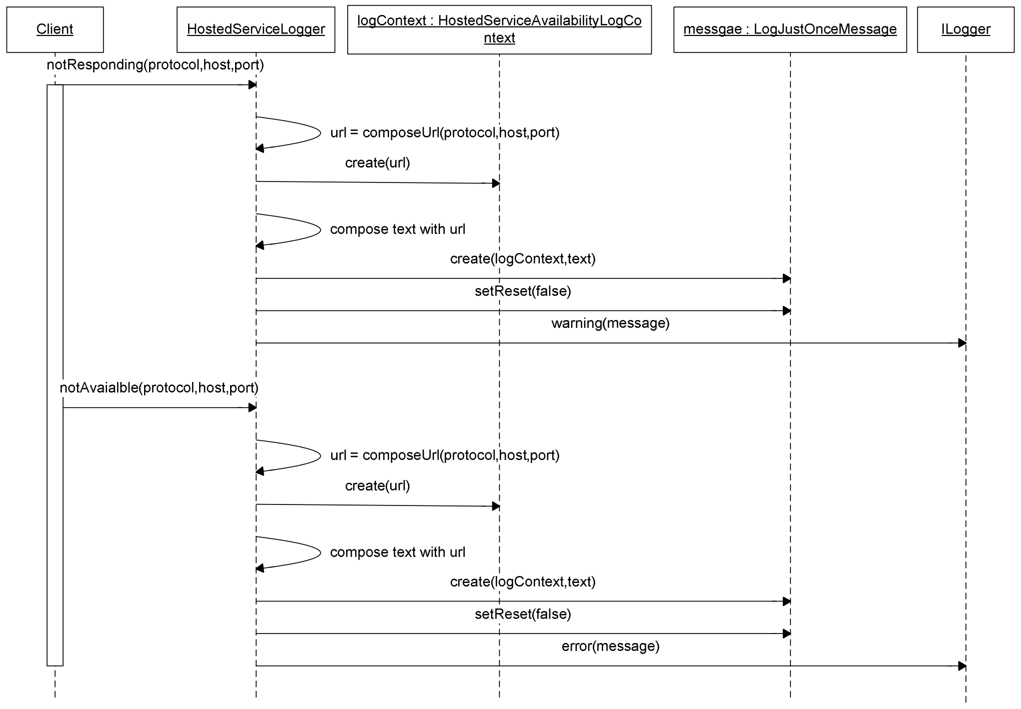
Client
HostedServiceLogger
logContext : HostedServiceAvailabilityLogContext
messgae : LogJustOnceMessage
ILogger
notResponding(protocol,host,port)
url = composeUrl(protocol,host,port)
create(url)
compose text with url
create(logContext,text)
setReset(false)
warning(message)
notAvaialble(protocol,host,port)
url = composeUrl(protocol,host,port)
create(url)
compose text with url
create(logContext,text)
setReset(false)
error(message)

Figure 8 Notifying that hosted service is not responding/available.

1. The client calls the `notResponding(…)` method on the `HostedServiceLogger` object, supplying protocol, host and port.
2. The `HostedServiceLogger` object composes an URL from protocol, host and port.
3. The `HostedServiceLogger` object creates a `HostedServiceAvailabiltyLogContext` object and composes a message text using the URL.
4. The `HostedServiceLogger` object creates a `LogJustOnceMessage` object using the log context object and the text, and sets the `triggReset` flag to false.
5. The `HostedServiceLogger` object calls the `warning(…)` method on the `ILogger` object, supplying the `LogJustOnceMessage` object.
6. The client calls the `notAvailable(…)` method on the `HostedServiceLogger` object, supplying protocol, host and port.
7. Repeats step 2-4.
8. The `HostedServiceLogger` object calls the `error(…)` method on the `ILogger` object, supplying the `LogJustOnceMessage` object.

4.1.6 Filtering repetitive messages

Below follows an example of filtering repetitive log messages. The instances of `LogJustMessage` are composed with a `HostedServiceAvailabilityLogContext` which has a wildcard based logic for determine if one log context implies another. E.g: `a.*` implies `a.b` but `a.b` does not imply `a.*`.

LogJustOnceMessage	Triggs reset	HostedServiceAvailabilityLogContext name
Message1	True	[HostedServiceAvailability;imap://luke:143].*
Message2	False	[HostedServiceAvailability;imap://luke:143].*
Message3	False	[HostedServiceAvailability;imap://luke:143].*
Message4	False	[HostedServiceAvailability;imap://luke:143].responding
Message5	False	[HostedServiceAvailability;imap://snake:357].*
Message6	True	[HostedServiceAvailability;imap://luke:143].*

Table 1 Example Message instance properties

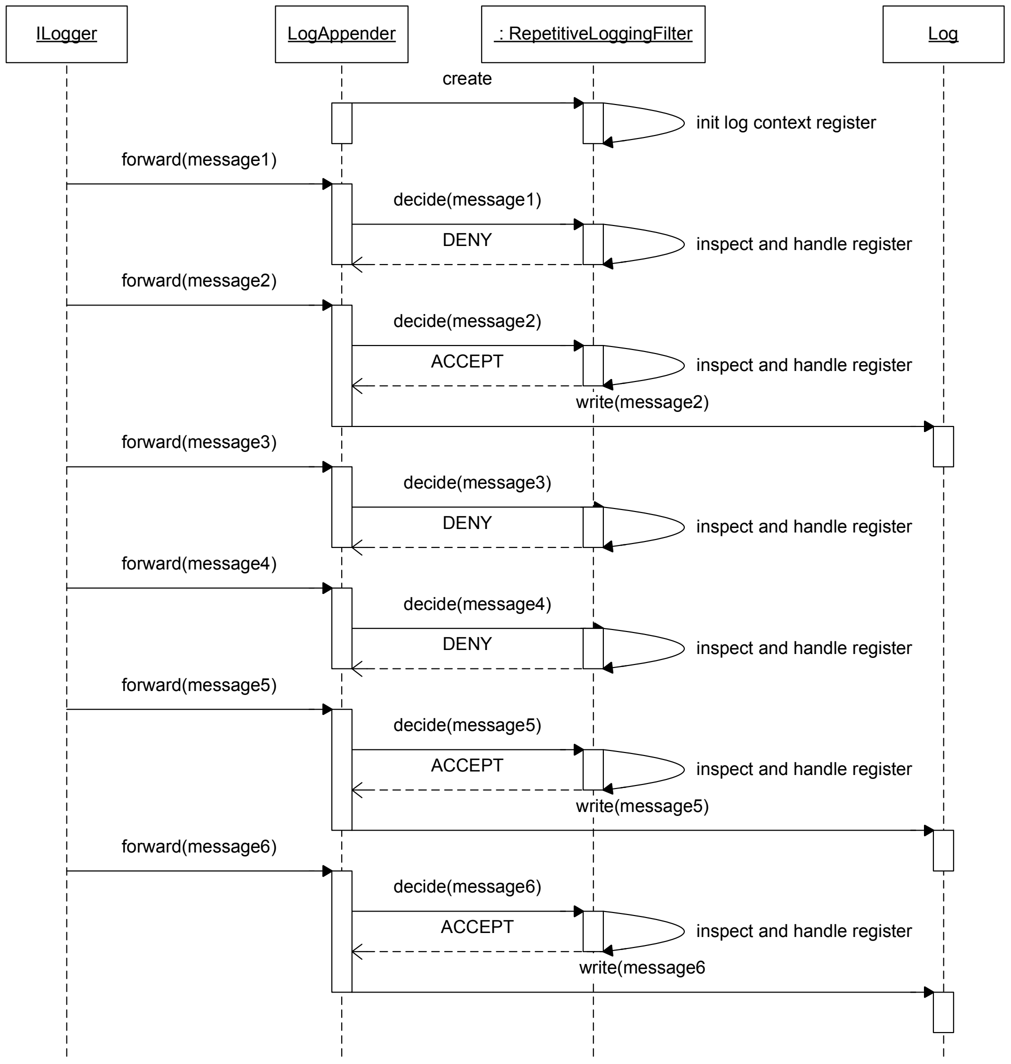

Figure 9 Example of repetitive log filtering.

1. System starts up. A `LogAppender` is configured to use a `RepetitiveLoggingFilter`. The `LogAppender` creates a filter instance which initializes its log context register (no entries).
2. `ILogger` forwards *message1* to `LogAppender`. *Message1* has the `triggReset` flag set to true and the log context register does not contain an entry equal to or implied by the log context of *message1*. Therefore will the filter decide to deny the `LogAppender` to write the message to the `Log`.
3. `ILogger` forwards *message2* to `LogAppender`. *Message2* has the `triggReset` flag set to false and the log context register does not contain an entry equal to or implying the log context of *message2*. Therefore will the filter decide to accept the `LogAppender` to write the message to the `Log`.
4. `ILogger` forwards *message3* to `LogAppender`. *Message3* has the `triggReset` flag set to false and the log context register does contain an entry equal to the log context of *message3*. Therefore will the filter decide to deny the `LogAppender` to write the message to the `Log`.
5. `ILogger` forwards *message4* to `LogAppender`. *Message4* has the `triggReset` flag set to false and the log context register does contain an entry implying the log context of *message4*. Therefore will the filter decide to deny the `LogAppender` to write the message to the `Log`.
6. `ILogger` forwards *message5* to `LogAppender`. *Message5* has the `triggReset` flag set to false and the log context register does not contain an entry equal to or implying the log context of *message5*. Therefore will the filter decide to accept the `LogAppender` to write the message to the `Log`.
7. `ILogger` forwards *message6* to `LogAppender`. *Message6* has the `triggReset` flag set to true and the log context register does contain an entry equal to the log context of *message6*. Therefore will the filter decide to accept the `LogAppender` to write the message to the `Log`.

5 References

[1] Logging Services Project at Apache Software Foundation, logging.apache.org as of 2005-06-08

6 Terminology

Term	Explanation