Centralized Logging with JDK 1.4

Abstract
Logging messages is an integral part of any application, particularly in a web-centric environment where it is difficult to run the code within a debugger. In fact, it is so important that a logging API is bundled with JDK 1.4. These log messages are not only used by developers for debugging their code but can also be used in production environment to troubleshoot problems that may result due to incorrect configuration. In a web-centric architecture sometime is it important to send all log messages to a centralized server, which eventually writes them to their final destination(s). This article introduces a generic mechanism to send and receive messages to a remote server by extending the logging API in JDK 1.4.

By Imran Hussain

Logging is a part of the infrastructure in virtually any development project. Its important is recognized by many software architects regardless of what platform or language they use to implement their system. Sun Microsystems will be bundling a logging API with JDK 1.4, which reaffirms its importance.

In a web-centric environment an HTTP user request can potentially pass through many tiers, possibly on different machines, each generating several log messages. Often it is desirable to collect these generated messages in one location in order to analyze them effectively. This mechanism is also known as centralized logging. This article talks about a generic way of adding centralized logging that is extensible to use any communication channel including JMS, RMI, UDP Messages, etc.

The logging API accompanied with JDK 1.4 is not only very simple but extremely extensible. The easiest way to create a centralized logging service would, obviously, be to extend the functionality provided by the JDK. Therefore, before going any further, it is important to talk about this API briefly. If you are already familiar with it you can skip the following section.

Logging API in JDK 1.4 – A primer

Logging API is an extensible, reliable and most importantly a high-performance API to send log messages to different destinations. This article does not explain all the features of the API in details. However, a short description is given below. For further details refer to the API documentation.

Loggers and Handlers

This API has two main components: Loggers and Handlers. Loggers are named entities that follow a hierarchical naming convention and are implemented by java.util.logging.Logger class. Creating a Logger object is the first step a developer would take in order to log a message. The hierarchical naming convention is achieved using a dot-separated name such as a package name in Java. In fact, Sun suggests that these names be aligned with package name, but is not required. This hierarchical nature allows children to inherit properties from its parent, which is its nearest extant ancestor in the logging namespace. Loggers inherit various attributes from their parent. The most important attributes are logging level and handlers. Besides its name, each logger object keeps track of a log level that it is interested in, and discards requests that are below this level. JDK defines seven built-in priority levels: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.

You can send the logged messages to multiple destinations using handler objects, embodied by java.util.logging.Handler class. This is the class that knows how and where to send a log message based on a configuration. Since this article deals with extending the JDK’s ability to send messages to a remote server, which can be treated as another destination, we will have to create classes that extend Handler.

Besides Logger and Handler, there are other important classes that must be studied before using the logging API effectively. These classes are LogManager, LogRecord, Formatter, FileHandler and ConsoleHandler. If you wish to learn more about these classes, refer to JDK documentation.

Let's look at an example code that creates a Logger object named "com.mycompany.myLogger" and then sends a simple message to its destination.

```
import java.util.logging.*;

public class RemoteLoggingClient {

    public static void main(String[] args) {
        RemoteLoggingClient remoteLoggingClient1 = new RemoteLoggingClient();
        remoteLoggingClient1.logTestMessages();
    }

    private void logTestMessages(){
        //Get an instance of Logger. The name of this logger
        //is myLogger.
        Logger logger = Logger.getLogger("com.mycompany.myLogger");

        //Next line sends a log message to the logging API. This
        //message will only get logged if the logging level for
        //com.mycompany.myLogger is set to an equal or higher than
        //WARNING
        logger.log(Level.WARNING, "Having sleepless nights");
    }
}
```

Notice that the above code does not specify what the actual destination of this message is. This information is provided by a configuration file. By default, JDK uses logging.properties file that is located in $JAVAHOME/jre/lib/ directory. The location and name of this file can be specified by setting a system property. An alternative method to modify configuration information is by manually setting properties of the Logger object within the code.

Problem analysis

Now let's talk about centralized logging, which has two modules. The first one is to send a message to a remove server and the second is the remote server itself that accepts a message from a client and forwards it to an appropriate destination. Another important factor to consider is the channel that is responsible for carrying our request from the client machine to the server. Before going any further, let's lay down some ground rules that our design must adhere to.

1. The design should be extensible enough to use any channel for data transfer that a programmer decides. For example JMS, RMI, Soap, etc.
2. Once a log message gets to the server, it should have to ability to send this logged message to other configurable destinations.

3. In case of a delivery failure, the logging framework should not block.

Figure 1 shows the architecture of this design.

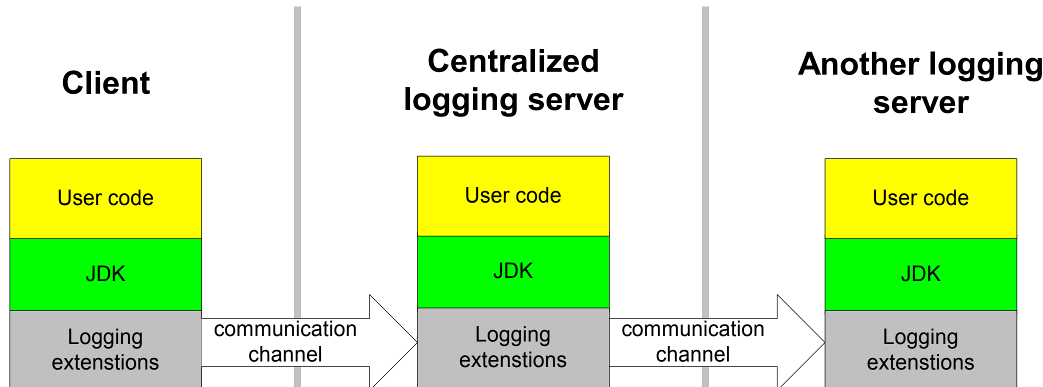

Figure 1

Writing client code

Now, let's get dive into the coding area. We begin by analyzing the Handler class in logging API. This is an abstract class and therefore, a descendent class must override a few methods. These are:

- close() – Allows the extended classes to free up all the resources
- flush() – Indicates the clients to flush any buffers that they may have.
- publish() – This is the main work-horse method that sends the data to its target and will get called automatically by the API if certain criteria is met – that is when the message's priority level is above the configured level for this logger and it passes through all the associated filters.

The publish method expects a LogRecord parameter, which holds the data that needs to be saved. LogRecord object is serializable, which makes our life easier to transmit it over to the network. However, there are a few catches in serializing this object.

First, there are a couple of methods, getSourceMethodName(), getSourceClassName(), in the LogRecord class that can potentially parse the stack trace to retrieve the method and class name of the caller. Obviously, the stack trace on a remote machine is not same as the client and a call to these methods on the server machine will not produce desired data. A work around to this problem is to pass the values for class and method name explicitly

in all calls to the log() method or call these function before sending the object over the network. Remember that if the framework has to parse the stack trace for the class and method names, it is going to take extra time for every message, which can add up to a significant number and can affect overall performance. In my opinion, the solution to this problem depends on your environment and what fields do you need in the output. In my projects I explicitly set these two parameters to an empty string if they are not provided by the caller.

The second point to keep in mind is that all the objects in the parameter array may not be serializable. Parameter array is a list of name-value pair that a user can use to add extra logging parameters. The value objects of this list are first converted to its string representation and then serialized. Therefore, if you add any object other can a String to the record's parameter list, by using LogRecord.setParameters(…) method, you will have to reconstruct it from its string representation on the other end.

The third catch is the availability of the resource bundle on server, which will not be same as the client. However, as long as the same resource exists on the remote machine with the same name, the API will try to locate it and return a reference to it.

Generic Channel

In order to accommodate for multiple transfer channels, we will create a RemoteHandler class that will delegate the actual logging work to a any class that implements a RemoteChannel interface. The relationship between RemoteHandler and RemoteChannel is displayed in figure 2.

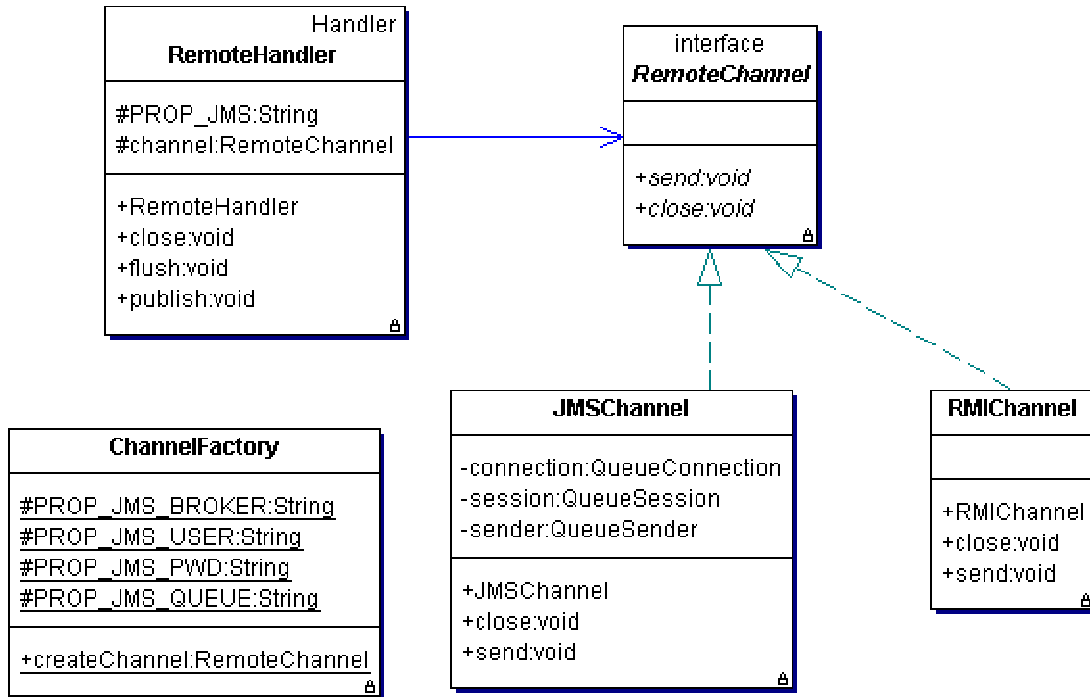

Figure 2 - Relationship between RemoteHandler and RemoteChannel

For the sake of this article I have only implemented two RemoteChannels - one uses JMS and the other uses RMI. However, the design can be extended to use any RPC mechanism including SOAP, CORBA, SMTP, or plain UDP sockets.

Notice that each remote channel that is implemented can have its own set of configuration parameters. These parameters can be set in the logging.properties file and can be read by using the LogManager.getLogManager().getProperty(…) function. In this case I have implemented a special factory class that reads parameters from the configuration file and creates appropriate channel object.

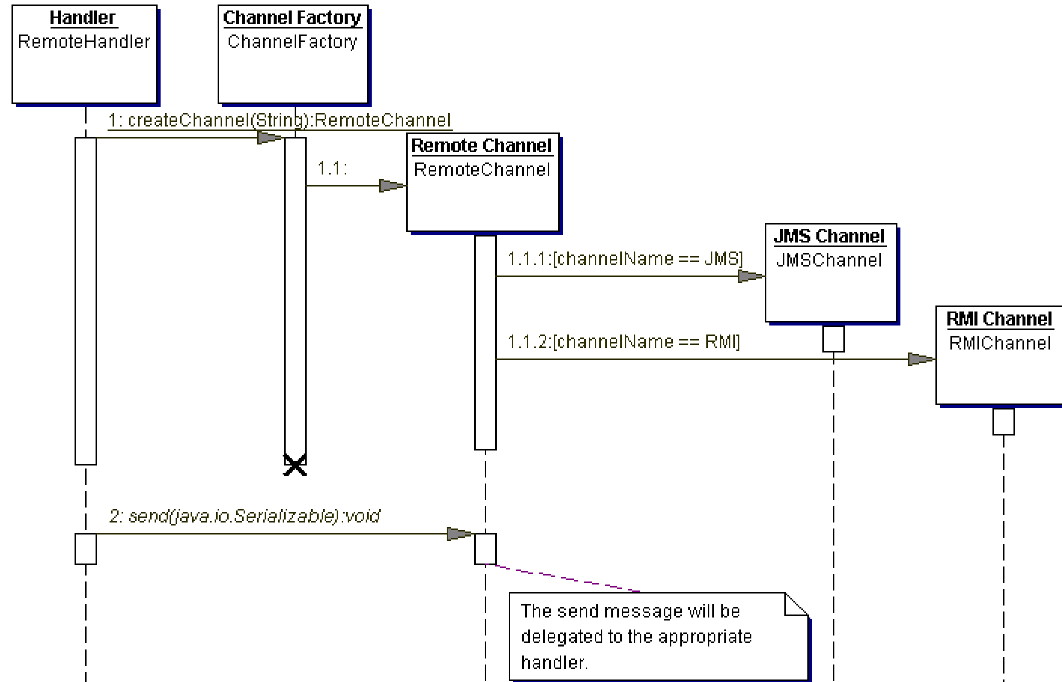

Figure 3 - Sequence of events on the client side

Figure 3 shows the sequence diagram explaining different steps that are taken by the framework. The figure can be summarized by following steps:

1. The RemoteHandler calls the createChannel(…) method in ChannelFactory to create appropriate channel
2. Once a handle to a channel is retrieved, its send method is called.

Implementation of the send method depends on the actual messaging channel being used. Lets dissect the JMSChannel class to see what it does. I recommend that you download the source code and refer to JMSChannel.java file as you are reading this text. There is only one constructor in this class that takes all the required parameters for JMS. This constructor is responsible for creating all necessary objects required to send messages to a JMS broker. If for any case the client cannot talk to the JMS broker, it will print out a stack trace message to the console notifying the user about the problem but the object will be created. The creation of this object even when no connection is established is important in any logging framework. This way the constructor does not cause null pointer exceptions to occur in the caller code and the framework continues to work as if everything is normal.

The second function that is important to understand is the send() method. Its code is listed below.

```java
public void send(java.io.Serializable messageToSend){
    try{
        if(sender != null){
            sender.send(session.createObjectMessage(messageToSend));
        }
    }catch(JMSException jmse){
        System.err.println("Could not send message to JMS broker");
        jmse.printStackTrace();
    }
}
```

Notice that if the sender object is null, the function simply exits without doing anything. The sender can be null if the constructor was not able to connect to the broker for any reason.

Another thing to notice is the type of the messageToSend object, which is declared as Serializable in the method signature. This is actually an instance of the LogRecord class provided by JDK.

Similar tactic is used in case of RMI. However, instead of connecting to a JMS broker a reference is obtained from the RMI registry and a remote method is called to transfer that data over to the server.

Writing server code

Now let's look at the other side of the picture. The objective here is to retrieve the message object sent by the client, cast it back to a LogRecord object and use the logging API to send it to its final destination.

The classes on this end are designed in a similar fashion, which is to accommodate for different communication channels. A ReceiverFactory class creates an appropriate receiver object, which is then used to receive messages. Figure 4 illustrates the relationship between the server side classes.

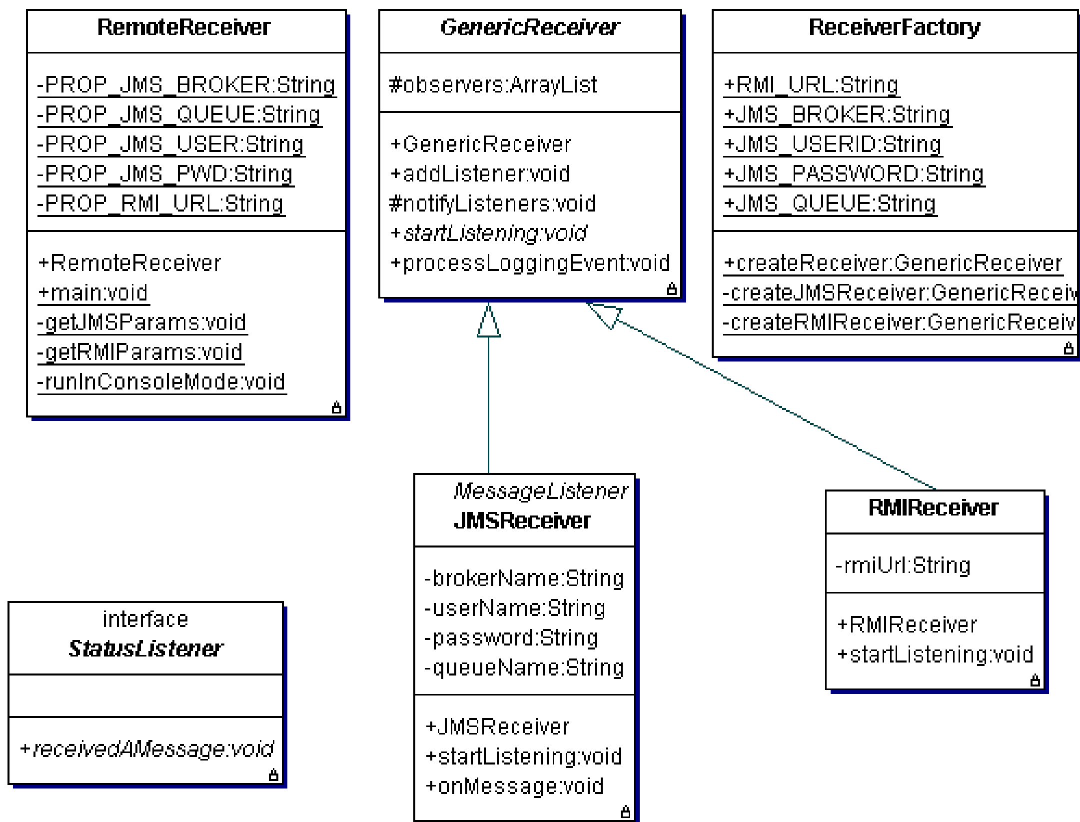

Figure 4 - Server side class diagram

Interaction between these classes is presented by two sequence diagrams: Figure 5 and 6. Both of these figures assume that we are using JMS as the RPC mechanism. The former diagram shows the communication that is initiated by a server application. In the example code this is a console based application represented by the RemoteReceiver class. First, this application uses the receiver factory to create an appropriate receiver message and gets a reference to the receiver. Second, the application calls the startListening() method on the receiver that establishes connection and registers to the receiving channel – a JMS broker in case of JMS and or an RMI registry in case of RMI.

Figure 6 illustrates the interaction initiated by the broker, which calls the onMessage method whenever a message is received. The onMessage method is responsible for the following tasks.

1. Extract the ObjectMessage from jms.Message object.
2. Cast this object to LogRecord.
3. If you used any additional parameters in the logRecord object, reconstruct it.
4. Call the processLoggingEvent() method, which will log a message using the local logging API.

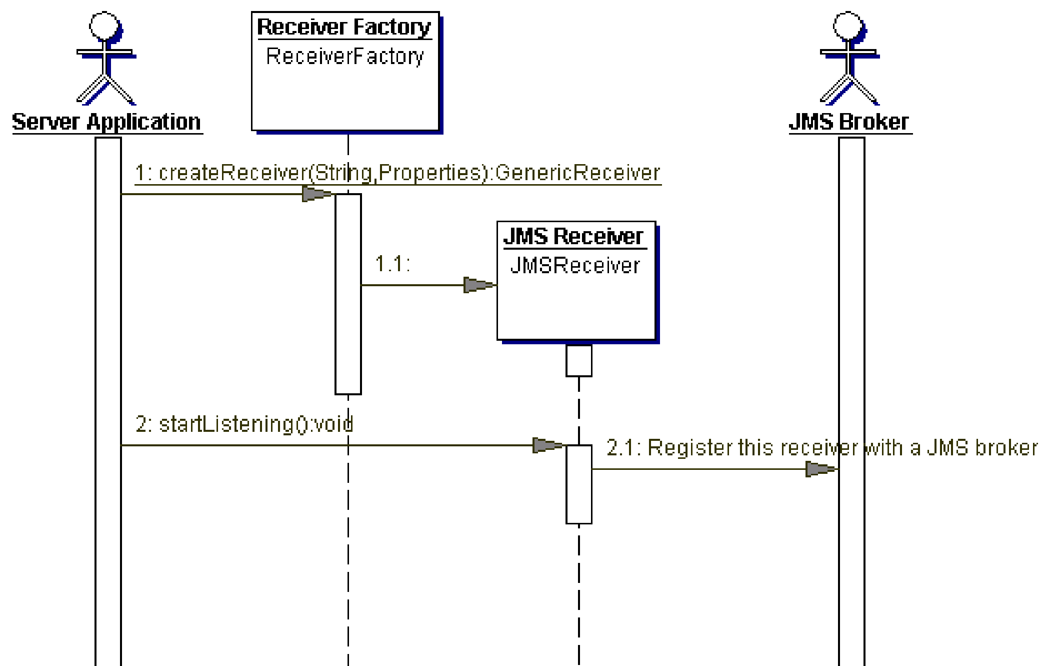

Figure 5 - Sequence of events initiated by a server application

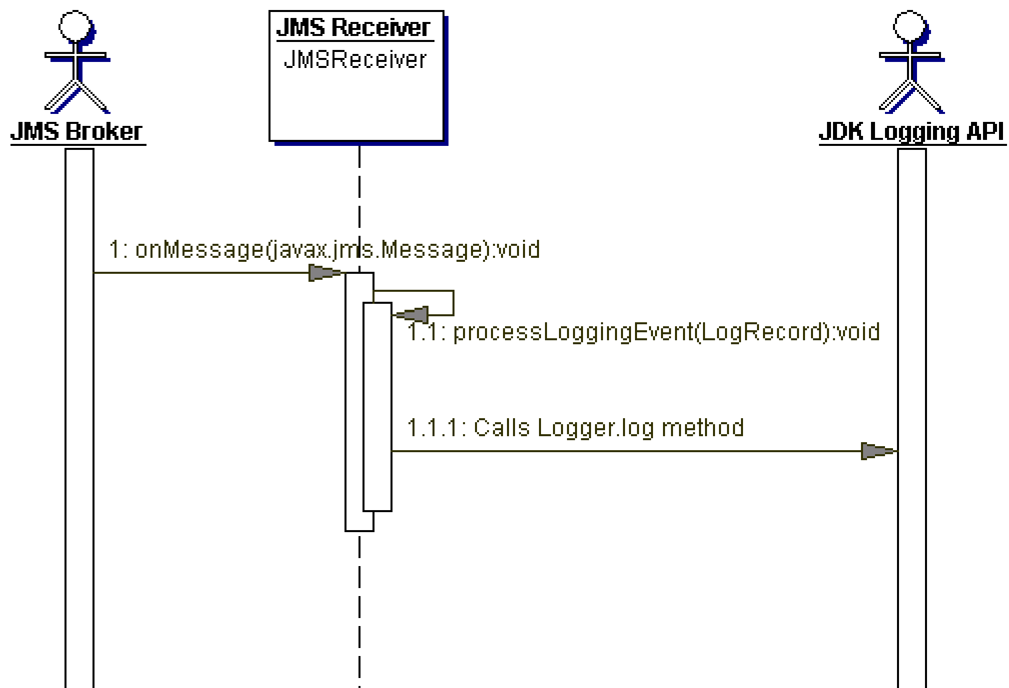

Figure 6 - Sequence of events initiated by a JMS broker

Message notification

Besides processing a remote client message, this article also demonstrates a mechanism to notify the application about the arrival of these messages. This feature is very useful if you need to keep track of how many messages are received on the server side and it is accomplished by implementing observer pattern (Gof). Server application in this case implements the StatusListener interface and calls the addListener() method of the GenericReceiver object. When a message arrives, the receiver notifies all listening objects, which update their respective buffers. Refer to the comments in the RemoteReceiver.java file for details.

Performance considerations

Performance is a key non-functional requirement for any logging API. In a production environment there can be, potentially, millions of request to log messages. Therefore, it is very important that our client as well as server code does not perform any lengthy tasks. The use of JMS on as a channel relieves the client from blocking. However, if the rate of sent messages is higher then what the server can handle, the queue will pile up and eventually saturate. Consider the following tips to improve performance.

When using JMS

- Avoid lengthy operations in the onMessage method
- If the rate of incoming message is higher than what the server can handle, use multiple JMS sessions. This will create multiple threads of execution and the message queue won't saturate with messages.
- Consider using clustered broker for load balancing
- Avoid using PERSISTENT delivery mode
- If your business logic allows, change the session acknowledge mode to DUPS_OK_ACKNOWLEDGE
- Avoid transactions.

When using RMI

- Use an in-memory queue on the server side to push incoming messages and pop these messages in a secondary thread.

Summary

In this article we demonstrated how to leverage the logging API in JDK 1.4 to send messages to a remote server, which acts as a logging client to a local API on that machine. We discussed a generic mechanism to transfer the log record objects using JMS and RMI. This mechanism is extensible to including any channel for data transfer

including SOAP, CORBA, UDP sockets or even home grown RPC methods. These techniques should help you design a remote logging framework in your system.

About the author

Imran Hussain is a Java Architect working for Sun Microsystems. Imran has worked on several projects that required logging and is responsible for designing logging frameworks within Sun ONE infrastructure. He holds bachelors degree in Computer Engineering and has over nine years of industry experience. He is a veteran C++ programmer and started Java in early 1997 and has been actively involved in J2EE architecture development for the pass two years.