DATA LOGGER FOR MEDICAL DEVICE COORDINATION FRAMEWORK

by

KARTHIK GUNDIMEDA

B. Tech., Jawaharlal Nehru Technological University, 2009

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2011

Approved by:

Major Professor
Dr. Daniel Andresen

ABSTRACT

A software application or a hardware device performs well under favorable conditions. Practically there can be many factors which effect the performance and functioning of the system. Scenarios where the system fails or performs better are needed to be determined. Logging is one of the best methodologies to determine such scenarios. Logging can be helpful in determining worst and effective performance. There is always an advantage of levels in logging which gives flexibility in logging different kinds of messages. Determining what messages to be logged is the key of logging. All the important events, state changes, messages are to be logged to know the higher level of progress of the system.

Medical Device Coordination Framework (MDCF) deals with device connectivity with MDCF server. In this report, we propose a logging component to the existing MDCF. Logging component for MDCF is inspired from the flight data recorder, "black box". Black box is a device used to log each and every message passing through the flight's system. In this way it is reliable and easy to investigate any failures in the system. We will also be able to simulate the replay of the scenarios. The important state changes in MDCF include device connection, scenario instantiation, initial state of MDCF server, destination creation. Logging in MDCF is implemented by wrapping Log4j logging framework. The interface provided by the logging component is used by MDCF in order to log. This implementation facilitates building more complex logging component for MDCF.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My special thanks to my major professor Dr. Daniel Andresen for giving me timely advice, encouragement, and guidance throughout the project.

I would also like to thank Dr. Gurdip Singh and Dr. Mitchell Neilsen for graciously accepting to serve on my committee.

I would like to thank the administrative and technical support staff of the department of CIS for their support throughout my graduate study.

CHAPTER 1: INTRODUCTION

Logging a system is considered as a significant aspect of the development process. It is beneficial is various ways, such as diagnosing problems, tracking system progress, and so on. The advantages of establishing a data logger motivates to create a logging component for the Medical Device Coordination Framework (MDCF). Data logger for MDCF is essential as it is a distributed architecture which runs on different machines. The server and its clients may be interrupted due to several reasons. Thus the important objective of data logger is to resolve any errors and diagnose the problem. Besides that data logger helps in storing important changes in the system. From these observations, introducing data logger into MDCF is a significant step.

In real world scenarios, medical servers in hospitals store information about patients. Servers can be setup for every ward or centered in small hospitals. MDCF serves as a distributed server for a hospital. Logging information in such environment is a challenging task. The system has to archive important log results. At the same time, as the number of patients increase the performance and scalability will be affected. Hence, determining an effective logging framework for MDCF is the mail goal of this report. The logging framework besides logging device messages and events also logs internal server messages of MDCF. Logging at device side provides details about the behavior of patient under medication in real world. Logging on server side determines the health of a device's connection. It is not possible for a nurse to attend a patient for the whole time. There should be some system that notifies a nurse when something abnormal has happened. To test the functionality of the server and to determine where things went wrong, there is a need for a logger to keep track of events. This helps in diagnosing any problem easily and taking respective measures to handle it.

The logger for MDCF is inspired from the black box recorder idea in an airplane cockpit. Here we try to log every message and event passing through MDCF. One of the main objectives of MDCF data logger is to deterministically replay the scenarios whenever necessary. The replay of the scenario can be based on various filters such as timestamps, scenarios, events, etc. For this every message and activity must be logged. In the existing MDCF, java.util.Logger is used to log the events. This approach of logging logs specific activities performed in MDCF Server. To overcome this issue, we have to build a more effective logging system which satisfies our

requirements. In this report, an independent data logger component is introduced to MDCF. The data logger can be configured to log all events and messages and also log only events and important messages. The data logger gets message to logged as object type as input. The messages and events will first get logged into in-memory log and then filtered into a log file. At the same time the performance and scalability of the server should be given importance as in a real world scenario, medical devices should be fast and effective within given resources. For this objective to be achieved, Log4j framework is used as an underlining framework to log the incoming objects. Among the available logging frameworks, Log4j is more effective and is suitable to achieve our requirements. This is discussed later in coming sections of the report.

The rest of this paper first discusses related work in chapter 2, and then describes our implementation in chapter 3. Chapter 4 describes how we evaluated our system and presents the results. Chapter 5 presents our conclusions and describes future work.

CHAPTER 2: RELATED WORK

Logging can be implemented in various ways. Choosing a methodology of logging depends on the requirements and importance of logging in the application. In a broader sense, there can be two approaches towards logging. First is to developing the logging framework from the scratch. Second approach is by using existing logging frameworks such as log4j, commons, etc. The existing logging system present in Medical Device Coordination Framework uses Java's logging API. The current logging system is a simple and ineffective approach. It only concentrates on logging connected device details. On the other hand, there is no logging provided on device side. Java's logging API provides less flexibility in appenders, layouts and handlers for logging. The implementation of logging proposed in this report using log4j is beneficial over using Java's logging API. It provides a base for building more complex logging framework. It has many features and options in developing custom appenders and layouts. It is easy to use and does not have a significant effect on the overall performance of the system.

Log4j developed by Apache Software Foundation is preferred over Java's Logging API due to several factors. They differ most in the areas of useful appender/handler implementations, useful formatter/layout implementations, and configuration flexibility. Log4j is considered as well-developed logging framework with good support. There are other logging frameworks besides Log4j. Table 1 presents a brief comparison of Log4j with other logging frameworks. As we can see, Log4j has many in-built appenders when compared to other logging frameworks. Log4j is widely used among the applications, since it provides flexibility is configuration and is performance effective.

Apache Commons Logging is used together with log4j as its underlining framework. This in turn depends on appenders and layout provided by underlining framework. It provides the same logging level priorities as of Log4j. By using commons log, we need to configure two configuration files. The logging framework implemented is dependent on the deployment environment.

SLF4J is better in configuration than Apache Commons Logging. It allows user to use any underlining logging framework. SLF4J is still in evolving stage and is a next generation logging proxy.

Framework	Supported log levels	Standard appenders	Popularity
Log4J	FATAL ERROR WARN INFO DEBUG TRACE	AsyncAppender, JDBCAppender, JMSAppender, NTEventLogAppender, NullAppender, SMTPAppender, SocketAppender, SocketHubAppender, SyslogAppender, TelnetAppender, WriterAppender	Widely used in many project and platforms
Java Logging API	SEVERE WARNING INFO CONFIG FINE FINER FINEST	ConsoleHandler, FileHandler, SocketHandler, MemoryHandler	Not widely used
Apache Commons Logging	FATAL ERROR WARN INFO DEBUG TRACE	Depends on the underlying framework	Widely used, in conjunction with log4j
SLF4J	ERROR WARN INFO DEBUG TRACE	Depends on the underlying framework, which is pluggable	

Table 1: Comparing logging frameworks

Based on the presented architecture, one can build a logging system which handles real-time constraints and also can archive important activities into a permanent storage. Medical devices publish data at a high frequency. Most of the data from the device is normal most of the times. We eventually need a memory efficient logging system. This logging implementation reduces the writes on disk by introducing an in-memory log. Log is written to a file in regular intervals by a filter. In the previous version of MDCF, logging is file based, which does not offer flexibility to log every message and event. Performance and scalability of the system is affected when we introduce a heavy logging system. To handle this case, the proposed architecture provides a configuration file where we can totally log events and messages and choose to log partially.

CHAPTER 3: IMPLEMENTATION

3.1 Java Messaging Service

Medical Device Coordination Framework (MDCF) is built on Java Messaging Service (JMS) as a basic architecture. To achieve the goal of developing an enterprise oriented service, JMS acts as message-oriented-middleware foundation to the system. JMS provides flexibility in choosing the type of topologies, point-to-point and publish-subscribe topologies. It is an open source technology which is performance effective in all the modes it provides. Modes involve delivery type, acknowledgments. In delivery types, it provides persistent and non-persistent types. In acknowledgements, it provides synchronous and asynchronous messaging.

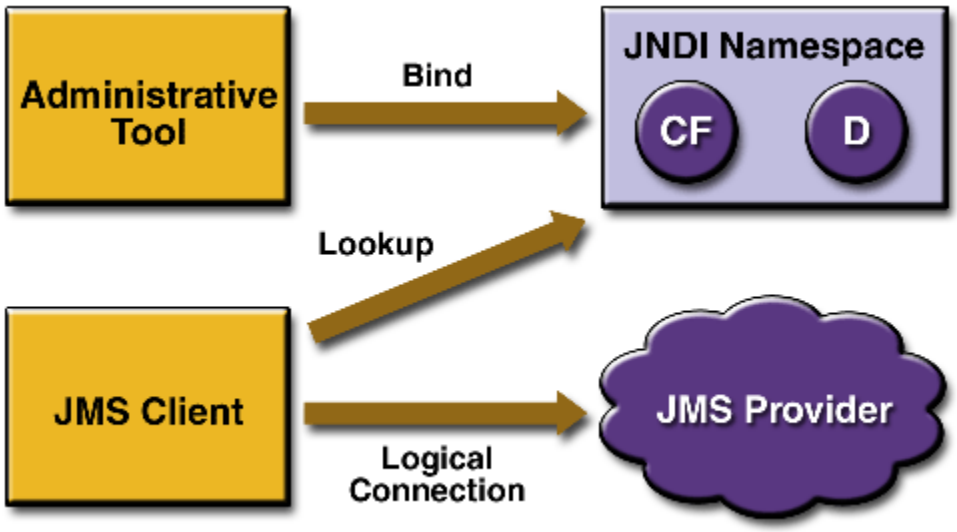

Figure 1: JMS API Architecture

http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/basics.html#1023437

Figure 1 presents the JMS programming architecture. The important parts in JMS architecture are Administrative tools, JNDI Namespace, JMS Client, JMS Provider and Messages. Administrative tools bind destinations and connection factories into a Java Naming and Directory Interface (JNDI) namespace. Administrative objects include connection factory, connection and session objects. A connection is established from session object. A JMS client looks up the administered objects in the JNDI and then establishes a logical connection to the same objects through the JMS provider.

3.2 OpenJMS vs ActiveMQ

The previous version of MDCF is implemented using OpenJMS as its messaging broker. OpenJMS is a provider of Java Messaging Service. Advantage of OpenJMS is its vendor neutral and each vendor's implementation is different from others. Users generally agree on it for its interoperability layers. OpenJMS provides key features such as synchronous and asynchronous message delivery, local transactions, authentication, XML-based configuration files, in-memory and database garbage collection, automatic client disconnection detection. It supports modules such as HTTP Tunnel, Common, etc. It has been very slow in its development versions and less support.

Apache ActiveMQ is a well-developed open source message broker implemented in Java. The current version of MDCF uses ActiveMQ as JMS provider. ActiveMQ integrates well into other products and is considered stable and high-performance application when compared to OpenJMS. It provides many extensions for messaging such as Virtual Destinations for load-balancing and failover for topics, Retroactive Subscriptions such that subscriber can receive some number of previous messages on connect, Mirrored queues to monitor queue messages and many more. ActiveMQ can be also used from .NET, C/C++ or from scripting languages like Perl, Python, PHP and Ruby via various Cross Language Clients together with connecting to protocols such as wire-level protocols, OpenWire protocol. It provides significant options for persistent messaging. Options include ability to use local files, database, VM, cache. ActiveMQ is easy to configure to serve many scenarios (high scalability, security,etc) as it is a XML configuration file driven application. It is more effective in distributed architecture and provides enterprise features such as clustering in two different strategies, Master/Slave and Network of Brokers. ActiveMQ's performance when real-world factors are considered is much faster than other JMS providers. Some of the real-world performance factors include number of producers and consumers, message size, clustering, hardware, etc. It provides many ways for performance tuning depending on the requirement of the system.

3.3 Medical Device Coordination Framework (MDCF)

Before introducing logging in MDCF, this section projects the important internal modules developed in MDCF. Device Manager in MDCF is useful in connecting new devices to the system. It listens on an administration channel for messages sent by mock devices which desire to connect to MDCF. Device Manager follows an authentication protocol that looks for the device in existing device database before approving the device. MDCF includes consoles to manage the devices and scenarios. Consoles such as clinician console which provides visualization of device, instantiation of device coordination scripts. The other important module is scenario manager. It manages the scenario scripts, determining the devices needed in the scripts and component creation.

3.4 Logging and Log4j

In general logging refers to recording of events and messages involved in the system. Logging is one of the important modules to be considered by the development team. **Basic Functionality of Logging:** Any logging framework has three major components, Logger, Formatter and the Handler/Appender. The logger acts as an interface between the system and the logging system. It is responsible for collecting the messages to be logged and sending it to the logging framework. The next step in the logging architecture is calling the Formatter. The Formatter takes the message object as input and formats it for output. The logging framework then takes the formatted message to the correct Appender associated with it. The job of Appender is to append the formatted logged message to appropriate destination. Appenders can be a file, console, in-memory, etc.

The system can log a message as object or exception along with message. It can also specify a severity level with a given a name. Name here refers to the class or package performing the logging. A message can be logged at certain levels. The common levels used in a logging framework are listed in the Table 1.

Level	Description
FATAL	FATAL is a message level indicating a serious failure.
ERROR	Other runtime errors or unexpected conditions.
WARNING	WARNING is a message level indicating a potential problem
INFO	INFO is a message level for informational messages.
DEBUG	Detailed information on the flow through the system. It is a message level providing tracing information.

Table 2: Common Priority Levels

There are many advantages of using logging framework in a system. Here, the basic idea of black box recorder is used to simulate a scenario which is completed. It benefits in tracing any problem occurred in a system. The foremost advantage of any logging API over plain System.out.println is in its ability to disable certain log statements while allowing others to print independently. At the same time, implementing logging will cause a performance overhead to the system. Considering the issue of performance and ease to use a logging system, Log4j framework is preferred over other logging frameworks.

Log4j:

Log4j is developed by The Apache Software Foundation. Log4j is fully configurable at run time using external configuration files. It also provides a feature to the developer to handle which log statements are to be logged with priorities. Each module in Log4j is implemented in the following way.

Named Hierarchy "A logger is said to be an *ancestor* of another logger if its name followed by a dot is a prefix of the *descendant* logger name. A logger is said to be a *parent* of a *child* logger if there are no ancestors between it and the descendant logger."

Level Inheritance "The *inherited level* for a given logger *C*, is equal to the first non-null level in the logger hierarchy, starting at *C* and proceeding upwards in the hierarchy towards the root logger."

Basic Selection Rule "A log request of level *p* in a logger with (either assigned or inherited, whichever is appropriate) level *q*, is enabled if *p* >= *q*."

Appender Additivity The output of a log statement of logger *C* will go to all the appenders in *C* and its ancestors. Loggers have their additivity flag set to true by default.

Layouts: The PatternLayout lets the user specify the output format according to conversion patterns similar to the C language printf function. For example, the PatternLayout with the conversion pattern "[%d {ISO8601}] %5p %6.6r [%t] %x - %C. %M (%F: %L) - %m%n" will output something as: [2011-04-19 18:44:55,508] INFO 0[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ (ActiveMQMessageBusManager.java:77) - Topic created: toAdmin

Configuration: The log4j environment is fully configurable programmatically. However, it is far more flexible to configure log4j using configuration files. Currently, configuration files can be written in XML or in Java properties format.

3.5 MDCF Data Logger Architecture

MDCF Data Logger provides an interface to the MDCF system. Server, Devices and Consoles use this interface to log the messages and events. MDCFLogger in turn uses Log4j logging framework to implement the logging system. It acts like a wrapper around Log4j and provides only the interface layer of logging to MDCF. It is easy to change the underlining implementation from Log4j to any other framework without affecting the code on MDCF side. Figure 2 represents the architectural overview of MDCF data logger. We can see that the important modules of MDCF as we discussed earlier communicates with MDCFLogger and send the messages (and excpetions) to be logged. Devices or clients can be both producers and consumers. These act like stand-alone components which instantiate a separate logger instance in it. For instance, a mock device named TickTockEmitter gets the logger instance from MDCFLogger which is independent from other instances of different devices. Thus we can differentiate each client by its name in the log file. Console as well does a similar functionality to implement the logger. Each type of console (such as AdminConsole, ClinicianConsole, ScenarioConsole) creates its own logger instance and sends the messages to be logged through

the instance. Unlike Devices and Consoles, MDCFServer and its inner managers utilize the logger in a different way.

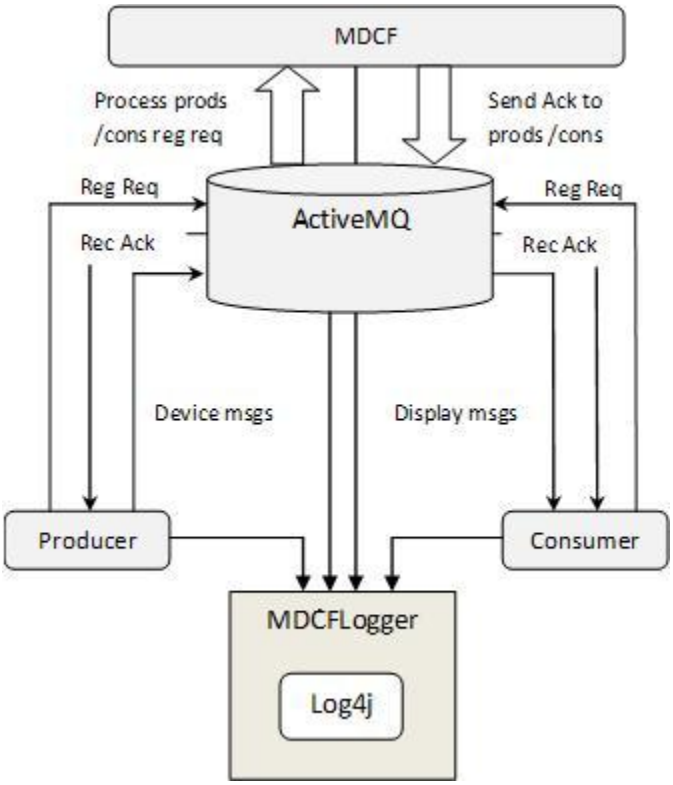

Figure 2: Overview of Architecture

In figure 2, the basic connection protocol of the devices with the server is also shown. All these are significant activities which are logged. For instance, a producer sends request for registration (Reg Req) to server through ActiveMQ. Server then receives this message, processes it and then sends back the acknowledgement (Rec Ack) to the producer. The producer starts emitting its message (device msgs) to ActiveMQ on a designated topic. The process is similar to consumer but the consumer listens for messages instead of publishing on a designated channel. All these steps are logged by MDCFLogger. When a producer or consumer is created a logger instance is also created for each client (producer or consumer). Similarly, when MDCF server is instantiated for the first time, a logger instance is created within the server and logging is done with the help of logger instance.

The logger handles total logging and partial logging in a different way. In partial logging, logger gets messages from the components (clients and server) directly. Each component makes use of the instance and the interface provided by the logger to send the messages. The instance is

used to collect the messages and is embedded in the component's code. As there are not many messages or events to be logged when compared to total logging the code is included in the component. Partial log runs when the priority level is INFO. In this case, once the message receives the logger, then a condition is checked if the level is in DEBUG. Since it is partial logging, every message on every topic is not logged. Thus only messages sent by the components are logged. In the appender of the logger, all these messages are collected in an in-memory data structure. This supports performance of the server by not getting connected to ActiveMQ. In partial logging, only important messages and events are logged which are sent by the components. Depending on the number of appenders the overall performance of the framework is affected.

In total logging, every message on all topics created is logged. Here, messages to be logged are received directly from components and also from ActiveMQ unlike partial logging, where in we only get log messages directly from the components. Total logging is enabled when the priority level of logging is in DEBUG mode. When the message to be logged is received by MDCFLogger, then appropriate log level is executed and the appenders are called. In MDCFAppender, append method is called for the messages to get appended in a specific format. Here, a condition checks if the level is in DEBUG. If it is true then we scan the message for new topic name. All the topic names are eventually collected. For each topic collected a new message listener is created in the appender. Thus, MDCFLogger is connected to ActiveMQ in DEBUG mode while total logging. Along with MDCF server and other components, now MDCF logger also tabs the topic for messages. It collects the messages and then adds to the in-memory logger. We can thus know when the device disconnects, the messages from that particular topic is stopped. This approach is more effective than getting messages directly from MDCF framework. Since, using the MDCF logger instance to send every message to logging system there would be an overhead of calling logger each time a message is sent. As the frequency of medical devices is high, this approach helps in improving the performance and scalability of the system. It is also reliable to get the messages from ActiveMQ. By following this method, the need to log every message received by consumer is avoided. Although we need an instance of logger to log connection protocol events of the consumer, we need to log every message received by the consumer. Total logging contains more overhead than partial logging as in total logging phase,

the logger gets messages to be logged from ActiveMQ. But it is effective than getting many calls from the components which have other overheads in them.

Logger instance in each component is used in different ways. Logger instance is simple in case of clients and consoles but different in case of MDCF server. In case of clients, logger instance is created for each client and used to call log method to send the messages to be logged. Similarly for consoles, for each kind of console a logger instance is created to used to send messages to be logged. MDCF server itself has many parts in it. Creating logger instance for each part will cause an overhead for the performance of the system.

The main parts in MDCF Server are MDCF config, Scenario Manager, Device manager and MessageBus Manager. When the server starts, the configuration object gets instantiated. It creates all the essential instances for the server to start and provides access to the server. As this config object is accessible by all other parts of MDCF server a MDCFLogger instance is created in config object along with other instantiations. This reduces the overload of creating logger instances for every segment of server. Figure 3 represents the communication between MDCF Server and MDCF data logger. All the components in MDCF Server access logger instance from config object. Config object is accessible from all other components. Scenario Manager uses logger instance to log the details of scenario such as guid and type mapping, instantiated scenarios etc. Device manager logs details about devices registered with the system. Details such as name, type and guid are logged. MessageBus Manager logs administrative topics and messages. All these log messages are passed to the data logger.

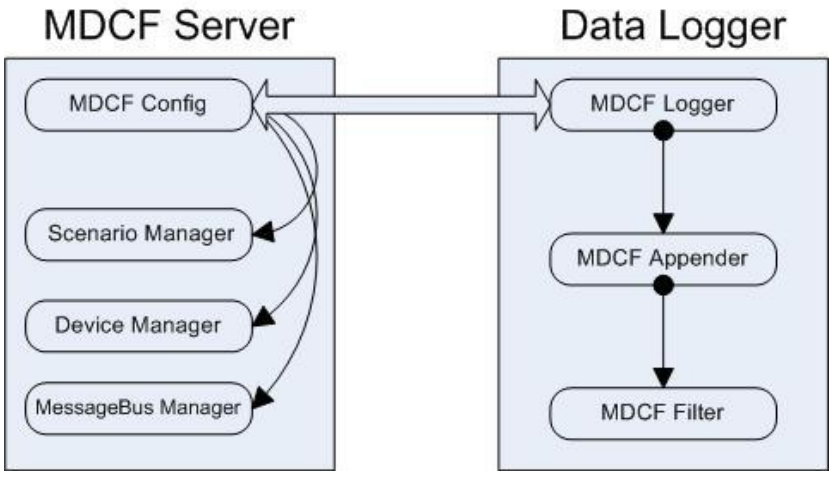

Figure 3: MDCF-DataLogger

On the data logger side, it takes message objects as input. Then the logger checks for the appender list to which the messages are to be appended. A custom appender called MDCFAppender is then called by the root category object. MDCFAppender is used to log the events and messages in-memory. MDCF Filter is a separate thread which accesses the in-memory log and then filters the logs and collects a more refined log. This filter then logs the refined log into a file. This Filter gains the reference to the in-memory log every five minutes and then clears the whole in-memory log which is used by the MDCFLogger.

From figure 2 we can derive many topologies. For instance, if there are n producers and m consumers then it is called an n - m topology. These topologies are considered as an important parameter in evaluation. We can simulate the real world scenarios from these topologies. When there are n producers and 1 consumer, we can compare that with the alarm scenario in a hospital. In this case, a nurse has a single display monitor which intimates when there is any unusual condition in any of the rooms in a ward i.e., many producers. Devices like temperature emitter have the same value to be displayed at many places. In such cases, the 1 – n topology simulates the scenario. Here, messages from one emitter are copied by the JMS provider and passed them to many display units in a room. Similarly, when a device is attached to a patient, a display unit should be associated to it. This is similar to n – n topology. Whenever that device is unplugged, then the display unit should also be disconnected. Logging these topologies is beneficial in a way that if anything goes wrong or behaves unusual, one can check the log first and then investigate the cause of the problem. Similarly, device connection with server is very crucial part of any medical system. All the details regarding the protocol should be logged for easy analysis.

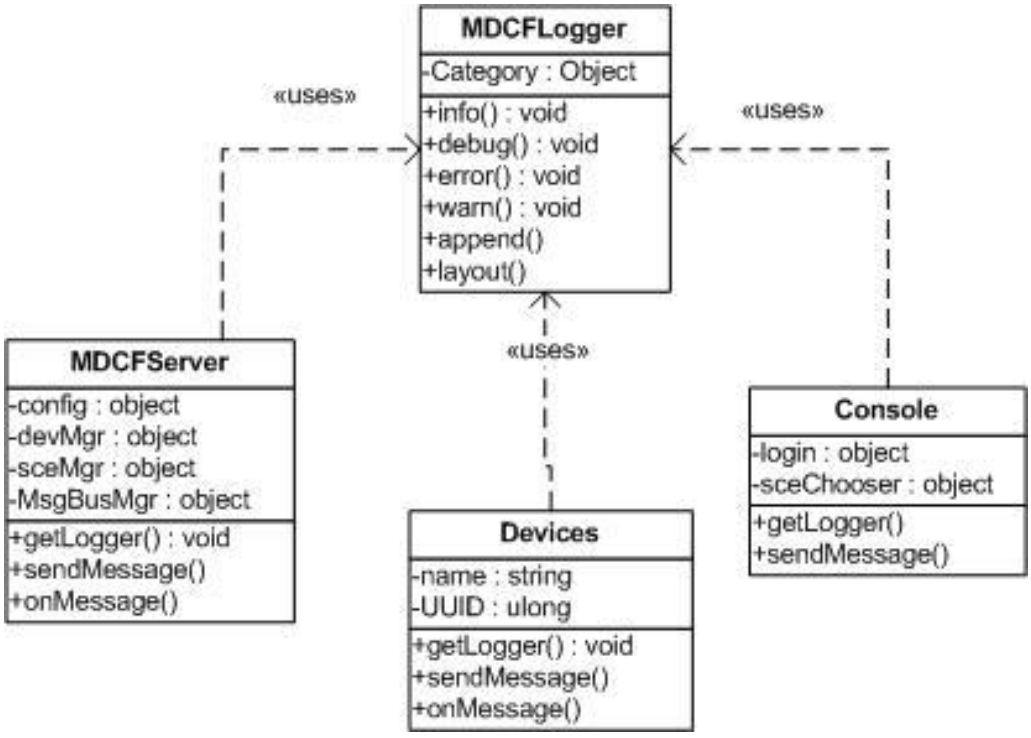

Figure 4: Class Diagram

The class diagram in Figure 4 describes the important operations and attributes involved in each component. MDCFLogger returns the instance of a logger when it is requested for one. It then defines all the log operations, info, debug, warn and error. When a component gets logger instance then it uses all the methods and logs its messages.

Security is to be considered while logging as the system logs significant information. In MDCF, there is a basic level of security at consoles. For instance, a nurse to manage devices, she/he should authenticate. Medical records are always confidential, so care must be taken to secure any information regarding a patient. Security from logger perspective depends on what kinds of appenders are listed. Depending on the appender, appropriate security measures should be taken. Here, the two main appenders are file and in-memory. In-memory is secure as it is implemented in java as a private data structure. On the other hand, file needs to be protected by giving permissions to it. If the file location is at remote location then encryption technique should be used to protect the file. As file contains important events and state changes of the system, it contains functioning details of the system. When any other appenders are added to the logging framework, then there are more ways to access the log which should be protected.

3.6 Sample Log

[2011-05-01 18:53:46,971] INFO 0[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:77) - Topic created: toAdmin

[2011-05-01 18:53:46,981] INFO 10[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:82) - Topic created: adminChan

[2011-05-01 18:53:46,982] INFO 11[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:87) - Topic created: setupChan

[2011-05-01 18:53:46,982] INFO 11[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:92) - Topic created: ADMINSERVICEBROADCAST

[2011-05-01 18:53:46,982] INFO 11[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:97) - Topic created: ADMINSERVICELISTEN

[2011-05-01 18:53:46,982] INFO 11[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:102) - Topic created: CLINICIANSERVICEBROADCAST

[2011-05-01 18:53:46,983] INFO 12[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:107) - Topic created: CLINICIANSERVICELISTEN

[2011-05-01 18:53:47,341] INFO 370[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:111) - class mdcf.chost.activemq.ActiveMQMessageBusManager connected to: setupChan

[2011-05-01 18:53:47,368] INFO 397[main] - mdcf.chost.activemq.ActiveMQMessageBusManager.initializeActiveMQ(ActiveMQMessageBusManager.java:117) - class mdcf.chost.activemq.ActiveMQMessageBusManager connected to: adminChan

CHAPTER 4: EVALUATION

Evaluation is done based on key parameters, topology and message size. Logging the system will cause an overhead to the performance of the system. Considering this issue, not only the logging code on MDCFServer needs to be polished but also ActiveMQ which is the JMS provider should also be tuned for better performance.

ActiveMQ is configured to handle maximum number of clients and acceptable performance while logging in each topology. During experiments the model uses non-persistent asynchronous messaging. The use of dedicated task runner (UseDedicatedTaskRunner) is disabled to improve the control over dispatching messages. This reduces the load on ActiveMQ in case of large number of threads. The producer flow control feature of ActiveMQ is disabled for all the topics (channels) to prevent consumers from halting when the producer is in full throttle. The transport connector in ActiveMQ is configured to use NIO (New I/O), which is same as TCP transport but with better performance. ActiveMQ opens a file descriptor for each topic it creates. This feature implies that it will eventually run out of file descriptors. We have used maximum (65535) file descriptors possible, which is achieved by 'ulimit –n65535' on unix machine.

Issues during performance testing on the model were pertaining to HeartBeat thread on device side. It effects in scaling up the number of devices. HeartBeat thread is associated with every device to check its health. HeartBeat thread has the function of publishing messages once the device has been connected. As the number of devices increases there are more number of HeartBeat threads. This has huge impact on the overall CPU utilization of the model. As we are considering rebuilding the HeartBeat component in the coming version of MDCF, we prevent the effect of HeartBeat by excluding it. This step is taken to test the scalability of MDCF. Excluding HeartBeat increased the number of clients handled by the model. Other issue was with frequency of messages for each producer. Emitters, which act as publishers, register with the server and get connected to the message bus. Once Emitter gets connected, it starts publishing messages on a dedicated channel. The frequency of messages plays a vital role on the durability of the channel and CPU utilization. We considered a delay of 50ms delay between messages. This delay helps in keeping the consumer from being flooded and prevents JVM on the consumer end to have memory issues. Memory issue arises when the message size is large.

In this model, MDCFLogger runs along with MDCF server on one machine. As the number of clients increase, logging every message and event will give rise to two main issues, CPU and memory utilization. CPU utilization on MDCF server machine is proportional to the number of clients. Memory utilization is proportional to the message size.

One of the goals of this model is to determine the throughput and latency for different topologies when logging is active. Throughput is calculated by considering number of messages received per second. Latency is the time taken for a message to go through the framework. Values of various samples are taken and average of it is calculated to determine throughput and latency. Performance various when total logging is enabled and when partial logging is enabled. We can choose for the type of logging in the log4j configuration file. When the level of logging is set to DEBUG, then a total logging takes place. When the level of logging is set to INFO, then a partial logging takes place. In total logging, all messages and events are logged. On the other hand, in partial logging, important messages and events are logged.

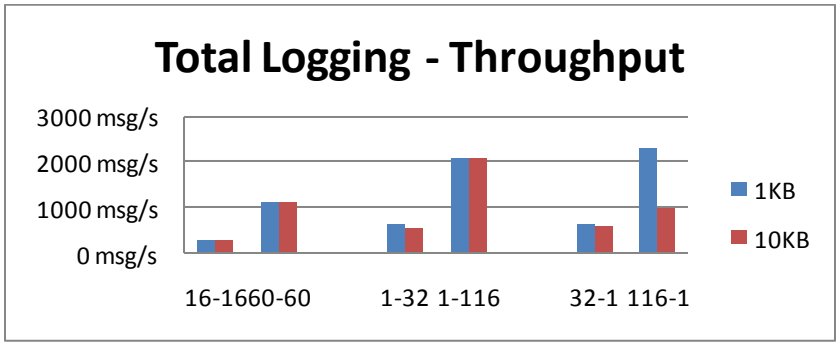

Figure 5: Total Logging - Throughput

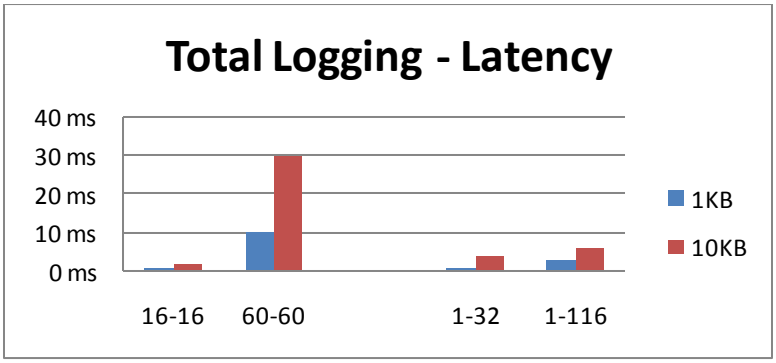

Figure 6: Total Logging – Latency

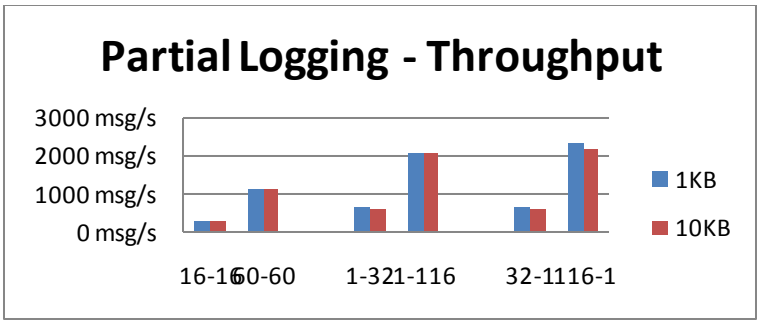

Figure 7: Partial Logging – Throughput

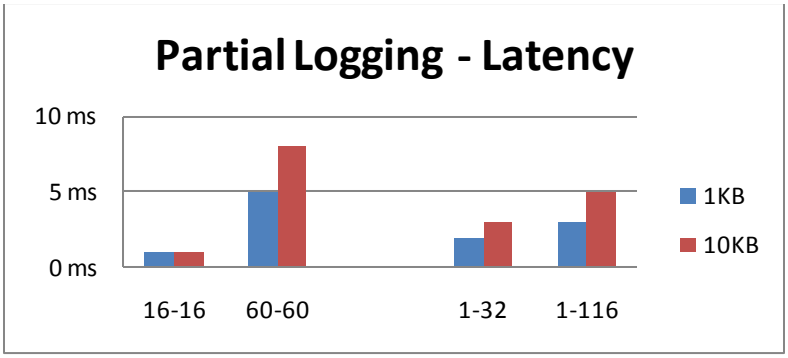

Figure 8: Partial Logging – Latency

All the experiments are performed in a distributed environment of three nodes. Each node is configured as CPU product: Intel(R) Xeon(R) CPU X5670 @ 2.93GHz, vendor: Intel Corp. physical id: 400, bus info: cpu@0, version: Intel(R) Xeon(R) CPU X5670 @ 2.93GHz, slot: Proc 1, size: 2933MHz, width: 64 bits, clock: 133MHz, 24GB of RAM, 1GB Ethernet card. Each node has 12 cores. To improve the performance of the system, we can configure the test in such a way that MDCFServer and ActiveMQ are running on same node. This reduces the usage of network bandwidth within the system. Clients can run on different machines as to simulate the real-world scenarios.

Above figures 5-8 represents the results of throughput and latency when total and partial logging is enabled. In real-time, the approximate size of a medical message ranges from 1KB to 10KB. Different topologies with 1KB, 10KB and 1MB message sizes are considered for evaluation. Topology represents the communication among number of producers to number of consumers. For instance, 60-60 represents associating 60 producers with 60 consumers. From the graphs, we can infer that as the message size increases with increasing device connections, the

latency also increases. It is because of the network traffic and marshalling and unmarshalling of messages.

Figure 5 & 7 shows the throughput results for the parameters topology and message size. As the number of devices increases the throughput increases for a particular message size. When the message size is increased we can see a slight decrease in the throughput. This is because of the internal structure of ActiveMQ, where it accumulates the messages and sends it to the consumer.

When the logging is enabled, the latency is affected in the scenario where there are many device connections. If we consider writing everything to a file, then writing onto a file which is on network can be a bottleneck to the system. Care should be taken such that only significant activities are to be written to a file when there are many devices connected to the system. There is a chance of hindering the scalability of the system. On the other hand, when the message size is 1MB or greater, there is a high increase in percentage of CPU utilization on the server node. As the message size increases, the memory utilization also increases which can be a bottleneck to the system and affects the scalability and performance of the system. 1MB message size works fine when there are small number of devices. In case of partial logging, throughput in 16 – 16 topology is 272 messages per second and in 32 – 1 topology is around 100 messages per second. When we have large size message, there is always delay in transfer of message. Latency in that case increases to unacceptable amount. Generally, the medical message size is approximately in the range of 1KB to 10KB.

The evaluation environment of the system assumes that the log file should be written on same node as it is executed. This reduces the overhead of writing a file on other node which uses network bandwidth. If we consider writing to a file in highly scalable system like MDCF, there is a chance of opening many files at the same time by each component. This can cause exceptions in log4j framework regarding writing to a file in a network. There are always network issues in a distributed environment. In this evaluation 1GB Ethernet card is being used. The packets passed per second are proportional to the message size and number of devices. The network (measured in mega bytes per second) reaches maximum in case of 1MB when compared to 1KB and 10KB. Hence, network bandwidth usage is also proportional to the message size and number of devices.

Considering the above issues, an estimate of the number of messages received when the appenders are in-memory, local file and remote file is tabulated. Assuming the partial logging and 60 – 60 topology as a common scenario Table 3 gives an estimate of the performance.

	In-memory (messages/second)	Local file (messages/second)	Remote file (messages/second)
1KB	1140	1140	900-1000
10KB	1140	1100-1140	850-950
1MB	1100-1140	500-600	50-100

Table 3: Estimate of performance in various appenders

The estimation is made keeping in mind the same test environment as previous. In remote file case, the throughput is very small if the message size is 1MB. This is because of the packets to be sent through the network along with the marshalling and unmarshalling of messages in ActiveMQ.

Introducing logging for a new component in MDCF is a simple task. Adding logging component to a device is beneficial and easy to perform. We only need to get the instance of the logger in a new device and send essential messages to the logger to be logged. Logging of devices is simple, since total logging and partial logging act same from device perspective. ActiveMQ handles collection of messages for new device destinations in total logging. Similarly, introducing logging for a new console is an easy task. When a new console is created, we need to get the logger instance and add the log messages describing the functionality of the console. For instance, in scenario choosing control log messages are added to describe the functionality of listing the scenario names, instantiation of scenario, etc. Unlike in devices and consoles, adding logger to MDCF server is a bit different.

CHAPTER 5: CONCLUSION AND FUTURE WORK

Logging MDCF activities with the idea of black box recorder is discussed in this report. Logging all messages and events are important for a medical system to diagnose a problem. Log4j is used as an underlining logging framework. A wrapper is written around Log4j and supplied to MDCF. A custom appender appends all the log messages to in-memory. A separate thread called a Filter collects the log messages from in-memory every five minutes and writes it to a file. We can refine the in-memory log based on many factors. Among the different logging frameworks available as open source products, Log4j is considered as efficient and performance effective framework. It is easy to configure and flexible to make changes to the logging system. The results obtained based on available resources are acceptable at real-time. The system is capable to scale to larger extent when the message size is small.

MDCFLogger provides a basic logging system, on which more sophisticated logging can be built. As a future work, one can create an interface which can filter the logs and access the logs which are event specific as well as component specific. It is ideal to produce a deterministic replay of events. Logging in real-time constraints can be considered to improve the performance of the logging system. In this system, security of logging and appenders are not considered. Providing security plug-in helps in privacy of the framework.

CHAPTER 6: REFERENCES

1. Log4j, Apache Software Foundation. http://logging.apache.org/log4j/index.html
2. **An Open Test Bed for Medical Device Integration and Coordination**, October 2008. *Andrew King, Sam Proctor, Dan Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, Sandy Weininger*.