JAVA Server Reliability in the Presence of Failures

Rich Coe
Marquette University

Recommended Citation
http://epublications.marquette.edu/theses_open/412
JAVA SERVER RELIABILITY IN THE
PRESENCE OF FAILURES

by

Richard H. Coe, B.S.

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Computing

Milwaukee, Wisconsin
May 2017
A design for the separation of a server interface and work processing. Numerous sources, Tanenbaum (Tanenbaum *Modern Operating Systems*, 493), Goscinski (Goscinski *Distributed operating systems*, 203), and Birman (Birman *Reliable distributed systems*, 265), all discuss the concept of Two-Phase Commit, where a coordinator directs one or more processes to perform a transaction. If the transaction or any of the processes fail, the coordinator can decide how to proceed by either retrying or aborting the request. The popular web browser Chrome utilizes a separate process for each tab displayed. Should the rendering and display of a web page cause a crash, the Chrome application itself does not. The implementation leads to a search of available Java IPC (Inter-process communication) methods, presenting a review of Java IPC implementations found. The implementation of IPC using JGroups is shown, including a code example. With results showing a 36 percent reduction in memory usage and a four times improvement in receipt and storage of DICOM C-Store images.
ACKNOWLEDGMENTS

Richard H. Coe, B.S.

I would like to thank my wife Charlotte for her support and encouragement. I am grateful to my colleague Doug Stelpflug for his assistance and comments during my numerous design conversations. My thanks to Dr Doug Harris for his valuable insight as my Graduate Advisor. My appreciation to Dr Tom Kaczmarek for leading my Thesis Committee.
TABLE OF CONTENTS

LIST OF TABLES..............................................................................................................iv
LIST OF FIGURES............................................................................................................v
I. INTRODUCTION............................................................................................................1
II. OVERVIEW OF THE DICOM NETWORK PROTOCOL..............................................4
   C-Echo..........................................................................................................................4
   C-Find..........................................................................................................................4
   C-Move.........................................................................................................................4
   C-Store........................................................................................................................4
III. OVERVIEW OF THE CURRENT DESIGN..................................................................6
IV. OVERVIEW OF THE PROPOSED DESIGN...................................................................9
V. OTHER DESIGN OPTIONS CONSIDERED.....................................................................14
   Server Socket Proxy....................................................................................................14
   Passing Received Socket to a Worker.........................................................................14
   “C” Server................................................................................................................15
   Java New I/O (NIO) Server........................................................................................15
VI. REVIEW OF JAVA IPC MECHANISMS.......................................................................16
   Aeron...........................................................................................................................18
   Akka.............................................................................................................................18
   Apache ActiveMQ......................................................................................................19
   Apache Qpid Proton....................................................................................................19
   Apache Kafka.............................................................................................................19
LIST OF TABLES

Table 1: Summary of Implementations.................................................................23
LIST OF FIGURES

figure 1 - DICOM Server Architecture.................................................................5
figure 2 – Received DICOM Connection.................................................................6
figure 3 – DICOM C-FIND existing design............................................................6
figure 4 – DICOM C-MOVE existing design.........................................................7
figure 5 – DICOM C-STORE existing design.......................................................8
figure 6 – C-MOVE Worker Architecture............................................................10
figure 7 - C-STORE Worker Architecture...........................................................11
figure 8 – DICOM C-MOVE proposed design.......................................................12
figure 9 – C-MOVE Worker Task.......................................................................12
figure 10 – DICOM C-STORE proposed design..................................................13
figure 11 – Implementation Overall Design.......................................................25
figure 12 – DicomWorkLeader Initialization......................................................26
figure 13 – WorkerLeader Thread......................................................................26
figure 14 – Worker Class....................................................................................27
figure 15 – C-MOVE Implementation Initialization.............................................28
figure 16 – C-MOVE Implementation Main Loop...............................................29
I. INTRODUCTION

A DICOM (Digital Imaging and Communications in Medicine) server handles network requests from networked medical devices for sending and receiving medical images among other transactions. DICOM files can be very large, on the order of a gigabyte or more of data.

DICOM is a network protocol and a medical image storage format. It was created by medical device manufacturers to standardize on an interoperability format between medical devices (DICOM, PS3.1 Scope and Field of Application). The DICOM network protocol defines a client/server architecture, but defines clients as Service Class User (SCU) and servers as Service Class Providers (SCP) (DICOM, PS3.7 7.2 DIMSE-Service-User Interaction). Any host or application can provide the SCP services and any application can implement the role of SCU.

A DICOM study is a set of images generated for a patient by one or more imaging technologies such as Magnetic Resonance (MR), X-Ray Computed Tomography (CT), or Ultrasound (US). A lifecycle of a typical study originates when the images are generated, sent via DICOM network to a Department Archive, and eventually sent via DICOM network to a Enterprise or Hospital Image Archive for long term storage. If, at a future date, the patient is scheduled for a new study, any previous studies will be requested for retrieval from the Image Archive for review or comparison to a newly generated study.

An application will send images as a SCU to a SCP destination using C-Store. An application can ask as a SCU with C-Find to a SCP if it has a Study or information about
a Patient. Given the results of a C-Find, an application can ask the SCP to send images of a Study to a destination with C-Move.

A computer server process listens for incoming connections, accepts the socket connection, and in some designs forks a new process to handle the incoming connection with an open file descriptor. In a Java server process, it is not possible to pass file descriptors to a forked process. In a common design, the Java server spawns a thread to handle each socket connection. This limits the total number of sockets a Java server process can handle to the number of threads limit and the open file descriptor limit within a single process. On some systems, the per-process open file limit is 1024. Additionally, in a thread-per-client-connection model, when the server dies due to a bug, it takes down all the other active connections.

It is desirable in a scalable server architecture to have a single server handle the most possible number of connections. This project for DICOM network server processing will create a design that allows the code written in Java to handle either the most allowed threads limited by memory or the file descriptor limit, with a maximum number of connections. It will offer better reliability so that a single failure does not take the server down.

This project will evaluate alternative solutions including a server that accepts connections and proxies sockets to other processes, a method to pass socket connections via unix domain sockets, starting a server written in “C”, spawning a Java Virtual Machine (JVM) to service each request.
The DICOM server responds to the requests from the applications. The only configurable limit in the current design of the services provided is the total number of network connections of all applications connecting and a total number of network connections per application. There is no restriction on the amount of work generated or rejection of commands when the server becomes overloaded.

There are no requirements for a minimum response time in responding to requests. SCU clients have a timeout for receiving response from the server. Typically the default timeout the SCU client allows the SCP server to respond is 30 seconds.

Even though a server is written in Java, a Java server can exit unexpectedly due to out-of-memory errors, JVM failures, or errors in a third-party library.
II. OVERVIEW OF THE DICOM NETWORK PROTOCOL

A simplified description of the DICOM protocol involves the following. A client initiates a DICOM network connection by creating a TCP/IP socket to a previously configured host and port and sends a DICOM Association Request packet (DICOM, PS3.7 7.4.1 Association Establishment). The server will reply with a DICOM Association Accept packet or a DICOM Association Reject packet. If the client receives an Accept packet, the client can issue commands supported by the server.

**C-Echo**
Similar to an ICMP ping, C-Echo allows a client to establish communications with a server and issue a command to test DICOM connectivity (DICOM, PS3.7 9.1.5 C-ECHO Service).

**C-Find**
Query for Patient or Study records (DICOM, PS3.7 9.1.2 C-FIND Service).

**C-Move**
Request server to transfer Image Objects to a requested destination (DICOM, PS3.7 9.1.4 C-MOVE Service).

**C-Store**
Request server to store a DICOM Image Object (DICOM, PS3.7 9.1.1 C-STORE Service). Processing continues for as long as the client issues commands, until either the client or the server sends an Association Release Request.
figure 1 - DICOM Server Architecture
III. OVERVIEW OF THE CURRENT DESIGN

The design of the current system is similar to a simple server that accepts socket connections (see figure 2). The new socket is accepted (1). A new thread is created (2) to process the connection. The DICOM protocol for connection negotiation (3) is performed. The thread is ready (4) to receive commands.

1: Connection established with new socket  
2: Spawn thread to handle connection  
3: Negotiate DICOM Protocol  
4: Handle Commands (DICOM C-FIND, DICOM C-MOVE, DICOM C-STORE)

figure 2 – Received DICOM Connection

When the requestor sends a DICOM C-FIND command (see figure 3), the DICOM attributes of the request are read (1). From the DICOM attributes, a DB query (2) is constructed. If any results are returned from the query, a DICOM response object is constructed (3) for each result entry and sent (4) in response. After all objects have been processed, the requestor is sent a DICOM status result object.

1: read DICOM query attributes from the socket  
2: query DB with query attributes  
3: format DB records into DICOM formatting  
4: send DICOM requestor results

figure 3 – DICOM C-FIND existing design

When the requestor sends a DICOM C-MOVE command (see figure 4), the DICOM attributes of the request are read (1) and validated. The configuration of the DICOM destination (2) is fetched. The objects requested to be sent (3) are located. If all
configuration and data is complete, the remote destination (4) is connected via DICOM. The requestor is notified of the current status (6), and the file is transferred via (8) DICOM. When all objects have been sent, the DICOM connection (9) is disconnected and the requestor is sent a DICOM status result object (10).

| 1: validate the arguments of the C-Move request |
| 2: get the configuration of the DICOM destination |
| 3: find file locations from DB |
| 4: connect to remote destination |
| 5: for each file |
| 6: send DICOM requestor transfer status and check for cancel request |
| 7: read file into memory |
| 8: send data to destination |
| 9: disconnect from remote destination |
| 10: send DICOM requestor C-MOVE result |

*figure 4 – DICOM C-MOVE existing design*

When the requestor sends a DICOM C-STORE command (see figure 5), the DICOM attributes of the request (1) and the contents of the DICOM header for the file being transferred are read. The DICOM attributes for Patient, Study, Series, and File are used to construct (2) the database (DB) record entries. The records are used to either find the existing entries or create new ones. A file path is created from the DB values and all the DICOM attributes are written (3) to the disk location. If the file is successfully created, the DB records are committed and the requestor is sent a DICOM status result object (5).
The requestor can continue to send C-STORE, C-MOVE, or C-FIND commands, or to end the Association by sending an Association Release Request. When the socket is closed, the connection handler thread exits and is returned to the connection thread handler pool.
IV. OVERVIEW OF THE PROPOSED DESIGN

The proposed design only modifies the DICOM C-MOVE and DICOM C-STORE command. These commands access and create large DICOM data sets. One motivation for this work is to prevent the crashing of the DICOM server while processing DICOM C-MOVE commands due to out-of-memory issues. Another motivation is to improve the reliability of the images stored during DICOM C-STORE. An architecture overview of the DICOM C-MOVE is shown in figure 6 and DICOM C-STORE is shown in figure 7. Crashes of the DICOM Server are prevented because the work of the C-MOVE request is transferred via IPC (Inter-Process Communication) to another process. The worker process could crash without affecting the server process. This design improves the reliability of the server as it maintains all the work it is doing independent of the worker task.
Reliability of the C-STORE process is improved because the image is stored but not processed. All the processing for recording the stored image into the system is completed by the C-STORE worker process.
In the proposed design, when the requestor sends a DICOM C-MOVE command (see figure 8), the DICOM attributes of the request are read (1) and validated. The configuration of the DICOM destination (2) is fetched. The objects requested to be sent (3) are located. If all configuration and data is complete, the request is sent to worker (4) task. While the worker task processes the request, the requestor is notified of the current status (8). When the worker task is finished, the requestor is sent a DICOM status result object (9).
1: validate the arguments of the C-Move request  
2: get the configuration of the DICOM destination  
3: find file locations from DB  
4: connect to a worker task  
5: send to worker task the request data  
6: while worker task is working  
7: get current status from worker task  
8: send DICOM requestor transfer status and check for cancel request  
9: send DICOM requestor C-MOVE result  

*figure 8 – DICOM C-MOVE proposed design*

The C-MOVE worker task (see figure 9) waits for an incoming request and processes the incoming data. The objects requested to be sent (2) are located, and the remote destination (3) is connected via DICOM. The worker's parent is notified of the current status (5), and the file is transferred via (7) DICOM. When all objects have been sent, the DICOM connection (8) is disconnected and the task parent is sent the status result (9).

1: receive task request  
2: find files from DB  
3: connect to remote destination  
4: for each file  
5: send transfer status and check for cancel request  
6: read file into memory  
7: send data to destination  
8: disconnect from remote destination  
9: send task completed status  

*figure 9 – C-MOVE Worker Task*
When the requestor sends a DICOM C-STORE command (see figure 10), the DICOM data is read and directly written (2) to a file on disk. A small record, recording the file path (3), is written to the DB so it can be processed later. The DICOM requestor is then sent a DICOM status result object (4).

1: read DICOM data from the socket  
2: write to disk  
3: create and commit DB record for file received  
4: send DICOM requestor C-STORE result

*figure 10 – DICOM C-STORE proposed design*
V. OTHER DESIGN OPTIONS CONSIDERED

The proposed design was selected because it required minimal changes to the existing implementation. The addition of an IPC mechanism and separation of the work meant the existing working code was to quickly maintain the existing functionality while extending the code to provide better reliability.

Server Socket Proxy

A server which proxies sockets and simply passes all data to another process is not efficient due to the delay caused by copying and relaying data.

Passing Received Socket to a Worker

Using the passed Socket approach, the server listens on the server socket and passes the accepted client socket onto a forked process for processing. This is difficult in Java because Java doesn't allow passing open file descriptors to new child processes like a traditional server written in “C”.

To pass a file descriptor in Java from one process to another requires creating a JNI (Java Native Interface) call to pass the file descriptor via a Unix Domain Socket (on Unix) like the implementation by Kellomaki (Kellomaki, BSD style file descriptor passing) or via a Windows API call to share a file descriptor between processes (Windows, Shared Sockets).
“C” Server

A server written in “C”, to accept an incoming socket, does not solve the immediate problem because there is no practical way to pass the opened socket to a new JVM. The amount of time required to start a new JVM is also prohibitive to responding in an expected timely fashion.

Java New I/O (NIO) Server

Using Java NIO, the server selects on all available sockets and reads available data as an event notification to process requests (java.nio, channels). To implement an event notification version based on the current software implementation would require a major restructuring and re-implementation of all the existing code.
VI. REVIEW OF JAVA IPC MECHANISMS

There are a myriad of different Java IPC mechanisms. Some implement a strict conformance to the Java Message Service (JMS) API, as described by Monson-Haefel and Chappell, “an abstraction of the interfaces and classes needed by messaging clients” (6). Others implement IPC mechanisms without the JMS API.

JMS has two messaging concepts, point-to-point (PTP) and publish/subscribe (PUB/SUB) (Monson-Haefel and Chappell 6). PTP requires a single receiver where messages are stored in queue prior to processing. PUB/SUB has one or more producers sending messages to a registered topic (destination) where each consumer receives every message.

JMS requires two operating servers, a Java Naming and Directory Interface (JNDI) and a JMS broker (Monson-Haefel and Chappell 21). The JNDI server is used to locate registered JMS connections and destinations. The JMS broker marshals JMS published requests to their subscriber destinations.

The following criteria was used to evaluate each Java IPC Implementation.

Easy to use? How hard is it to configure, incorporate it into an application, and use it effectively? If something goes wrong, is it easy to figure out from error messages or documentation how to fix the issue?

Is it under active development? It is important the selected mechanism be actively used and supported. “Actively used” means that a community is using it, satisfied with
its use, and reporting bugs when found. It also means that the developers actively respond to and fix the reported bugs. Not having active development or community means that there is no one to collaborate with, no one to guide using it, or fixing problems when a bug occurs.

Does it have a native Java implementation? Some implementations are written in “C” and use a JNI to provide a bridge to the actual implementation from Java. A non-native Java implementation makes it hard to debug or fix issues because the actual implementation cannot be seen from Java.

Does it require a broker? A broker is a separate application or application thread which maintains state, responds to queries, cache and forward requests, and may also direct traffic between communication endpoints. A broker makes implementation of IPC features easy. One drawback is that reliability becomes difficult when the broker goes down or becomes unresponsive or swamped with requests.

Zero configuration? Does the IPC mechanism require any preconfigured parameters? How are these parameters configured into the system? Do all of the communication endpoints have to be predetermined? A complicated configuration system means that mistakes are easy to make and testing is more difficult.

Is software released under an Open Source License? An open source license means that the software is unencumbered, allowing it to be included into a production product without fees (Open-source software). It also means that the code is available for debugging and modification should the need arise.
The following section describes and compares alternatives for the IPC mechanism. Table 1 follows and summarizes the observations.

**Aeron**

Aeron is designed to supply efficient and reliable UDP unicast, UDP multicast, and IPC message transport (Thompson, *Aeron Wiki*). It is an easy to use and easy to configure messaging system, that is supported and has been under active development since it was released under the open source Apache License. Aeron requires Java 1.8 and needs a broker to operate, making it unsuitable for use with a Java 1.7 installation.

**Akka**

Akka is designed using an Actor Model and provides a high level of abstraction for writing concurrent and distributed systems (*Akka Documentation*). Since it alleviates the developer from having to deal with explicit locking and thread management, the developers claim that it makes it easier to write correct, concurrent, and parallel systems. Actors, defined in a 1973 paper by Carl Hewitt, have been made popular by the Erlang language and used at Ericsson with great success to build highly concurrent and reliable telecom systems (ref. in *Akka Documentation*).

The large amount of terminology used in the documentation presents a steep learning curve in understanding how to configure and use the software. The amount of implementation, which is hidden from the developer, could make it difficult to implement and debug.
The tutorial examples “HelloWorld” and “Camel” are not simple to understand. The code is under active development and requires Java 1.8. The documentation implies that Akka does not need a broker, but will interact with other broker systems such as JMS and AMPQ. There is little or no configuration needed. The code is released and licensed under the open source Apache License.

**Apache ActiveMQ**

Apache ActiveMQ is a JMS broker providing PUB/SUB topics implemented in Java (*Active MQ*). It is widely used, under active development, and the configuration is simple. This Apache project was released under the Apache License.

**Apache Qpid Proton**

Apache Qpid Proton is an implementation of AMQP (Advanced Message Queueing Protocol) messaging toolkit (*Qpid Proton*). AMQP is an application layer protocol, unlike JMS which is an API and a programming framework. AMQP only describes the format of the data shared between two applications.

**Apache Kafka**

Apache Kafka relies on Apache ZooKeeper to provide a Naming Service, Configuration, and a Message Queue (*Apache Kafka*). It also requires a Kafka broker for processing requests.
Appia

Appia is a set of protocols that provides group communication (*Appia Communication Framework*). The software has not had any updates since version 4.1.2 was released in 2011. There are a few demo programs in the source code which show some limited capabilities, but there is not a simple getting-started or example program with documentation showing how to use Appia. The source code is written in Java and the implementation uses configuration data or network broadcasting to discover peers. The code is released and licensed under the open source Apache License.

Fast-cast

Fast-cast is written in Java and implements PUB/SUB using tcp/ip multicast (Moeller *fast-cast*). It does not use a broker, but instead relies on application startup configuration for port assignments and control attributes. There has been no active development since November 2015, and it was released under the GNU Lesser General Public license (LGPL).

jGCS

jGCS is written in Java and does not provide a group communication implementation, but instead provides a single generic interface over different implementations such as Appia, JGroups, Spread, and NeEM (*jGCS*). It is an interesting concept that makes it easier when an installation needs to work with two or more of the supported implementations. jGCS is licensed under a BSD License, is not under active development, and has had no published changes since 2007.
JGroups

JGroups is a flexible communication protocol written in Java (Ban and Blagojevic *Reliable group communication with JGroups*). It does not use a broker, but uses protocols to discover application memberships and failures. It has been in continuous active development since 1998, uses a reasonable set of default settings and several simple demo programs that show JGroups capabilities. JGroups is licensed under the open source Apache License.

Nanomsg

Nanomsg is a “C” language socket library that provides several common communication patterns (*nanomsg*). It aims to make the networking layer fast, scalable, and easy to use. The Java version is implemented as a java interface over JNI.

NeEM

NeEM is an acronym for Network-friendly Epidemic Multicast (*NeEM*). It is written in Java and implements a simple API for sending messages via Multicast networking. Aside from a few recent changes in 2015 to support Maven and Ivy, the code has not been changed since 2007. It uses a modified MIT License for distribution.

Spread

Spread is a toolkit for reliable, scalable messaging, and group communication (*Spread*). The toolkit requires a hardcoded configuration file which contains ip-addresses
of all participation hosts. It also requires a central daemon, written in “C”, which acts as a broker.

**ZeroMQ**

ZeroMQ provides sockets in a variety of forms including TCP and multicast (ZeroMQ). It also supports familiar patterns including PUB/SUB and request-reply. The license is GNU LGPL which makes it accessible to production code. The drawback to ZeroMQ is that it does not have a native Java implementation. The Java implementation calls the “C” implementation via JNI.
### Summary of Implementations

<table>
<thead>
<tr>
<th>Package</th>
<th>Active Development</th>
<th>Native Java</th>
<th>No Broker</th>
<th>Zero Configuration</th>
<th>License</th>
<th>Accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aeron</td>
<td>Yes</td>
<td>Java-8</td>
<td>No</td>
<td>Yes</td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Akka</td>
<td>Yes</td>
<td>Java-8</td>
<td>Yes</td>
<td>Yes</td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Apache ActiveMQ</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Apache Kafka</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Apache Qpid Proton</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Appia</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Apache</td>
<td>No</td>
</tr>
<tr>
<td>Fast-cast</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>LGPL</td>
<td>No</td>
</tr>
<tr>
<td>jGCS</td>
<td>No</td>
<td>Yes</td>
<td>--</td>
<td>--</td>
<td>BSD</td>
<td>No</td>
</tr>
<tr>
<td>JGroups</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Apache</td>
<td>Yes</td>
</tr>
<tr>
<td>Nanomsg</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>MIT</td>
<td>No</td>
</tr>
<tr>
<td>NeEM</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>modified MIT</td>
<td>No</td>
</tr>
<tr>
<td>Spread</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>modified MIT</td>
<td>No</td>
</tr>
<tr>
<td>ZeroMQ</td>
<td>Yes</td>
<td>No</td>
<td>--</td>
<td>--</td>
<td>LGPL</td>
<td>No</td>
</tr>
</tbody>
</table>

*Table 1: Summary of Implementations*
VII. REVIEW OF THE SELECTED JAVA IPC MECHANISM

JGroups was an attractive solution because it implemented application discovery and message delivery services without a broker.

The method to add JGroups support to a Java application is to create a class that extends org.jgroups.ReceiverAdapter. A working Java example “MessageIPC” is shown in Appendix A.

The application creates an extended class of MessageIPC to implement any application specific code. “SampleServerIPC” class (line 25) is an empty template of what a class might look like. During initialization the application creates an instance of the IPC class and calls the class method “start(null, 'app-name')” (line 394). For sending messages, the class method “send('dest-app-name', MessageContent)” (line 484) is used, where 'MessageContent' can be any Java object which is serializable.
VIII. IMPLEMENTATION

OVERVIEW

The overall design is shown in figure 11. A DICOM listener process services requests from port 1 and port 2. The requests are processed and sent to a DICOM worker for processing.
The “DicomWorkLeader” (figure 12) is a singleton class that holds the state of all the workers. The Java implementation is shown in Appendix B (line 516).

```
1: initialize the ipc mechanism as “DicomWorkLeader”
2: create a WorkLeader thread to handle remote messages and internal activities
3: createWorker() API to create a remote thread for handling remote work
4: receiveMessage() called by the IPC mechanism to deliver messages

figure 12 – DicomWorkLeader Initialization
```

The “WorkerLeader” (figure 13) main thread is shown at Appendix B (line 258).

```
1: start number of configured workers
2: while running
3:  look for and process deferred incoming worker messages
4:   - “clientCommand” a message from the remote worker for the local process
5:   - “idle” the thread associated with the associated thread key is terminated
6:   - “started” the worker process has finished initializing and is ready
7:  look for and process allocate worker requests

figure 13 – WorkerLeader Thread
```

The “WorkerClass” (figure 14) holds the remote worker state and the worker process main thread. The working implementation is shown in Appendix C (starts at line 331).
1: starts IPC thread for receiving messages
2: initialize the ipc mechanism as “WorkName,” a unique name assigned by the WorkLeader
3: send message “started” to the WorkerLeader now the initialization is complete
4: while running
5:     look for and process deferred incoming worker messages
6:         - “clientCommand” a message from the local process for a specific thread
7:         - “createCStore” create a CStore Worker
8:         - “createCMove” create a CMove Worker
9:         - “ctxt” record thread context information for a Worker
10:        - “finish” the local process is done with the named Worker thread
11:        look for exiting threads and remove thread from the thread pool list
12:        shutdown process if commanded to stop

figure 14 – Worker Class

The “WorkMessage” class is used to hold a pending message received on the IPC thread for deferred delivery (Appendix B, line 111).

The “WorkProcessor” class (Appendix C, line 52) is an API interface, which holds the thread that the worker defines, and is called by the “AssociationWorkerInfo” class.

The “AssociationWorkerInfo” class (Appendix C, line 60) provides an API to the remote process for two-way communication as the remote “Worker”. A remote “Worker” extends “AssociationWorkerInfo” and implements the “process” method to be called when a new message arrives.
DICOM C-MOVE

The C-Move Implementation initialization (see figure 15) is similar to the C-Move design (see figure 4) up to the point where the connection to a worker process (figure 15, line 6) is established. The arguments for the C-Move request are sent to the worker for processing (figure 15, lines 8 through 13).

1: validate the arguments of the C-Move request
2: get the configuration of the destination
3: find files from DB
4: if no files found
5: - stop the C-Move processing
6: connect to an available worker
7: send to the worker
8: - the arguments of the C-Move request
9: - the originator config
10: - the destination config
11: - the requested study info
12: - the count of expected images to send
13: - start the C-Move command

figure 15 – C-MOVE Implementation Initialization

The C-Move Implementation Main loop (see figure 16) reads messages from the C-Move worker as it processes through each of the C-Move steps.
The C-Move implementation of WorkerInfo class (Appendix D, line 17) processes the C-Move Implementation messages. It creates a CMoveProcessor object (Appendix D, line 34) that does the work and starts a new thread (Appendix D, line 66).

**DICOM C-STORE**

The C-Store implementation of WorkerInfo class (Appendix E, line 11) processes the C-Store messages. It creates a CStoreProcessor object (Appendix E, line 17) that does the work and starts a new thread.
IX. METHOD

The hardware setup consisted of two computers, the server computer an Intel Core i7-3720QM (2.60GHz) running Linux 4.8 64-bit and the client computer an Intel Core i3-3227U (1.90GHz) running Windows 8.1 32-bit.

Testing was performed with a client task that every 15 seconds creates a thread which connects to the server, loops performing a query to the server for a study to move and requests the study to be moved to the client. If the move fails, it closes the connection and exits the thread. The task keeps creating threads up to a maximum of 95 threads.

Up to five tasks were started in succession, and the total number of threads running simultaneously was monitored. The testing failed because the Java Out-of-Memory issue could not be reproduced with the test setup. Two recent modifications that blocked the error from occurring were identified and installed, yet the testing still failed. All attempts at recreating the heap error issue after reducing the java server heap size from 1280MB to 640MB, and subsequently 320MB were unsuccessful.

The source of Out-of-Memory error “java.lang.OutOfMemoryError: requested 20267370 bytes for jbyte” was identified. The symptom of the issue is that server crashes because of an unsatisfied memory heap allocation. This is a catastrophic error that cannot be caught. The assumption is that the software was over allocating heap memory.

The error scenario is that the running Java server has many heap allocations that have not yet been garbage collected (GC). A JNI call is made to the “C” code, the “C”
code calls back into the Java code, and the Java code tries to make a heap allocation. Since the heap allocation is made from inside a JNI call and all GC is blocked during a JNI call, the heap allocation fails and the JVM crashes. The correct software solution is to allocate the required Java heap allocation prior to the JNI call.
X. CONCLUSION

After 301 active DICOM connections, the dumped java heap was 1107.78 MB for the Current Design. For the Proposed Design, with 321 active DICOM connections, the dumped java heap was 703.88 MB. The current design is able to handle the up to 300 connections which is sufficient to handle the current capacity as it is six times the current default maximum number of connections.

The Proposed design supports recoverability when during the processing of a DICOM C-Move, a memory error unexpectedly causes the processing to abort. The server is able to recover and restart the operation.

The purpose of the Proposed design is to protect the DICOM server from crashing when a Java memory issue causes the server to crash. Under the new design, the DICOM server continues to process requests because the processing of the requests are handled in a separate process. The new design is being integrated into a future release.

The Proposed design does not support a significant, sustained increase in concurrent connections. There was only a 10% increase in the number of concurrent connections, within the range of 330 to 340. More observations are needed in order to determine what is causing connections to drop during processing.

One bottleneck observed during testing was constant querying and fetching of unchanging configuration data. A cache was implemented to hold configuration so as to
eliminate redundant configuration lookups with nominal improvements. Further study of failures is required.

To provide a high quality of service, the server should actively measure performance metrics and reduce accepting new work to prevent the DICOM server from reaching a crisis point and preventing critical failure. The DICOM server currently limits the total number of connections and the number of connections per host. This is independent of how much work these connections generate.

Future work would include finding an automated systemic way to provide back-pressure into the work generating commands C-Move and C-Store to reduce the incoming workload. Without back-pressure, the system may become overloaded as DICOM C-Stores are accepted at 80 images per second, while those images are inserted at 20 images per second or slower. For C-Move, options to investigate are:

- limit any additional requests until the total number of requests are reduced
- prevent additional C-Move requests when a CPU threshold has been reached
- prevent additional C-Move requests until the amount of heap space available is increased.

For C-Store, it would be practical to slow down image store response time or stop the receipt of images until the total number of images to be inserted has reached a specified threshold.
BIBLIOGRAPHY


Sample code MessageIPC.java showing the server interface to implement messaging using jgroups.

```java
/* //////////
* MessageIPC.java
* Created on Sep 3, 2015 by rcoe
* Copyright 2015 */
package edu.marquette.rcoe.messaging;

// A simple messaging interface to jgroups
// Sample Usage
// Initialize:
// SampleServerIPC.getInstance();

// -- SampleServerIPC.java --
// Server Message receiver
// package edu.marquette.rcoe.server.core;
// import java.io.Serializable;
// import org.apache.log4j.Logger;
// import edu.marquette.rcoe.messaging.MessageIPC;
// public class SampleServerIPC extends MessageIPC {
// private static final Logger logger
// = Logger.getLogger(SampleServerIPC.class);
// private static SampleServerIPC instance
// = new SampleServerIPC();
// private SampleServerIPC()
// { }
// public static SampleServerIPC getInstance()
// { return instance;
```
```java
// @Override
public void receiveMessage(String from, Serializable obj)
{
    String cmd = (String) obj;
    logger.info("received command " + cmd);
    if (cmd.equals("stop"))
        System.exit(0);
}

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collection;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import org.apache.log4j.Logger;
import org.jgroups.Address;
import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.ReceiverAdapter;
import org.jgroups.util.Util;
import org.jgroups.View;

public class MessageIPC extends ReceiverAdapter
{
    private static final Logger logger = Logger.getLogger(MessageIPC.class);

    // diff from default:
    // set UDP:loopback to true for windows
    // set GMS:print_local_addr to false

    public static final String DEFAULT_PROTOCOL_STACK=
        "UDP(mcast_port=45588;ip_ttl=4;tos=8;"
        + "ucast_recv_buf_size=200K;ucast_send_buf_size=200K;"
```

+ "mcast_recv_buf_size=200K;mcast_send_buf_size=200K;"
+ "max_bundle_size=64K;max_bundle_timeout=30;"
+ "enable_diagnostics=true;thread_naming_pattern=cl;"
+ "timer_type=new3;timer.min_threads=2;"
+ "timer.max_threads=4;timer.keep_alive_time=3000;"
+ "thread_pool.keep_alive_time=5000;"
+ "thread_pool.queue_enabled=true;"
+ "thread_pool.queue_max_size=10000;"
+ "thread_pool.rejection_policy=discard;"
+ "log_discard_msgs=false;"
+ "oob_thread_pool.enabled=true;"
+ "oob_thread_pool.min_threads=1;"
+ "oob_thread_pool.max_threads=8;"
+ "oob_thread_pool.queue_enabled=false;"
+ "oob_thread_pool.queue_max_size=100;"
+ "oob_thread_pool.rejection_policy=discard;"
+ "PING:"
+ "MERGE3(max_interval=30000;min_interval=10000):"
+ "FD_SOCK:"
+ "FD_ALL:"
+ "VERIFY_SUSPECT(timeout=1500):"
+ "BARRIER:"
+ "pbcast.NAKACK2(xmit_interval=500;xmit_table_num_rows=100;"
    + "xmit_table_msgs_per_row=2000;"
    + "xmit_table_max_compaction_time=30000;"
    + "max_msg_batch_size=500;use_mcast_xmit=false;"
    + "discard_delivered_msgs=true):"
+ "UNICAST(xmit_interval=500;xmit_table_num_rows=100;"
    + "xmit_table_msgs_per_row=2000;"
    + "xmit_table_max_compaction_time=60000;"
    + "conn_expiry_timeout=0:max_msg_batch_size=500):"
+ "pbcast.STABLE(stability_delay=1000;"
    + "desired_avg_gossip=50000:max_bytes=4M):"
+ "pbcast.GMS(print_local_addr=false;join_timeout=2000;"
    + "view_bundling=true):"
+ "UFC(max_credits=2M;min_threshold=0.4);"
+ "MFC(max_credits=2M;min_threshold=0.4);"
+ "FRAG2(frag_size=60K):"
+ "pbcast.STATE_TRANSFER();"
public static List<String> msgName = new ArrayList<String>;

static {
    msgName.add("" );
    msgName.add("HELLO");
    msgName.add("HELLOREPLY");
    msgName.add("GOODBYE");
    msgName.add("APPMSG");
}

private char msgType;
Serializable imsg;
	public iMessageIPC(char mType, Serializable obj)
{
    msgType = mType;
    imsg = obj;
}

public byte[] serialize()
{
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    try {
        ObjectOutputStream output = new ObjectOutputStream(bos);
        output.writeObject(this);
        output.flush();
    } catch (Exception ex) {
        logger.error("cannot encode message", ex);
        return null;
    }
    logger.debug("created message of ",
        + bos.toByteArray().length);
    return bos.toByteArray();
}

    public static iMessageIPC deserialize(byte[] imsg, int off,
        int len)
    {
        iMessageIPC msg = null;
        try {
            ByteArrayInputStream bis
                = new ByteArrayInputStream(imsg, off, len);
            ObjectInputStream input = new ObjectInputStream(bis);
            msg = (iMessageIPC) input.readObject();
        } catch (Exception ex) {
            logger.error("cannot decode message", ex);
            return null;
        }
    }
        return msg;
    }

    public char getType()
    {
        return msgType;
    }

    public Serializable getMessage()
    {
        return imsg;
    }

    public static String msgTypeToString(int mType)
    {
        return msgName.get(mType);
    }

    private static class Members
    {
        private String appName;
        private Address appAddr;
        private String host;
        private int tstamp;
        private boolean exiting = false;

        public Members(String name, Address addr)
        {
            appName = name;
            appAddr = addr;
            host = addrToHost(addr.toString());
            tstamp = 0;
        }

        public String getName()
        {
            return appName;
        }

        public void setName(String name)
        {
            appName = name;
        }

        public Address getAddr()
        {
            return appAddr;
        }
    }
public int getStamp() {
    return tstamp;
}

public void setStamp(int stamp) {
    tstamp = stamp;
}

public String getHost() {
    return host;
}

public void setExiting() {
    exiting = true;
}

public boolean isExiting() {
    return exiting;
}

public String toString() {
    return new String(appName + "@" + appAddr);
}

private static class IPCProcessor extends Thread {
    private MessageIPC mipc;
    private ConcurrentLinkedQueue<Message> queue = new ConcurrentLinkedQueue<Message>();
    private boolean running = true;

    public IPCProcessor(MessageIPC mipc) {
        super("IPCProcessor");
        this.mipc = mipc;
    }

    public void run() {
        while (running) {
            Message msg;
while (null != (msg = queue.poll())) {
    String appname;
    boolean reply = false;
    iMessageIPC imsg
        = iMessageIPC.deserializer(msg.getRawBuffer(),
                                    msg.getOffset(), msg.getLength());

    if (null == imsg) {
        logger.error("invalid message of "
                     + msg.getRawBuffer().length);
    } else
        switch (imsg.getType()) {
            // when an app starts up, it sends HELLO
            // when an app receives HELLO, reply with a HELLOREPLY
            case iMessageIPC.HELLO:
                reply = true;
            case iMessageIPC.HELLOREPLY:
                appname = (String) imsg.getMessage();
                logger.debug("hello from " + appname + " status: "
                             + mipc.app2addr.get(appname));
                Members memb = mipc.addr2app.get(msg.getSrc());
                if (null != memb) {
                    // existing, add it's name
                    memb.setName(appname);  
                    mipc.app2addr.put(appname, memb);
                } else {
                    // create new
                    memb = new Members(appname, msg.getSrc());
                }
                if (reply) {
                    try {
                        mipc.send(iMessageIPC.HELLOREPLY, msg.getSrc(),
                                  mipc.appName);
                    } catch (Exception ex) {
                        logger.error("cannot return HELLO to "
                                     + appname, ex);
                    }
                }
                break;
            // an app is exiting
            case iMessageIPC.GOODBYE:
                appname = (String) imsg.getMessage();
                logger.debug("goodbye from " + appname);
                mipc.removeApp(appname);
                break;
            // an app upper layer message
            case iMessageIPC.APPMSG:
                memb = mipc.addr2app.get(msg.getSrc());
String from = (null != memb) ? memb.getName() : "unknown";
mipc.receiveMessage(from, imsg.getMessage());
break;
default:
    logger.error("Unknown message received type: "+ imsg.getType());
    break;
}
}
try {
    synchronized (this) {
        wait();
    }
} catch (Exception ex) {
    logger.error("wait", ex);
}
}

public void add(Message msg) {
    queue.add(msg);
    try {
        synchronized (this) {
            notify();
        }
    } catch (Exception ex) {
        logger.error("add notify", ex);
    }
}

public void stopRunning() {
    running = false;
    try {
        synchronized (this) {
            notify();
        }
    } catch (Exception ex) {
        logger.error("stop notify", ex);
    }
}

private JChannel jch = null;
private String appName;
private Map<String, Members> app2addr
    = new TreeMap<String, Members>();
private Map<Address, Members> addr2app
    = new TreeMap<Address, Members>();

private int tstamp = 0;
private IPCProcessor msgThread;

public void start(String props, String name) throws Exception
{
    String localhost = InetAddress.getLocalHost().getHostName();
    start(props, name, localhost);
}

public void start(String props, String name, String channel)
          throws Exception
{
    if (null != jch)
        return;

    logger.debug("starting IPC for " + name);
    props = (null == props) ? DEFAULT_PROTOCOL_STACK : props;
    jch = new JChannel(props);

    appName = name;
    if (null != name) jch.setName(name);         // later release
    jch.setDiscardOwnMessages(true);             // added in 3.0

    msgThread = new IPCProcessor(this);
    msgThread.start();

    jch.setReceiver(this);
    logger.info("connecting to group " + channel);
    jch.connect(channel);

    addApp(new Members(appName, jch.getAddress()));

    send(iMessageIPC.HELLO, null, name);
}

public void shutdown() throws Exception
{
    if (null != jch) {
        send(iMessageIPC.GOODBYE, null, appName);
        jch.disconnect();
        msgThread.stopRunning();
    }
}
        Util.close(jch);
    jch = null;
  }
  
  public void viewAccepted(View newView) {
    tstamp++;
    for (Object obj : newView.getMembers()) {
      Address addr = (Address) obj;
      Members app = addr2app.get(addr);
      if (null == app) {  
        app = new Members(null, addr);
        addApp(app);
      }
      app.setStamp(tstamp);
    }
    Iterator<Members> vit = addr2app.values().iterator();
    while (vit.hasNext()) {
      Members memb = vit.next();
      if (tstamp != memb.getStamp()) {
        vit.remove();
        if (null != memb.getName())
          app2addr.remove(memb.getName());
      }
    }
  }
  
  public Collection<Members> getMembers() {
    return addr2app.values();
  }
  
  public void receive(Message msg) {
    msgThread.add(msg);
  }
  
  // to be overridden
  public void receiveMessage(String from, Serializable obj) {
  }
  
  private void send(char mType, Address dest, Serializable obj)
    throws Exception {
    logger.debug("send " + iMessageIPC.msgTypeToString(mType));
    iMessageIPC imsg = new iMessageIPC(mType, obj);
    Message msg = new Message(dest, null, imsg.serialize());
    jch.send(msg);
public boolean send(String dest, Serializable obj) throws Exception {
    if (null == jch)
        return false;
    Address ecdest = null;
    if (null != dest) {
        Members memb = app2addr.get(dest);
        if (null == memb) {
            logger.error("cannot send to non-member" + " destination " + dest);
            logger.info("rebroadcasting app discovery");
            send(iMessageIPC.HELLO, null, appName);
            return false;
        }
        if (memb.isExiting())
            return false;
        ecdest = memb.getAddr();
    }
    send(iMessageIPC.APPMSG, ecdest, obj);
    return true;
}

public Set<String> getApps() {
    logger.debug("getApps");
    return app2addr.keySet();
}

public List<String> getLocalApps() {
    logger.debug("getLocalApps");
    List<String> rapps = new ArrayList<String>();
    for (Members memb : app2addr.values()) {
        logger.debug("member addr " + memb.getHost());
        rapps.add(memb.getName());
    }
    return rapps;
}

private void addApp(Members app) {
    String name = app.getName();
    Address addr = app.getAddr();
    logger.debug("adding app" + name + " from " + addr);
    if (null != name)
app2addr.put(name, app);
addr2app.put(addr, app);
}

private void removeApp(String appName)
{
    Members memb;
    logger.debug("removing app " + appName);
    if (null != (memb = app2addr.get(appName))) {
        memb.setExiting();
    }
}

private static String addrToHost(String addr)
{
    int colon = addr.indexOf(':');
    if (-1 != colon)
        return addr.substring(0, colon);
    return addr;
}
Sample code DicomWorkLeader.java showing the server interface to implement messaging between the Server and the Workers.

```java
/*
 * DicomWorkLeader.java
 * Created by rcoe
 * Copyright 2016
 */
package edu.marquette.rcoe.server.core;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.Serializable;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.TreeMap;

import org.apache.log4j.Logger;

import edu.marquette.rcoe.messaging.MessageIPC;
// [ ... ] Other server interface imports
/**
 *  DicomWorkLeader -- work leader for the DicomServer
 */
public class DicomWorkLeader extends MessageIPC
{
    private static final Logger logger = Logger.getLogger(DicomWorkLeader.class);
    private static DicomWorkLeader theServer = null;

    private WorkLeader workLeader;

    private Map<String, AssociationInfo> ainfos = new TreeMap<String, AssociationInfo>();
    private Map<String, Worker> workers = new TreeMap<String, Worker>();
    private ConcurrentLinkedQueue<AssociationInfo> msgQueue
```
= new ConcurrentLinkedQueue<AssociationInfo>();

private static List<String> worker_args = new ArrayList<String>();

private int sendStudyCount = 0;

static {
  worker_args.add("-DT=DW${INDEX}");
  worker_args.add("-DlogPropertyFile=
      +"${XNS_HOME}/config/DicomWorkerlog.properties");
  worker_args.add("-DPROCNAME=${PROCNAME}");
  worker_args.add("-DLOGPORT=${logport}");
  worker_args.add("-Djava.net.preferIPv4Stack=true");
  worker_args.add("-server");
  worker_args.add("-DXNS_HOME=${XNS_HOME}");
  // worker_args.add("-Xms128k");                // initial heap
  worker_args.add("-Xmx1024m");                  // max heap
}

public enum WorkerState {
  STARTING,       // jvm launched
  IDLE,           // received alive message
  ACTIVE,         // working on a task
  STOPPING,       // IDLE or ACTIVE jvm sent stop message
  ZOMBIE          // jvm stopped
}

public enum AssocState {
  PENDING,        // messages pending
  IDLE            // no messages pending
}

// class Worker manages the remote worker proc
private static class Worker {
  String workName;
  WorkerState state = WorkerState.STARTING;
  int activeThreads = 0;

  public Worker(String name) {
    workName = name;
  }

  public void setState(WorkerState st) {
    state = st;
  }
}
public boolean isActive() {
    return state.equals(WorkerState.ACTIVE);
}

public boolean isStopping() {
    return state.equals(WorkerState.STOPPING);
}

public boolean isStopped() {
    return state.equals(WorkerState.ZOMBIE);
}

public void setStopped() {
}

public static class WorkMessage {
    private String command;
    private String value;
    private String appkey;
    private Properties message;

    public WorkMessage(String cmd, String val) {
        command = cmd;
        value = val;
    }

    public WorkMessage(String cmd, String val, String key, Properties msg) {
        command = cmd;
        value = val;
        appkey = key;
        message = msg;
    }

    public String getCommand() {
        return command;
    }

    public String getValue() {
{ return value; }

public String getAppKey()
{
    return appkey;
}

public Properties getMessage()
{
    return message;
}

// class AssociationInfo represents the DICOM work being
// managed by the worker
public static class AssociationInfo
{
    String assocPk;
    private String ctxt;
    private AssocState state = AssocState.IDLE;
    Worker worker = null;
    ConcurrentHashMap<String> msgs
        = new ConcurrentHashMap<String>();
    ConcurrentHashMap<WorkMessage> replies
        = new ConcurrentHashMap<WorkMessage>();
    DicomWorkLeader mipc;

    AssociationInfo(DicomWorkLeader parent, String apk)
    {
        assocPk = apk;
        mipc = parent;
    }

    public void send(String command)
    {
        send(command, (String) null);
    }

    public void send(String command, String msg)
    {
        StringBuffer sb = new StringBuffer();
        sb.append("cmd=clientCommand\n");
        sb.append("id=").append(assocPk).append("\n");
        sb.append("clientCommand=").append(command).append("\n");
        if (null != msg)
            sb.append("clientMessage=").append(msg).append("\n");
        addMessage( sb.toString() );
private void sendInternal(String command, String value) {
    StringBuffer sb = new StringBuffer();
    sb.append("cmd=").append(command).append("\n");
    sb.append("id=").append(assocPk).append("\n");
    if (null != value)
        sb.append("value=").append(value).append("\n");
    addMessage( sb.toString() );
}

public void addMessage(String msg) {
    msgs.add(msg);
    if (state.equals(AssocState.IDLE)) {
        mipc.msgQueue.add(this);
        state = AssocState.PENDING;
    }
    synchronized(mipc.workers) {
        mipc.workers.notify();
    }
}

public WorkMessage getMessage() {
    return replies.poll();
}

public void addReply(WorkMessage msg) {
    replies.add(msg);
}

public boolean isPending() {
    return state.equals(AssocState.PENDING);
}

// class WorkLeader manages the running Workers
private static class WorkLeader extends Thread {
    private static int MAXWORKERS = 10;
    private Map<String, Worker> workers;
    private Map<String, AssociationInfo> ainfos;
    private boolean shutdown = false;
    private long shutdownTime = 0;
private int debugPort = 5020;

private ConcurrentLinkedQueue<Worker> idleQueue = new ConcurrentLinkedQueue<Worker>();
private ConcurrentLinkedQueue<WorkMessage> msgs = new ConcurrentLinkedQueue<WorkMessage>();

DicomWorkLeader mipc;

private int keynum = 0;

public WorkLeader(DicomWorkLeader parent, Map<String, Worker> wrx, Map<String, AssociationInfo> aix) {
    workers = wrx;
    ainfos = aix;
    mipc = parent;
}

public void run() {
    Thread t = Thread.currentThread();
    t.setName("DicomWorkLeader");
    Logger.pushLoggingContext("DicomWorkLeader");

    // start workers
    try {
        for (int i=0; i < MAXWORKERS; i++)
            startWorker(mipc);
    } catch (Exception ex) {
        logger.error("Cannot start initial workers", ex);
    }

    try {
        while (true) {
            boolean dostop = true;
            AssociationInfo ainfo;

            synchronized(workers) {
                WorkMessage msg;
                while (null != (msg = msgs.poll())) {
                    Worker worker;
                    switch (msg.getCommand()) {
                    case "clientCommand":
                        logger.debug("worker " + msg.getValue() + " " + msg.getCommand() + " " + msg.message);
                        ainfo = ainfos.get(msg.appkey);
                        if (null == ainfo)
                            logger.error("cc from: " + msg.getValue() + " " + msg.message);
                        else
                            ainfo.process((msg关键) = ainfo.keynum = 0);
                    case "shutdown":
                        logger.debug("worker " + msg.getValue() + " " + msg.getCommand() + " " + msg.message);
                        ainfo = ainfos.get(msg.appkey);
                        if (null == ainfo)
                            logger.error("cc from: " + msg.getValue() + " " + msg.message);
                        else
                            ainfo.process((msg关键) = ainfo.keynum = 0);
                    default:
                        logger.error("Unknown command: " + msg.getCommand());
                    }
                }
            }
            synchronized(ainfos) {
                // do something with ainfo
            }
        }
    }
}

// handle incoming messages

private void handleMessage(WorkMessage msg) {
    switch (msg.getCommand()) {
    case "clientCommand":
        logger.debug("worker " + msg.getValue() + " " + msg.getCommand() + " " + msg.message);
        ainfo = ainfos.get(msg.appkey);
        if (null == ainfo)
            logger.error("cc from: " + msg.getValue() + " " + msg.message);
        else
            ainfo.process((msg关键) = ainfo.keynum = 0);
    case "shutdown":
        logger.debug("worker " + msg.getValue() + " " + msg.getCommand() + " " + msg.message);
        ainfo = ainfos.get(msg.appkey);
        if (null == ainfo)
            logger.error("cc from: " + msg.getValue() + " " + msg.message);
        else
            ainfo.process((msg关键) = ainfo.keynum = 0);
    default:
        logger.error("Unknown command: " + msg.getCommand());
    }
}

// start worker

private void startWorker(DicomWorkLeader parent) {
    // do something to start the worker
    // set it as running
    parent.running = true;
    // add it to the running list
    parent.runningWorkers.add(worker);
}

// stop worker

private void stopWorker(DicomWorkLeader parent) {
    // do something to stop the worker
    // set it as stopped
    parent.running = false;
    // remove it from the running list
    parent.runningWorkers.remove(worker);
}
else {
    ainfo.addReply(
        new WorkMessage(
            msg.message.getProperty("clientCommand"),
            msg.message.getProperty("clientMessage")
        ) );
}
}
break;

case "idle":
    logger.debug("worker " + msg.getValue() + " "
    + msg.getCommand());
    ainfo = ainfos.get(msg.appkey);
    if (null == ainfo)
        logger.error("idle from: " + msg.getValue()
        + " unknown assoc info key: " + msg.appkey);
    else {
        ainfos.remove(msg.appkey);
        logger.debug("Worker " + ainfo.worker.workName
            + " thread " + msg.appkey + " active threads="
            + ainfo.worker.activeThreads);
    }
}

worker = workers.get( msg.getValue() );
if (null != worker) {
    worker.activeThreads--;  
    if (0 == worker.activeThreads) {
        logger.debug("worker added to idle queue "
            + worker.workName);
        idleQueue.add(worker);  
        worker.setState(WorkerState.IDLE);
    } else {
        logger.debug("worker " + worker.workName
            + " active=" + worker.activeThreads);
    }
} else
    logger.error("worker not found '"
            + msg.getValue() + "'");
break;

case "started":
    logger.debug("worker " + msg.getValue() + " "
    + msg.getCommand());
    worker = workers.get( msg.getValue() );
    if (null != worker) {
        logger.debug("worker added to idle queue "
            + worker.workName);
        idleQueue.add(worker);  
        worker.setState(WorkerState.IDLE);
    } else
logger.error("worker not found \
+ msg.getValue() + ")
break;
}
}

// look for pending messages
AssociationInfo assoc;
List<AssociationInfo> backup = new ArrayList<AssociationInfo>();
while (null != (assoc = mipc.msgQueue.poll())) {
  if (null == assoc.worker) {
    assoc.worker = getIdleWorker();
  }
  if (null == assoc.worker) {
    backup.add(assoc);
    continue;
  }
  String msg;
  while (null != (msg = assoc.msgs.poll())) {
    try {
      mipc.send(assoc.worker.workName, msg);
    } catch (Exception ex) {
      logger.error("assoc has error " + assoc.assocPk);
      logger.error("assoc worker " + assoc.worker);
      logger.error("cannot send worker message to "
        + assoc.worker.workName, ex);
    }
  }
  assoc.state = AssocState.IDLE;
}
if (0 != backup.size())
  mipc.msgQueue.addAll(backup);
else
  synchronized(mipc.workers) {
    try {
      mipc.workers.wait();
    } catch (Exception ex) {
      // no-op
    }
  }
} catch (Throwable th) {
  logger.error("error processing DicomWorkLeader "+ "thread, exiting", th);
}
private static String abstractPath(String path) {
  if (null == path)
    return null;
  return path.replace('\\', '/');
}

private Worker getIdleWorker() {
  Worker worker = idleQueue.poll();
  int workerQueue = 0;

  synchronized(workers) {
    workerQueue = workers.size();
  }

  try {
    if (0 == idleQueue.size() && MAXWORKERS > workerQueue)
      startWorker(mipc);
  } catch (Exception ex) {
    logger.error("Cannot start idle worker", ex);
  }

  if (null != worker) {
    worker.setState(WorkerState.ACTIVE);
    worker.activeThreads++;
    logger.debug("idle worker " + worker.workName + " activeThreads=" + worker.activeThreads);
    return worker;
  }

  // todo.
  // get least active worker to add work to.
  synchronized(workers) {
    int minThreads = 10000;
    for (Worker witem : workers.values()) {
      // if activeThreads was 0 it would be on the idleQueue
      // skip these
      if (0 != witem.activeThreads
            && minThreads > witem.activeThreads)
        worker = witem;
      minThreads = witem.activeThreads;
    }
  }
  worker.activeThreads++;
  logger.debug("busy worker " + worker.workName + " activeThreads=" + worker.activeThreads);
return worker;

private int nextSlot()
{
    int ret = keynum;
    keynum = (100 < keynum) ? 0 : 1 + keynum;
    return ret;
}

private static String formatWorkerName(int nnum)
{
    return String.format("DW%04d", nnum);
}

private void startWorker(DicomWorkLeader mipc)
    throws Exception
{
    int num = nextSlot();
    int last = num;
    String sname = formatWorkerName(num);
    synchronized(workers) {
        while (workers.containsKey(sname)) {
            num = nextSlot();
            if (last == num) {
                logger.warn("no more worker slots available");
                return;
            }
            sname = formatWorkerName(num);
        }
        logger.debug("waiting for ecore to become available");
        int count = 10;
        while (0 < count) {
            if (mipc.appReady("ecore"))
                break;
            Thread.sleep(500);
            count--;
        }
        mipc.send("ecore", "define " + sname + " dworker "
            + "com.teramedica.server.core.DicomWorker");
        // properties
        mipc.send("ecore", String.format("setArg %s prop INDEX=%d",
            sname, num));
        mipc.send("ecore", "setArg " + sname + " prop "
            + "logport=${DS1_LOG}";
        mipc.send("ecore", "setArg " + sname + " prop PROCNAME="
            + sname);
// jvm args
for (String arg : worker_args) {
mipc.send("ecore", "setArg " + sname + " arg " + arg);
}
mipc.send("ecore", "setArg " + sname + " arg -agentlib:" + "jdwp=transport=dt_socket,server=y,suspend=n,address=");
// class args
mipc.send("ecore", "setArg " + sname + " class " + sname);
mipc.send("ecore", "build " + sname);
logger.info("Starting worker " + sname);
mipc.send("ecore", "start " + sname);
Worker worker = new Worker(sname);
synchronized(workers) {
  workers.put(sname, worker);
}
count = 10;
while (0 < count) {
  if (mipc.appReady("DW" + sname))
    break;
  Thread.sleep(500);
  count--;
}
}
private DicomWorkLeader()
{
  try {
    String localhost = InetAddress.getLocalHost().getHostName();
    start(null, "DicomWorkLeader-" + localhost);
  } catch (Exception ex) {
    logger.error("error forking channel", ex);
  }
  workLeader = new WorkLeader(this, workers, ainfos);
  workLeader.start();
}
public static DicomWorkLeader getInstance()
{
  if (null == theServer)
    theServer = new DicomWorkLeader();
  return theServer;
public void shutdown()
{
    workLeader.shutdown = true;
    workLeader.shutdownTime = System.currentTimeMillis() + 180000;
    synchronized(workers) {
        workers.notify();
    }
}

public AssociationInfo createWorker(RCIdentifyingContext ctxt,
    String command, String name)
{
    AssociationInfo ainfo = new AssociationInfo(this, name);
    ainfo.sendInternal(command, null);
    ainfo.sendInternal("ctxt", ctxt.serialize());
    synchronized(workers) {
        ainfos.put(name, ainfo);
    }
    return ainfo;
}

public void finishWorker(AssociationInfo worker)
{
    worker.sendInternal("finish", null);
}

@Override
public void receiveMessage(String from, Serializable obj)
{
    String cmd = (String) obj;
    logger.debug("recvMsg: from: " + from + " msg: " + cmd);
    try {
        if (cmd.equals("stop")) {
            workLeader.shutdown = true;
            workLeader.shutdownTime
                = System.currentTimeMillis() + 180000;
            synchronized(workers) {
                workers.notify();
            }
            return;
        }
    }
}
if (cmd.equals("started")) {
    logger.debug("queued started from " + from);
    workLeader.msgs.add( new WorkMessage(cmd, from) );
    return;
}

Properties msg = new Properties();
try {
    ByteArrayInputStream bis = new ByteArrayInputStream(cmd.getBytes());
    msg.load(bis);
} catch (Exception ex) {
    logger.error("receiving message", ex);
}

cmd = msg.getProperty("cmd");
String apk = msg.getProperty("id");
String info;

AssociationInfo ainfo;
switch (cmd) {
    case "idle":
        workLeader.msgs.add( new WorkMessage(cmd, from, apk, null));
        break;
    case "clientCommand":
        workLeader.msgs.add( new WorkMessage(cmd, from, apk, msg) );
        break;
    default:
        logger.warn("unknown command: " + cmd);
        break;
}
synchronized(workers) {
    workers.notify();
}

} catch (Throwable th) {
    logger.error("failed receiving message ", th);
}
}
APPENDIX C - DicomWorker.java

Sample code DicomWorker.java showing the worker interface to implement messaging between the Server and the Workers.

```java
/*
 * DicomWorker.java
 * Created by rcoe
 * Copyright 2016
 */
package edu.marquette.rcoe.server.core;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.Serializable;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.TreeMap;

import org.apache.log4j.Logger;

import edu.marquette.rcoe.messaging.MessageIPC;
import edu.marquette.rcoe.server.core.DicomWorkLeader.WorkMessage;
import edu.marquette.rcoe.server.handler.CMoveHandler;
import edu.marquette.rcoe.server.handler.CStoreHandler;
// [ ... ] Other server interface imports

/**
 * DicomWorker -- worker for work items from the DicomServer
 */
public class DicomWorker extends MessageIPC {

    private static final Logger logger
            = Logger.getLogger(DicomWorker.class);

    private DicomWorkerShutdownHook shutdownHook;

    private Map<String, AssociationWorkerInfo> ainfos
            = new TreeMap<String, AssociationWorkerInfo>();
    private ConcurrentLinkedQueue<WorkMessage> msgQueue
            = new ConcurrentLinkedQueue<WorkMessage>();
```
private boolean shutdown = false;
private long shutdownTime = 0;

public enum AssocState {
    PENDING, // messages pending
    IDLE, // no messages pending
    STOPPING, // processing complete
    ZOMBIE
}

public static class WorkProcessor extends Thread {
    public WorkProcessor(String name) {
        super(name);
    }
}

public static class AssociationWorkerInfo {
    String assocPk;
    String requestor;
    RCIIdentifyingContext ctxt;
    AssocState state = AssocState.IDLE;
    WorkProcessor processor;
    DicomWorker parent;

    ConcurrentLinkedQueue<WorkMessage> msgs = new ConcurrentLinkedQueue<WorkMessage>();

    public AssociationWorkerInfo(DicomWorker worker, String from, String apk) {
        logger.debug("this: " + this + " parent: " + worker + " from: " + from + " apk: " + apk);
        parent = worker;
        requestor = from;
        assocPk = apk;
    }

    public RCIIdentifyingContext getContext() {
        return ctxt;
    }

    public void setContext(RCIIdentifyingContext ctx) {
        ctxt = ctx;
    }
public void send(String cmd)
{
    send(cmd, (String) null);
}

public void send(String cmd, Integer msg)
{
    send(cmd, msg.toString());
}

public void send(String cmd, Long msg)
{
    send(cmd, msg.toString());
}

public void send(String cmd, String msg)
{
    StringBuffer sb = new StringBuffer();
    sb.append("cmd=clientCommand\n");
    sb.append("id=").append(assocPk).append("\n");
    sb.append("clientCommand=").append(cmd).append("\n");
    if (null != msg)
        sb.append("clientMessage=").append(msg)
            .append("\n");
    try {
        parent.send(requestor, sb.toString());
    } catch (Exception ex) {
        logger.error("cannot send message to " + requestor
                + " cmd " + cmd + " message " + msg, ex);
    }
    // error recovery ??
    // propagate parent.send return val ??
}

public void addMessage(WorkMessage msg)
{
    msgs.add(msg);
}

public boolean isPending()
{
    return state.equals(AssocState.PENDING);
}

public void idle()
{
    StringBuffer sb = new StringBuffer();
sb.append("cmd=idle\n");
sb.append("id=").append(assocPk).append("\n");
try {
    parent.send(requestor, sb.toString());
} catch (Exception ex) {
    logger.error("cannot send message to " + requestor + " cmd idle message ", ex);
}

public boolean stop()
{
    if (state.equals(AssocState.STOPPING) && processor.getState().equals(Thread.State.TERMINATED)) {
        try {
            processor.join();
        } catch (Exception ex) {
            logger.error("cleaning up running thread", ex);
        }
        state = AssocState.ZOMBIE;
        return true;
    }
    return state.equals(AssocState.ZOMBIE);
}

public boolean isStopped()
{
    return state.equals(AssocState.ZOMBIE);
}

public WorkProcessor getProcessor()
{
    return processor;
}

public void setProcessor(WorkProcessor proc)
{
    processor = proc;
}

// overridden
public void process(WorkMessage msg)
{
}

private void process()
{
    WorkMessage msg;
while (null != (msg = msgs.poll())) {
    process(msg);
}
}

private class DicomWorkerShutdownHook
{
    DicomWorker mipc;

    public DicomWorkerShutdownHook(DicomWorker dwl, String appName)
    {
        super(appName);
        mipc = dwl;
    }

    protected void shutdown()
    {
        mipc.shutdown = true;
    }

    @Override
    public void receiveMessage(String from, Serializable obj)
    {
        String cmd = (String) obj;
        logger.debug("recvMsg: from: " + from + " msg: " + cmd);
        if (cmd.equals("stop")) {
            shutdown = true;
            shutdownTime = System.currentTimeMillis() + 180000;
            synchronized(ainfos) {
                ainfos.notify();
            }
            System.exit(0);
        }
        Properties msg = new Properties();
        try {
            ByteArrayInputStream bis = new ByteArrayInputStream(cmd.getBytes());
            msg.load(bis);
        } catch (Exception ex) {
            logger.error("receiving message", ex);
        }
    }
}
cmd = msg.getProperty("cmd");
String apk = msg.getProperty("id");

msgQueue.add( new WorkMessage(cmd, from, apk, msg) );

private void processMessage(DicomWorkLeader.WorkMessage wm) {
    String info;
    AssociationWorkerInfo ainfo;

    Properties msg = wm.getMessage();

    String cmd = wm.getCommand();
    String from = wm.getValue();
    String apk = wm.getAppKey();

    try {
        switch (cmd) {
        case "clientCommand":
            ainfo = ainfos.get(apk);
            logger.debug("lookup " + apk + " node: " + ainfo);
            if (null != ainfo) {
                logger.debug(String.format("cmd: %s  msg: %s",
                    msg.getProperty("clientCommand"),
                    msg.getProperty("clientMessage")));
                wm = new DicomWorkLeader.WorkMessage(
                    msg.getProperty("clientCommand"),
                    msg.getProperty("clientMessage")) ;
                ainfo.process(wm);
            }
            break;
        case "createCStore":
            logger.debug(String.format("CStoreHandler from %s for" + " %s : %s", from, apk, this));
            ainfo = new CStoreHandler.WorkerInfo(this, from, apk);
            ainfos.put(apk, ainfo);
            break;
        case "createCMove":
            logger.debug(String.format("CMoveHandler from %s for" + " %s : %s", from, apk, this));
            ainfo = new CMoveHandler.WorkerInfo(this, from, apk);
            logger.debug("insert " + apk + " node: " + ainfo);
            ainfos.put(apk, ainfo);
            break;
        case "ctxt":
            info = msg.getProperty("value");
            ainfo = ainfos.get(apk);
            if (null != ainfo) {
                
            }
// set ctxt
ainfo.setContext(
    RCIdentifyingContext.deserialize(info) );
}
break;

case "finish" :
    ainfo = ainfos.get(apk);
    logger.debug("finish: lookup " + apk + " node: "
        + ainfo);
    if (null != ainfo) {
        ainfos.remove(apk);
    }
    ainfos.remove(apk);
break;

    default:
    logger.warn("unknown command: " + cmd);
    break;
}
}
} catch (Throwable th) {
    logger.error("receiving message", th);
}

public DicomWorker(String appName)
{
    try {
        start(null, appName);
    } catch (Exception ex) {
        logger.error("cannot create main ipc");
    }
}

public DicomWorker(DicomWorker parent, String appName)
{
    try {
        // fork(parent, "dcm", appName, "DICOM");
        start(null, appName);
    } catch (Exception ex) {
        logger.error("cannot create forked ipc");
    }
}

public static void main(String args[])
{
    String shortName = null;
    System.out.println("Dicom worker starting with args=");
    for (int i = 0; i < args.length; i++) {
        if (0 == i)
            shortName = args[i];
System.out.println(args[i]);

try {
    if (null == shortName) {
        shortName = System.getProperty("T");
        if (shortName == null)
            shortName = "DS99";
    }

    Thread t = Thread.currentThread();
t.setName(shortName);
    Logger.pushLoggingContext( shortName );

    logger.debug("DicomWorker shortName=" + shortName);
    // DicomWorker mipc = new DicomWorker(shortName);
    DicomWorker theServer = new DicomWorker(null, shortName);
    theServer.setShutdownHook();
    theServer.run();
}
    } catch (Exception e) {
    logger.error("An exception has been caught starting " + " the DicomWorker", e);
    System.exit(1);
    }

public void run()
{
    logger.info("starting");
    Hibernator.getInstance().createSession();
    DicomConfigCacheManager cache
        = DicomConfigCacheManager.getInstance();

    try {
        String localhost
            = InetAddress.getLocalHost().getHostName();
        int count = 10;
        while (0 < count) {
            if (appReady("DicomWorkLeader-" + localhost))
                break;
            Thread.sleep(500);
            count--;
        }

        send("DicomWorkLeader-" + localhost, "started");
    }
try { catch (Exception ex) {
    logger.error("cannot send start to Leader", ex);
}

while (true) {
    boolean dostop = true;

    // look for pending messages
    WorkMessage msg;
    while (null != (msg = msgQueue.poll())) {
        processMessage(msg);
    }

    for (AssociationWorkerInfo ainfo : ainfos.values()) {
        if (ainfo.stop())
            ainfos.remove(ainfo.assocPk);
        dostop = dostop && ainfo.isStopped();
    }

    if (dostop && shutdown)
        break;

    if (shutdown
        && shutdownTime < System.currentTimeMillis())
        ;  /// what to do

    synchronized(ainfos) {
        try {
            ainfos.wait(1000);
        } catch (Exception ex) {
            // no-op
        }
    }
}

private void setShutdownHook() {
    if (shutdownHook == null) {
        shutdownHook = new DicomWorkerShutdownHook(this,
            "DicomWorker");
        Runtime.getRuntime().addShutdownHook( shutdownHook );
    }
}
APPENDIX D - CMoveProcess.java

Sample code CMoveProcess.java showing the received message processing in the Worker process.

```java
public static class WorkerInfo extends AssociationWorkerInfo {
    private CMoveProcessor cMoveProc;
    private CMoveHandler handler;
    private boolean abort = false;
    private long lastStudyPk = 0;
    private Map<Long, StudyInfo> studySops = new TreeMap<Long, StudyInfo>();

    public WorkerInfo(DicomWorker dwork, String from, String name) {
        super(dwork, from, name);
    }

    @Override
    public void process(DicomWorkLeader.WorkMessage msg) {
        logger.debug("process: " + msg.getCommand() + " msg: " + msg.getValue());
        if (abort) {
            logger.warn("not processing command in ABORT state: " + msg.getCommand());
            return;
        }
        try {
            switch (msg.getCommand()) {
                case "moveinfo":
                    try {
                        TMMoveInfo moveInfo = new TMMoveInfo( msg.getValue() );
                        handler = new CMoveHandler(getContext(), moveInfo);
                        cMoveProc = new CMoveProcessor(this, handler);
                    } catch (Exception ex) {
                        logger.error("cannot create a moveInfo", ex);
                        throw new DicomException("cannot create moveInfo", ex);
                    }
                    break;
            }
        } catch (Exception ex) {
            logger.error("cannot create a moveInfo", ex);
            throw new DicomException("cannot create moveInfo", ex);
        }
    }
}
```
    case "aefrom":
        cMoveProc.setListener(Long.valueOf(msg.getValue()));
        break;
    case "aedest":
        cMoveProc.setDestination(
            Long.valueOf(msg.getValue()));
        break;
    case "start":
        cMoveProc.setContext(getContext());
        setProcessor(cMoveProc);
        break;
    case "cmove":
        cMoveProc.addStudy(
            lastStudyPk = Long.valueOf(msg.getValue()));
        break;
    case "imageCount":
        studySops.put(lastStudyPk,
            new StudyInfo(Integer.valueOf(msg.getValue()),
                null));
        break;
    case "noSops":
        abort = true;
        break;
    case "cmove.start":
        cMoveProc.setThreadName(msg.getValue());
        cMoveProc.start();
        break;
    case "cancel":
        cMoveProc.cancel();
        break;
    case "finish":
        cMoveProc.join();
        idle();
        break;
    default:
        break;
    }
}
} catch (Throwable th) {
    logger.error("process: " + msg.getCommand(), th);
    abort = true;
    send("sendFinalResponse",
        DimseUtil.SUB_OP_COMPLETE_WITH_FAILURES);
    try {
        cMoveProc.join();
    } catch (Throwable tth) {
        logger.error("join failed or interrupted", tth);
    }
    idle();
}
public static class CMoveProcessor
    extends DicomWorker.WorkProcessor
{
    private TMIdentifyingContext ctxt;
    private CMoveHandler cmove;
    private WorkerInfo winfo;
    private List<Long> studyPks = new ArrayList<Long>();
    private Long listenPk;
    private Long destPk;
    private String thName;

    public CMoveProcessor(WorkerInfo winfo, CMoveHandler cmove)
    {
        super("CMoveProcessor");
        this.cmove = cmove;
        this.winfo = winfo;
    }

    public void setContext(TMIdentifyingContext ctxt)
    {
        this.ctxt = ctxt.clone();
    }

    public void addStudy(Long stupk)
    {
        studyPks.add(stupk);
    }

    public CMoveHandler getHandler()
    {
        return cmove;
    }

    public void setDestination(Long pk)
    {
        destPk = pk;
    }

    public void setListener(Long pk)
    {
        listenPk = pk;
    }

    public void setThreadName(String nm)
    {
        thName = nm;
    }
public void cancel()
{
    cmove.cancelStatus = true;
}

public void run()
{
    try {
        TMIdentifyingContext parent =
            cmove.requestorIC.getParentContext();
        cmove.requestorIC.setParentContext( parent.clone() );
        TMContextUtil.
            setCurrentIdentifyingContext( cmove.requestorIC );

        setName( thName );

        Session session =
            Hibernator.getInstance().createSession();
        List<TMStudy> studies = new ArrayList<TMStudy>();
        TMStudyDAO studyDAO =
            TMDaoFactory.getFactory().getStudyDAO();
        cmove.setListener( listenPk );
        cmove.setDestination( destPk );

        for (Long spk : studyPks) {
            try {
                TMStudy study = studyDAO.findByPk(spk, false);
                if (null != study)
                    studies.add( study );
                if (null == cmove.patient)
                    cmove.patient = study.getPatient();
            } catch (Exception ex) {
                logger.error("cannot find study by pk: " + spk,
                    ex);
            }
        }

        if (0 != studies.size())
            cmove.sendStudies(winfo, studies);

        Hibernator.getInstance().closeSession();
    } catch (Throwable th) {
        logger.error("processing cmove request", th);
        winfo.send("sendFinalResponse",
            DimseUtil.SUB_OP_COMPLETE_WITH_FAILURES);
    }
}
188     }
189     }

Sample code CStoreProcess.java showing the received message processing in the Worker process.

```java
public static class WorkerInfo extends AssociationWorkerInfo {
  private CStoreProcessor cstore;

  public WorkerInfo(DicomWorker dwork, String from, String name) {
    super(dwork, from, name);
  }

  @Override
  public void process(DicomWorkLeader.WorkMessage msg) {
    switch (msg.getCommand()) {
      case "start":
        CStoreHandler handler =
            new CStoreHandler(getContext());
        cstore = new CStoreProcessor(this, handler);
        cstore.setContext( getContext() );
        setProcessor(cstore);
        cstore.start();
        break;
      case "sop":
        cstore.add( Long.valueOf(msg.getValue()) );
        break;
      case "assocEnd":
        cstore.stopRunning();
        break;
      default:
        break;
    }
  }
}

public static class CStoreProcessor extends DicomWorker.WorkProcessor {
  private TMIdentifyingContext ctxt;
  private ConcurrentLinkedQueue<Long> queue =
      new ConcurrentLinkedQueue<Long>();
  private boolean running = true;
```
private CStoreHandler cstore;

public CStoreProcessor(WorkerInfo winfo, CStoreHandler cstore)
{
    super("CStoreProcessor");
    this.cstore = cstore;
}

// processes each sop as it arrives.

public void run()
{
    TMIdentifyingContext parent = ctxt.getParentContext();
    ctxt.setParentContext(parent.clone());
    TMContextUtil.setCurrentIdentifyingContext(ctxt);

    cstore.startStore();

    Session sess = Hibernator.getInstance().createSession();

    while (running || null != queue.peek()) {
        Long cpk;
        while (null != (cpk = queue.poll())) {
            logger.debug("processing c-store sop: " + cpk);
            try {
                cstore.deferredHandleEvent(cpk);
            } catch (Throwable th) {
                logger.error("cstore deferred failed", th);
            }
        }
        try {
            synchronized (this) {
                if (running)
                    wait();
            }
        } catch (Exception ex) {
            logger.error("wait", ex);
        }
    }
    logger.debug("exiting cstore thread, size " + queue.size());
    cstore.endStore();

    Hibernator.getInstance().closeSession();
}

public void add(Long csoppk)
{
    queue.add(csoppk);
try {
    synchronized (this) {
        notify();
    }
} catch (Exception ex) {
    logger.error("add notify", ex);
}

public void stopRunning() {
    running = false;
    try {
        synchronized (this) {
            notify();
        }
    } catch (Exception ex) {
        logger.error("stop notify", ex);
    }
}

public void setContext(TMIdentifyingContext ctxt) {
    this.ctxt = ctxt.clone();
}