JAVA Server Reliability in the Presence of Failures

Rich Coe
Marquette University

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

ABSTRACT
JAVA SERVER RELIABILITY IN THE PRESENCE OF FAILURES

Richard H. Coe, B.S.

Marquette University, 2016

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

LIST OF FIGURES

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 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 tecnologies 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.

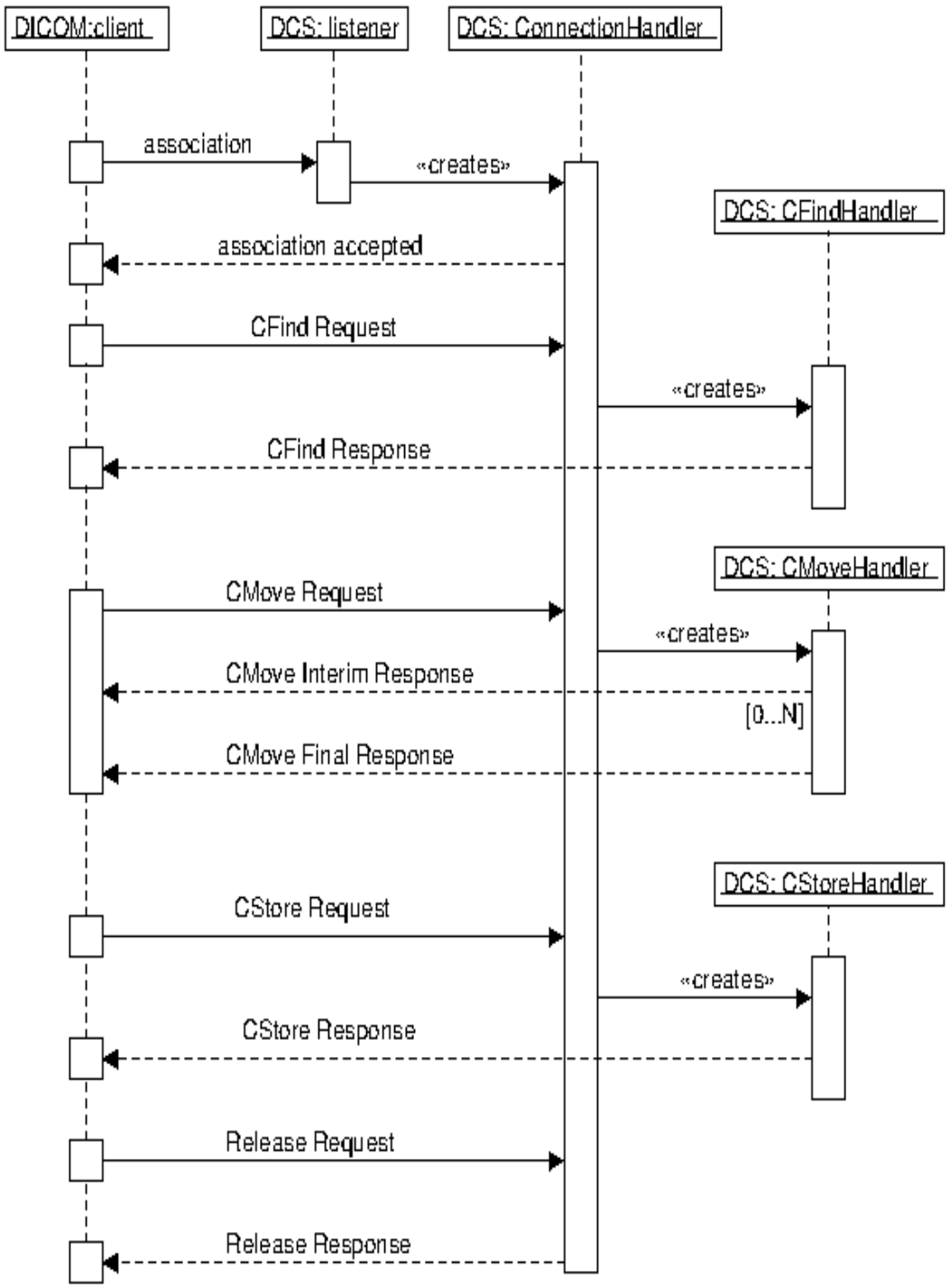

figure 1 - DICOM Server Architecture

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).

1: read DICOM data from the socket
2: create DB records patient, study, series, and file
3: write to disk location
4: commit db records
5: send DICOM requestor C-STORE result

figure 5 – DICOM C-STORE existing design

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.

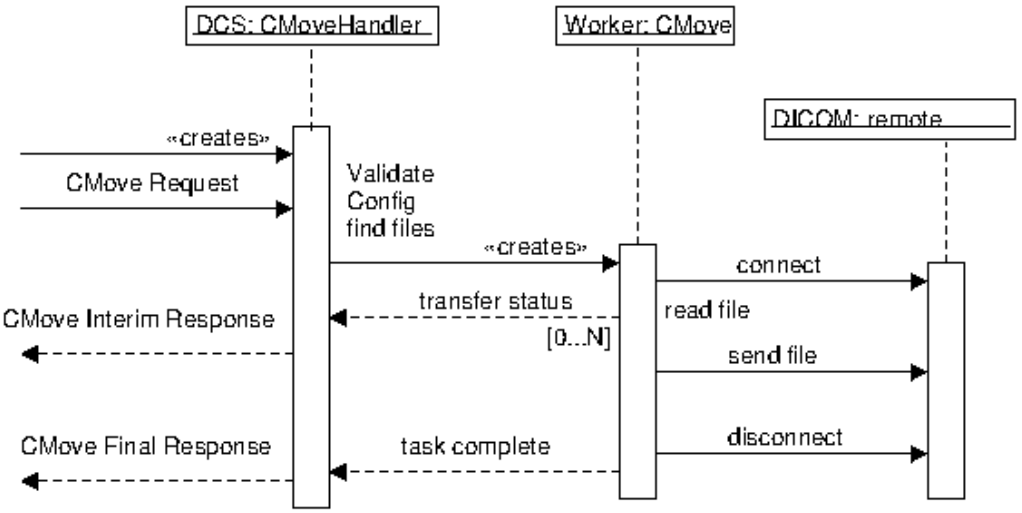

figure 6 – C-MOVE Worker Architecture

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.

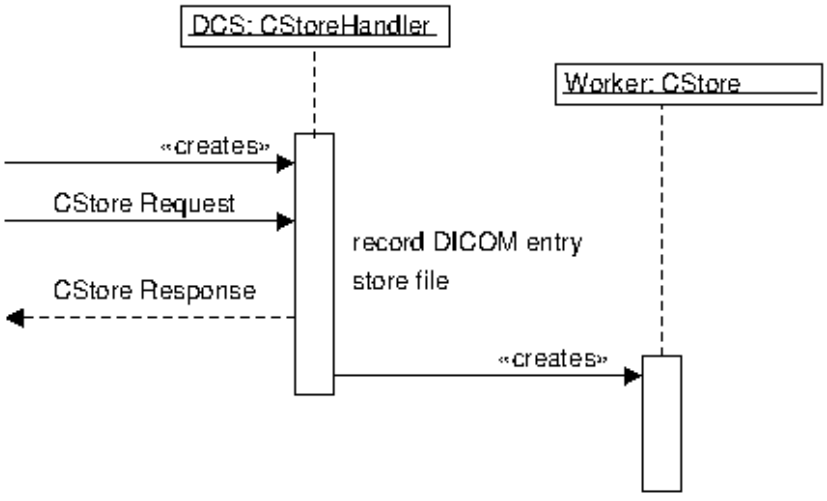

figure 7 - C-STORE Worker Architecture

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).

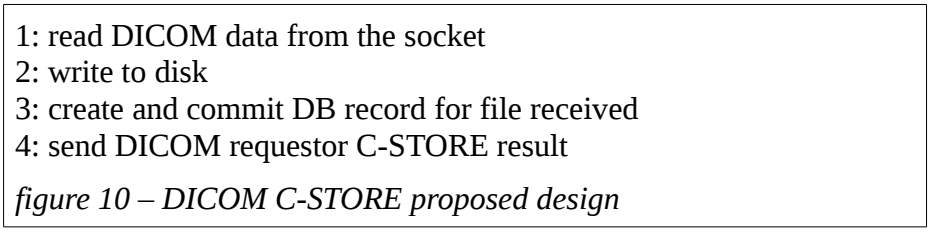

figure 10 – DICOM C-STORE proposed design

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

Package	Active Development	Native Java	No Broker	Zero Configuration	License	Accept
Aeron	Yes	Java-8	No	Yes	Apache	No
Akka	Yes	Java-8	Yes	Yes	Apache	No
Apache ActiveMQ	Yes	Yes	No	No	Apache	No
Apache Kafka	Yes	Yes	No	No	Apache	No
Apache Qpid Proton	Yes	Yes	No		Apache	No
Appia	No	Yes	Yes	No	Apache	No
Fast-cast	No	Yes	Yes	Yes	LGPL	No
jGCS	No	Yes	--	--	BSD	No
JGroups	Yes	Yes	Yes	Yes	Apache	Yes
Nanomsg	Yes	No	Yes	No	MIT	No
NeEM	No	Yes	Yes	Yes	modified MIT	No
Spread	No	No	No	No	modified MIT	No
ZeroMQ	Yes	No	--	--	LGPL	No

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.

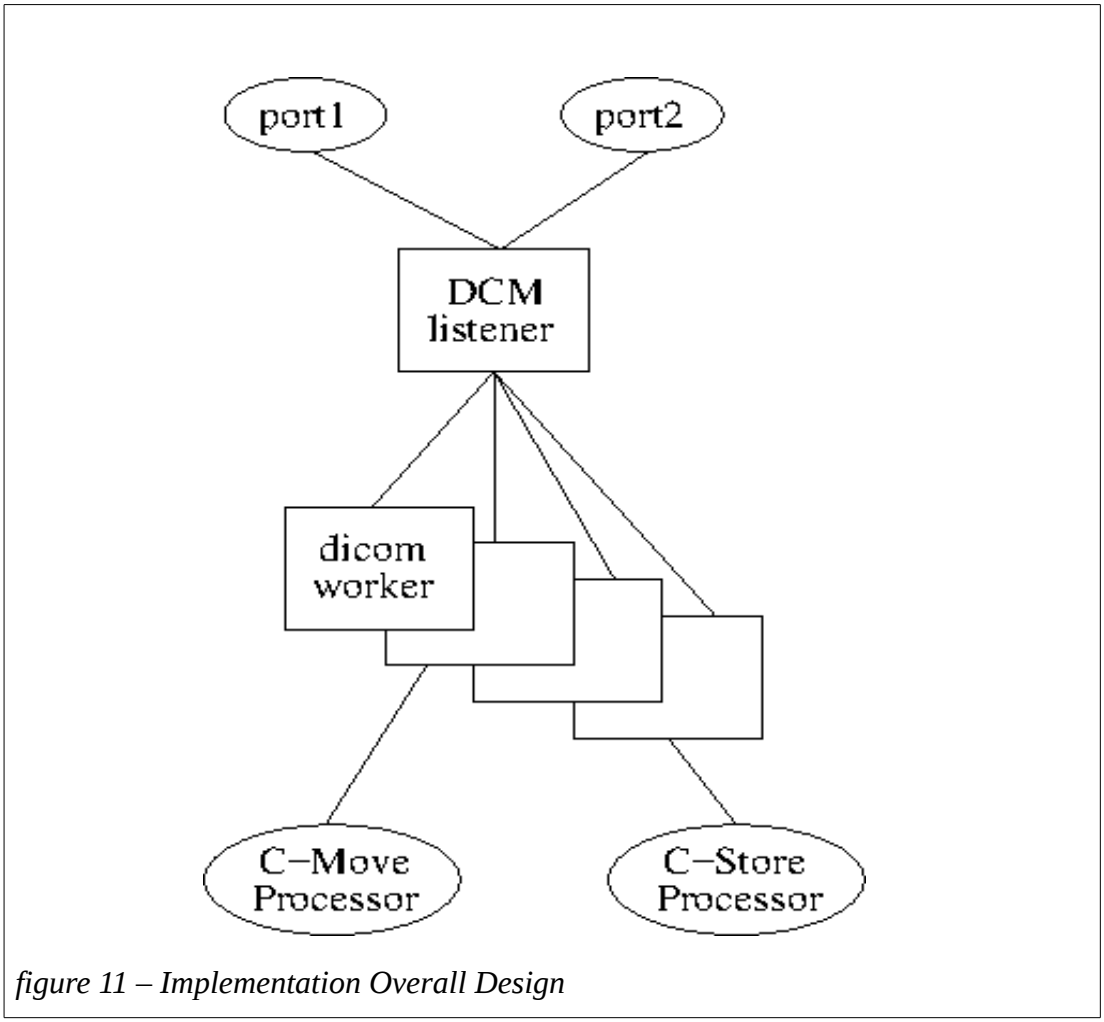

figure 11 – Implementation Overall Design

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.

```
1: while waiting to be done
2:   receive a command, if available, from the worker
3:    - image send started
4:    - study send started
5:    - requested file to send not found
6:    - file failed to send
7:    - file sent (good status)
8:    - file sent with warning
9:    - status message from a failure
10:   - receiver aborted the remote connection
11:   - request complete
12:  if DICOM C-Move requestor has issued a cancel request
13:   - send to the worker to cancel all processing
14:  send update of status to the DICOM requestor
15:  if no message available from the worker
16:   - sleep waiting for a message
17: end of waiting to be done
18: send to the worker the finish command
19: disconnect from the worker
```

figure 16 – C-MOVE Implementation Main Loop

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 catastophic 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

"Active MQ." Apache ActiveMQ. activemq.apache.org, 14 December 2014. Web. 14 December 2014.

"Advanced Message Queueing Protocol." Advanced Message Queueing Protocol. Wikipedia, January 2015. Web. January 2015.

"Akka Documentation." Akka Documentation. akka.io, Novemeber 2014. Web. 14 December 2014.

"Apache Kafka." Apache Kafka. kafka.apache.org, 27 December 2014. Web. 27 December 2014.

"Apache Kafka for Beginners." Cloudera Engineering Blog. blog.cloudera.com, 12 September 2014. Web. 27 December 2014.

"Appia Communication Framework." Main Page Appia. appia.di.fc.ul.pt, 4 January 2011. Web. 21 January 2015.

Ban, B., and V. Blagojevic. *Reliable group communication with JGroups 3*. x. Technical Report. Red Hat, Inc. Web. 2002.

Birman, Kenneth P. "Reliable distributed systems: technologies, web services, and applications." Springer Science & Business Media, 2005.

Blagojevic, Vladimir. "Implementing totem's total ordering protocol in javagroups reliable group communication toolkit." *York University*, 2000.

Carvalho, Nuno, José Pereira, and Luís Rodrigues. "Towards a generic group communication service." *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*. Springer Berlin Heidelberg, 2006. 1485-1502.

"Chord Wiki." Home sit/dht Wiki. github, 10 March 2013. Web. 30 November 2014.

"DICOM." Digital imaging and communications in medicine. National Electrical Manufacturers Association. dicom.nema.org, 1998. Web. January 2016.

"GNU Lesser General Public License." GNU Lesser General Public License. Wikipedia, February 2016. Web. February 2016.

"GNUnet." GNU's Framework for Secure Peer-to-Peer Networking. gnunet.org, 30 November 2014. Web. 30 November 2014.

Goscinski, Andrzej. Distributed operating systems. Addison-Wesley, 1991.

Hewitt, Carl, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence." Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Morgan Kaufman Publishers Inc., 1973.

Hintjens, Pieter. "ZeroMQ The Guide." 0MQ The Guide, 2014. Web. 27 October 2014.

Hoff, Todd. "Aeron: Do we really need another messaging system?" High Scalability. highscalability.com, 17 November 2014. Web. 17 November 2014.

"java.nio." Java Platform SE 7, doc.oracle.com. Web. October 2016.

"jGCS." Group Communication Service for Java. jgcs.sourceforge.net, 2008. Web. 21 January 2015.

"JGroups." JGroups wiki page. developer.jboss.org, 11 July 2014. Web. 21 January 2015.

Jones, Evan P C. "Efficient Java I/O: byte[], ByteBuffers, and OutputStreams" evanjones.ca, 21 October 2009. Web. 22 December 2015.

Kellomaki, Sampo. "BSD style file descriptor passing over unix domain sockets" cpansearch.perl.org, 30 January 2000. Web. September 2014.

Mangold, Eric P. "AMP." Asynchronous Messaging Protocol. amp-protocol.net, 2010. Web. 7 November 2014.

Mitev, Eduardo L. "EventDance, a peer-to-peer inter-process communication library." Bless the uncertainty - Blog Archive. blogs.igalia.com, 14 October 2010. Web. 30 November 2014.

Monson-Haefel, Richard and David A. Chappell. Java Message Service. "O'Reilly Media, Inc.", 2000.

Moeller, Ruediger. "fast-cast: hp low-latency reliable multicast messaging." github, 26 December 2014. 27 December 2014.

"nanomsg." nanomsg. nanomsg.org, 2014. Web. 14 December 2014.

"NeEM." NeEM Network-friendly Epidemic Multicast. neem.sourceforge.net, 10 February 2013. Web. 21 January 2015.

"Open-source software." Open-source software. Wikipedia, February 2016. Web. February 2016.

"Pastry." Pastry - A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications. www.freepatry.org, 13 March 2009. Web. 30 November 2014.

"Qpid Proton – Apache Qpid." Qpid Proton – Apache Qpid. qpid.apache.org, 14 December 2014. Web. 14 December 2014.

Ratnasamy, Sylvia. "A Scalable Content-Addressable Network." Ph.D. Thesis. www.icir.org, October 2002. Web. 30 November 2014.

"Spread." The Spread Toolkit. www.spread.org, 28 May 2014. Web. 21 January 2015.
Stanton, Jonathan R. "Spread Users Guide." The Spread Toolkit, 21 October 2002. Web. 21 January 2015.

Strachan, James. "How do I embed a Broker inside a Connection." Apache Active MQ. apache.org, 11 November 2014. Web. 17 April 2014.

Tanenbaum, Andrew S. "Modern Operating Systems." Prentice Hall. Englewood Cliffs, NJ, 1992.

Tate, Bruce. Bitter Java. Greenwich, Conn: Manning, 2002.

Thompson, Martin. "Aeron Wiki." Home real-logic/Aeron Wiki. github, 24 July 2015. Web. 9 July 2015.

Treat, Tyler. "What You Want Is What You Don't: Understanding Trade-Offs in Distributed Messaging." Brave New Geek, 23 August 2015. Web. 27 August 2015.

Windows Documentation. "Shared Sockets." Microsoft Developer Network, ms740478. September 2014.

"ZeroMQ." Distributed Messaging zeromq. zeromq.org, 2014. Web. 14 December 2014.

APPENDIX A – MessageIPC.java

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

```
 1  /* ///////
 2   * MessageIPC.java
 3   * Created on Sep 3, 2015 by rcoe
 4   * Copyright 2015
 5   */
 6  package edu.marquette.rcoe.messaging;
 7
 8  // A simple messaging interface to jgroups
 9  //
10  // Sample Usage
11  // Initialize:
12  //     SampleServerIPC.getInstance();
13
14  // -- SampleServerIPC.java --
15  // Server Message receiver
16  //
17  // package edu.marquette.rcoe.server.core;
18  //
19  // import java.io.Serializable;
20  //
21  // import org.apache.log4j.Logger;
22  //
23  // import edu.marquette.rcoe.messaging.MessageIPC;
24  //
25  // public class SampleServerIPC extends MessageIPC
26  // {
27  //   private static final Logger logger
28  //       = Logger.getLogger(SampleServerIPC.class);
29  //
30  //   private static SampleServerIPC instance
31  //       = new SampleServerIPC();
32  //
33  //   private SampleServerIPC()
34  //   {
35  //   }
36  //
37  //   public static SampleServerIPC getInstance()
38  //   {
39  //     return instance;
40  //   }
```

```
41  //
42  //   @Override
43  //   public void receiveMessage(String from, Serializable obj)
44  //   {
45  //     String cmd = (String) obj;
46  //
47  //     logger.info("received command " + cmd);
48  //
49  //     if (cmd.equals("stop"))
50  //       System.exit(0);
51  //   }
52  // }
53  // -- SampleServerIPC.java --
54
55  import java.io.ByteArrayInputStream;
56  import java.io.ByteArrayOutputStream;
57  import java.io.ObjectInputStream;
58  import java.io.ObjectOutputStream;
59  import java.io.Serializable;
60  import java.net.InetAddress;
61  import java.util.ArrayList;
62  import java.util.Collection;
63  import java.util.concurrent.ConcurrentLinkedQueue;
64  import java.util.Iterator;
65  import java.util.List;
66  import java.util.Map;
67  import java.util.Set;
68  import java.util.TreeMap;
69
70  import org.apache.log4j.Logger;
71  import org.jgroups.Address;
72  import org.jgroups.JChannel;
73  import org.jgroups.Message;
74  import org.jgroups.ReceiverAdapter;
75  import org.jgroups.util.Util;
76  import org.jgroups.View;
77
78  public class MessageIPC extends ReceiverAdapter
79  {
80    private static final Logger logger
81        = Logger.getLogger(MessageIPC.class);
82
83    // diff from default:
84    // set UDP:loopback to true for windows
85    // set GMS:print_local_addr to false
86
87    public static final String DEFAULT_PROTOCOL_STACK=
88      "UDP(mcast_port=45588;ip_ttl=4;tos=8;"
89          + "ucast_recv_buf_size=200K;"ucast_send_buf_size=200K;"
```

```
 90          + "mcast_recv_buf_size=200K;mcast_send_buf_size=200K;"
 91          + "max_bundle_size=64K;max_bundle_timeout=30;"
 92          + "enable_diagnostics=true;thread_naming_pattern=cl;"
 93          + "timer_type=new3;timer.min_threads=2;"
 94          + "timer.max_threads=4;timer.keep_alive_time=3000;"
 95          + "timer.queue_max_size=500;thread_pool.enabled=true;"
 96          + "thread_pool.min_threads=2;thread_pool.max_threads=8;"
 97          + "thread_pool.keep_alive_time=5000;"
 98          + "thread_pool.queue_enabled=true;"
 99          + "thread_pool.queue_max_size=10000;"
100          + "thread_pool.rejection_policy=discard;"
101          + "log_discard_msgs=false;"
102          + "oob_thread_pool.enabled=true;"
103          + "oob_thread_pool.min_threads=1;"
104          + "oob_thread_pool.max_threads=8;"
105          + "oob_thread_pool.keep_alive_time=5000;"
106          + "oob_thread_pool.queue_enabled=false;"
107          + "oob_thread_pool.queue_max_size=100;"
108          + "oob_thread_pool.rejection_policy=discard):"
109      + "PING:"
110      + "MERGE3(max_interval=30000;min_interval=10000):"
111      + "FD_SOCK:"
112      + "FD_ALL:"
113      + "VERIFY_SUSPECT(timeout=1500):"
114      + "BARRIER:"
115      + "pbcast.NAKACK2(xmit_interval=500;xmit_table_num_rows=100;"
116          + "xmit_table_msgs_per_row=2000;"
117          + "xmit_table_max_compaction_time=30000;"
118          + "max_msg_batch_size=500;"use_mcast_xmit=false;"
119          + "discard_delivered_msgs=true):"
120      + "UNICAST(xmit_interval=500;xmit_table_num_rows=100;"
121          + "xmit_table_msgs_per_row=2000;"
122          + "xmit_table_max_compaction_time=60000;"
123          + "conn_expiry_timeout=0;max_msg_batch_size=500):"
124      + "pbcast.STABLE(stability_delay=1000;"
125          + "desired_avg_gossip=50000;max_bytes=4M):"
126      + "pbcast.GMS(print_local_addr=false;join_timeout=2000;"
127          + "view_bundling=true):"
128      + "UFC(max_credits=2M;min_threshold=0.4):"
129      + "MFC(max_credits=2M;min_threshold=0.4):"
130      + "FRAG2(frag_size=60K):"
131      + "pbcast.STATE_TRANSFER()";
132
133    private static class iMessageIPC implements Serializable
134    {
135      public static final char HELLO = 1;
136      public static final char HELLOREPLY = 2;
137      public static final char GOODBYE = 3;
138      public static final char APPMSG = 4;
```

```
139
140       public static List<String> msgName = new ArrayList<String>();
141
142       static {
143         msgName.add("");
144         msgName.add("HELLO");
145         msgName.add("HELLOREPLY");
146         msgName.add("GOODBYE");
147         msgName.add("APPMSG");
148       };
149
150       private char msgType;
151       Serializable imsg;
152
153       public iMessageIPC(char mType, Serializable obj)
154       {
155         msgType = mType;
156         imsg = obj;
157       }
158
159       public byte[] serialize()
160       {
161         ByteArrayOutputStream bos = new ByteArrayOutputStream();
162         try {
163           ObjectOutputStream output = new ObjectOutputStream(bos);
164           output.writeObject(this);
165           output.flush();
166         } catch (Exception ex) {
167           logger.error("cannot encode message", ex);
168           return null;
169         }
170         logger.debug("created message of "
171             + bos.toByteArray().length);
172         return bos.toByteArray();
173       }
174
175       public static iMessageIPC deserialize(byte[] imsg, int off,
176           int len)
177       {
178         iMessageIPC msg = null;
179         try {
180           ByteArrayInputStream bis
181               = new ByteArrayInputStream(imsg, off, len);
182           ObjectInputStream input = new ObjectInputStream(bis);
183           msg = (iMessageIPC) input.readObject();
184         } catch (Exception ex) {
185           logger.error("cannot decode message", ex);
186           return null;
187         }
```

```
188       return msg;
189     }
190
191     public char getType()
192     {
193       return msgType;
194     }
195
196     public Serializable getMessage()
197     {
198       return imsg;
199     }
200
201     public static String msgTypeToString(int mType)
202     {
203       return msgName.get(mType);
204     }
205   }
206
207   private static class Members
208   {
209     private String appName;
210     private Address appAddr;
211     private String host;
212     private int tstamp;
213     private boolean exiting = false;
214
215     public Members(String name, Address addr)
216     {
217       appName = name;
218       appAddr = addr;
219       host = addrToHost(addr.toString());
220       tstamp = 0;
221     }
222
223     public String getName()
224     {
225       return appName;
226     }
227
228     public void setName(String name)
229     {
230       appName = name;
231     }
232
233     public Address getAddr()
234     {
235       return appAddr;
236     }
```

```
237
238       public int getStamp()
239       {
240         return tstamp;
241       }
242
243       public void setStamp(int stamp)
244       {
245         tstamp = stamp;
246       }
247
248       public String getHost()
249       {
250         return host;
251       }
252
253       public void setExiting()
254       {
255         exiting = true;
256       }
257
258       public boolean isExiting()
259       {
260         return exiting;
261       }
262
263       public String toString()
264       {
265         return new String(appName + "@" + appAddr);
266       }
267     }
268
269     private static class IPCProcessor extends Thread
270     {
271       private MessageIPC mipc;
272       private ConcurrentLinkedQueue<Message> queue
273           = new ConcurrentLinkedQueue<Message>();
274       private boolean running = true;
275
276       public IPCProcessor(MessageIPC mipc)
277       {
278         super("IPCProcessor");
279         this.mipc = mipc;
280       }
281
282       public void run()
283       {
284         while (running) {
285           Message msg;
```

```
286           while (null != (msg = queue.poll())) {
287             String appname;
288             boolean reply = false;
289             iMessageIPC imsg
290                 = iMessageIPC.deserialize(msg.getRawBuffer(),
291                     msg.getOffset(), msg.getLength());
292
293             if (null == imsg) {
294               logger.error("invalid message of "
295                   + msg.getRawBuffer().length);
296             } else
297               switch (imsg.getType()) {
298               // when an app starts up, it sends HELLO
299               // when an app receives HELLO, reply with a HELLOREPLY
300               case iMessageIPC.HELLO:
301                 reply = true;
302               case iMessageIPC.HELLOREPLY:
303                 appname = (String) imsg.getMessage();
304                 logger.debug("hello from " + appname + " status: "
305                     + mipc.app2addr.get(appname));
306                 Members memb = mipc.addr2app.get(msg.getSrc());
307
308                 if (null != memb) {
309                   // existing, add it's name
310                   memb.setName(appname);
311                   mipc.app2addr.put(appname, memb);
312                 } else {
313                   // create new
314                   memb = new Members(appname, msg.getSrc());
315                 }
316                 if (reply) {
317                   try {
318                     mipc.send(iMessageIPC.HELLOREPLY, msg.getSrc(),
319                         mipc.appName);
320                   } catch (Exception ex) {
321                     logger.error("cannot return HELLO to "
322                         + appname, ex);
323                   }
324                 }
325                 break;
326               // an app is exiting
327               case iMessageIPC.GOODBYE:
328                 appname = (String) imsg.getMessage();
329                 logger.debug("goodbye from " + appname);
330                 mipc.removeApp(appname);
331                 break;
332               // an app upper layer message
333               case iMessageIPC.APPMSG:
334                 memb = mipc.addr2app.get(msg.getSrc());
```

```
335                String from = (null != memb) ? memb.getName()
336                          : "unknown";
337                mipc.receiveMessage(from, imsg.getMessage());
338                break;
339              default:
340                logger.error("Unknown message received type: "
341                    + imsg.getType());
342                break;
343              }
344          }
345          try {
346            synchronized (this) {
347              wait();
348            }
349          } catch (Exception ex) {
350            logger.error("wait", ex);
351          }
352        }
353      }
354
355      // write to head read from tail
356      public void add(Message msg)
357      {
358        queue.add(msg);
359        try {
360          synchronized (this) {
361            notify();
362          }
363        } catch (Exception ex) {
364          logger.error("add notify", ex);
365        }
366      }
367
368      public void stopRunning()
369      {
370        running = false;
371        try {
372          synchronized (this) {
373            notify();
374          }
375        } catch (Exception ex) {
376          logger.error("stop notify", ex);
377        }
378      }
379    }
380
381    private JChannel jch = null;
382
383    private String appName;
```

```
384
385     private Map<String, Members> app2addr
386       = new TreeMap<String, Members>();
387     private Map<Address, Members> addr2app
388       = new TreeMap<Address, Members>();
389
390     private int tstamp = 0;
391
392     private IPCProcessor msgThread;
393
394     public void start(String props, String name) throws Exception
395     {
396       String localhost = InetAddress.getLocalHost().getHostName();
397       start(props, name, localhost);
398     }
399
400     public void start(String props, String name, String channel)
401           throws Exception
402     {
403       if (null != jch)
404           return;
405
406       logger.debug("starting IPC for " + name);
407       props = (null == props) ? DEFAULT_PROTOCOL_STACK : props;
408       jch = new JChannel(props);
409
410       appName = name;
411       if (null != name) jch.setName(name);         // later release
412       jch.setDiscardOwnMessages(true);             // added in 3.0
413
414       msgThread = new IPCProcessor(this);
415       msgThread.start();
416
417       jch.setReceiver(this);
418       logger.info("connecting to group " + channel);
419       jch.connect(channel);
420
421       addApp(new Members(appName, jch.getAddress()));
422
423       send(iMessageIPC.HELLO, null, name);
424     }
425
426     public void shutdown() throws Exception
427     {
428       if (null != jch) {
429         send(iMessageIPC.GOODBYE, null, appName);
430         jch.disconnect();
431         msgThread.stopRunning();
432       }
```

```
433       Util.close(jch);
434       jch = null;
435     }
436
437     public void viewAccepted(View newView)
438     {
439       tstamp++;
440       for (Object obj : newView.getMembers()) {
441         Address addr = (Address) obj;
442         Members app = addr2app.get(addr);
443         if (null == app) {
444           app = new Members(null, addr);
445           addApp(app);
446         }
447         app.setStamp(tstamp);
448       }
449       Iterator<Members> vit = addr2app.values().iterator();
450       while (vit.hasNext()) {
451         Members memb = vit.next();
452         if (tstamp != memb.getStamp()) {
453           vit.remove();
454           if (null != memb.getName())
455             app2addr.remove(memb.getName());
456           }
457       }
458     }
459
460     public Collection<Members> getMembers()
461     {
462       return addr2app.values();
463     }
464
465     public void receive(Message msg)
466     {
467       msgThread.add(msg);
468     }
469
470     // to be overridden
471     public void receiveMessage(String from, Serializable obj)
472     {
473     }
474
475     private void send(char mType, Address dest, Serializable obj)
476         throws Exception
477     {
478       logger.debug("send " + iMessageIPC.msgTypeToString(mType));
479       iMessageIPC imsg = new iMessageIPC(mType, obj);
480       Message msg = new Message(dest, null, imsg.serialize());
481       jch.send(msg);
```

```
482    }
483
484    public boolean send(String dest, Serializable obj)
485        throws Exception
486    {
487      if (null == jch)
488        return false;
489      Address ecdest = null;
490      if (null != dest) {
491        Members memb = app2addr.get(dest);
492        if (null == memb) {
493          logger.error("cannot send to non-member" + " destination "
494              + dest);
495          logger.info("rebroadcasting app discovery");
496          send(iMessageIPC.HELLO, null, appName);
497          return false;
498        }
499        if (memb.isExiting())
500          return false;
501        ecdest = memb.getAddr();
502      }
503      send(iMessageIPC.APPMSG, ecdest, obj);
504      return true;
505    }
506
507    public Set<String> getApps()
508    {
509      logger.debug("getApps");
510      return app2addr.keySet();
511    }
512
513    public List<String> getLocalApps()
514    {
515      logger.debug("getLocalApps");
516      List<String> rapps = new ArrayList<String>();
517
518      for (Members memb : app2addr.values()) {
519        logger.debug( "member addr " + memb.getHost() );
520        rapps.add(memb.getName());
521      }
522      return rapps;
523    }
524
525    private void addApp(Members app)
526    {
527      String name = app.getName();
528      Address addr = app.getAddr();
529      logger.debug("adding app " + name + " from " + addr);
530      if (null != name)
```

```
531         app2addr.put(name, app);
532       addr2app.put(addr, app);
533     }
534
535     private void removeApp(String appName)
536     {
537       Members memb;
538       logger.debug("removing app " + appName);
539       if (null != (memb = app2addr.get(appName))) {
540         memb.setExiting();
541       }
542     }
543
544     private static String addrToHost(String addr)
545     {
546       int colon = addr.indexOf(':');
547       if (-1 != colon)
548         return addr.substring(0, colon);
549       return addr;
550     }
551   }
```

APPENDIX B - DicomWorkLeader.java

Sample code DicomWorkLeader.java showing the server interface to implement messaging between the Server and the Workers.

```
 1  /*
 2   * DicomWorkLeader.java
 3   * Created by rcoe
 4   * Copyright 2016
 5   */
 6  package edu.marquette.rcoe.server.core;
 7
 8  import java.io.ByteArrayInputStream;
 9  import java.io.File;
10  import java.io.Serializable;
11  import java.net.InetAddress;
12  import java.util.ArrayList;
13  import java.util.concurrent.ConcurrentLinkedQueue;
14  import java.util.List;
15  import java.util.Map;
16  import java.util.Properties;
17  import java.util.TreeMap;
18
19  import org.apache.log4j.Logger;
20
21  import edu.marquette.rcoe.messaging.MessageIPC;
22  // [ ... ] Other server interface imports
23
24  /**
25   *  DicomWorkLeader -- work leader for the DicomServer
26   */
27  public class DicomWorkLeader extends MessageIPC
28  {
29    private static final Logger logger
30        = Logger.getLogger(DicomWorkLeader.class);
31
32    private static DicomWorkLeader theServer = null;
33
34    private WorkLeader workLeader;
35
36    private Map<String, AssociationInfo> ainfos
37        = new TreeMap<String, AssociationInfo>();
38    private Map<String, Worker> workers
39        = new TreeMap<String, Worker>();
40    private ConcurrentLinkedQueue<AssociationInfo> msgQueue
```

```
41        = new ConcurrentLinkedQueue<AssociationInfo>();
42
43    private static List<String> worker_args
44        = new ArrayList<String>();
45
46    private int sendStudyCount = 0;
47
48    static {
49      worker_args.add("-DT=DW${INDEX}");
50      worker_args.add("-DlogPropertyFile="
51          + "${XNS_HOME}/config/DicomWorkerlog.properties");
52      worker_args.add("-DPROCNAME=${PROCNAME}");
53      worker_args.add("-DLOGPORT=${logport}");
54      worker_args.add("-Djava.net.preferIPv4Stack=true");
55      worker_args.add("-server");
56      worker_args.add("-DXNS_HOME=${XNS_HOME}");
57      // worker_args.add("-Xms128k");                // initial heap
58      worker_args.add("-Xmx1024m");                  // max heap
59    };
60
61    public enum WorkerState {
62      STARTING,             // jvm launched
63      IDLE,                 // received alive message
64      ACTIVE,               // working on a task
65      STOPPING,             // IDLE or ACTIVE jvm sent stop message
66      ZOMBIE                // jvm stopped
67    }
68
69    public enum AssocState {
70      PENDING,                // messages pending
71      IDLE                    // no messages pending
72    }
73
74      // class Worker manages the remote worker proc
75    private static class Worker
76    {
77      String workName;
78      WorkerState state = WorkerState.STARTING;
79      int activeThreads = 0;
80
81      public Worker(String name)
82      {
83        workName = name;
84      }
85
86      public void setState(WorkerState st)
87      {
88        state = st;
89      }
```

```
 90
 91      public boolean isActive()
 92      {
 93        return state.equals(WorkerState.ACTIVE);
 94      }
 95
 96      public boolean isStopping()
 97      {
 98        return state.equals(WorkerState.STOPPING);
 99      }
100
101      public boolean isStopped()
102      {
103        return state.equals(WorkerState.ZOMBIE);
104      }
105
106      public void setStopped()
107      {
108      }
109    }
110
111    public static class WorkMessage
112    {
113      private String command;
114      private String value;
115      private String appkey;
116      private Properties message;
117
118      public WorkMessage(String cmd, String val)
119      {
120        command = cmd;
121        value = val;
122      }
123
124      public WorkMessage(String cmd, String val, String key,
125          Properties msg)
126      {
127        command = cmd;
128        value = val;
129        appkey = key;
130        message = msg;
131      }
132
133      public String getCommand()
134      {
135        return command;
136      }
137
138      public String getValue()
```

```
139       {
140         return value;
141       }
142
143       public String getAppKey()
144       {
145         return appkey;
146       }
147
148       public Properties getMessage()
149       {
150         return message;
151       }
152     }
153
154     // class AssocationInfo represents the DICOM work being
155     // managed by the worker
156     public static class AssociationInfo
157     {
158       String assocPk;
159       private String ctxt;
160       private AssocState state = AssocState.IDLE;
161       Worker worker = null;
162       ConcurrentLinkedQueue<String> msgs
163           = new ConcurrentLinkedQueue<String>();
164       ConcurrentLinkedQueue<WorkMessage> replies
165           = new ConcurrentLinkedQueue<WorkMessage>();
166       DicomWorkLeader mipc;
167
168       AssociationInfo(DicomWorkLeader parent, String apk)
169       {
170         assocPk = apk;
171         mipc = parent;
172       }
173
174       public void send(String command)
175       {
176         send(command, (String) null);
177       }
178
179       public void send(String command, String msg)
180       {
181         StringBuffer sb = new StringBuffer();
182         sb.append("cmd=clientCommand\n");
183         sb.append("id=").append(assocPk).append("\n");
184         sb.append("clientCommand=").append(command).append("\n");
185         if (null != msg)
186           sb.append("clientMessage=").append(msg).append("\n");
187         addMessage( sb.toString() );
```

```
188       }
189
190       private void sendInternal(String command, String value)
191       {
192         StringBuffer sb = new StringBuffer();
193         sb.append("cmd=").append(command).append("\n");
194         sb.append("id=").append(assocPk).append("\n");
195         if (null != value)
196           sb.append("value=").append(value).append("\n");
197         addMessage( sb.toString() );
198       }
199
200       public void addMessage(String msg)
201       {
202         msgs.add(msg);
203         if (state.equals(AssocState.IDLE)) {
204           mipc.msgQueue.add(this);
205           state = AssocState.PENDING;
206         }
207         synchronized(mipc.workers) {
208           mipc.workers.notify();
209         }
210       }
211
212       public WorkMessage getMessage()
213       {
214         return replies.poll();
215       }
216
217       public void addReply(WorkMessage msg)
218       {
219         replies.add(msg);
220       }
221
222       public boolean isPending()
223       {
224         return state.equals(AssocState.PENDING);
225       }
226     }
227
228       // class WorkLeader manages the running Workers
229     private static class WorkLeader extends Thread
230     {
231       private static int MAXWORKERS = 10;
232
233       private Map<String, Worker> workers;
234       private Map<String, AssociationInfo> ainfos;
235       private boolean shutdown = false;
236       private long shutdownTime = 0;
```

```
237     private int debugPort = 5020;
238
239     private ConcurrentLinkedQueue<Worker> idleQueue
240         = new ConcurrentLinkedQueue<Worker>();
241     private ConcurrentLinkedQueue<WorkMessage> msgs
242         = new ConcurrentLinkedQueue<WorkMessage>();
243
244     DicomWorkLeader mipc;
245
246     private int keynum = 0;
247
248     public WorkLeader(DicomWorkLeader parent,
249         Map<String, Worker> wrx, Map<String, AssociationInfo> aix)
250     {
251       workers = wrx;
252       ainfos = aix;
253       mipc = parent;
254     }
255
256     public void run()
257     {
258       Thread t = Thread.currentThread();
259       t.setName("DicomWorkLeader");
260       Logger.pushLoggingContext("DicomWorkLeader");
261
262       // start workers
263       try {
264         for (int i=0; i < MAXWORKERS; i++)
265           startWorker(mipc);
266       } catch (Exception ex) {
267         logger.error("Cannot start initial workers", ex);
268       }
269
270       try {
271       while (true) {
272         boolean dostop = true;
273         AssociationInfo ainfo;
274
275         synchronized(workers) {
276           WorkMessage msg;
277           while (null != (msg = msgs.poll())) {
278             Worker worker;
279             switch (msg.getCommand()) {
280             case  "clientCommand":
281               logger.debug("worker " + msg.getValue() + " "
282                   + msg.getCommand() + " " + msg.message);
283               ainfo = ainfos.get(msg.appkey);
284               if (null == ainfo)
285                 logger.error("cc from: " + msg.getValue()
```

```
286                     + " unknown assoc info key: " + msg.appkey);
287                 else {
288                   ainfo.addReply(
289                       new WorkMessage(
290                           msg.message.getProperty("clientCommand"),
291                           msg.message.getProperty("clientMessage")
292                       ) );
293                 }
294                 break;
295               case  "idle":
296                 logger.debug("worker " + msg.getValue() + " "
297                     + msg.getCommand());
298                 ainfo = ainfos.get(msg.appkey);
299                 if (null == ainfo)
300                   logger.error("idle from: " + msg.getValue()
301                       + " unknown assoc info key: " + msg.appkey);
302                 else {
303                   ainfos.remove(msg.appkey);
304                   logger.debug("Worker " + ainfo.worker.workName
305                       + " thread " + msg.appkey + " active threads="
306                       + ainfo.worker.activeThreads);
307                 }
308
309                 worker = workers.get( msg.getValue() );
310                 if (null != worker) {
311                   worker.activeThreads--;
312                   if (0 == worker.activeThreads) {
313                     logger.debug("worker added to idle queue "
314                         + worker.workName);
315                     idleQueue.add(worker);
316                     worker.setState(WorkerState.IDLE);
317                   } else {
318                     logger.debug("worker " + worker.workName
319                         + " active=" + worker.activeThreads);
320                   }
321                 } else
322                   logger.error("worker not found '"
323                       + msg.getValue() + "'");
324                 break;
325               case  "started":
326                 logger.debug("worker " + msg.getValue() + " "
327                     + msg.getCommand());
328                 worker = workers.get( msg.getValue() );
329                 if (null != worker) {
330                   logger.debug("worker added to idle queue "
331                       + worker.workName);
332                   idleQueue.add(worker);
333                   worker.setState(WorkerState.IDLE);
334                 } else
```

```
335                 logger.error("worker not found '"
336                     + msg.getValue() + "'");
337               break;
338             }
339           }
340         }
341
342         // look for pending messages
343         AssociationInfo assoc;
344         List<AssociationInfo> backup
345             = new ArrayList<AssociationInfo>();
346         while (null != (assoc = mipc.msgQueue.poll())) {
347           if (null == assoc.worker) {
348             assoc.worker = getIdleWorker();
349           }
350           if (null == assoc.worker) {
351             backup.add(assoc);
352             continue;
353           }
354           String msg;
355           while (null != (msg = assoc.msgs.poll())) {
356             try {
357               mipc.send(assoc.worker.workName, msg);
358             } catch (Exception ex) {
359               logger.error("assoc has error " + assoc.assocPk);
360               logger.error("assoc worker " + assoc.worker);
361               logger.error("cannot send worker message to "
362                   + assoc.worker.workName, ex);
363             }
364           }
365           assoc.state = AssocState.IDLE;
366         }
367         if (0 != backup.size())
368           mipc.msgQueue.addAll(backup);
369         else
370           synchronized(mipc.workers) {
371             try {
372               mipc.workers.wait();
373             } catch (Exception ex) {
374               // no-op
375             }
376           }
377         }
378         } catch (Throwable th) {
379           logger.error("error processing DicomWorkLeader "
380               + "thread, exiting", th);
381         }
382       }
383
```

```
384      private static String abstractPath(String path)
385      {
386        if (null == path)
387          return null;
388        return path.replace('\\', '/');
389      }
390
391      private Worker getIdleWorker()
392      {
393        Worker worker = idleQueue.poll();
394        int workerQueue = 0;
395
396        synchronized(workers) {
397          workerQueue = workers.size();
398        }
399
400        try {
401          if (0 == idleQueue.size() && MAXWORKERS > workerQueue)
402            startWorker(mipc);
403        } catch (Exception ex) {
404          logger.error("Cannot start idle worker", ex);
405        }
406
407        if (null != worker) {
408          worker.setState(WorkerState.ACTIVE);
409          worker.activeThreads++;
410          logger.debug("idle worker " + worker.workName
411              + " activeThreads=" + worker.activeThreads);
412          return worker;
413        }
414
415        // todo.
416        // get least active worker to add work to.
417        synchronized(workers) {
418          int minThreads = 10000;
419          for (Worker witem : workers.values()) {
420            // if activeThreads was 0 it would be on the idleQueue
421            // skip these
422            if (0 != witem.activeThreads
423                && minThreads > witem.activeThreads) {
424              worker = witem;
425              minThreads = witem.activeThreads;
426            }
427          }
428        }
429        worker.activeThreads++;
430        logger.debug("busy worker " + worker.workName
431            + " activeThreads=" + worker.activeThreads);
432
```

```
433       return worker;
434     }
435
436     private int nextSlot()
437     {
438       int ret = keynum;
439       keynum = (100 < keynum) ? 0 : 1 + keynum;
440       return ret;
441     }
442
443     private static String formatWorkerName(int nnum)
444     {
445       return String.format("DW%04d", nnum);
446     }
447
448     private void startWorker(DicomWorkLeader mipc)
449         throws Exception
450     {
451       int num = nextSlot();
452       int last = num;
453       String sname = formatWorkerName(num);
454       synchronized(workers) {
455         while (workers.containsKey(sname)) {
456           num = nextSlot();
457           if (last == num) {
458             logger.warn("no more worker slots available");
459             return;
460           }
461           sname = formatWorkerName(num);
462         }
463       }
464
465       logger.debug("waiting for ecore to become available");
466       int count = 10;
467       while (0 < count) {
468         if (mipc.appReady("ecore"))
469           break;
470         Thread.sleep(500);
471         count--;
472       }
473       mipc.send("ecore", "define " + sname + " dworker "
474           + "com.teramedica.server.core.DicomWorker");
475       // properties
476       mipc.send("ecore", String.format("setArg %s prop INDEX=%d",
477           sname, num));
478       mipc.send("ecore", "setArg " + sname + " prop "
479           + "logport=${DS1_LOG}");
480       mipc.send("ecore", "setArg " + sname + " prop PROCNAME="
481           + sname);
```

```
482         // jvm args
483         for (String arg : worker_args) {
484           mipc.send("ecore", "setArg " + sname + " arg " + arg);
485         }
486         mipc.send("ecore", "setArg " + sname + " arg -agentlib:"
487             + "jdwp=transport=dt_socket,server=y,suspend=n,address="
488             + debugPort++);
489         // class args
490         mipc.send("ecore", "setArg " + sname + " class " + sname);
491         mipc.send("ecore", "build " + sname);
492 
493         logger.info("Starting worker " + sname );
494         mipc.send("ecore", "start " + sname);
495 
496         Worker worker = new Worker(sname);
497 
498         synchronized(workers) {
499           workers.put(sname, worker);
500         }
501 
502         count = 10;
503         while (0 < count) {
504           if (mipc.appReady("DW" + sname))
505             break;
506           Thread.sleep(500);
507           count--;
508         }
509       }
510     }
511 
512     private DicomWorkLeader()
513     {
514       try {
515         String localhost = InetAddress.getLocalHost().getHostName();
516         start(null, "DicomWorkLeader-" + localhost);
517       } catch (Exception ex) {
518         logger.error("error forking channel", ex);
519       }
520 
521       workLeader = new WorkLeader(this, workers, ainfos);
522       workLeader.start();
523     }
524 
525     public static DicomWorkLeader getInstance()
526     {
527       if (null == theServer)
528         theServer = new DicomWorkLeader();
529 
530       return theServer;
```

```
531    }
532
533    public void shutdown()
534    {
535      workLeader.shutdown = true;
536      workLeader.shutdownTime = System.currentTimeMillis() + 180000;
537      synchronized(workers) {
538        workers.notify();
539      }
540    }
541
542    public AssociationInfo createWorker(RCIdentifyingContext ctxt,
543        String command, String name)
544    {
545      AssociationInfo ainfo = new AssociationInfo(this, name);
546
547      ainfo.sendInternal(command, null);
548      ainfo.sendInternal("ctxt", ctxt.serialize());
549
550      synchronized(workers) {
551        ainfos.put(name, ainfo);
552      }
553
554      return ainfo;
555    }
556
557    public void finishWorker(AssociationInfo worker)
558    {
559      worker.sendInternal("finish", null);
560    }
561
562    @Override
563    public void receiveMessage(String from, Serializable obj)
564    {
565      String cmd = (String) obj;
566
567      logger.debug("recvMsg: from: " + from + " msg: " + cmd);
568
569      try {
570        if (cmd.equals("stop")) {
571          workLeader.shutdown = true;
572          workLeader.shutdownTime
573              = System.currentTimeMillis() + 180000;
574          synchronized(workers) {
575          workers.notify();
576        }
577        return;
578      }
579
```

```
580       if (cmd.equals("started")) {
581         logger.debug("queued started from " + from);
582         workLeader.msgs.add( new WorkMessage(cmd, from) );
583         return;
584       }
585
586       Properties msg = new Properties();
587       try {
588         ByteArrayInputStream bis
589             = new ByteArrayInputStream(cmd.getBytes());
590         msg.load(bis);
591       } catch (Exception ex) {
592         logger.error("receiving message", ex);
593       }
594
595       cmd = msg.getProperty("cmd");
596       String apk = msg.getProperty("id");
597       String info;
598
599       AssociationInfo ainfo;
600       switch (cmd) {
601       case "idle":
602         workLeader.msgs.add( new WorkMessage(cmd, from, apk, null));
603         break;
604       case  "clientCommand":
605         workLeader.msgs.add( new WorkMessage(cmd, from, apk, msg) );
606         break;
607       default:
608         logger.warn("unknown command: " + cmd);
609         break;
610       }
611       synchronized(workers) {
612         workers.notify();
613       }
614
615       } catch (Throwable th) {
616         logger.error("failed receiving message ", th);
617       }
618     }
619   }
```

APPENDIX C - DicomWorker.java

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

```
 1  /*
 2   * DicomWorker.java
 3   * Created by rcoe
 4   * Copyright 2016
 5   */
 6  package edu.marquette.rcoe.server.core;
 7
 8  import java.io.ByteArrayInputStream;
 9  import java.io.File;
10  import java.io.Serializable;
11  import java.net.InetAddress;
12  import java.util.ArrayList;
13  import java.util.concurrent.ConcurrentLinkedQueue;
14  import java.util.List;
15  import java.util.Map;
16  import java.util.Properties;
17  import java.util.TreeMap;
18
19  import org.apache.log4j.Logger;
20
21  import edu.marquette.rcoe.messaging.MessageIPC;
22  import edu.marquette.rcoe.server.core.DicomWorkLeader.WorkMessage;
23  import edu.marquette.rcoe.server.handler.CMoveHandler;
24  import edu.marquette.rcoe.server.handler.CStoreHandler;
25  // [ ... ] Other server interface imports
26
27  /**
28   *  DicomWorker -- worker for work items from the DicomServer
29   */
30  public class DicomWorker extends MessageIPC
31  {
32      private static final Logger logger
33          = Logger.getLogger(DicomWorker.class);
34
35      private DicomWorkerShutdownHook shutdownHook;
36
37      private Map<String, AssociationWorkerInfo> ainfos
38          = new TreeMap<String, AssociationWorkerInfo>();
39      private ConcurrentLinkedQueue<WorkMessage> msgQueue
40          = new ConcurrentLinkedQueue<WorkMessage>();
```

```
41
42      private boolean shutdown = false;
43      private long shutdownTime = 0;
44
45      public enum AssocState {
46          PENDING,                // messages pending
47          IDLE,                   // no messages pending
48          STOPPING,               // processing complete
49          ZOMBIE
50      }
51
52      public static class WorkProcessor extends Thread
53      {
54          public WorkProcessor(String name)
55          {
56              super(name);
57          }
58      }
59
60      public static class AssociationWorkerInfo
61      {
62          String assocPk;
63          String requestor;
64          RCIdentifyingContext ctxt;
65          AssocState state = AssocState.IDLE;
66          WorkProcessor processor;
67          DicomWorker parent;
68
69          ConcurrentLinkedQueue<WorkMessage> msgs
70              = new ConcurrentLinkedQueue<WorkMessage>();
71
72          public AssociationWorkerInfo(DicomWorker worker,
73              String from, String apk)
74          {
75              logger.debug("this: " + this + " parent: " + worker
76                  + " from: " + from + " apk: " + apk);
77              parent = worker;
78              requestor = from;
79              assocPk = apk;
80          }
81
82          public RCIdentifyingContext getContext()
83          {
84              return ctxt;
85          }
86
87          public void setContext(RCIdentifyingContext ctx)
88          {
89              ctxt = ctx;
```

```
 90          }
 91
 92          public void send(String cmd)
 93          {
 94              send(cmd, (String) null);
 95          }
 96
 97          public void send(String cmd, Integer msg)
 98          {
 99              send(cmd, msg.toString());
100          }
101
102          public void send(String cmd, Long msg)
103          {
104              send(cmd, msg.toString());
105          }
106
107          public void send(String cmd, String msg)
108          {
109              StringBuffer sb = new StringBuffer();
110              sb.append("cmd=clientCommand\n");
111              sb.append("id=").append(assocPk).append("\n");
112              sb.append("clientCommand=").append(cmd).append("\n");
113              if (null != msg)
114                  sb.append("clientMessage=").append(msg)
115                      .append("\n");
116              try {
117                  parent.send(requestor, sb.toString());
118              } catch (Exception ex) {
119                  logger.error("cannot send message to " + requestor
120                      + " cmd " + cmd + " message " + msg, ex);
121              }
122              // error recovery ???
123              // propagate parent.send return val ??
124          }
125
126          public void addMessage(WorkMessage msg)
127          {
128              msgs.add(msg);
129          }
130
131          public boolean isPending()
132          {
133              return state.equals(AssocState.PENDING);
134          }
135
136          public void idle()
137          {
138              StringBuffer sb = new StringBuffer();
```

```
139             sb.append("cmd=idle\n");
140             sb.append("id=").append(assocPk).append("\n");
141             try {
142                 parent.send(requestor, sb.toString());
143             } catch (Exception ex) {
144                 logger.error("cannot send message to " + requestor
145                     + " cmd idle message ", ex);
146             }
147         }
148 
149         public boolean stop()
150         {
151             if (state.equals(AssocState.STOPPING)
152                 && processor.getState()
153                     .equals(Thread.State.TERMINATED)) {
154                 try {
155                     processor.join();
156                 } catch (Exception ex) {
157                     logger.error("cleaning up running thread", ex);
158                 }
159                 state = AssocState.ZOMBIE;
160                 return true;
161             }
162             return state.equals(AssocState.ZOMBIE);
163         }
164 
165         public boolean isStopped()
166         {
167             return state.equals(AssocState.ZOMBIE);
168         }
169 
170         public WorkProcessor getProcessor()
171         {
172             return processor;
173         }
174 
175         public void setProcessor(WorkProcessor proc)
176         {
177             processor = proc;
178         }
179 
180         // overriden
181         public void process(WorkMessage msg)
182         {
183         }
184 
185         private void process()
186         {
187             WorkMessage msg;
```

```
188             while (null != (msg = msgs.poll())) {
189                 process(msg);
190             }
191         }
192     }
193
194     private class DicomWorkerShutdownHook
195     {
196         DicomWorker mipc;
197
198         public DicomWorkerShutdownHook(DicomWorker dwl,
199             String appName)
200         {
201             super(appName);
202             mipc = dwl;
203         }
204
205         protected void shutdown()
206         {
207             mipc.shutdown = true;
208         }
209     }
210
211     @Override
212     public void receiveMessage(String from, Serializable obj)
213     {
214         String cmd = (String) obj;
215
216         logger.debug("recvMsg: from: " + from + " msg: " + cmd);
217
218         if (cmd.equals("stop")) {
219             shutdown = true;
220             shutdownTime = System.currentTimeMillis() + 180000;
221             synchronized(ainfos) {
222                 ainfos.notify();
223             }
224             System.exit(0);
225         }
226
227
228         Properties msg = new Properties();
229         try {
230             ByteArrayInputStream bis
231                 = new ByteArrayInputStream(cmd.getBytes());
232             msg.load(bis);
233         } catch (Exception ex) {
234             logger.error("receiving message", ex);
235         }
236
```

```
237             cmd = msg.getProperty("cmd");
238             String apk = msg.getProperty("id");
239 
240             msgQueue.add( new WorkMessage(cmd, from, apk, msg) );
241         }
242 
243         private void processMessage(DicomWorkLeader.WorkMessage wm)
244         {
245             String info;
246             AssociationWorkerInfo ainfo;
247 
248             Properties msg = wm.getMessage();
249 
250             String cmd = wm.getCommand();
251             String from = wm.getValue();
252             String apk = wm.getAppKey();
253 
254             try {
255             switch (cmd) {
256             case "clientCommand":
257                 ainfo = ainfos.get(apk);
258                 logger.debug("lookup " + apk + " node: " + ainfo);
259                 if (null != ainfo) {
260                     logger.debug(String.format("cmd: %s  msg: %s",
261                         msg.getProperty("clientCommand"),
262                         msg.getProperty("clientMessage")));
263                     wm = new DicomWorkLeader.WorkMessage(
264                         msg.getProperty("clientCommand"),
265                         msg.getProperty("clientMessage") );
266                     ainfo.process(wm);
267                 }
268                 break;
269             case "createCStore" :
270                 logger.debug(String.format("CStoreHandler from %s for"
271                     + " %s : %s", from, apk, this));
272                 ainfo = new CStoreHandler.WorkerInfo(this, from, apk);
273                 ainfos.put(apk, ainfo);
274                 break;
275             case "createCMove" :
276                 logger.debug(String.format("CMoveHandler from %s for"
277                     + " %s : %s", from, apk, this));
278                 ainfo = new CMoveHandler.WorkerInfo(this, from, apk);
279                 logger.debug("insert " + apk + " node: " + ainfo);
280                 ainfos.put(apk, ainfo);
281                 break;
282             case "ctxt" :
283                 info = msg.getProperty("value");
284                 ainfo = ainfos.get(apk);
285                 if (null != ainfo) {
```

```
286                 // set ctxt
287                 ainfo.setContext(
288                     RCIdentifyingContext.deserialize(info) );
289             }
290             break;
291         case "finish" :
292             ainfo = ainfos.get(apk);
293             logger.debug("finish: lookup " + apk + " node: "
294                 + ainfo);
295             if (null != ainfo) {
296                 ainfos.remove(apk);
297             }
298             break;
299         default:
300             logger.warn("unknown command: " + cmd);
301             break;
302         }
303         } catch (Throwable th) {
304             logger.error("receiving message", th);
305         }
306     }
307
308     public DicomWorker(String appName)
309     {
310         try {
311             start(null, appName);
312         } catch (Exception ex) {
313             logger.error("cannot create main ipc");
314         }
315     }
316
317     public DicomWorker(DicomWorker parent, String appName)
318     {
319         try {
320             // fork(parent, "dcm", appName, "DICOM");
321             start(null, appName);
322         } catch (Exception ex) {
323             logger.error("cannot create forked ipc");
324         }
325     }
326
327     public static void main(String args[])
328     {
329         String shortName = null;
330
331         System.out.println("Dicom worker starting with args=");
332         for (int i = 0; i < args.length; i++) {
333             if (0 == i)
334                 shortName = args[i];
```

```
335             System.out.println(args[i]);
336         }
337
338         try {
339
340             if (null == shortName) {
341                 shortName = System.getProperty("T");
342                 if (shortName == null)
343                     shortName = "DS99";
344             }
345
346             Thread t = Thread.currentThread();
347             t.setName(shortName);
348             Logger.pushLoggingContext( shortName );
349
350             logger.debug("DicomWorker shortName=" + shortName);
351
352             // DicomWorker mipc = new DicomWorker(shortName);
353             DicomWorker theServer = new DicomWorker(null,
354                 shortName);
355             theServer.setShutdownHook();
356             theServer.run();
357
358         } catch (Exception e) {
359             logger.error("An exception has been caught starting "
360                 + " the DicomWorker", e);
361             System.exit(1);
362         }
363     }
364
365     public void run()
366     {
367         logger.info("starting");
368         Hibernator.getInstance().createSession();
369         DicomConfigCacheManager cache
370             = DicomConfigCacheManager.getInstance();
371
372         try {
373             String localhost
374                 = InetAddress.getLocalHost().getHostName();
375             int count = 10;
376             while (0 < count) {
377                 if (appReady("DicomWorkLeader-" + localhost))
378                     break;
379                 Thread.sleep(500);
380                 count--;
381             }
382
383             send("DicomWorkLeader-" + localhost, "started");
```

```
384          } catch (Exception ex) {
385              logger.error("cannot send start to Leader", ex);
386          }
387
388          while (true) {
389              boolean dostop = true;
390
391              // look for pending messages
392              WorkMessage msg;
393              while (null != (msg = msgQueue.poll())) {
394                  processMessage(msg);
395              }
396
397              for (AssociationWorkerInfo ainfo : ainfos.values()) {
398                  if (ainfo.stop())
399                      ainfos.remove(ainfo.assocPk);
400                  dostop = dostop && ainfo.isStopped();
401              }
402
403              if (dostop && shutdown)
404                  break;
405
406              if (shutdown
407                  && shutdownTime < System.currentTimeMillis())
408                  ;  /// what to do
409
410              synchronized(ainfos) {
411                  try {
412                      ainfos.wait(1000);
413                  } catch (Exception ex) {
414                      // no-op
415                  }
416              }
417          }
418      }
419
420      private void setShutdownHook()
421      {
422          if (shutdownHook == null) {
423              shutdownHook = new DicomWorkerShutdownHook(this,
424                  "DicomWorker");
425              Runtime.getRuntime().addShutdownHook( shutdownHook );
426          }
427      }
428  }
```

APPENDIX D - CMoveProcess.java

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

```
 1  public static class WorkerInfo extends AssociationWorkerInfo
 2  {
 3      private CMoveProcessor cMoveProc;
 4      private CMoveHandler handler;
 5      private boolean abort = false;
 6      private long lastStudyPk = 0;
 7      private Map<Long, StudyInfo> studySops =
 8          new TreeMap<Long, StudyInfo>();
 9
10
11      public WorkerInfo(DicomWorker dwork, String from, String name)
12      {
13          super(dwork, from, name);
14      }
15
16      @Override
17      public void process(DicomWorkLeader.WorkMessage msg)
18      {
19          logger.debug("process: " + msg.getCommand() + " msg: "
20              + msg.getValue());
21          if (abort) {
22              logger.warn("not processing command in ABORT state: "
23                  + msg.getCommand());
24              return;
25          }
26          try {
27          switch (msg.getCommand()) {
28          case  "moveinfo":
29              try {
30                  TMMoveInfo moveInfo = new
31                      TMMoveInfo( msg.getValue() );
32                  handler = new
33                      CMoveHandler(getContext(), moveInfo);
34                  cMoveProc = new CMoveProcessor(this, handler);
35              } catch (Exception ex) {
36                  logger.error("cannot create a moveInfo", ex);
37                  throw new
38                      DicomException("cannot create moveInfo", ex);
39              }
40              break;
```

```
41          case  "aefrom":
42              cMoveProc.setListener( Long.valueOf(msg.getValue()) );
43              break;
44          case  "aedest":
45              cMoveProc.setDestination(
46                  Long.valueOf(msg.getValue()) );
47              break;
48          case  "start":
49              cMoveProc.setContext( getContext() );
50              setProcessor( cMoveProc );
51              break;
52          case  "cmove":
53              cMoveProc.addStudy(
54                  lastStudyPk = Long.valueOf(msg.getValue()) );
55              break;
56          case  "imageCount":
57              studySops.put(lastStudyPk,
58                  new StudyInfo(Integer.valueOf(msg.getValue()),
59                      null));
60              break;
61          case   "noSops":
62              abort = true;
63              break;
64          case  "cmove.start":
65              cMoveProc.setThreadName( msg.getValue() );
66              cMoveProc.start();
67              break;
68          case  "cancel":
69              cMoveProc.cancel();
70              break;
71          case  "finish":
72              cMoveProc.join();
73              idle();
74              break;
75          default:
76              break;
77          }
78          } catch (Throwable th) {
79              logger.error("process: " + msg.getCommand(), th);
80              abort = true;
81              send("sendFinalResponse",
82                  DimseUtil.SUB_OP_COMPLETE_WITH_FAILURES);
83              try {
84                  cMoveProc.join();
85              } catch (Throwable tth) {
86                  logger.error("join failed or interrupted", tth);
87              }
88              idle();
89          }
```

```
 90      }
 91  }
 92
 93  public static class CMoveProcessor
 94      extends DicomWorker.WorkProcessor
 95  {
 96      private TMIdentifyingContext ctxt;
 97      private CMoveHandler cmove;
 98      private WorkerInfo winfo;
 99      private List<Long> studyPks = new ArrayList<Long>();
100      private Long listenPk;
101      private Long destPk;
102      private String thName;
103
104      public CMoveProcessor(WorkerInfo winfo, CMoveHandler cmove)
105      {
106          super("CMoveProcessor");
107          this.cmove = cmove;
108          this.winfo = winfo;
109      }
110
111      public void setContext(TMIdentifyingContext ctxt)
112      {
113          this.ctxt = ctxt.clone();
114      }
115
116      public void addStudy(Long stupk)
117      {
118          studyPks.add(stupk);
119      }
120
121      public CMoveHandler getHandler()
122      {
123          return cmove;
124      }
125
126      public void setDestination(Long pk)
127      {
128          destPk = pk;
129      }
130
131      public void setListener(Long pk)
132      {
133          listenPk = pk;
134      }
135
136      public void setThreadName(String nm)
137      {
138          thName = nm;
```

```
139     }
140
141     public void cancel()
142     {
143         cmove.cancelStatus = true;
144     }
145
146     public void run()
147     {
148         try {
149         TMIdentifyingContext parent =
150             cmove.requestorIC.getParentContext();
151         cmove.requestorIC.setParentContext( parent.clone() );
152         TMContextUtil.
153             setCurrentIdentifyingContext( cmove.requestorIC );
154
155         setName( thName );
156
157         Session session =
158             Hibernator.getInstance().createSession();
159         List<TMStudy> studies = new ArrayList<TMStudy>();
160         TMStudyDAO studyDAO =
161             TMDataAccessObjectFactory.getFactory().getStudyDAO();
162
163         cmove.setListener( listenPk );
164         cmove.setDestination( destPk );
165
166         for (Long spk : studyPks) {
167             try {
168                 TMStudy study = studyDAO.findByPk(spk, false);
169                 if (null != study)
170                     studies.add( study );
171                 if (null == cmove.patient)
172                     cmove.patient = study.getPatient();
173             } catch (Exception ex) {
174                 logger.error("cannot find study by pk: " + spk,
175                     ex);
176             }
177         }
178
179         if (0 != studies.size())
180             cmove.sendStudies(winfo, studies);
181
182         Hibernator.getInstance().closeSession();
183         } catch (Throwable th) {
184             logger.error("processing cmove request", th);
185             winfo.send("sendFinalResponse",
186                 DimseUtil.SUB_OP_COMPLETE_WITH_FAILURES);
187         }
```

```
188       }
189   }
```

APPENDIX E - CStoreProcess.java

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

```
 1  public static class WorkerInfo extends AssociationWorkerInfo
 2  {
 3      private CStoreProcessor cstore;
 4
 5      public WorkerInfo(DicomWorker dwork, String from, String name)
 6      {
 7          super(dwork, from, name);
 8      }
 9
10      @Override
11      public void process(DicomWorkLeader.WorkMessage msg)
12      {
13          switch (msg.getCommand()) {
14          case  "start":
15              CStoreHandler handler =
16                  new CStoreHandler(getContext());
17              cstore = new CStoreProcessor(this, handler);
18              cstore.setContext( getContext() );
19              setProcessor(cstore);
20              cstore.start();
21              break;
22          case  "sop":
23              cstore.add( Long.valueOf(msg.getValue()) );
24              break;
25          case  "assocEnd":
26              cstore.stopRunning();
27              break;
28          default:
29              break;
30          }
31      }
32  }
33
34  public static class CStoreProcessor
35      extends DicomWorker.WorkProcessor
36  {
37      private TMIdentifyingContext ctxt;
38      private ConcurrentLinkedQueue<Long> queue =
39          new ConcurrentLinkedQueue<Long>();
40      private boolean running = true;
```

```
41      private CStoreHandler cstore;
42
43      public CStoreProcessor(WorkerInfo winfo, CStoreHandler cstore)
44      {
45          super("CStoreProcessor");
46          this.cstore = cstore;
47      }
48
49      // processes each sop as it arrives.
50
51      public void run()
52      {
53          TMIdentifyingContext parent = ctxt.getParentContext();
54          ctxt.setParentContext( parent.clone() );
55          TMContextUtil.setCurrentIdentifyingContext(ctxt);
56
57          cstore.startStore();
58
59          Session sess = Hibernator.getInstance().createSession();
60
61          while (running || null != queue.peek()) {
62              Long cpk;
63              while (null != (cpk = queue.poll())) {
64                  logger.debug("processing c-store sop: " + cpk);
65                  try {
66                      cstore.deferredHandleEvent(cpk);
67                  } catch (Throwable th) {
68                      logger.error("cstore deferred failed", th);
69                  }
70              }
71              try {
72                  synchronized (this) {
73                      if (running)
74                          wait();
75                  }
76              } catch (Exception ex) {
77                  logger.error("wait", ex);
78              }
79          }
80          logger.debug("exiting cstore thread, size "
81              + queue.size());
82          cstore.endStore();
83
84          Hibernator.getInstance().closeSession();
85      }
86
87      public void add(Long csoppk)
88      {
89          queue.add(csoppk);
```

```
 90          try {
 91              synchronized (this) {
 92                  notify();
 93              }
 94          } catch (Exception ex) {
 95              logger.error("add notify", ex);
 96          }
 97      }
 98
 99      public void stopRunning()
100      {
101          running = false;
102          try {
103              synchronized (this) {
104                  notify();
105              }
106          } catch (Exception ex) {
107              logger.error("stop notify", ex);
108          }
109      }
110
111      public void setContext(TMIdentifyingContext ctxt)
112      {
113          this.ctxt = ctxt.clone();
114      }
115  }
```