/\*

\* @(#) FloodController.java

\*

\* Copyright 2018 Verizon Technologies. All Rights Reserved.

\*

\* Verizon Technologies owns all right title, and interest in this listing.

\*

\* This listing is confidential and proprietary. Any unauthorized use,

\* reproduction, altering, distribution, or transmission of this listing, or any

\* part thereof, in any form or by any means, electronic or mechanical, is

\* strictly prohibited without a license from Verizon Technologies.

\*/

/\*\*

\* @author Madhu Repala

\* @date 04/13/2018

\*/

package com.vz.vpns.mse.floodcontrol;

import java.util.Calendar;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Timer;

import java.util.TimerTask;

import com.vz.vnms.py.MessageParserHelper;

import com.vz.vnms.util.logging.VNMSLogger;

import com.vz.vnms.util.properties.VerizonProperties;

import com.vz.vpns.epnm.main.EMSNE;

import com.vz.vpns.medsvc.common.MediationConstants;

import com.vz.vpns.mse.handler.MSEMessageDispatcher;

import com.vz.vpns.services.alarm.AlarmConstants;

import com.vz.vpns.services.alarm.AlarmSkulkingFromDB;

public class FloodController {

private static final String EVENT\_FLOOD\_CLEARED = "EVENT FLOOD CLEARED. Number of events received: ";

private static final String EVENT\_FLOOD\_DETECTED = "EVENT FLOOD DETECTED. Number of events received: ";

private static final String MSE = "MSE";

private static final String EVENT\_FLOOD = "EVENT\_FLOOD";

private static final String COMMA = ",";

private static final String NODE\_ID = "\"node-id\": \"";

private VNMSLogger \_logger = null;

private static FloodController floodController;

private HashMap<String, Boolean> floodDetectdTIDsMap = null;

private HashMap<String, Integer> floodCounterTIDsMap = null;

private HashMap<String, Long> startTimeForFloodCheck = null;

private final int FLOOD\_EVENT\_THRESHOLD = VerizonProperties.getIntegerProperty("FloodEventThreshold", 5000);

private final long FLOOD\_CNTRL\_INTERVAL = VerizonProperties.getIntegerProperty("FloodControlIntervalInMins", 5) \* 60 \* 1000;

private MessageParserHelper parser\_helper = null;

private FloodController(){

\_logger = VNMSLogger.getLogger(getClass().getName());

parser\_helper = MessageParserHelper.getInstance(MediationConstants.TYPE\_MSE);

floodDetectdTIDsMap = new HashMap<String, Boolean>();

floodCounterTIDsMap = new HashMap<String, Integer>();

startTimeForFloodCheck = new HashMap<String, Long>();

startClearFloodEventThread();

}

public static synchronized FloodController getInstance() {

if (floodController == null)

{

floodController = new FloodController();

}

return floodController;

}

public boolean isFloodDetected(String message) {

boolean floodDetected = false;

//\_logger.info("entered into isFloodDetected");

try {

int nodeIdStartIndex = message.indexOf(NODE\_ID);

if(nodeIdStartIndex == -1){ return false; }

int nodeIdEndIndex = message.indexOf(COMMA, nodeIdStartIndex);

if(nodeIdEndIndex == -1){ return false; }

String nodeId = message.substring(nodeIdStartIndex+12, nodeIdEndIndex-1);

if(nodeId == null || nodeId.isEmpty() || !parser\_helper.get\_node\_map().containsKey(nodeId)){ return false; }

//\_logger.info("Inside isFloodDetected. TID: "+nodeId);

int numOfAlarms = floodCounterTIDsMap.compute(nodeId, (key, value) -> value == null ? 1 : value + 1);

floodDetected = floodDetectdTIDsMap.compute(nodeId, (key, value) -> value == null ? false : value);

boolean timeElapsed = isFloodTimeIntervalElapsed(nodeId);

if(floodDetected && numOfAlarms < FLOOD\_EVENT\_THRESHOLD && timeElapsed){

clearFloodEvent(nodeId);

floodDetectdTIDsMap.put(nodeId, false);

//\_logger.info("EVENT FLOOD CLEARED and going to reset time and counter "+nodeId);

resetCounter(nodeId);

return false;

}

else if(!floodDetected && numOfAlarms >= FLOOD\_EVENT\_THRESHOLD){

declareFloodEvent(nodeId);

floodDetectdTIDsMap.put(nodeId, true);

//\_logger.info("EVENT FLOOD DETECTED and going to reset time and counter "+nodeId);

resetCounter(nodeId);

return true;

}

if(timeElapsed){

//\_logger.info(" timeElapsed "+timeElapsed);

resetCounter(nodeId);

}

} catch (Exception e) {

e.printStackTrace();

}

//\_logger.debug("Returning from isFloodDetected. floodDetected "+floodDetected);

return floodDetected;

}

private boolean isFloodTimeIntervalElapsed(String nodeId){

long lastFloodCheckTime = getLastFloodCheckTime(nodeId);

return System.currentTimeMillis() - lastFloodCheckTime >= FLOOD\_CNTRL\_INTERVAL;

}

private void clearFloodEvent(String nodeId) {

EMSNE ne = parser\_helper.get\_node\_map().get(nodeId);

long lastFloodCheckTime = getLastFloodCheckTime(nodeId);

String eventDescription = EVENT\_FLOOD\_CLEARED+floodCounterTIDsMap.get(nodeId);

\_logger.info("TID: "+nodeId+EVENT\_FLOOD\_CLEARED+" "+floodCounterTIDsMap.get(nodeId)+" in duration: "+((System.currentTimeMillis() - lastFloodCheckTime))+" milliseconds.");

sendFloodEvent(nodeId, eventDescription, EVENT\_FLOOD, MSE, AlarmConstants.SEVERITY\_CLEAR, ne.getNeType());

}

private void declareFloodEvent(String nodeId) {

EMSNE ne = parser\_helper.get\_node\_map().get(nodeId);

long lastFloodCheckTime = getLastFloodCheckTime(nodeId);

String eventDescription = EVENT\_FLOOD\_DETECTED+floodCounterTIDsMap.get(nodeId);

\_logger.info("TID: "+nodeId+EVENT\_FLOOD\_DETECTED+" "+floodCounterTIDsMap.get(nodeId)+" in duration: "+((System.currentTimeMillis() - lastFloodCheckTime))+" milliseconds.");

sendFloodEvent(nodeId, eventDescription, EVENT\_FLOOD, MSE, AlarmConstants.SEVERITY\_CRITICAL, ne.getNeType());

}

private void resetCounter(String nodeId) {

floodCounterTIDsMap.put(nodeId, 0);

startTimeForFloodCheck.put(nodeId, System.currentTimeMillis());

}

private long getLastFloodCheckTime(String nodeId) {

return startTimeForFloodCheck.compute(nodeId, (key, value) -> value == null ? System.currentTimeMillis() : value);

}

private void sendFloodEvent(String tid, String floodDesc, String condType, String aid, int severity, String \_neType)

{

LinkedList<HashMap> alarms = new LinkedList<HashMap>();

HashMap<String, String> m = new HashMap<String, String>();

m.put(AlarmConstants.ALARM\_FIELD\_EMSIP,"");

m.put(AlarmConstants.ALARM\_FIELD\_CLLI, tid);

m.put(AlarmConstants.ALARM\_FIELD\_ALARMSOURCE, \_neType);

if(\_neType.equalsIgnoreCase("CIENA-6500S")){

m.put(AlarmConstants.ALARM\_FIELD\_EQUIPCLASS, "ULH");

}

else if(\_neType.equalsIgnoreCase("CIENA-5410") || \_neType.equalsIgnoreCase("CIENA-5430")){

m.put(AlarmConstants.ALARM\_FIELD\_EQUIPCLASS, "OTN");

}

else if(\_neType.equalsIgnoreCase("CISCO-NCS4K") || \_neType.equalsIgnoreCase("CISCO-NCS2K") || \_neType.equalsIgnoreCase("CIENA-6500T") || \_neType.equalsIgnoreCase("CISCO-NCS2002") || \_neType.equalsIgnoreCase("CISCO-NCS2006")|| \_neType.equalsIgnoreCase("CIENA-6500S-2S")|| \_neType.equalsIgnoreCase("CIENA-6500S-7S")){

m.put(AlarmConstants.ALARM\_FIELD\_EQUIPCLASS, "NGROADM");

}

else if(\_neType.equalsIgnoreCase("CISCO-NCS6208") || \_neType.equalsIgnoreCase("JUNIPER-MX2020")){

m.put(AlarmConstants.ALARM\_FIELD\_EQUIPCLASS, MSE);

}

m.put(AlarmConstants.NE\_TID, tid);

m.put(AlarmConstants.SUB\_NEADDRESS, "-1");

m.put(AlarmConstants.ALARM\_FIELD\_SEVERITY, Integer.toString(severity));

m.put(AlarmConstants.ALARM\_FIELD\_ATAG, Long.toString(System.currentTimeMillis()));

m.put(AlarmConstants.ALARM\_FIELD\_GENERATIONTIME, String.valueOf(System.currentTimeMillis()));

m.put(AlarmConstants.ALARM\_FIELD\_CONDTYPE, condType);

m.put(AlarmConstants.ALARM\_FIELD\_AID, tid);

floodDesc = floodDesc + '\n';

m.put(AlarmConstants.ALARM\_FIELD\_DESCRIPTION, floodDesc );

m.put(AlarmConstants.ALARM\_FIELD\_RAWMSG,floodDesc);

m.put(AlarmConstants.ALARM\_FIELD\_SERVICEAFFECT, "true");

m.put(AlarmConstants.ALARM\_FIELD\_LOCATION, "");

m.put(AlarmConstants.ALARM\_FIELD\_DIRECTION, "");

alarms.add(m);

MSEMessageDispatcher.getMessageDispatcher().addMsgToQueue(alarms);

}

private boolean isThresholdReached(String nodeId){

return floodCounterTIDsMap.get(nodeId) < FLOOD\_EVENT\_THRESHOLD ? false : true;

}

public void startClearFloodEventThread(){

try {

\_logger.info("Entered into startClearFloodEventThread...");

Timer timer = new Timer();

Calendar date = Calendar.getInstance();

date.add(Calendar.SECOND, (int)(FLOOD\_CNTRL\_INTERVAL/1000/60));

timer.schedule(new ClearFloodEventTask(), date.getTime(), FLOOD\_CNTRL\_INTERVAL);

\_logger.info("Exit from startClearFloodEventThread...");

} catch (Exception e) {

e.printStackTrace();

}

}

private class ClearFloodEventTask extends TimerTask{

@Override

public void run() {

try {

\_logger.info("ClearFloodEventTask started...");

floodDetectdTIDsMap.keySet().stream()

.filter(nodeId -> floodDetectdTIDsMap.get(nodeId))

.filter(nodeId -> isFloodTimeIntervalElapsed(nodeId))

.forEach(nodeId -> {

if(!isThresholdReached(nodeId)){

clearFloodEvent(nodeId);

}

resetCounter(nodeId);

});

\_logger.info("ClearFloodEventTask completed...");

} catch (Exception e) {

e.printStackTrace();

}

}

}

}