

Distributed Database and Tasks System

מגיש: איתי בן שלום

ת.ז. 209479161

אורט ע"ש נעמי שמר גן יבנה

מוגש ל: אנטולי פיימר

אפריל 2018

תוכן עניינים:

רקע 3

מבוא 4

תיאור המוצר 4

אופן פעולת המערכת 6

פעולת המערכת באופן כללי 7

חלוקה למודולים 8

Communicator 11

Database 11

Decider 11

DriverCommunicator 11

WDM-DKOM 11

ControllerConnection 11

Manager 11

# רקע

סביבת פיתוח:

Microsoft Visual Studio 2015 community

שפות תכנות:

Node: C, C++, C#, Python 2.6/2.7

Directory Server: Python 2.6/2.7

דרישות נוספות:

Python packages:

psutil – package for retrieving information on processes and system.

netifaces – network interfaces data extraction package

scapy – packets manipulation package

הערה: פונקציות מסויימות אינן נתמכות במחשבים עם מערכת הפעלה של 64 סיביות (הפונקצייה של DKOM), הסבר נוסף בהמשך.

מבוא

היום עם התקדמות הטכנולוגיה יש דרישה בתחומים רבים לביצוע חישובים שצורכים זמן ומשאבים רבים במיוחד בארגונים טכנולוגיים גדולים. לשם כך ניתן לרכוש מחשבים עם יכולות חישוב גבוהות או שרתים עם זיכרון רב.

המערכת הזו מספקת פתרון לבעיה הזו ללא רכישת מחשבי על ושרתים בעלויות גבוהות אלא על ידי רתימת כוח חישוב וזיכרון לא מנוצל של מחשבים אישיים בחברה (כל מחשב שמשתמשים בו בחברה – מחשבים של מזכירים/ות, מתכנתים/ות, מעצבים/ות וכו').

קהל היעד המרכזי של המערכת הוא תאגידים טכנולוגיים גדולים שצריכים יכולות חישוב גבוהות ו/או זיכרון רב ויש ברשותם כמות נכבדת של מחשבים אישיים.

# תיאור המוצר

שם המוצר:

DDTS – Distributed Database and Tasks System

חלקים:

Node:

זהו החלק שמותקן בכל מחשב שמהווה חלק מהמערכת והוא משמש למספר דברים:

* אחראי על ביצוע המשימות שנשלחות אל המחשב ושליחת התוצאות של הביצוע בחזרה
* אחראי על ביצוע פעולות במסד הנתונים שבמחשב ושליחת התוצאות שמחזירות הפעולות בחזרה.
* אפשור חיבור של Controller ל-Node ובעצם זאת אפשור של שליחת משימות אל Node-ים אחרים וקבלת תוצאות בחזרה.

Directory server:

זהו החלק שנמצא בשרת אחר (לא על מחשב שמותקן עליו Node) והוא אחראי על אפשור התקשורת בין Node-ים שונים. הוא עושה זאת על ידי העברת הכתובת של כל Node ל- Node-ים האחרים.

כלומר המידע שנשלח מ-Node אחד ל-Node אחר לא עובר דרך ה-Directory server, המידע הזה נשלח ישירות לכתובות שכל Node מקבל מה-Directory server.

Controller:

זהו החלק שדרכו ניתן לשלוח משימות למערכת, החלק הזה מתחבר ל-Node שרץ על המחשב הנוכחי ושולח דרכו משימות ושאילתות ל-Node-ים אחרים.

את החלק הזה ניתן להפעיל בשתי דרכים:

* עם ממשק גרפי
* עם ממשק טקסטואלי

הממשק הטקסטואלי נועד בעיקר לשם אפשור של חיבור אפליקציות שונות למערכת בעזרת ניתוב מחדש של ה-stdin וה-stdout.

הסבר נוסף על פעולתו של כל חלק ועל אופן השימוש במערכת יוצג בהמשך.

# אופן פעולת המערכת

## פעולת המערכת באופן כללי

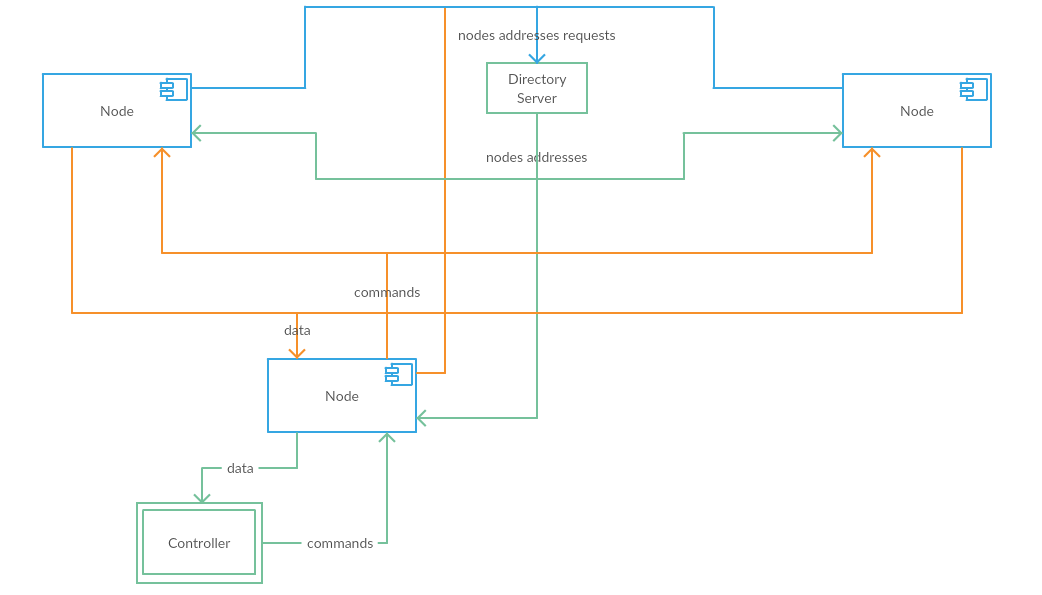
ה- DirectoryServer הוא שרת שמשתף את הכתובות של ה- Node-ים.

ה- Controller מתחבר ל- Node במחשב הנוכחי ושולח דרכו משימות ושאילתות לשאר ה- Node-ים.

ה- Node הוא החלק המרכזי שרץ על כל מחשב ומאפשר ביצוע של משימות והתחברות.

כאשר אין אף חיבור של Controller המערכת רק מתקשרת עם ה- DirectoryServer, אך לאחר התחברות של Controller והתחלת פעולה, ה- Node במחשב הנוכחי שולח משימות ל- Node-ים אחרים, והם מבצעים את המשימות ושולחים בחזרה מידע שהתקבל מביצוע הפעולות. לאחר מכן, ה- Controller מציג את המידע שהתקבל ושולח את המשימות הבאות וכך חוזר חלילה.

להלן שרטוט המציג את אופן הפעולה של המערכת באופן כללי:



## חלוקה למודולים

במערכת הזו ישנם מספר חלקים והם כאמור:

Node, Directory Server ו Controller.

החלקים Directory Server ו Controller בנויים כל אחד כרכיב יחיד והחלק Node בנוי ממספר רכיבים/מודולים והם:

Communication:

מערכת התקשורת של כל Node, היא כתובה ב- Python ובשימוש נרחב ב- Scapy. התקשורת היא בשימוש בפקטות UDP בכדי להימנע מחסימות של התקשורת ב- Firewall.

Database:

זהו המודול שאחראי על בסיס הנתונים בכל Node. כתוב ב- Python ומשתמש ב- SQLite.

Decider:

למודול זה מועבר המידע מכל דבר שקורה בתוכנה והוא מעביר אותו לאן שנדרש. המודול הזה מתווך בין דברים שקורים בNode הנוכחי ומידע שמתקבל "מבחוץ". המודול כתוב ב- Python.

DriverCommunicator:

מטרת המודול הזה הוא לתקשר עם הדרייבר. נכתב ב- C++.

WDM-DKOM:

זהו הדרייבר המדובר, זהו Kernel Driver שמטרתו להחביא את המערכת (או לפחות את החלק המרכזי בה) מהמשתמש כדי שהוא לא יכבה את התוכנה. חלק זה פועל רק עבור מערכת הפעלה מבוססת 32 סיביות. הדרייבר נכתב ב- C.

ControllerConnection:

חלק זה מאפשר את החיבור של Controller במידת הצורך. נכתב ב- Python.

Manager:

חלק זה אחראי על העברת המידע בין כל המודולים והתחלת תהליכים חדשים (וניהולם) במידת הצורך. נכתב ב- C#.

**הערה:** המידע עובר בין המודלים בעזרת ניתוב מחדש של ה- stdin/stdout שה- Manager מבצע. בהסבר המורחב על המודולים, שיוצג בהמשך, לא אציג את אופן העברת המידע ואת שינוי הפורמט של המידע בדרך אלא רק את אופן פעולת המודול, אבל לשם שלמות ההסבר אציין כאן כי עם קבלת המידע, תהליכון בכל מודול שומר את המידע בתור באופן בטוח ותהליכון אחר מוציא את המידע באופן בטוח מהתור ומבצע את תפקידו של המודול.

## Communication

בכדי להפעיל את המודול הזה ה- Manager מריץ את הקובץ communicator.py הוא משנה את הפורמט של המידע ומטפל בהעברתו באופן הדרוש. בשביל התקשורת הוא משתמש במחלקה Communication שהיא מטפלת בתקשורת עם ה- Node-ים האחרים.

המחלקה Communication משתמשת בשני דברים מרכזיים בכדי לתקשר עם ה- Node-ים האחרים והם: Encoder ו- Lowlevelcommunicator עליהם יוסבר בקצרה בהמשך. להלן קוד המחלקה Communicator:

#!/usr/bin/python

**from** time **import** sleep

**import** communicationUtils

**from** encoder **import** Encoder

**from** lowLevelCommunicator **import** LowLevelCommunicator

**class** **Communication:**

**def** \_\_init\_\_**(**self**,** ID**,** communicationKey**,** holePunchingAddr **=** communicationUtils**.**GetDirServerAddr**()):**

self**.**port **=** communicationUtils**.**defaultPort**()**

**while** communicationUtils**.**IsPortTaken**(**self**.**port**):** # take over a port

self**.**port **+=** 1

**if** self**.**port **>=** 2**\*\***16**:**

self**.**port **=** 2000 # after saved ports

self**.**\_\_encoder **=** Encoder**(**communicationKey**)**

**with** open**(**"config.cfg"**,** "w"**)** **as** f**:**

f**.**write**(**self**.**\_\_encoder**.**encrypt**(**str**(**self**.**port**)))** # save the port for controller

self**.**\_\_communicator **=** LowLevelCommunicator**(**self**.**port**,** holePunchingAddr**,** ID**)**

self**.**\_\_communicator**.**start**()**

**def** send**(**self**,** qryOrTsk**,** toId**):** # FIN

self**.**\_\_communicator**.**log**.**write**(**"communication, sending: " **+** qryOrTsk **+** "\n"**)**

self**.**\_\_communicator**.**log**.**flush**()**

self**.**\_\_communicator**.**sendTo**(**self**.**\_\_encoder**.**encrypt**(**qryOrTsk**),** self**.**getAddrById**(**toId**))** # to = (host,port)

**def** getReceivedMessages**(**self**):**

QandT **=** self**.**\_\_communicator**.**getReceivedMessages**()** #getRecievedQuerriesAndTasks()

**if** len**(**QandT**)** **>** 0**:**

self**.**\_\_communicator**.**log**.**write**(**"communication, recieved messages: " **+** str**(**self**.**\_\_encoder**.**decrypt**(**QandT**))** **+** "\n"**)**

self**.**\_\_communicator**.**log**.**flush**()**

**return** self**.**\_\_encoder**.**decrypt**(**QandT**)**

**def** getAddrById**(**self**,** ID**):**

contacts **=** dict**(**self**.**\_\_communicator**.**getContacts**())**

**try:**

**return** contacts**[**str**(**ID**)]**

**except:**

**try:**

**return** self**.**\_\_communicator**.**getAllPossibleContacts**()[**str**(**ID**)]**

**except** Exception **as** e**:**

**raise** e

**def** refreshContacts**(**self**,** timeout **=** 0.2**):**

self**.**\_\_communicator**.**refreshContacts**()**

sleep**(**timeout**)**

**def** getContacts**(**self**):**

**return** **[**c**[**0**]** **for** c **in** self**.**\_\_communicator**.**getContacts**()]** # only id's

המחלקה Encoder:

מחלקה זו אחראית על ההצפנה, זוהי הצפנה יחסית פשוטה בעזרת פעולת xor עם מפתח בגודל 1024 סיביות שלאחריה התוצאה מקודדת בפורמט של Base64. להלן הקוד:

#!/usr/bin/python

**from** base64 **import** b64encode**,** b64decode

**class** **Encoder:** # encryption with xor (list of values) and base64

**def** \_\_init\_\_**(**self**,** key**):**

# key = 1024bit integer

self**.**\_\_keys **=** **[]**

**while** key **!=** 0**:**

self**.**\_\_keys**.**append**(**int**(**key**%**256**))**

key **/=** 256

**def** decrypt**(**self**,** msgs**):** # in format [(ID, msg)...]

**return** msgs # debug

**if(**type**(**msgs**)** **==** type**(**list**())):**

retLst **=** **[]**

**for** item **in** msgs**:**

retLst**.**append**((**item**[**0**],** self**.**\_\_decrypt**(**item**[**1**])))**

**return** retLst

**return** self**.**\_\_decrypt**(**msgs**)** # if single item

**def** \_\_decrypt**(**self**,** msg**):** # type(msg) = string

notBase64 **=** b64decode**(**msg**)**

newMsg **=** ""

i **=** 0

**for** ch **in** notBase64**:**

newMsg **+=** chr**(**ord**(**ch**)** **^** self**.**\_\_keys**[**i **%** len**(**self**.**\_\_keys**)])**

i **+=** 1

**return** newMsg

**def** encrypt**(**self**,** msg**):** # type(msg) = string

#return msg # debug

newMsg **=** ""

i **=** 0

**for** ch **in** msg**:**

newMsg **+=** chr**(**ord**(**ch**)** **^** self**.**\_\_keys**[**i **%** len**(**self**.**\_\_keys**)])**

i **+=** 1

**return** b64encode**(**newMsg**)** # base 64 to disable some spacial characters

אציג כאן את הקוד של LowLevelCommunicator ששולח ומקבל הודעות בפועל בעזרת Scapy אבל לא ארחיב עליו לעומק. באופן כללי החלק הזה מקבל את כל הפקטות שהמחשב מקבל ומסנן רק את אלה שנשלחו אל ה- Node ושומר אותם. תהליכון אחר מסנן את הפקטות לפי המטרה שלהן בעזרת מידע שנמצא בפקטה ומחלץ מהן מידע וברגע שנקראת הפונקציה getReceivedMessages מורכבות הודעות מהמידע שהתקבל קודם לכן וכל ההודעות השלמות (כלומר שהתקבלו במלואן) מוחזרות ברשימה. יש לשים לב שההודעות לא בהכרח יתקבלו באותו הסדר.

ברגע שפקטה מתקבלת נשלחת פקטת תשובה שמעידה על קבלת הפקטה, במידה ו- Node שולח הודעה ולא מקבל פקטת תשובה, הוא שולח את הפקטה שוב, כך אפשר לאמת שכל המידע התקבל. להלן הקוד המדובר:

#!/usr/bin/python

**from** scapy**.**all **import** **\***

**from** thread **import** start\_new\_thread

**from** socket **import** **\***

**from** time **import** sleep**,** time

**import** communicationUtils

**class** **LowLevelCommunicator:** # FIN

**def** \_\_init\_\_**(**self**,** port**,** holePunchingAddr**,** ID**):** # FIN

self**.**\_\_port **=** port

self**.**\_\_recvFromAddr **=** **{}** # store addresses that sended to this node by format {ID:ADDR}

self**.**\_\_sendedAndNotResponded **=** **[]**

self**.**\_\_sniffed **=** **[]**

self**.**\_\_rawMessages **=** **{}**

self**.**\_\_pacTimeout **=** 1 # 1 = default # in seconds

self**.**\_\_isPortProtectionServiceStarted **=** **False**

self**.**\_\_isRecievingThreadStarted **=** **False**

self**.**\_\_isSendedValidationThreadStarted **=** **False**

self**.**\_\_shutdown **=** **False**

self**.**\_\_holePunchingAddr **=** holePunchingAddr

self**.**\_\_dirSerAddr **=** communicationUtils**.**GetDirServerAddr**();**

self**.**\_\_ID **=** ID

self**.**\_\_seq **=** 0

self**.**\_\_recentMessagesIdSeq **=** **[]** # last 20 seconds msgs (id,seq)

self**.**\_\_recentMessagesTimeStamp **=** **[]** # last 500 msgs (id,seq)

self**.**otherNodes **=** **[]** # [(ID,ADDR)...]

self**.**allOtherNodes **=** **{}** # {ID:ADDR, ...}

self**.**EOM **=** "<EOF>" # end of message

self**.**SOM **=** "<SOM>" # start of message

self**.**MAX\_SEQ **=** 2**\*\***16

self**.**log **=** open**(**"lowLevelCom.log"**,** "w"**)**

# self.pcapWriter = PcapWriter("sniffLog" + str(ID) + ".pcap", append=True, sync=True) for debugging

**def** start**(**self**):** # FIN

self**.**startPortProtectionService**()**

self**.**startRecievingThread**()**

self**.**startSendedMsgsValidationThread**()**

**def** startPortProtectionService**(**self**):** # FIN

**if** self**.**\_\_isPortProtectionServiceStarted**:** **return** # already started

start\_new\_thread**(**self**.**\_\_portProtectionService**,** **())**

self**.**\_\_isPortProtectionServiceStarted **=** **True**

self**.**log**.**write**(**"port protection started\n"**)**

self**.**log**.**flush**()**

**def** \_\_portProtectionService**(**self**,** gapBetweenPunches**=**1**):** # FIN

i **=** 0

**while** **not** self**.**\_\_shutdown**:**

# optional: maybe block any connection that attemps to bind or use that port

i **+=** 1

**if** i **==** 2**:**

self**.**requestContacts**()**

i **=** 0

**else:**

self**.**log**.**write**(**"hole punching...\n"**)**

self**.**log**.**flush**()**

holePunchPac **=** IP**(**dst**=**self**.**\_\_holePunchingAddr**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**self**.**\_\_holePunchingAddr**[**1**])/**str**(**self**.**\_\_ID**)**

send**(**holePunchPac**,** verbose**=False)**

sleep**(**gapBetweenPunches**)**

**def** startSendedMsgsValidationThread**(**self**):** #FIN

**if** self**.**\_\_isSendedValidationThreadStarted**:** **return** # already started

start\_new\_thread**(**self**.**\_\_sendedValidationThread**,** **())**

self**.**\_\_isSendedValidationThreadStarted **=** **True**

self**.**log**.**write**(**"sended messages validation started\n"**)**

self**.**log**.**flush**()**

**def** startRecievingThread**(**self**):** # FIN

**if** self**.**\_\_isRecievingThreadStarted**:** **return** # already started

start\_new\_thread**(**self**.**\_\_recievingThread**,** **())**

self**.**\_\_isRecievingThreadStarted **=** **True**

self**.**log**.**write**(**"recieving started\n"**)**

self**.**log**.**flush**()**

**def** \_\_recievingThread**(**self**):** # FIN

start\_new\_thread**(**self**.**\_\_sniffingThread**,())**

**while** **not** self**.**\_\_shutdown**:**

sniffedCpy **=** self**.**\_\_sniffed**[**0**:]**

self**.**\_\_sniffed **=** **[]**

**for** pac **in** sniffedCpy**:** # self.\_\_sniffed:

# check if the packet is valid

origChecksum **=** pac**[**UDP**].**chksum

**del** pac**[**UDP**].**chksum

pac **=** pac**.**\_\_class\_\_**(**str**(**pac**))** # dump to string and rebuild the packet

**if** origChecksum **!=** pac**[**UDP**].**chksum**:** # packet is damaged

**continue** # ignore this packet

# then extract data to recData

**try:**

splt **=** pac**[**Raw**].**load**.**split**(**","**)**

self**.**\_\_recvFromAddr**[**splt**[**0**]]** **=** **(**pac**[**IP**].**src**,** pac**[**UDP**].**sport**)** # add to addresses by ID's dictionary

**if** splt**[**2**]** **==** "r"**:** # recieved response => remove from self.\_\_sendedAndNotResponded

# print "r received" # for debug

**for** i **in** self**.**\_\_sendedAndNotResponded**:**

**if** i**[**0**]** **==** int**(**splt**[**3**]):** # remove

# print "removed: " + str(i) # for debug

self**.**\_\_sendedAndNotResponded**.**remove**(**i**)**

**break**

**elif** splt**[**0**]** **==** "0"**:** # ID == 0 => its a directory server

# its the node connections data... use it, save it to list or ID

**for** i **in** eval**(**","**.**join**(**splt**[**3**:])):** # list

**if** i **not** **in** self**.**otherNodes**:**

self**.**otherNodes**.**append**(**i**)** # (ID, ADDR), ADDR = (IP, PORT)

self**.**allOtherNodes**.**update**(**dict**(**self**.**otherNodes**))**

**elif** splt**[**2**]** **==** "m"**:** # msg

**if** **(**splt**[**0**],** splt**[**1**])** **not** **in** self**.**\_\_recentMessagesIdSeq**:** # not recieved yet

self**.**\_\_recentMessagesIdSeq**.**append**((**splt**[**0**],** splt**[**1**]))**

self**.**\_\_recentMessagesTimeStamp**.**append**(**time**())**

self**.**\_\_rawMessages**[(**int**(**splt**[**0**]),** int**(**splt**[**1**]))]** **=** ","**.**join**(**splt**[**3**:])** # dict[ID, Seq] = data

self**.**\_\_sendRecievedResponse**(**splt**)**

**else:** # unimplemented packet type

**raise** Exception**(**"not implemented packet type (LLC reached to else statment)"**)**

**except** Exception **as** e**:**

**raise** Exception**(**"illegal packet, exception: " **+** str**(**e**))**

# remove messages that older than 20 seconds, by binary search -> fast

currTime **=** time**()** # faster

start **=** 0

end **=** len**(**self**.**\_\_recentMessagesTimeStamp**)** **-** 1

mid **=** 0

**while** start **<=** end**:**

mid **=** **(**start **+** end**)** **/** 2

**if** currTime **-** self**.**\_\_recentMessagesTimeStamp**[**mid**]** **>** 10**:**

start **=** mid **+** 1

**else:**

end **=** mid **-** 1

self**.**\_\_recentMessagesIdSeq **=** self**.**\_\_recentMessagesIdSeq**[**mid**:]**

self**.**\_\_recentMessagesTimeStamp **=** self**.**\_\_recentMessagesTimeStamp**[**mid**:]**

**if** **(**len**(**self**.**\_\_sniffed**)** **==** 0**):**

sleep**(**0.1**)**

**def** \_\_sniffingThread**(**self**):** # FIN

myIntIps **=** communicationUtils**.**GetMachineInternalIps**()**

myIntIps **=** **[**ip**.**encode**(**'ascii'**,** 'ignore'**)** **for** ip **in** myIntIps**]** # from unicode to str

pacFilter **=** **lambda** p**:** p**.**haslayer**(**UDP**)** **and** p**[**UDP**].**dport **==** self**.**\_\_port **and** p**.**haslayer**(**IP**)** **and** p**[**IP**].**dst **in** myIntIps

stopFilter **=** **lambda** x**:** self**.**\_\_shutdown

sniff**(**lfilter**=**pacFilter**,** prn**=**self**.**\_\_appendSniffedPac**,** stop\_filter**=**stopFilter**)**

**def** \_\_appendSniffedPac**(**self**,** p**):** # FIN

#print "appended: ",

#p.summary()

self**.**\_\_sniffed**.**append**(**p**)**

**def** sendTo**(**self**,** msg**,** to**):** # FIN

**if** type**(**to**)** **==** type**(**list**()):**

**for** node **in** to**:**

self**.**\_\_sendTo**(**msg**,** Node**)**

**else:** # to single node

self**.**\_\_sendTo**(**msg**,** to**)**

# returning recieved messages in format: [(ID1, msg1), (ID2, msg2), (ID3, msg3)...]

**def** getReceivedMessages**(**self**):** # FIN

**if** len**(**self**.**\_\_rawMessages**)** **==** 0**:** # if raw messages empty exit

**return** **[]**

messages **=** **[]**

sortedRawMessagesKeys **=** list**(**sorted**(**self**.**\_\_rawMessages**.**keys**()))** # sort by (primaryElement, secondaryElement)

isMsgValid **=** **True**

lastSeqId **=** sortedRawMessagesKeys**[**0**]** # first key[0] = first key id

lastKeys **=** **[**sortedRawMessagesKeys**[**0**]]** # initialize with the first key

msgParts **=** **[]** #[self.\_\_rawMessages[sortedRawMessagesKeys[0]]]

**for** i **in** sortedRawMessagesKeys**:** #[1:]:

#print "i: ", type(i), " last: ", type(lastSeqId)

**if** self**.**\_\_rawMessages**[**i**]** **==** self**.**SOM**:**

isMsgValid **=** **True** # reinitialize for the msg

lastKeys **=** **[]** # reinitialize for the msg

msgParts **=** **[]** # reinitialize for the msg

**elif** isMsgValid **and** lastSeqId**[**0**]** **==** i**[**0**]** **and** **(**lastSeqId**[**1**]** **+** 1**)** **%** self**.**MAX\_SEQ **==** i**[**1**]** **and** self**.**\_\_rawMessages**[**i**]** **==** self**.**EOM**:**

lastKeys**.**append**(**i**)**

**if** isMsgValid**:**

messages**.**append**((**i**[**0**],** ""**.**join**(**msgParts**)))** # ID, msg

**for** key **in** lastKeys**:** # remove "used" raw messages

self**.**\_\_rawMessages**.**pop**(**key**)**

isMsgValid **=** **False**

**elif** isMsgValid **and** lastSeqId**[**0**]** **==** i**[**0**]** **and** **(**lastSeqId**[**1**]** **+** 1**)** **%** self**.**MAX\_SEQ **==** i**[**1**]:** # same id and next seq

msgParts**.**append**(**self**.**\_\_rawMessages**[**i**])**

lastKeys**.**append**(**i**)**

**else:** # lastSeqId[0] == i[0] and lastSeqId[1] + 1 != i[1]

isMsgValid **=** **False**

lastSeqId **=** i

**return** messages # in the format (ID, msg)

**def** \_\_sendedValidationThread**(**self**):** # FIN

"""

re-send packets that havn't recieved or recieved incorrectly

"""

**while** **not** self**.**\_\_shutdown**:**

**for** i **in** self**.**\_\_sendedAndNotResponded**:** # i = (seq, time(), pac)

**if** time**()** **-** i**[**1**]** **>=** self**.**\_\_pacTimeout**:**

self**.**log**.**write**(**"re sended: " **+** str**(**i**[**2**])** **+** "\n"**)**

self**.**log**.**flush**()**

send**(**i**[**2**],** verbose **=** **False)**

i**[**1**]** **=** time**()**

# print "re-sent: " + str(i) # for debug

sleep**(**0.2**)**

**def** \_\_sendTo**(**self**,** msg**,** to**):** # FIN

# to = (ip, port)

self**.**log**.**write**(**"sending: " **+** msg **+** "\n"**)**

self**.**log**.**flush**()**

msgIndicatorLen **=** len**(**"m,"**)** # indicates thats a message

maxIdAndSeqLen **=** len**(**str**(**2**\*\***48**))** **+** len**(**str**(**self**.**MAX\_SEQ**))**

maxPortLength **=** len**(**str**(**2**\*\***16**))**

dataPerPac **=** **(**508 **-** maxIdAndSeqLen **-** maxPortLength **-** msgIndicatorLen**)** # 508 = for sure safe length

numToSend **=** int**(**len**(**msg**)** **/** dataPerPac**)** **+** 1 # round down + 1 =~ round up

toSend **=** **[]**

toSend**.**append**(**IP**(**dst**=**to**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**to**[**1**])/(**str**(**self**.**\_\_ID**)** **+** "," **+** str**(**self**.**\_\_seq**)** **+** ",m," **+** self**.**SOM**))** # start message

self**.**\_\_sendedAndNotResponded**.**append**([**self**.**\_\_seq**,** time**(),** toSend**[-**1**]])**

self**.**\_\_incSeq**()**

**for** i **in** xrange**(**numToSend **-** 1**):** # split data to packets with max len of "dataPerPac"

# ID,Seq,messageIndicator,data

# the response address would be found by the ID

raw **=** str**(**self**.**\_\_ID**)** **+** "," **+** str**(**self**.**\_\_seq**)** **+** ",m," **+** msg**[**i**\***dataPerPac**:(**i**+**1**)\***dataPerPac**]**

toSend**.**append**(**IP**(**dst**=**to**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**to**[**1**])/**raw**)**

self**.**\_\_sendedAndNotResponded**.**append**([**self**.**\_\_seq**,** time**(),** toSend**[-**1**]])**

self**.**\_\_incSeq**()**

toSend**.**append**(**IP**(**dst**=**to**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**to**[**1**])/**

**(**str**(**self**.**\_\_ID**)** **+** "," **+** str**(**self**.**\_\_seq**)** **+** ",m," **+** msg**[(**numToSend**-**1**)\***dataPerPac**:]))** # last data packet

self**.**\_\_sendedAndNotResponded**.**append**([**self**.**\_\_seq**,** time**(),** toSend**[-**1**]])**

self**.**\_\_incSeq**()**

toSend**.**append**(**IP**(**dst**=**to**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**to**[**1**])/(**str**(**self**.**\_\_ID**)** **+** "," **+** str**(**self**.**\_\_seq**)** **+** ",m," **+** self**.**EOM**))** # end message

self**.**\_\_sendedAndNotResponded**.**append**([**self**.**\_\_seq**,** time**(),** toSend**[-**1**]])**

self**.**\_\_incSeq**()**

#print "tosend: " + str(toSend) # fir debugging

send**(**toSend**,** verbose **=** **False)**

**def** \_\_sendRecievedResponse**(**self**,** pacDataSplt**):** # FIN

self**.**log**.**write**(**"sended recieved response to: " **+** str**(**pacDataSplt**)** **+** "\n"**)**

self**.**log**.**flush**()**

splt **=** pacDataSplt # pacData.split(",")

**try:**

to **=** self**.**\_\_recvFromAddr**[**splt**[**0**]]** # self.getAddrById(int(splt[0]))

**except** Exception **as** e**:**

**print** "send recv response err: " **+** str**(**e**)**

# ID,Seq,recvResponseIndicator,recPacSeq

raw **=** str**(**self**.**\_\_ID**)** **+** "," **+** str**(**self**.**\_\_seq**)** **+** ",r," **+** splt**[**1**]** #r=recieved , splt[1] = other node's seq

# print "sending resp: " + raw # for debugging

send**(**IP**(**dst**=**to**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**to**[**1**])/**raw**,** verbose **=** **False)**

**def** \_\_incSeq**(**self**):** # FIN

""" icrease sequence indicator """

self**.**\_\_seq **+=** 1

**if** self**.**\_\_seq **>=** self**.**MAX\_SEQ**:** self**.**\_\_seq **=** 0 # reset seq # like % but faster

**def** shutdown**(**self**):** # FIN

self**.**\_\_shutdown **=** **True**

sleep**(**1**)**

**def** refreshContacts**(**self**):**

self**.**log**.**write**(**"refreshed contacts\n"**)**

self**.**log**.**flush**()**

self**.**otherNodes **=** **[]**

self**.**requestContacts**()**

**def** requestContacts**(**self**):**

self**.**log**.**write**(**"requested for contacts\n"**)**

self**.**log**.**flush**()**

requestPac **=** IP**(**dst**=**self**.**\_\_dirSerAddr**[**0**])/**UDP**(**sport**=**self**.**\_\_port**,** dport**=**self**.**\_\_dirSerAddr**[**1**])/(**">" **+** str**(**self**.**\_\_ID**))**

send**(**requestPac**,** verbose**=False)**

**def** getContacts**(**self**):**

**return** self**.**otherNodes**[**0**:]** # copy

**def** getAllPossibleContacts**(**self**):** # return {ID:ADDR,...} for all nodes from all times

ret **=** self**.**\_\_recvFromAddr**.**copy**()**

ret**.**update**(**self**.**allOtherNodes**)**

**return** ret

## 

## Database

## Decider

## DriverCommunicator

## WDM-DKOM

## ControllerConnection

## Manager