DIAS-GDELT Documentation v0.2

2018-09-19 - D. Vangelis

- ☑ Create DIAS-GDELT Documentation v0.2
- ☑ Should keep track of the terminology -> DIAS Documentation Glossary
- ☑ In section Getting Started
    - ☑ Outline of steps
    - ☑ number the steps: step 1, step 2, etc
- ☑ Mock device
    - ☑ Not clear that there is 1 mock device is per DIAS
    - ☑ Can we explain this better
- ☑ Big picture of the deployment
    - See Vangelis' diagram
    - add the steps
- ☑ Intro
    - Purpose is not data analytics of GDELT
    - ☑ Purpose is to test DIAS in a real-time setup
- ☑ See Vangelis notes -> DIAS-GDELT Documentation v0.1 Review

2018-09-12 (Last update)

Intro

Have you ever wondered how many news event are produced every day in Europe?

To demonstrate the real-world applicability and feasibility to perform real-time distributed data analytics, we connected DIAS into GDELT to measure the number of news events in Europe, updated every 15 minutes. We believe this is a good proxy for inferring the overall activity level of the continent. Indeed, if more news items are being produced, then it is likely that more physical events are taking place. And vice-versa.

The following application connects DIAS into GDELT. GDELT is the largest publicly available collection of near real-time news feed in the World, connecting to hundreds of newsfeeds and delivering updates every 15-minutes.

This DIAS application covers the 28 countries in Europe. Each country is represented by a *DIAS node*, so there are 28 *DIAS nodes* in the network that are cooperating to provide analytics about the network. Each *DIAS node* aggregates the number of news items produced by itself as well as the 27 other nodes in the network.

Application Setup

At a high level, data for the 28 European countries is retrieved from GDELT every 15 minutes and published to the 28 *Mock Devices.* The latter are required to compute *selected* and *possible states* from the number of news events generated in the GDELT stream. The *selected* and *possible states* are then passed along to the *DIAS nodes* in the DIAS Network, each *DIAS node* being paired with Mock Device. The real-time aggregation takes place inside the DIAS Network.

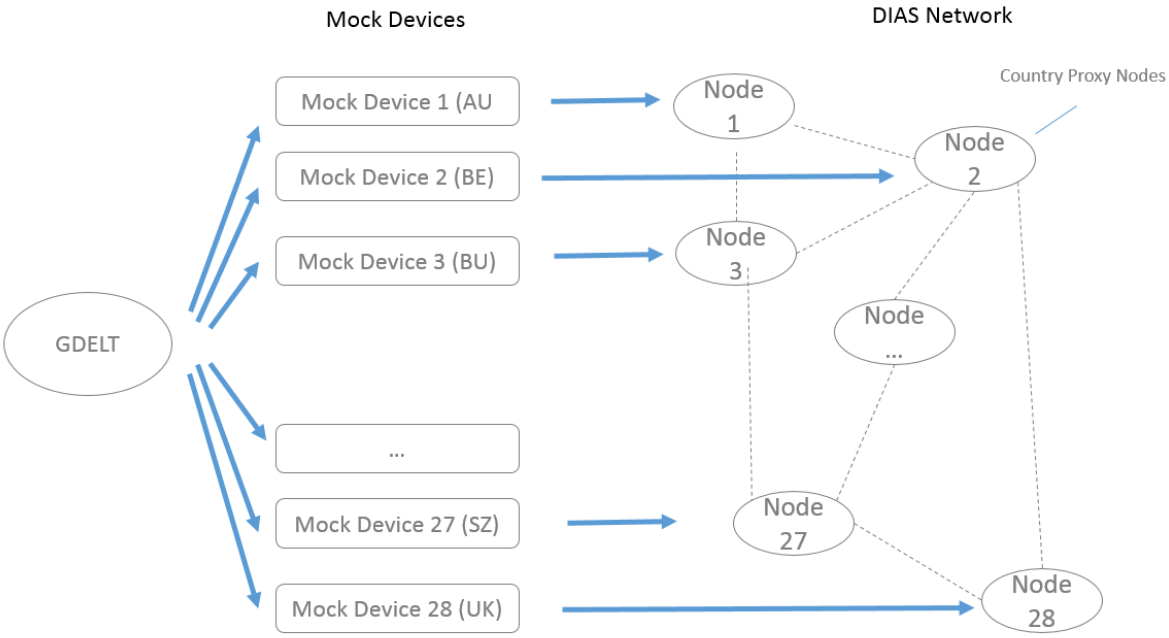


Figure {}. Overview of the DIAS-GDELT application.

Screenshots

Here is a screenshot of DIAS-GDELT in action. You can observe the fluctuations in the numbers of news events generated per 15-minute window.

For example, the number 200 (on the y-axis) indicates that 200 news events were generated concerning European countries over the relevant 15-minute window.

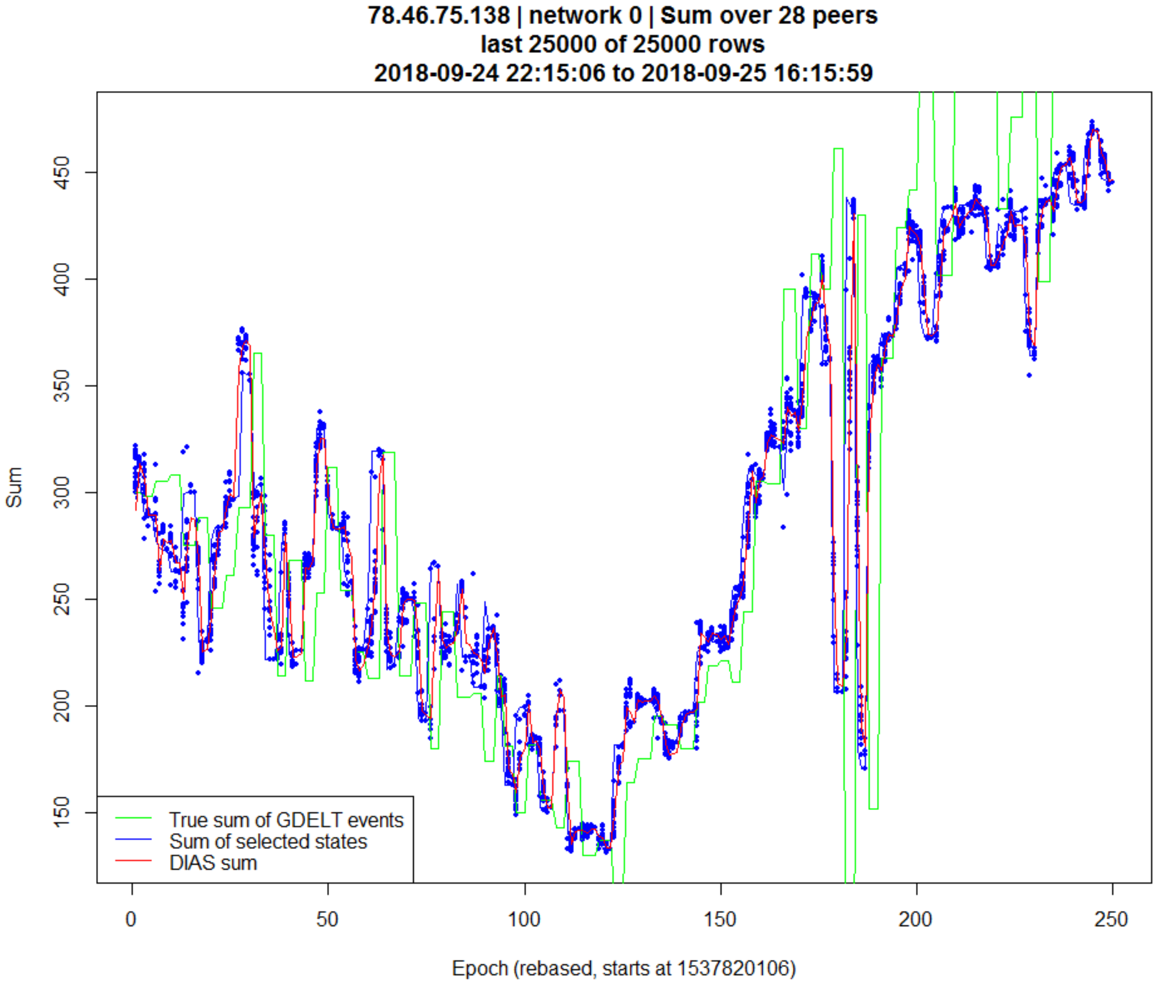


Figure {}. DIAS-GDELT aggregation process. Each blue dot represents the estimated sum fromeacha DIAS node. The green line shows the true sum of news events from GDELT. The blue line shows the true sum of selected states, whilst the red line shows the sum of the DIAS aggregates.

You can also view the same aggregation process on our website, http://dias-net.org/gdelt/

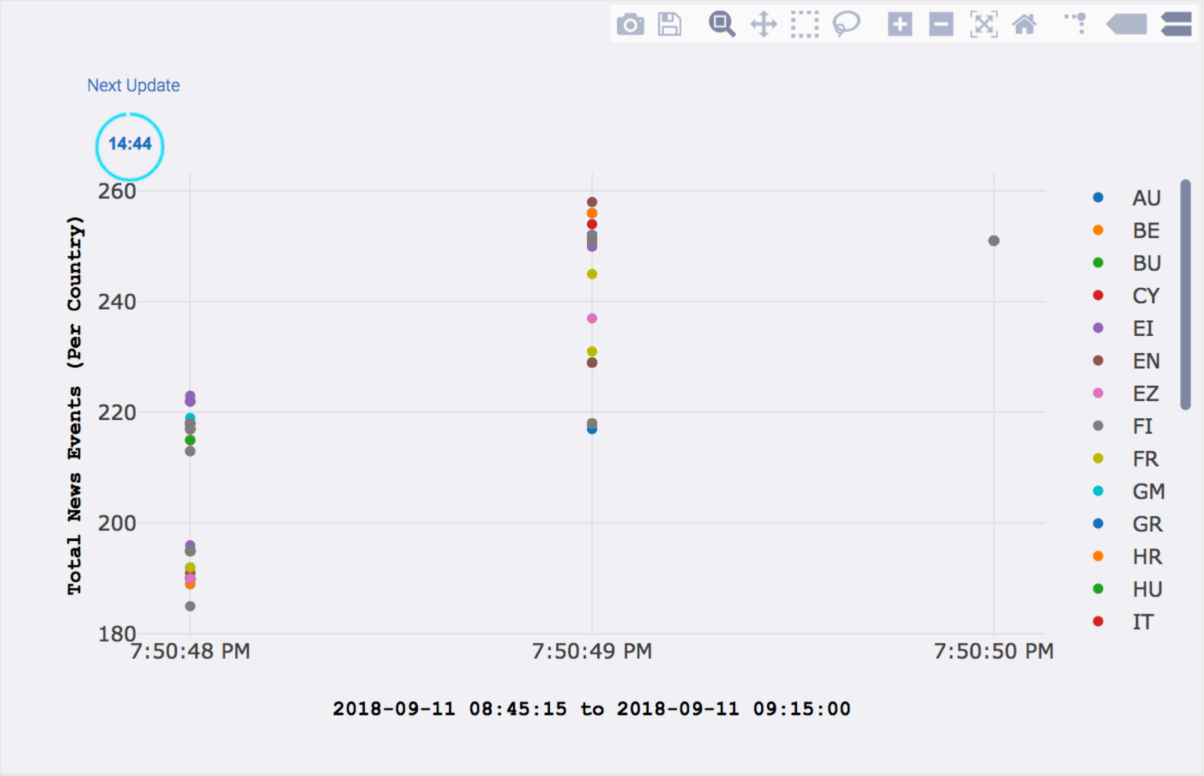


Figure {}. DIAS-GDELT aggregation process as viewed in the webpage http://dias-net.org/gdelt/

Getting Started

This section is organised into the following sub-sections:

1. Installation
2. Start Commands for launching:
   2.1 Logging Daemon
   2.2 Protopeer Boostrap Server and DIAS Gateway Server

   2.3 DIAS Nodes
   2.4 GDELT Subscription
   2.5 Mock Devices

1.Installation

See DIAS Installation Guide (-> DIAS Installation Guide) for pre-requisites. Assuming the DIAS application has been successfully installed, you only need to install the DIAS-GDELT application to make this demo work.

```
git clone https://github.com/epournaras/DIAS-GDELT.git
```

2.Start Commands

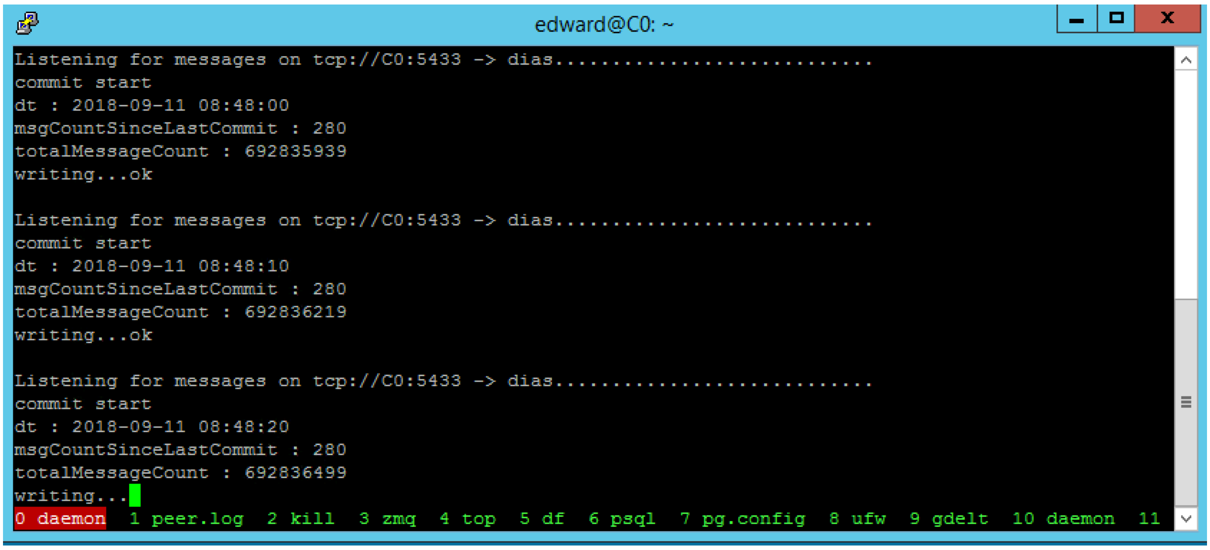
First, as with all DIAS applications, we need to start the persistence daemon, so that real-time data can be efficiently stored to a database whilst reducing persistence work to a minimum within each *DIAS node*.

```
cd DIAS-Logging-System

./start.daemon.sh
```

2.1 Logging Daemon

The logging daemon is launched in a separate screen (that is, the GNU *screen* utility), as shown just below.



```
Listening for messages on tcp://C0:5433 -> dias...........................
commit start
dt : 2018-09-11 08:48:00
msgCountSinceLastCommit : 280
totalMessageCount : 692835939
writing...ok

Listening for messages on tcp://C0:5433 -> dias...........................
commit start
dt : 2018-09-11 08:48:10
msgCountSinceLastCommit : 280
totalMessageCount : 692836219
writing...ok

Listening for messages on tcp://C0:5433 -> dias...........................
commit start
dt : 2018-09-11 08:48:20
msgCountSinceLastCommit : 280
totalMessageCount : 692836499
writing...
0 daemon  1 peer.log  2 kill  3 zmq  4 top  5 df  6 psql  7 pg.config  8 ufw  9 gdelt  10 daemon  11
```

Figure {}. DIAS-Logging Daemon receiving messages and persisting them to a PostgreSQL database for offline and online system logging analysis

2.2 Protopeer Bootstrap Server and DIAS Gateway Server

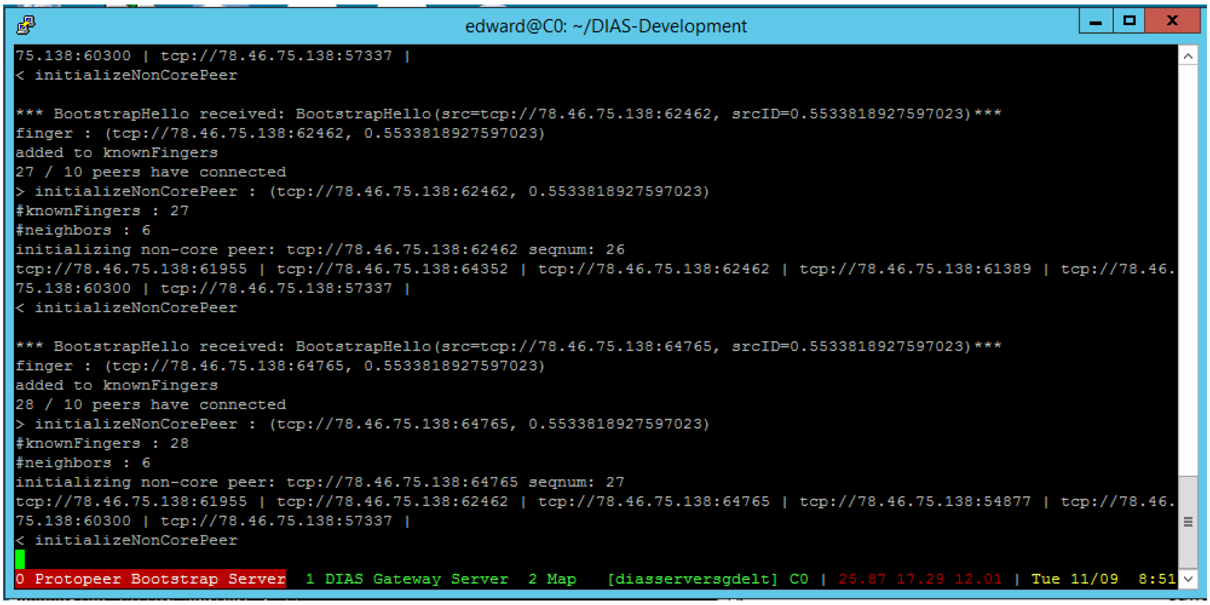
Next, we start a *Protopeer Bootstrap Server* and *DIAS Gateway Server*. The bootstrap server allows *DIAS nodes* to discover each other at the beginning when they initially connect with the bootstrap server. . The gateway, on the other hand, allows *mock devices* to be assigned to *DIAS nodes*.

Note that we are using the *gdelt* deployment, which contains suitable configuration for running DIAS-GDELT. You can adjust the configuration by editing the file deployments/gdelt/dias.conf. The DIAS Installation Guide (-> DIAS Installation Guide) explains in further detail what can be configured.

```
cd DIAS-Development

./start.servers.sh deployments/gdelt
```

After starting the *Protopeer Bootstrap Server* and *DIAS Gateway Server*, you should see something like this.



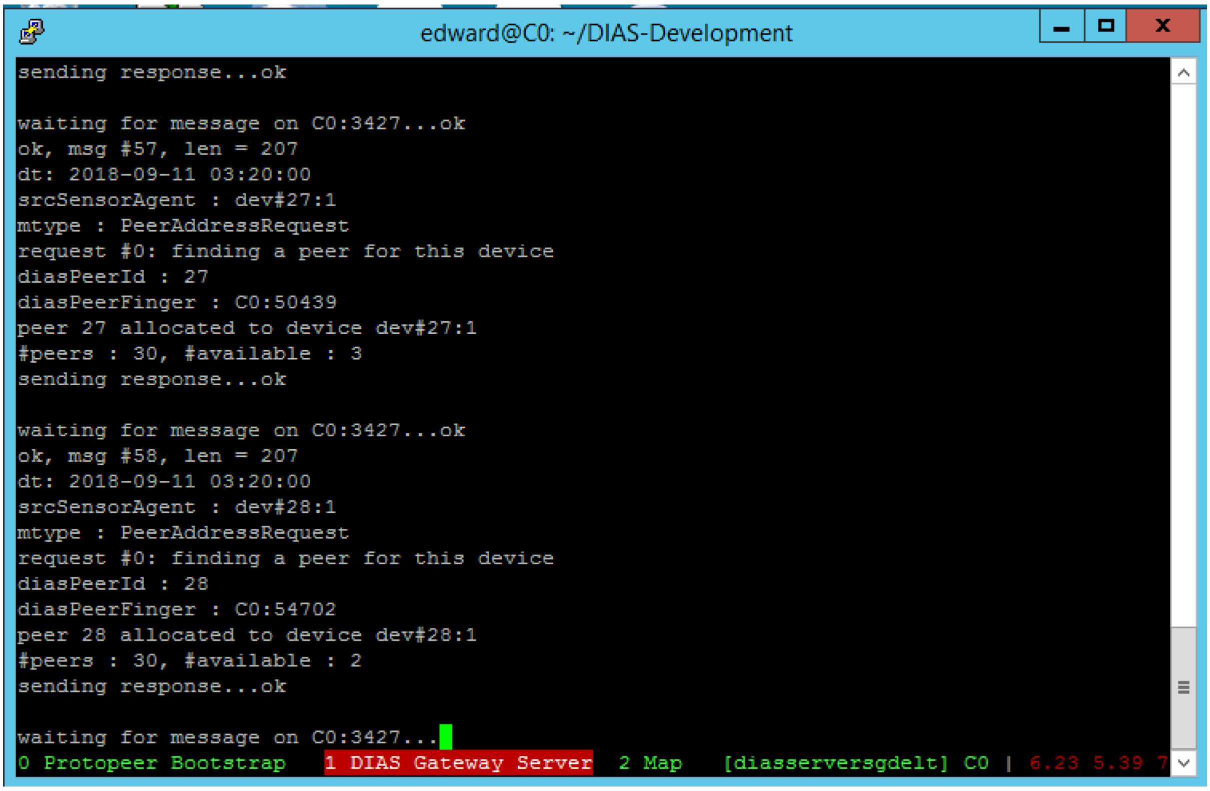
```
75.138:60300 | tcp://78.46.75.138:57337 |
< initializeNonCorePeer

*** BootstrapHello received: BootstrapHello(src=tcp://78.46.75.138:62462, srcID=0.5533818927597023)***
finger : (tcp://78.46.75.138:62462, 0.5533818927597023)
added to knownFingers
27 / 10 peers have connected
> initializeNonCorePeer : (tcp://78.46.75.138:62462, 0.5533818927597023)
#knownFingers : 27
#neighbors : 6
initializing non-core peer: tcp://78.46.75.138:62462 seqnum: 26
tcp://78.46.75.138:61955 | tcp://78.46.75.138:64352 | tcp://78.46.75.138:62462 | tcp://78.46.75.138:61389 | tcp://78.46.
75.138:60300 | tcp://78.46.75.138:57337 |
< initializeNonCorePeer

*** BootstrapHello received: BootstrapHello(src=tcp://78.46.75.138:64765, srcID=0.5533818927597023)***
finger : (tcp://78.46.75.138:64765, 0.5533818927597023)
added to knownFingers
28 / 10 peers have connected
> initializeNonCorePeer : (tcp://78.46.75.138:64765, 0.5533818927597023)
#knownFingers : 28
#neighbors : 6
initializing non-core peer: tcp://78.46.75.138:64765 seqnum: 27
tcp://78.46.75.138:61955 | tcp://78.46.75.138:62462 | tcp://78.46.75.138:64765 | tcp://78.46.75.138:54877 | tcp://78.46.
75.138:60300 | tcp://78.46.75.138:57337 |
< initializeNonCorePeer

0 Protopeer Bootstrap Server  1 DIAS Gateway Server  2 Map   [diasserversgdelt] C0 | 25.87 17.29 12.01 | Tue 11/09  8:51
```

Figure {}. The *Protopeer Boostrap Server*, that allows the *DIAS nodes* to initially connect with each other.



```
sending response...ok

waiting for message on C0:3427...ok
ok, msg #57, len = 207
dt: 2018-09-11 03:20:00
srcSensorAgent : dev#27:1
mtype : PeerAddressRequest
request #0: finding a peer for this device
diasPeerId : 27
diasPeerFinger : C0:50439
peer 27 allocated to device dev#27:1
#peers : 30, #available : 3
sending response...ok

waiting for message on C0:3427...ok
ok, msg #58, len = 207
dt: 2018-09-11 03:20:00
srcSensorAgent : dev#28:1
mtype : PeerAddressRequest
request #0: finding a peer for this device
diasPeerId : 28
diasPeerFinger : C0:54702
peer 28 allocated to device dev#28:1
#peers : 30, #available : 2
sending response...ok

waiting for message on C0:3427...
0 Protopeer Bootstrap   1 DIAS Gateway Server  2 Map   [diasserversgdelt] C0 | 6.23 5.39 7
```

Figure {}. The *DIAS Gateway Server,* that binds devices such as mobiles phones to *DIAS nodes*

2.3 DIAS Nodes

Now that we have the Protopeer Bootstrap Server and DIAS Gateway Server running, we can launch the *DIAS nodes*, also known as aggregation peers. As the DIAS-GDELT simulation covers the European countries only, we will launch 28 *DIAS nodes*, one per European country. Note that, in this application, there is no need for carrier nodes to enable the join+leave process since countries don't leave the planet (under normal conditions).

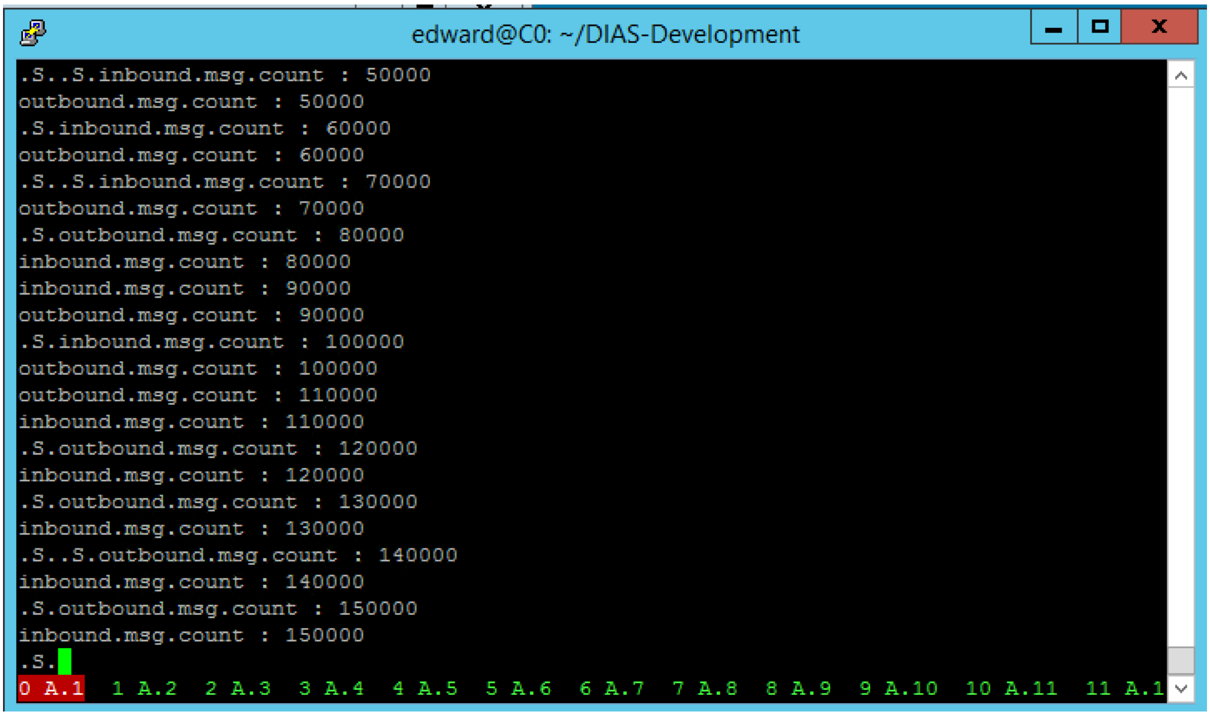
```
cd DIAS-Development

./start.aggregation.peers.sh deployments/gdelt 28 1 1
```

You should a screen that looks like below, with one *DIAS node* per screen tab.

To move to a different *DIAS node*, use the command ctrl+A then *n* or *p* to cycle forwards or backwards amongst the tabs of the screen. Since we have launched *DIAS nodes* that perform aggregation, they were given the prefix *A* (for aggregation), and then suffixed with the id of the *DIAS node*.

```
# moving to a different peer within a screen window
ctrl+A then n or p
```



```
.S..S.inbound.msg.count : 50000
outbound.msg.count : 50000
.S.inbound.msg.count : 60000
outbound.msg.count : 60000
.S..S.inbound.msg.count : 70000
outbound.msg.count : 70000
.S.outbound.msg.count : 80000
inbound.msg.count : 80000
inbound.msg.count : 90000
outbound.msg.count : 90000
.S.inbound.msg.count : 100000
outbound.msg.count : 100000
outbound.msg.count : 110000
inbound.msg.count : 110000
.S.outbound.msg.count : 120000
inbound.msg.count : 120000
.S.outbound.msg.count : 130000
inbound.msg.count : 130000
.S..S.outbound.msg.count : 140000
inbound.msg.count : 140000
.S.outbound.msg.count : 150000
inbound.msg.count : 150000
.S.
0 A.1  1 A.2  2 A.3  3 A.4  4 A.5  5 A.6  6 A.7  7 A.8  8 A.9  9 A.10  10 A.11  11 A.1
```

Figure {}. The *DIAS nodes* , that will perform the aggregation. Each *DIAS node* is launched in a separate tab of the *screen* utility

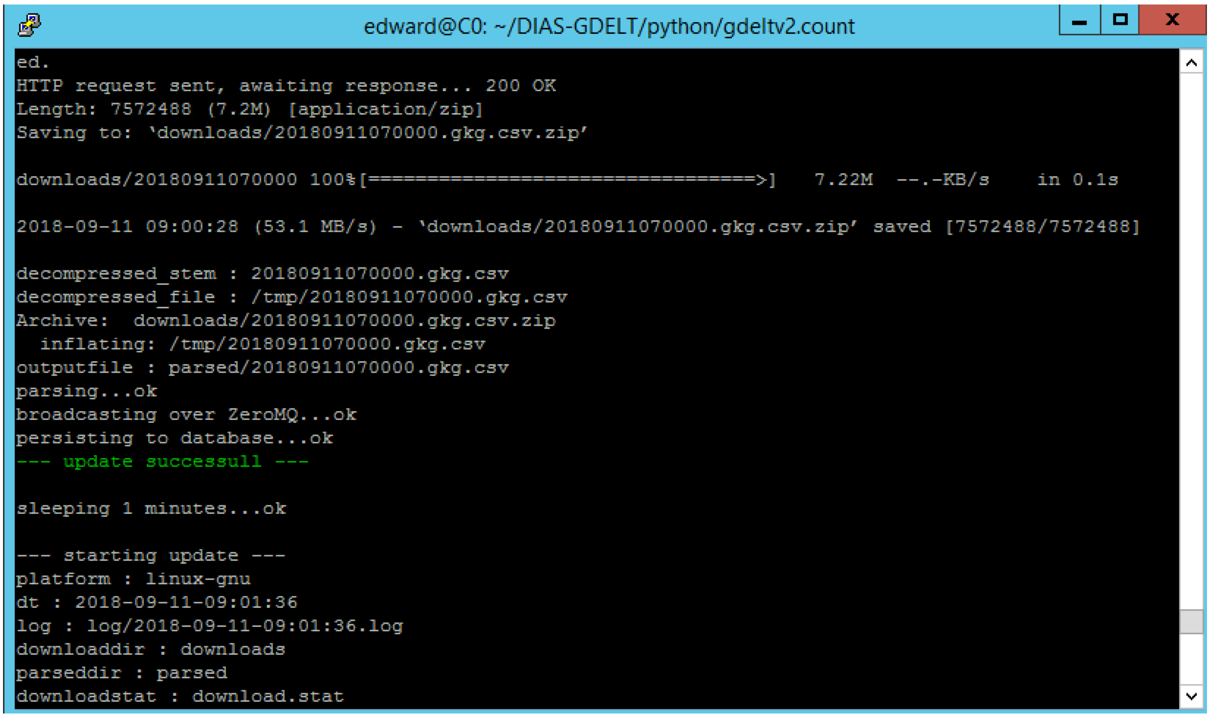
2.4 *GDELT Subscription*

Now we can start the GDELT subscription. GDELT collects data from thousands of websites and makes this data available every 15 minutes as a compressed text file. The *GDELT Subscription* is a bash script that retrieves this file, decompresses it, and delivers the data to the 28 *mock devices.*

The bash and Python scripts obtain an update of the news events from GDELT every 15 minutes. The data is only obtained once from the GDELT website, then broadcast locally to each of the 28 mock devices. More details are provided below.

```
cd /DIAS-GDELT/python/gdeltv2.count

./auto.update.sh
```



```
ed.
HTTP request sent, awaiting response... 200 OK
Length: 7572488 (7.2M) [application/zip]
Saving to: 'downloads/20180911070000.gkg.csv.zip'

downloads/20180911070000 100%[=================================>]   7.22M  --.-KB/s    in 0.1s

2018-09-11 09:00:28 (53.1 MB/s) - 'downloads/20180911070000.gkg.csv.zip' saved [7572488/7572488]

decompressed_stem : 20180911070000.gkg.csv
decompressed_file : /tmp/20180911070000.gkg.csv
Archive:  downloads/20180911070000.gkg.csv.zip
  inflating: /tmp/20180911070000.gkg.csv
outputfile : parsed/20180911070000.gkg.csv
parsing...ok
broadcasting over ZeroMQ...ok
persisting to database...ok
--- update successull ---

sleeping 1 minutes...ok

--- starting update ---
platform : linux-gnu
dt : 2018-09-11-09:01:36
log : log/2018-09-11-09:01:36.log
downloaddir : downloads
parseddir : parsed
downloadstat : download.stat
```

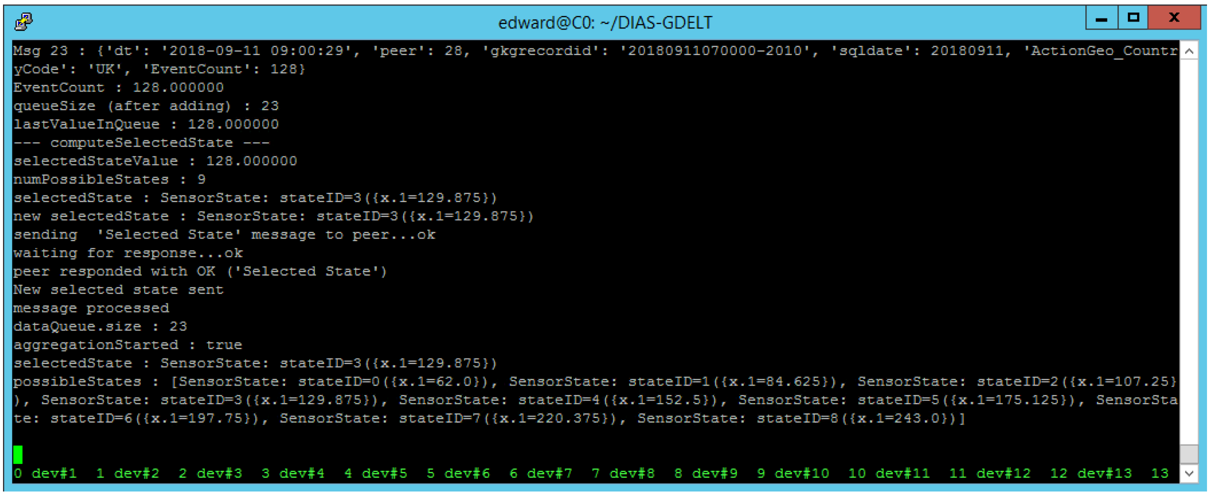
Figure {}. The script that retrieves the news feed from GDELT every 15 minutes. Once new data has arrived, it is parsed and broadcast over ZeroMQ for the *mock devices* to process

2.5 Mock Devices

Finally, we are ready to start the *mock devices*. In the DIAS-GDELT application, each *mock device* represents a country. Thus we must launch 28 *mock devices* which listen to data updates over the ZeroMQ broadcast (mentioned just before) and update their selected and possible states accordingly.

```
cd DIAS-GDELT

./start.mock.devices.sh deployments/gdelt 28
```



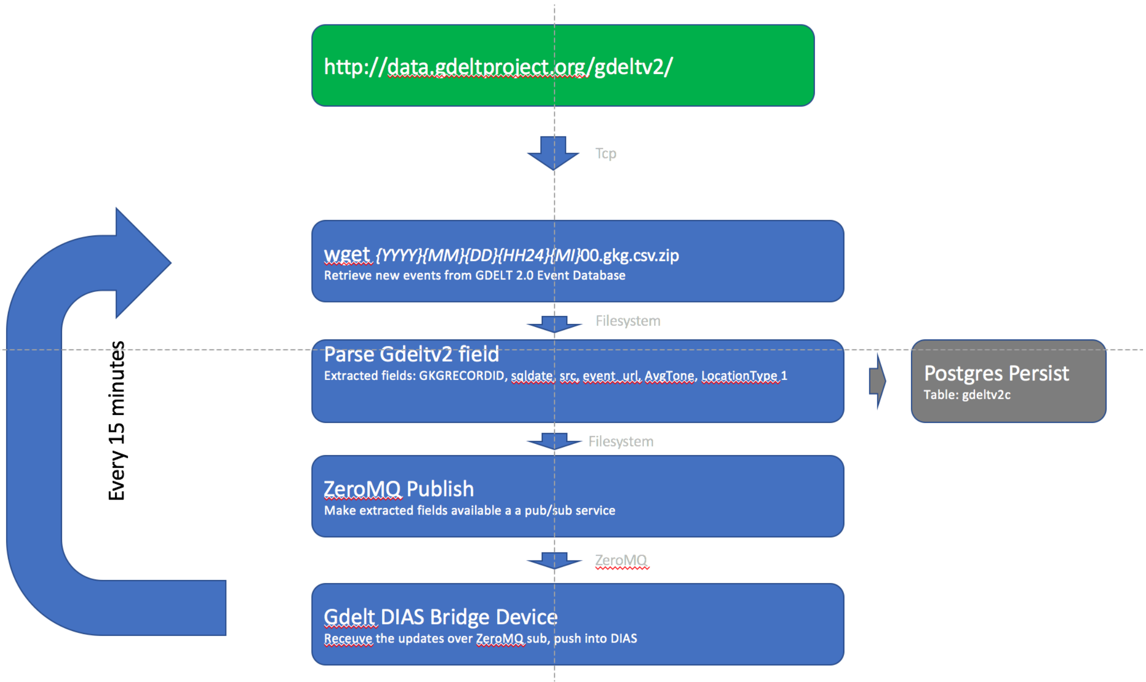
```
Msg 23 : {'dt': '2018-09-11 09:00:29', 'peer': 28, 'gkgrecordid': '20180911070000-2010', 'sqldate': 20180911, 'ActionGeo_Countr
yCode': 'UK', 'EventCount': 128}
EventCount : 128.000000
queueSize (after adding) : 23
lastValueInQueue : 128.000000
--- computeSelectedState ---
selectedStateValue : 128.000000
numPossibleStates : 9
selectedState : SensorState: stateID=3({x.1=129.875})
new selectedState : SensorState: stateID=3({x.1=129.875})
sending  'Selected State' message to peer...ok
waiting for response...ok
peer responded with OK ('Selected State')
New selected state sent
message processed
dataQueue.size : 23
aggregationStarted : true
selectedState : SensorState: stateID=3({x.1=129.875})
possibleStates : [SensorState: stateID=0({x.1=62.0}), SensorState: stateID=1({x.1=84.625}), SensorState: stateID=2({x.1=107.25}
), SensorState: stateID=3({x.1=129.875}), SensorState: stateID=4({x.1=152.5}), SensorState: stateID=5({x.1=175.125}), SensorSta
te: stateID=6({x.1=197.75}), SensorState: stateID=7({x.1=220.375}), SensorState: stateID=8({x.1=243.0})]

0 dev#1  1 dev#2  2 dev#3  3 dev#4  4 dev#5  5 dev#6  6 dev#7  7 dev#8  8 dev#9  9 dev#10  10 dev#11  11 dev#12  12 dev#13  13
```

Figure {}. The mock devices, each processing messages from the ZeroMQ broadcast and listening for changes in the selected state.

How it works?

diagram



http://data.gdeltproject.org/gdeltv2/
Tcp
wget {YYYY}{MM}{DD}{HH24}{MI}00.gkg.csv.zip
Retrieve new events from GDELT 2.0 Event Database
Filesystem
Parse Gdeltv2 field
Extracted fields: GKGRECORDID, sqldate, src, event_url, AvgTone, LocationType 1
Postgres Persist
Table: gdeltv2c
Filesystem
ZeroMQ Publish
Make extracted fields available a a pub/sub service
ZeroMQ
Gdelt DIAS Bridge Device
Receuve the updates over ZeroMQ sub, push into DIAS
Every 15 minutes

Figure {}. High level view of the news flow

Raw data download from GDELT

At a high level, the process works as follows. The raw gdelt2 data files are retrieved from the GDELT website using the linux *wget* command-line utility. Files are decompressed and parsed in order to extract the following fields of interest for the DIAS-GDELT application:

- GKGRECORDID
- sqldate
- src

- event_url
- AvgTone
- LocationType1

A detailed description of the meanings of these fields can be found in the GDELT 2.0 Event Database Codebook located here (http://data.gdeltproject.org/documentation/GDELT-Event_Codebook-V2.0.pdf)

The LocationType1 contains the FIPS 2-letter country code of the location of the event. It is used to assign the event to one of the 28 European countries in our application.

Note that this field does not capture which countries are affected by events, but rather where the event actually took place. FIPS country codes can be looked up on Wikipedia (https://en.wikipedia.org/wiki/List_of_FIPS_country_codes).

Local Publication Service
There are 28 *DIAS nodes* in this application, each representing a European country.

For sake of efficiency, list of news events (compressed text file) is only pulled once from the GDELT server every 15 minutes. After the decompressing and parsing, data is published using a ZeroMQ publish socket over a TCP port on the server. This allows the 28 *mock devices* to connect to this port and subscribe to the parsed GDELT events as ZeroMQ subscribers, whilst only retrieving the raw data once from GDELT.

GDELT provides an update every 15 minutes. The Local Publication Service is responsible for filtering European countries and for computing the number of events per country over this 15-minute time window. The field name for determining the country is LocationType1.

Mock Device
As with all DIAS applications, the *DIAS nodes* that perform the aggregation require selected and possible states as inputs. In the case of the DIAS-GDELT application, we only have the number of events per country provided by the Local Publication Service generated in the last 15 minutes. Thus an intermediate step is required to transform the most recent count of events into a list of *possible states* and a *selected state* chosen from the *possible states*. This task is handled by the *mock device*.

Each mock device is a separate JVM process, that subscribes to the Local Publication Service to receive the number of events per country. From this metric, each mock device computes:

- The list of possible states
- Selects the selected state from the list of possible states

Each *mock device* computes it's possible states individually. The *possible states* are computed using a rolling window of the latest 27 observations. Each observation is the number of GDELT news events generated over a time window of 15 minutes for the relevant country that has been mapped to the mock device. In this case of this application, there are 9 possible states, that are equally spaced between the minimum and maximum of the 27 observations.

The mock device then communicates the possible and selected state to it's assigned *DIAS node*, so that the DIAS network can perform the aggregation.

It could be argued that the functionality of the mock devices should be integrated into the *DIAS nodes* in order to reduce the complexity of the setup. The present design was chosen due to it's modularity. Indeed, a number of different applications can use the exact same *DIAS nodes*. Only the mock devices are application specific. At time of writing, the applications that use *DIAS nodes* are DIAS-GDELT, Smart Agora, and the basic DIAS testbeds.

Database Persistence
Following publication over ZeroMQ, the parsed GDELT data is persisted to a PostgreSQL database. This allows the mock DIAS-GDELT devices to "warmup" on recent historical data, so that the queue for computing the possible states can be backfilled on startup, which allows each mock device to recover it's state after a restart.

References

- GitHub repo
  - `https://github.com/epournaras/DIAS-GDELT.git`
- GDELT
  - http://data.gdeltproject.org/
  - http://data.gdeltproject.org/documentation/GDELT-Event_Codebook-V2.0.pdf
- DIAS-GDELT microsite
  - http://dias-net.org/gdelt/