

DIAS-GDELT Documentation

2018-09-12 (Last update)

Intro

Have you ever wondered how many news event are produced every day in Europe? 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 (data.gdeltproject.org). 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 application covers the 28 countries in Europe.

Screenshots

Here is a screenshot of DIAS-GDELT in action. You can observe the fluctuations in the numbers of news events generate 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.

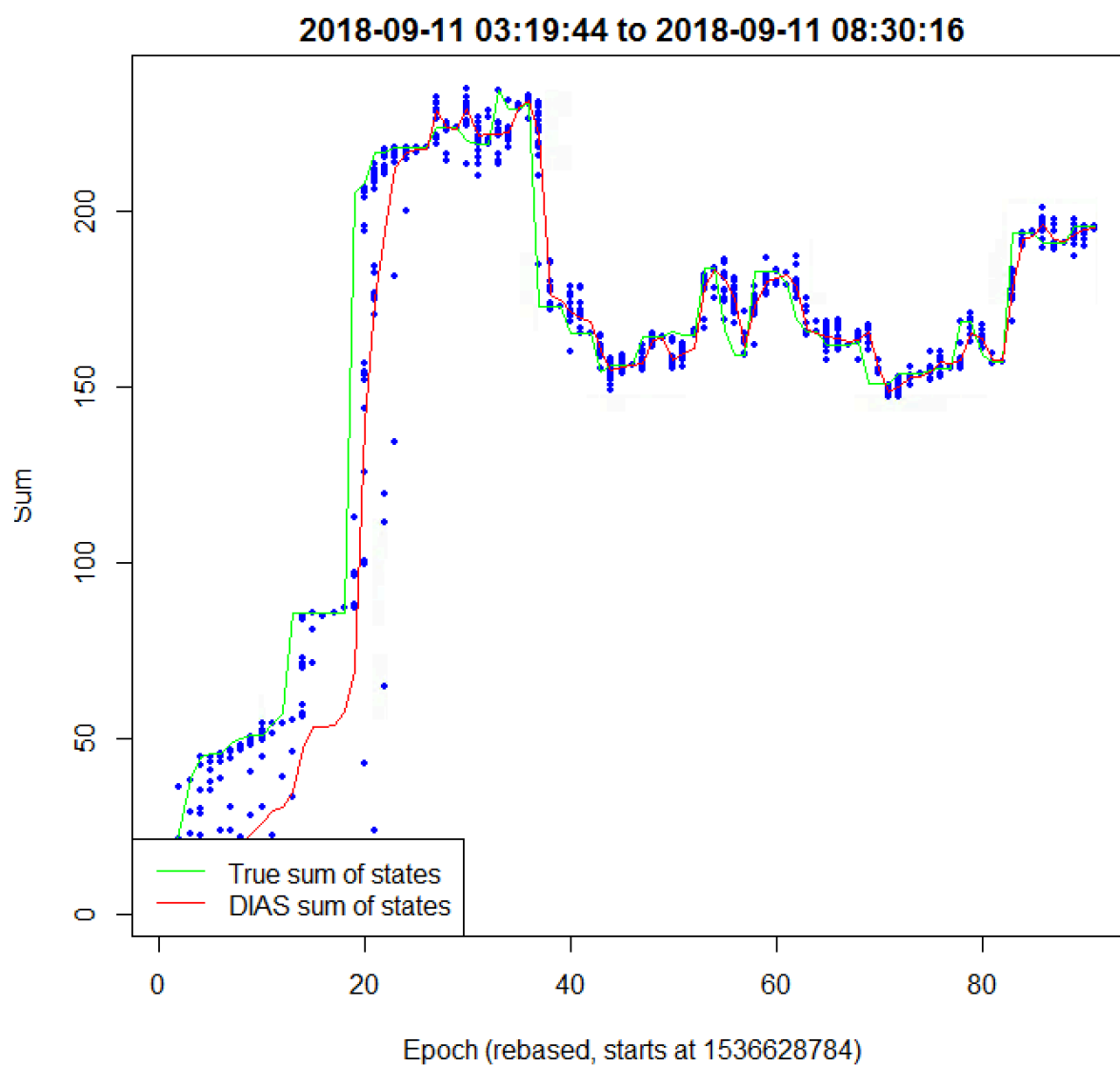


Figure 1. DIAS-GDELT aggregation process. Each blue dot represents the inferred sum from a DIAS peer. The green line shows the true sum of states, whilst the red line shows the DIAS aggregation (sum).

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

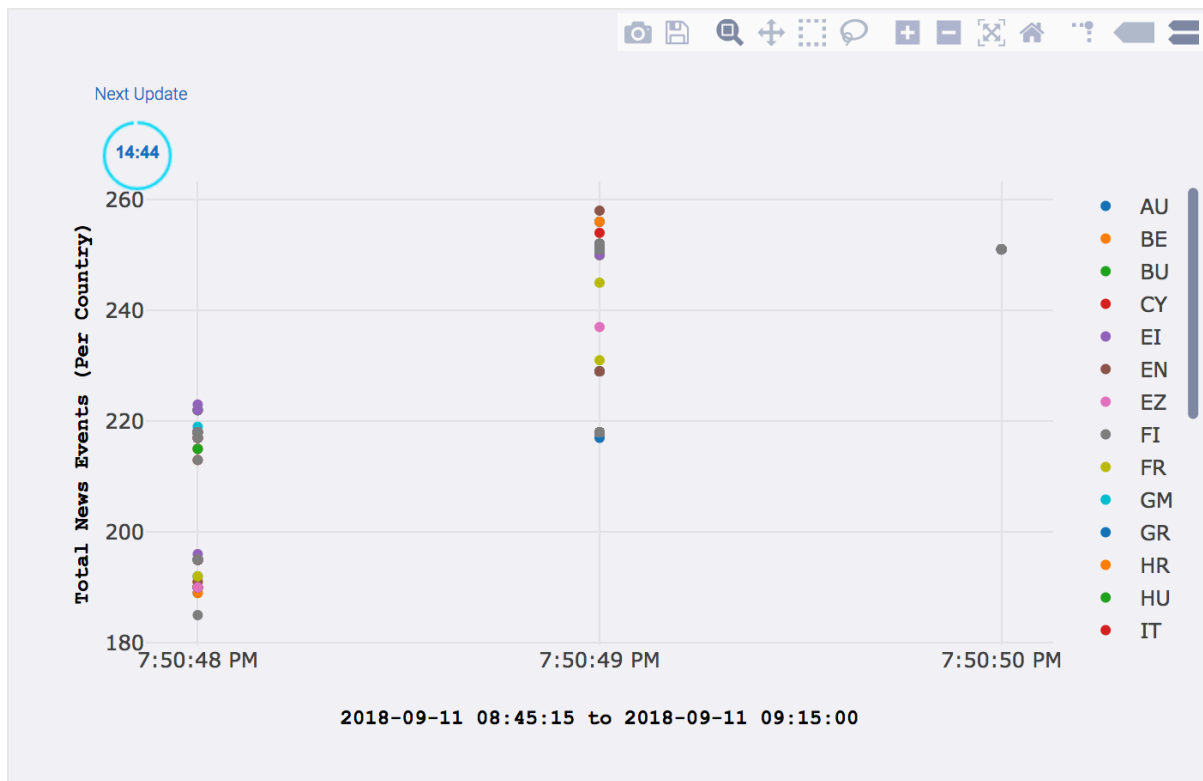


Figure {}. DIAS-GDELT aggregation process as viewed in the webpage <http://dias-net.org/gdelt/>

Getting Started

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
```

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 interfering to a minimum with the performance of the individual peers.

```
cd DIAS-Logging-System  
./start.daemon.sh
```

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

```

edward@C0: ~
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 1. DIAS-Logging Daemon receiving messages and persisting them to a PostgreSQL database for offline and online analytics

Next, we start a basic DIAS Protopeer Bootstrap Server and DIAS Gateway Server. The bootstrap server allows DIAS peers to communicate with each other. The gateway, on the other hand, allows devices to be assigned to DIAS peers.

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 servers, you should see something like this.

```

edward@C0: ~/DIAS-Development
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 1. The Proropeer Bootstrap Server, that allows the DIAS peers to initially connect with each other.

A terminal window titled 'edward@C0: ~/DIAS-Development' showing the logs of the Proropeer Bootstrap Server. The logs show two successful peer address requests. The first request is from device 'dev#27:1' with peer ID 27 and finger C0:50439. The second request is from device 'dev#28:1' with peer ID 28 and finger C0:54702. The server responds to both requests, indicating that the peers are allocated and the response is sent. The terminal also shows a status bar at the bottom with tabs for '0 Protopeer Bootstrap', '1 DIAS Gateway Server', and '2 Map', along with a timestamp and version information.

```
edward@C0: ~/DIAS-Development
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
```

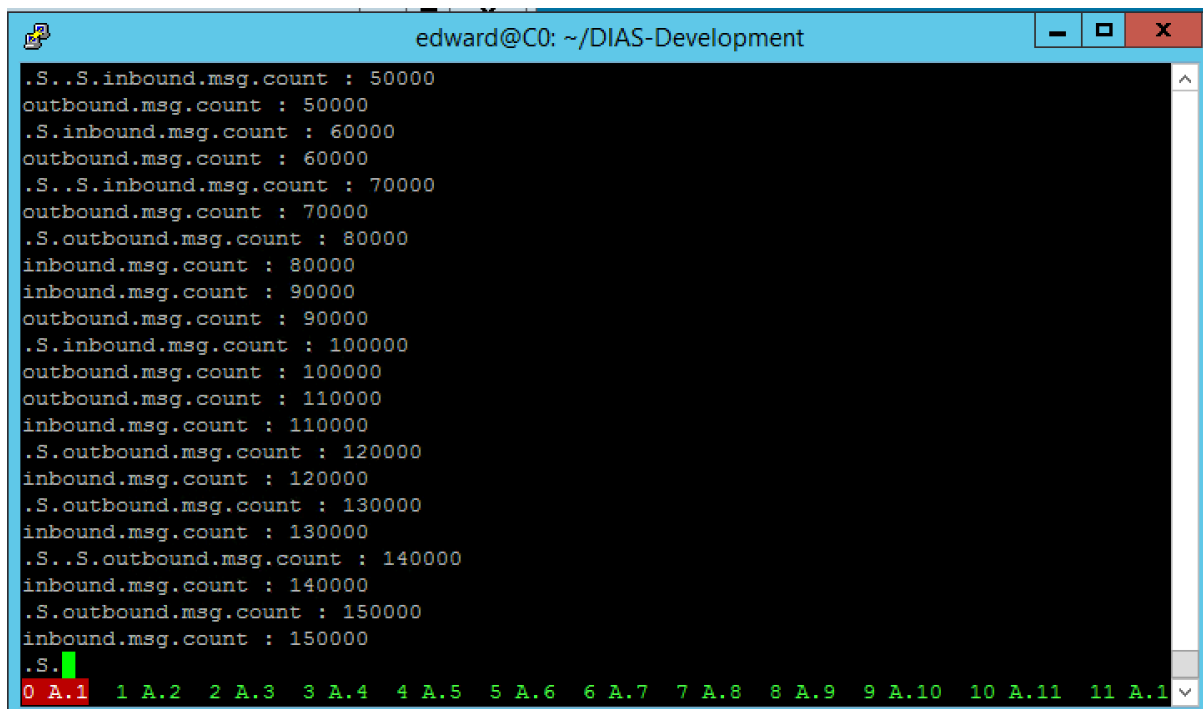
Figure 1. The DIAS Gateway Server, that binds devices such as mobiles phones to DIAS peers

Now that we have the protopeer and gateway servers running, we can launch the DIAS peers, also known as aggregation peers. As the DIAS-GDELT simulation covers the European countries only, we will launch 28 DIAS peers, 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 peer per screen tab. Use the command `ctrl+A` then `n` or `p` to cycle forwards or backwards amongst the tabs of the screen. Since we have launched aggregation peers, they were given the prefix `A` (for aggregation), and then suffixed with the id of the DIAS peer.

A terminal window titled 'edward@C0: ~/DIAS-Development' displays a list of message counts for various peers. The peers are identified by labels like '.S..S.inbound.msg.count', '.S.inbound.msg.count', and '.S.outbound.msg.count'. The counts range from 50,000 to 150,000. At the bottom, a status bar shows 12 aggregation peers labeled '0 A.1' through '11 A.11' in a green font.

```
.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.11
```

Figure 1. The DIAS Aggregation Peers, that will perform the aggregation. Each peer is launched in a separate tab of the *screen* utility

Now we can start the GDELT subscription. 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
```

```

edward@C0: ~/DIAS-GDELT/python/gdeltv2.count
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 successfull ---

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
downloadaddr : downloads
parseddir : parsed
downloadstat : download.stat

```

Figure 1. 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

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. The mock devices 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

```

```

edward@C0: ~/DIAS-GDELT
Msg 23 : {'dt': '2018-09-11 09:00:29', 'peer': 28, 'gkgrecordid': '20180911070000-2010', 'sqldate': '20180911', 'ActionGeo_CountryCode': '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}), SensorState: 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 1. The mock devices, each processing messages from the ZeroMQ broadcast and listening for changes in the selected state.

How it works?

diagram

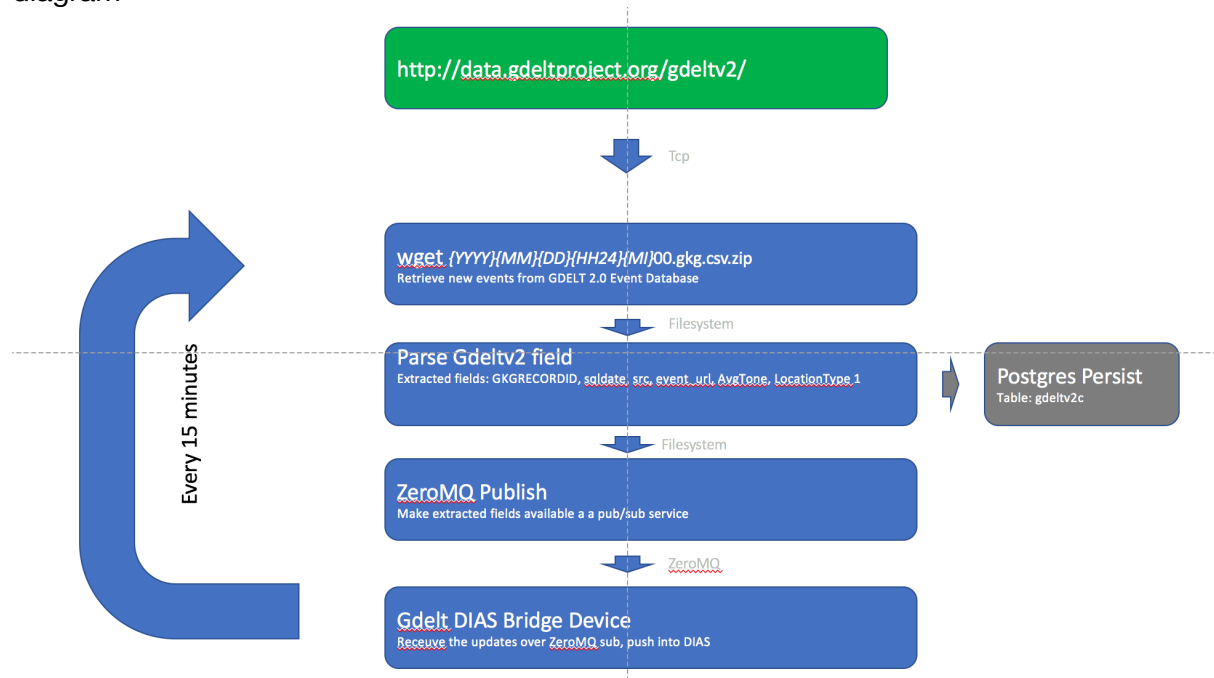


Figure 1. High level view of the news flow

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 peers in this application, each representing a European country. However, we do not want to

For sake of efficiency, data is only pulled once from the GDELT server. After the decompressing and parsing, data is published over a ZeroMQ pub socket. This allows the 28 peers to individually consume 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 DIAS-GDELT Device

As with all DIAS applications, the DIAS peers 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 DIAS-GDELT device, hereafter "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

The mock device then communicates the possible and selected state to it's assigned DIAS aggregation peer, 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 peers 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 peers. Only the mock devices are application specific. At time of writing, the applications that use DIAS peers 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/>