Transfer Services.

Functions

- `append_data_to_log_file`
- `build_thread`
- `generate_query`
- `get_endpoints_rcsites`
- `get_fts_optimizer_events`
- `get_metrics_link`
- `get_rcsites_ip_address`
- `get_rcsites_ip_addresses`
- `get_subendpoints_rcsites`
- `inspect_transfers`
- `load_json_from_url`
- `load_yaml`
- `main`
- `make_decision_before_alert`
- `make_decision_link`
- `monitor_queue_before_alert`
- `monitor_transfers`
- `query_cric_database`
- `send_email`
- `sense_dynamic_circuit`
- `start_threads`

Module **`main`**

NOTED: a framework to optimise network traffic via the analysis of data from File Transfer Services.

```
Authors:
    * Carmen Misa Moreira
    * Edoardo Martelli

Company:
    CERN (IT-CS-NE)
```

`$ python3 main.py config-example.py [--verbosity debug/info/warning]`

```
python3 main.py -h
usage: main.py [-h] [-v VERBOSITY] config_file

NOTED: a framework to optimise network traffic via the analysis of data from

positional arguments:
  config_file                           the name of the configuration file [c

optional arguments:
  -h, --help                            show this help message and exit
  -v VERBOSITY, --verbosity VERBOSITY   defines the logging level [debug, inf
```

```
Structure of the project:
.
├── TransferBroker.py
├── __init__.py
├── config
│   └── config-example.yaml
├── documentation
│   ├── noted_main_function_documentation.pdf
│   └── noted_transfer_broker_class_documentation.pdf
├── html
│   ├── TransferBroker.html
│   └── main.html
├── log
│   ├── sense.log
│   └── transfer_broker.log
├── main.py
├── noted_email.txt
├── params
│   └── params.ini
├── plot_transfers.py
├── query
│   ├── query_monit_prod_fts_raw_queue_dst_rcsite
│   └── query_monit_prod_fts_raw_queue_src_rcsite
├── sense-o
│   ├── sense-cancel.sh
│   └── sense-provision.sh
└── transfers
    ├── transfer_broker_all_transfers.txt
    ├── transfer_broker_dst_rcsite.txt
    └── transfer_broker_src_rcsite.txt
```

▸ EXPAND SOURCE CODE

Functions

`def append_data_to_log_file(filename, df_last_event)`

Function to append data to a log file.

Args

`filename` : `str`

name of the file to append data.

`df_last_event` : `dataframe`

last FTS optimizer event.

▼ EXPAND SOURCE CODE

```
def append_data_to_log_file(filename, df_last_event):
    """Function to append data to a log file.

    Args:
        filename (str): name of the file to append data.
        df_last_event (dataframe): last FTS optimizer event.
    """
    logging.debug('Transfer broker: append data to log file %s.' % filename)
    f = open(filename + '.txt', 'a+')
    if isinstance(df_last_event, pd.DataFrame):
        data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp = get
        f.write('timestamp: ' + str(timestamp) + ', datetime: ' + str(datetime.now()) + ', source:
    f.close()
    return
```

`def build_thread(params, type)`

Function to create a thread per link for monitoring the transfers.

Args

`params` : `configparser`
parameters file.

`type` : `str`
direction of the link, it can take two values: {tx, rx}.

Returns

`thread`
pointing to a defined link.

▼ EXPAND SOURCE CODE

```
def build_thread(params, type):
    """Function to create a thread per link for monitoring the transfers.

    Args:
        params (configparser): parameters file.
        type (str): direction of the link, it can take two values: {tx, rx}.

    Returns:
        thread: pointing to a defined link.
    """
    logging.debug('Building thread %s%s.' % ('transfer_broker_', type))
    # Launch a thread to monitor transfers for a defined link {src, dst}
    transfers = Thread(name = 'transfer_broker_' + type, target = monitor_transfers, args = [param
    return transfers
```

`def generate_query(config, params, type, df_cric_database)`

Function to generate the queries for downloading the FTS raw queues.

Args

`config` : `dict`
dictionary with the yaml configuration file.

`params` : `configparser`
parameters file.

`type` : `str`
direction of the link. It can take two values: src_rcsite or dst_rcsite.

`df_cric_database` : `DataFrame`
CRIC database information in a dataframe structure.

Returns

`list`
list of endpoints for a defined link.

▼ EXPAND SOURCE CODE

```
def generate_query(config, params, type, df_cric_database):
    """Function to generate the queries for downloading the FTS raw queues.

    Args:
        config (dict): dictionary with the yaml configuration file.
        params (configparser): parameters file.
        type (str): direction of the link. It can take two values: src_rcsite or dst_rcsite.
        df_cric_database (DataFrame): CRIC database information in a dataframe structure.
```

```
    Returns:
        list: list of endpoints for a defined link.
    """
    # Get a list with the endpoints
    rcsite_type = config[type]
    list_endpoints = df_cric_database.query('Rcsite in @rcsite_type').reset_index(drop = True)['E
    # Write query into a file without extension
    if 'src_rcsite' in type:
        query_2nd_line_2 = '"data.source_se": ' + json.dumps(list_endpoints)
        with open(params.get('QUERY PARAMETERS', 'filename_src_query'), 'w') as f: f.write(params.
    else:
        query_2nd_line_2 = '"data.dest_se": ' + json.dumps(list_endpoints)
        with open(params.get('QUERY PARAMETERS', 'filename_dst_query'), 'w') as f: f.write(params.
    logging.debug('Generating query for %s, number of endpoints: %d' % (type, len(list_endpoints))
    return list_endpoints
```

`def get_endpoints_rcsites(df_cric_database, df_cric_endpoints)`

Function to get endpoints and rcsites.

Args

`df_cric_database` : `dataframe`

CRIC database.

`df_cric_endpoints` : `dataframe`

CRIC endpoints.

Returns

`dataframe`

columns Rcsite, Federation and Endpoint of CRIC database in a dataframe structure.

▼ EXPAND SOURCE CODE

```
def get_endpoints_rcsites(df_cric_database, df_cric_endpoints):
    """Function to get endpoints and rcsites.

    Args:
        df_cric_database (dataframe): CRIC database.
        df_cric_endpoints (dataframe): CRIC endpoints.

    Returns:
        dataframe: columns Rcsite, Federation and Endpoint of CRIC database in a dataframe structu
    """
    logging.debug('Querying CRIC database: get endpoints rcsites.')
    # Separate the endpoints into two categories: endpoint and sub-endpoint
    df_endpoint     = df_cric_endpoints[df_cric_endpoints['endpoint'].notnull()]
    df_sub_endpoint = df_cric_endpoints[df_cric_endpoints['endpoint'].isnull()] # endpoints are i
    # Fill dataframe with the data from query
    df_cric_database['Rcsite']     = df_endpoint['rcsite'].values
    df_cric_database['Federation'] = df_endpoint['federation'].values
    df_cric_database['Endpoint']   = df_endpoint['endpoint'].values
    # Get sub-endpoints in rcsites
    list_endpoints = get_subendpoints_rcsites(df_sub_endpoint)
    # Append endpoints to dataframe, drop duplicates and sort data by rcsites
    df_cric_database = pd.concat([df_cric_database, pd.DataFrame(list_endpoints, columns = ['Endp
    df_cric_database = df_cric_database.drop_duplicates(subset = ['Endpoint'])
    df_cric_database = df_cric_database.sort_values(by = ['Rcsite'], ignore_index = True)
    # Format data: remove port in endpoints: davs://webdav.lcg.triumf.ca:2880 -> davs://webdav.lc
    df_cric_database['Endpoint'] = df_cric_database['Endpoint'].replace({r':\d+': ''}, regex = Tru
    return df_cric_database
```

`def get_fts_optimizer_events(cmd)`

Function to get the FTS optimizer events.

Args

`cmd` : `str`

curl command to execute.

Returns

`dataframe`

FTS optimizers events in a dataframe structure.

▼ EXPAND SOURCE CODE

```
def get_fts_optimizer_events(cmd):
    """Function to get the FTS optimizer events.

    Args:
        cmd (str): curl command to execute.

    Returns:
        dataframe: FTS optimizers events in a dataframe structure.
    """
```

```
    logging.debug('Inspecting transfers: get FTS optimizer events.')
    # Get all the FTS Optimizer events:
    #   Connections:    gives the maximum number of transfers that can be held (optimizer decisio
    #   Rationale:      if 'Range fixes' means that connections is the limit value set by the orga
    #                   if 'Queue emptying' then connections is the maximum value set by the orga
    #   Active_count:   gives the number of parallel transfers (TCP windows)
    #   Submitted_count gives the number of transfers in the queue
    try:
        response = json.loads(os.popen(cmd).read())['responses']
        df_hits  = pd.DataFrame(pd.DataFrame(response)['hits'][0]['hits'])
        df_items = pd.DataFrame(df_hits['_source'].values.tolist())
        df_fts_optimizer_data = pd.DataFrame(df_items['data'].values.tolist())
    # Return an empty dataframe because there are none FTS Optimizer Events
    except (KeyError, ValueError): df_fts_optimizer_data = pd.DataFrame(columns = ['source_se', 'c
    return df_fts_optimizer_data
```

`def get_metrics_link(df_last_event)`

Function to get the metrics of the link.

Args

`df_last_event` : `dataframe`
: last event of FTS optimizer.

Returns

`float64`
: amount of data in GB.

`float64`
: throughput of the link in Gb/s.

`int64`
: number of TCP parallel transfers.

`int64`
: number of transfers in the queue.

`int64`
: timestamp of the current FTS optimizer event.

▼ EXPAND SOURCE CODE

```
def get_metrics_link(df_last_event):
    """Function to get the metrics of the link.

    Args:
        df_last_event (dataframe): last event of FTS optimizer.

    Returns:
        float64: amount of data in GB.
        float64: throughput of the link in Gb/s.
        int64: number of TCP parallel transfers.
        int64: number of transfers in the queue.
        int64: timestamp of the current FTS optimizer event.
    """
    logging.debug('Inspecting transfers: get metrics of the link.')
    data_gigabytes = df_last_event['filesize_avg'].values[0]/1e9      # Amount of data [GB]
    throughput_gigabits = 8*df_last_event['throughput'].values[0]/1e9 # Throughput [Gb/s]
    parallel_transfers  = df_last_event['active_count'].values[0]     # TCP parallel transfers
    queued_transfers    = df_last_event['submitted_count'].values[0]  # Transfers in the queue
    timestamp           = df_last_event['timestamp'].values[0]        # Timestamp
    return data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp
```

`def get_rcsites_ip_address(df_cric_database, df_query, ip_version)`

Function to get IPv4/IPv6 addresses, fills the columns 'IPv4' and 'IPv6' of df_cric_database.

Args

`df_cric_database` : `dataframe`
: CRIC database.

`df_query` : `dataframe`
: FTS optimizer events.

`ip_version` : `str`
: IP version, it can take two values: {ipv4, ipv6}.

▼ EXPAND SOURCE CODE

```
def get_rcsites_ip_address(df_cric_database, df_query, ip_version):
    """Function to get IPv4/IPv6 addresses, fills the columns 'IPv4' and 'IPv6' of df_cric_databas
```

```
    Args:
        df_cric_database (dataframe): CRIC database.
        df_query (dataframe): FTS optimizer events.
        ip_version (str): IP version, it can take two values: {ipv4, ipv6}.
    """
    logging.debug('Querying CRIC database: get %s%s addresses.' % (ip_version[:2].upper(), ip_vers
    list_ip = []
    # Iterate over netroutes parameter
    for i in range(df_query.shape[0]):
        name = df_query['name'].values[i]
        df_networks = pd.DataFrame(df_query['netroutes'].values[i]).T['networks']
        df_ip = pd.DataFrame(df_networks.values.tolist())
        # Get IPv4/IPv6 addresses
        if ip_version in df_ip.columns: df_ip_version = pd.DataFrame(df_ip[ip_version].dropna().re
        else: continue # There are none ip address
        for index, row in df_ip_version.iterrows():
            items = pd.DataFrame(row.values.tolist())
            for i in range(items.shape[1]):
                if items[i].values[0] not in list_ip: list_ip.append(items[i].values[0])
        # Fill dataframe with the data from query
        duplicated_index = df_cric_database.query('Rcsite in @name').index # Note: can be more tha
        if 'ipv4' in ip_version: df_cric_database.loc[duplicated_index, 'IPv4'] = [list_ip.copy()]
        else: df_cric_database.loc[duplicated_index, 'IPv6'] = [list_ip.copy()]
        # Clear the content of the list for the next iteration
        list_ip.clear()
    return
```

`def get_rcsites_ip_addresses(df_cric_database, df_cric_ip_address)`

Function to get rcsites and IPv4/IPv6 addresses.

Args

`df_cric_database` : `dataframe`

CRIC database.

`df_cric_ip_address` : `dataframe`

CRIC IPv4/IPv6 addresses.

Returns

`dataframe`

columns 'IPv4' and 'IPv6' of CRIC database in a dataframe structure.

▼ EXPAND SOURCE CODE

```
def get_rcsites_ip_addresses(df_cric_database, df_cric_ip_address):
    """Function to get rcsites and IPv4/IPv6 addresses.

    Args:
        df_cric_database (dataframe): CRIC database.
        df_cric_ip_address (dataframe): CRIC IPv4/IPv6 addresses.

    Returns:
        dataframe: columns 'IPv4' and 'IPv6' of CRIC database in a dataframe structure.
    """
    # Get rcsites from fts_queues json file
    list_rcsites = df_cric_database['Rcsite'].drop_duplicates().tolist()
    # Get parameters from CRIC json file, i.e. df_cric_ip_address
    df_query = df_cric_ip_address.query('name in @list_rcsites', engine = 'python').reset_index(dr
    # Get IPv4/IPv6 addresses
    get_rcsites_ip_address(df_cric_database, df_query, 'ipv4')
    get_rcsites_ip_address(df_cric_database, df_query, 'ipv6')
    return df_cric_database
```

`def get_subendpoints_rcsites(df_sub_endpoint)`

Function to get sub-endpoints in rcsites.

Args

`df_sub_endpoint` : `dataframe`

CRIC database sub-endpoints.

Returns

`list`

endpoints of rc_site in a list structure.

▼ EXPAND SOURCE CODE

```
def get_subendpoints_rcsites(df_sub_endpoint):
    """Function to get sub-endpoints in rcsites.

    Args:
        df_sub_endpoint (dataframe): CRIC database sub-endpoints.

    Returns:
        list: endpoints of rc_site in a list structure.
    """
```

```
    logging.debug('Querying CRIC database: get sub-endpoints rcsites.')
    list_endpoints = []
    # Iterate over services parameter
    for i in range(df_sub_endpoint.shape[0]):
        df_protocols = pd.DataFrame(df_sub_endpoint['protocols'].values[i]).T
        # Get endpoints
        if 'endpoint' in df_protocols.columns: df_endpoints = pd.DataFrame(df_protocols['endpoint'
        else: continue # there are none endpoints
        rcsite = df_sub_endpoint['rcsite'].values[i]
        federation = df_sub_endpoint['federation'].values[i]
        # Iterate over subendpoints parameter
        for index, row in df_endpoints.iterrows():
            items = pd.DataFrame(row.values.tolist())
            for j in range(items.shape[1]):
                if [items[j].values[0], rcsite] not in list_endpoints: list_endpoints.append([item
    return list_endpoints
```

`def inspect_transfers(params)`

Function to inspect transfers parameters in FTS.

Args

params : `configparser`

parameters file.

Returns

`dataframe`

last event of FTS optimizer.

▼ EXPAND SOURCE CODE

```
def inspect_transfers(params):
    """Function to inspect transfers parameters in FTS.

    Args:
        params (configparser): parameters file.

    Returns:
        dataframe: last event of FTS optimizer.
    """
    logging.debug('Inspecting transfers.')
    # Get list of endpoints [needed to execute the curl command]
    list_src_endpoints = class_transfer_broker_.get_list_src_endpoints()
    list_dst_endpoints = class_transfer_broker_.get_list_dst_endpoints()
    # Get FTS Optimizer events for a given {src, dst} pair and drop duplicates events
    df_fts_optimizer_data = get_fts_optimizer_events(params.get('CMD', 'cmd'))
    if 'src_rcsite' in threading.current_thread().name: df_query = df_fts_optimizer_data.query(par
    else: df_query = df_fts_optimizer_data.query(params.get('QUERY PARAMETERS', 'query_dst_site'),
    # Get latest FTS Optimizer event (first row), i.e. the most recent event generated by FTS Opti
    df_query = df_query.query('throughput != 0 & active_count > 0').reset_index(drop = True)
    # The link is 'inactive', i.e. df_query empty -> FTS Optimizer is updated every 5 min because
    if not df_query.empty: df_last_event = pd.DataFrame({'source_se': [df_query['source_se'][0]],
    else:
        # Create an empty dataframe
        logging.warning('No transfers found for the given {src, dst} pair.')
        time.sleep(5*60) # FTS OptimizerSteadyInterval = 300s = 5min
        df_last_event = pd.DataFrame(columns = ['source_se', 'dest_se', 'timestamp', 'throughput',
    return df_last_event
```

`def load_json_from_url(json_url)`

Function to load a json file from URL.

Args

json_url : `str`

url of the json file.

Returns

`dict`

data in a dictionary structure.

▼ EXPAND SOURCE CODE

```
def load_json_from_url(json_url):
    """Function to load a json file from URL.

    Args:
        json_url (str): url of the json file.

    Returns:
        dict: data in a dictionary structure.
    """
```

```
    logging.debug('Loading JSON from URL: %s' % json_url)
    # Self signed certificate issue
    ssl_context = ssl.create_default_context();
    ssl_context.check_hostname = False
    ssl_context.verify_mode = ssl.CERT_NONE
    # Load json file from URL
    with urllib.request.urlopen(json_url, context = ssl_context, timeout = None) as url:
        data = json.loads(url.read().decode())
        return data
```

def load_yaml(filename)

Function to load a yaml file.

Args

filename : `str`
: name of the yaml file.

Returns

`dict`
: data in a dictionary structure.

▼ EXPAND SOURCE CODE

```
def load_yaml(filename):
    """Function to load a yaml file.

    Args:
        filename (str): name of the yaml file.

    Returns:
        dict: data in a dictionary structure.
    """
    with open(filename) as file:
        data = yaml.load(file, Loader = yaml.FullLoader)
        return data
```

def main()

Main function.

▼ EXPAND SOURCE CODE

```
def main():

    """Main function."""
    # Logging
    logging.basicConfig(level = logging.NOTSET, filename = 'log/transfer_broker_' + str(time.time(
    logging.getLogger('numexpr.utils').setLevel(logging.WARNING) # hide logging messages from nume
    # Config parser
    params = configparser.ConfigParser(interpolation = configparser.ExtendedInterpolation())
    params.read('params/params.ini')
    # Argument parser
    args_parser = ArgumentParser(description = 'NOTED: a framework to optimise network traffic via
    args_parser.add_argument('config_file', help = 'the name of the configuration file [config-exa
    args_parser.add_argument('-v', '--verbosity', help = 'defines the logging level [debug, info,
    args = args_parser.parse_args()
    # Set verbosity level
    if args.verbosity is not None:
        if   'debug'   in args.verbosity: logging.getLogger().setLevel(logging.DEBUG)
        elif 'info'    in args.verbosity: logging.getLogger().setLevel(logging.INFO)
        elif 'warning' in args.verbosity: logging.getLogger().setLevel(logging.WARNING)
    # Load yaml config file
    logging.debug('Loading YAML file: %s' % args.config_file.split('/')[1])
    config = load_yaml(args.config_file)
    logging.info('Source rcsite: %s' % config['src_rcsite'])
    logging.info('Destination rcsite: %s' % config['dst_rcsite'])
    # CRIC database
    df_cric_database = query_cric_database(params)
    # Generate queries
    list_src_endpoints = generate_query(config, params, 'src_rcsite', df_cric_database)
    list_dst_endpoints = generate_query(config, params, 'dst_rcsite', df_cric_database)
    # Set the attributes of the class transfer broker
```

def make_decision_before_alert(params, action, message)

Function to make a decision on the link for loooking into the FTS queue before alert.

Args

params : `configparser`
: parameters file.

action : `str`
: action to be execute on the link, it can take two values: {start, stop}

message : `str`
: message to send in the email notification

▼ EXPAND SOURCE CODE

```
def make_decision_before_alert(params, action, message):
    """Function to make a decision on the link for loooking into the FTS queue before alert.

    Args:
        params (configparser): parameters file.
        action (str): action to be execute on the link, it can take two values: {start, stop}
        message (str): message to send in the email notification
    """
    logging.warning('Link under supervision: make decision before alert: action %s' % action)
    # Look to FTS queue for X events to see if it is just instaneous traffic fluctuations or not,
    bool_email = monitor_queue_before_alert(params, action)
    if bool_email:
        # Check the current state of the link. This is used to synchronize TX and RX threads becau
        if not class_transfer_broker_.get_unidirectional():
            # If TX and RX are not congested -> activate dynamic circuit and update state of the
            if action == 'start' and not class_transfer_broker_.get_link_src_state() and not class
                logging.warning('Link under supervision: update link state: True')
                if 'src_rcsite' in threading.current_thread().name: class_transfer_broker_.set_li
                else: class_transfer_broker_.set_link_dst_state(True)
                sense_dynamic_circuit(params, 'provision', action, message)
            # If TX congested but RX not: set only RX and do not send an email
            elif action == 'start' and class_transfer_broker_.get_link_src_state() and not class_t
                logging.warning('Link under supervision: update dst link state: True')
                class_transfer_broker_.set_link_dst_state(True)
            # If RX congested but TX not: set only TX and do not send an email
            elif action == 'start' and not class_transfer_broker_.get_link_src_state() and class_t
                logging.warning('Link under supervision: update link state: True')
                class_transfer_broker_.set_link_src_state(True)
            # Stop condition -> TX are RX are congested, not send an email
            elif action == 'stop' and class_transfer_broker_.get_link_src_state() and class_trans
```

`def make_decision_link(params, df_last_event)`

Function to make a decision on the link [start/stop events].

Args

`params` : `configparser`
: parameters file.

`df_last_event` : `dataframe`
: last event of FTS optimizer.

▼ EXPAND SOURCE CODE

```
def make_decision_link(params, df_last_event):
    """Function to make a decision on the link [start/stop events].

    Args:
        params (configparser): parameters file.
        df_last_event (dataframe): last event of FTS optimizer.
    """
    logging.debug('Inspecting transfers: make decision.')
    # Get the metrics of the link
    data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp = get_met
    # Get current state of the link
    if 'src_rcsite' in threading.current_thread().name: link_state = class_transfer_broker_.get_li
    else: link_state = class_transfer_broker_.get_link_dst_state()
    # If throughput > X Gb/s and the link is not congested send an email because the link will be
    if   throughput_gigabits > class_transfer_broker_.get_max_throughput() and not link_state: mak
    # If the link was congested but now the transfers takes throughput < X Gb/s, the link will not
    elif throughput_gigabits < class_transfer_broker_.get_min_throughput() and link_state: make_de
    return
```

`def monitor_queue_before_alert(params, action)`

Function to monitor FTS queue for X events to check if it is just instantaneous traffic or a huge transfer is going to take place.

Args

`params` : `configparser`
: parameters file.

`action` : `str`
: action to execute, it can take two values {start, stop}.

▼ EXPAND SOURCE CODE

```
def monitor_queue_before_alert(params, action):
    """Function to monitor FTS queue for X events to check if it is just instantaneous traffic or

    Args:
        params (configparser): parameters file.
        action (str): action to execute, it can take two values {start, stop}.
    """
    logging.warning('Link under supervision: monitor queue before alert.')
    number_of_events = 0
    last_timestamp = ''
    while True:
        # Inspect the parameters of the transfers in FTS and make a decision on the link based on
        df_last_event = inspect_transfers(params)
        # Get the metrics of the link
```

```
        data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp = get
        # Count events if there are with different timestamp, i.e. if there are different FTS Opti
        if last_timestamp != str(timestamp):
            # Interrupt the start/stop sequence because the throughput changed its expected behavi
            if (throughput_gigabits < class_transfer_broker_.get_max_throughput() and action == 's
            # Update last timestamp to the new one and count the events
            last_timestamp = str(timestamp)
            number_of_events = number_of_events + 1
            logging.warning('Link under supervision: monitor queue before alert: action %s, number
            logging.warning('df_last_event: timestamp: %s, source_se: %s, dest_se: %s, throughput 
            logging.warning('There are %d transfers with a total of %f GB - Throughput: %f Gb/s' %
            if number_of_events == class_transfer_broker_.get_events_to_wait(): return True
        time.sleep(60) # FTS OptimizerInterval = 60s
```

`def monitor_transfers(params)`

Function to monitor transfers in FTS, this function is used by TX/RX threads.

Args

`params` : `configparser`

parameters file.

▼ EXPAND SOURCE CODE

```python
def monitor_transfers(params):
    """Function to monitor transfers in FTS, this function is used by TX/RX threads.

    Args:
        params (configparser): parameters file.
    """
    logging.info('Monitoring transfers.')
    if 'src_rcsite' in threading.current_thread().name: filename = params.get('FILENAME PARAMETERS
    else: filename = params.get('FILENAME PARAMETERS', 'filename_transfers_dst')
    # Declare variables
    last_timestamp = '0'
    while True:
        # Inspect the parameters of the transfers in FTS and make a decision on the link based on
        df_last_event = inspect_transfers(params)
        # Get the metrics of the link
        if df_last_event.empty: continue # There are none transfers
        data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp = get
        if throughput_gigabits != 0:
            if 'src_rcsite' in threading.current_thread().name: link_state = class_transfer_broker
            else: link_state = class_transfer_broker_.get_link_dst_state()
            make_decision_link(params, df_last_event)
            # Append data to a log file for traceability purposes of the events
            if link_state and timestamp != last_timestamp: append_data_to_log_file(filename, df_la
            if timestamp != last_timestamp:
                append_data_to_log_file(params.get('FILENAME PARAMETERS', 'filename_all_transfers'
                logging.info('df_last_event: timestamp: %s, source_se: %s, dest_se: %s, throughput
                logging.info('There are %d transfers with a total of %f GB - Throughput: %f Gb/s'
            last_timestamp = timestamp
            time.sleep(60) # FTS OptimizerInterval = 60s
        # If the link is active and suddenly it goes down
```

def query_cric_database(params)

Query CRIC database.

Args

params : `configparser`

parameters file.

Returns

`dataframe`

CRIC database information in a dataframe structure.

▼ EXPAND SOURCE CODE

```python
def query_cric_database(params):
    """Query CRIC database.

    Args:
        params (configparser): parameters file.

    Returns:
        dataframe: CRIC database information in a dataframe structure.
    """
    # Get data from CRIC database
    dict_cric_ip_address = load_json_from_url(params.get('CRIC PARAMETERS', 'cric_ip_address_url')
    dict_cric_endpoints  = load_json_from_url(params.get('CRIC PARAMETERS', 'cric_endpoints_url'))
    # Convert dictionaries to dataframes
    df_cric_ip_address = pd.DataFrame.from_dict(dict_cric_ip_address).T # columns = admin_email, a
    df_cric_ip_address = df_cric_ip_address[df_cric_ip_address['netroutes'] != {}].reset_index(dro
    df_cric_endpoints  = pd.DataFrame.from_dict(dict_cric_endpoints).T  # columns = arch, country,
    # Process data: get endpoints
    logging.debug('Querying CRIC database.')
    df_cric_database = pd.DataFrame(columns = ['Endpoint', 'Rcsite', 'Federation', 'IPv4', 'IPv6']
    df_cric_database = get_endpoints_rcsites(df_cric_database, df_cric_endpoints)     # Get endpoi
    df_cric_database = get_rcsites_ip_addresses(df_cric_database, df_cric_ip_address) # Get rcsite
    logging.info('There are %d rcsites defined in CRIC database'   % df_cric_ip_address.shape[0])
    logging.info('There are %d endpoints defined in CRIC database' % df_cric_endpoints.shape[0])
    return df_cric_database
```

def send_email(params, message)

Function to send an email for advertising the congestion of the link.

Args

params : `configparser`

parameters file.

message : `str`

message to stream

▼ EXPAND SOURCE CODE

```python
def send_email(params, message):
    """
    Function to send an email for advertising the congestion of the link.

    Args:
        params (configparser): parameters file.
        message (str): message to stream
    """
```

```
    logging.warning('Sending email: LHCOPN Source: %s, LHCOPN Destination: %s, message: %s' % (cla
    # Inspect parameters of the transfers and get the metrics of the link
    df_last_event = inspect_transfers(params)
    data_gigabytes, throughput_gigabits, parallel_transfers, queued_transfers, timestamp = get_met
    # Write content of the email into a txt file
    with open('noted_email.txt', 'w') as f:
        f.write('From: noted@cern.ch\n')
        f.write('To: carmen.misa.moreira@cern.ch\n')
        f.write('Subject: [NOTED] Notification\n\n')
        f.write('Hi,\n\n')
        f.write('New transfer detected...\n\n')
        f.write(message + '\n\n')
        if 'src_rcsite' in threading.current_thread().name:
            f.write('LHCOPN Source: '      + str(class_transfer_broker_.get_list_src_rcsites()) +
            f.write('LHCOPN Destination: ' + str(class_transfer_broker_.get_list_dst_rcsites()) +
        else:
            f.write('LHCOPN Source: '      + str(class_transfer_broker_.get_list_dst_rcsites()) +
            f.write('LHCOPN Destination: ' + str(class_transfer_broker_.get_list_src_rcsites()) +
        f.write('\tTimestamp: '             + str(timestamp)            + '\n')
        f.write('\tNumber of transfers: '   + str(parallel_transfers)   + '\n')
        f.write('\tQueued transfers: '      + str(queued_transfers)     + '\n')
        f.write('\tThroughput [Gb/s]: '     + str(throughput_gigabits)  + '\n')
```

def sense_dynamic_circuit(params, task, action, message)

Function to provide or cancel a dynamic circuit by using sense-o autogole northbound API.

Args

params : `configparser`
parameters file.

task : `str`
action to be executed in sense-o, it can take two values: {provision, cancel}.

action : `str`
action to be execute on the link, it can take two values: {start, stop}

message : `str`
message to send in the email notification

▼ EXPAND SOURCE CODE

```
def sense_dynamic_circuit(params, task, action, message):
    """Function to provide or cancel a dynamic circuit by using sense-o autogole northbound API.

    Args:
        params (configparser): parameters file.
        task (str): action to be executed in sense-o, it can take two values: {provision, cancel}.
        action (str): action to be execute on the link, it can take two values: {start, stop}
        message (str): message to send in the email notification
    """
    logging.warning('Calling to sense-o API to %s a dynamic circuit.' % task)
    # Provision dynamic circuit with sense-o
    if 'provision' in task:
        cmd = os.popen(str('sh sense-o/sense-provision.sh ' + class_transfer_broker_.get_sense_uui
        cmd = os.popen(str('sh sense-o/sense-provision.sh ' + class_transfer_broker_.get_sense_uui
    # Cancel dynamic circuit with sense-o
    else:
        cmd = os.popen(str('sh sense-o/sense-cancel.sh ' + class_transfer_broker_.get_sense_uuid()
        cmd = os.popen(str('sh sense-o/sense-cancel.sh ' + class_transfer_broker_.get_sense_uuid_
    send_email(params, message)
    if 'src_rcsite' in threading.current_thread().name: filename = params.get('FILENAME PARAMETERS
    else: filename = params.get('FILENAME PARAMETERS', 'filename_transfers_dst')
    append_data_to_log_file(filename, action)
    return
```

def start_threads(transfers_tx, transfers_rx)

Function to start thread and monitor the transfers.

Args

transfers_tx : `thread`
tx thread.

transfers_rx : `thread`
rx thread.

▼ EXPAND SOURCE CODE

```
def start_threads(transfers_tx, transfers_rx):
    """Function to start thread and monitor the transfers.

    Args:
        transfers_tx (thread): tx thread.
        transfers_rx (thread): rx thread.
    """
    logging.debug('Starting thread %s.' % transfers_tx.name)
    logging.debug('Starting thread %s.' % transfers_rx.name)
    # Start threads
    transfers_tx.start()
    if not class_transfer_broker_.get_unidirectional(): transfers_rx.start()
    transfers_tx.join()
    transfers_rx.join()
    return
```

Generated by *pdoc* 0.10.0.