```
def run_udf_locals(udf_name,inputs):
    ids_events=[]
    for node in nodes:
        task=Task(ExecUDF, (udf_name,inputs))
        ( id, event)=tasks_queue.push(node, task)
        ids_events.append((id, event))

result=[]
    for id, event in ids_events:
        event.wait(timeout=config.timeout)
    try:
        result.append(results[id])

return result
```

```
class TasksQueue:

def push(node,task):

completion_event=threading.Event()
task_id=task_counter.get_next_id()
self.tasks_queue.push( (task_id, node, completion_event, task) )
return ( task_id, completion_event)
```

tasks_queu	
i	

```
def run_udf_locals(udf_name,inputs):
    ids_events=[]
    for node in nodes:
        task=Task(ExecUDF, (udf_name,inputs))
        ( id, event)=tasks_queue.push(node, task)
        ids_events.append((id, event))
    result=[]
    for id, event in ids_events:
        event.wait(timeout=config.timeout)
        try:
        result.append(results[id])
```

```
class TasksQueue:

def push(node,task):

completion_event=threading.Event()
task_id=task_counter.get_next_id()
self.tasks_queue.push( (task_id, node, completion_event, task) )
return ( task_id, completion_event)
```

tasks_queu

(6, node3, event, task) (5, node2, event, task) (4, node1, event, task)

```
def run_udf_locals(udf_name,inputs):

ids_events=[]
for node in nodes:
    task=Task(ExecUDF, (udf_name,inputs))
    ( id, event)=tasks_queue.push(node, task)
    ids_events.append((id, event))

result=[]
for id, event in ids_events:
    event.wait(timeout=config.timeout)
    try:
    result.append(results[id])

return result
```

```
class TasksQueue:

def push(node,task):

completion_event=threading.Event()
task_id=task_counter.get_next_id()
self.tasks_queue.push( (task_id, node, completion_event, task) )
return ( task_id, completion_event)
```

```
(6, node3, event, task)
(5, node2, event, task)
(4, node1, event, task)
```

```
pop()
```

```
while(!tasks_queue.empty()):
  (id,node, event, task)=tasks_queue.pop() threading.Thread( target=exec_task, args=(id, nodes, event, task) )
```

results[task_ event.set()

def exec_task(task_id, node, event, task):
 response= ## http request to node, with parameters from task ## AWAITED
 results[task_id]=response

```
def run_udf_locals(udf_name,inputs):

ids_events=[]
for node in nodes:
    task=Task(ExecUDF, (udf_name,inputs))
    (id, event)=tasks_queue.push(node, task)
    ids_events.append((id, event))

result=[]
for id, event in ids_events:
    event.wait(timeout=config.timeout)
    try:
    result.append(results[id])

return result
```

```
class TasksQueue:

def push(node,task):

completion_event=threading.Event()
task_id=task_counter.get_next_id()
self.tasks_queue.push((task_id, node, completion_event, task))
return (task_id, completion_event)
```

```
(6, node3, event, task)
(5, node2, event, task)
```

```
while(!tasks_queue.empty()):
    (id,node, event, task)=tasks_queue.pop()
    threading.Thread( target=exec_task, args=(id, nodes, event, task) )
...
```

```
def exec_task(task_id, node, event, task):
    response= ## http request to node, with parameters from task ## ANAITED
    results[task_id]=response
    event.set()
```

```
(4: [table_name])
(3: [ <some response> ])
(2: [ <some response> ])
(1: [ <some response> ])
```

results

```
def run_udf_locals(udf_name,inputs):
                                                              class TasksQueue:
  ids events=[]
                                                                def push(node,task):
  for node in nodes:
     task=Task(ExecUDF, (udf_name,inputs))
                                                                   completion event=threading.Event()
     (id, event)=tasks queue.push(node, task)
                                                                  task id=task counter.get next id()
     ids events.append((id, event))
                                                                   self.tasks queue.push( (task id, node, completion event, task) )
  result=∏
                                                                return ( task id, completion event)
  for id, event in ids events:
     event.wait(timeout=config.timeout)
     try:
       result.append(results[id])
  return result
```

```
response= ## http request to node, with parameters from task ## ANAITED
```

def exec task(task_id, node, event, task):

results[task id]=response

event.set()

```
while(!tasks queue.empty()):
  (id,node, event, task)=tasks queue.pop()
  threading. Thread(target=exec task, args=(id, nodes, event, task))
```

```
results
(4: [table name])
(3: [ <some response> ])
(2: [ <some response> ])
(1: [ <some response> ])
```

tasks queu

(6, node3, event, task)

(5, node2, event, task)

```
def run_udf_locals(udf_name,inputs):
  ids events=[]
  for node in nodes:
     task=Task(ExecUDF, (udf_name,inputs))
     (id, event)=tasks queue.push(node, task)
     ids events.append((id, event))
  for id, event in ids events:
     event.wait(timeout=config.timeout)
       result.append(results[id])
  return result
```

result=∏

try:

```
class TasksQueue:
  def push(node,task):
    completion event=threading.Event()
    task id=task counter.get next id()
    self.tasks queue.push( (task id, node, completion event, task) )
  return ( task id, completion event)
```

```
def exec task(task_id, node, event, task):
  response= ## http request to node, with parameters from task ## ANAITED
  results[task id]=response
  event.set()
```

```
while(!tasks queue.empty()):
  (id,node, event, task)=tasks queue.pop()
  threading.Thread(target=exec task, args=(id, nodes, event, task))
```

results (6: [table name]) (5 : [table name]) (4: [table name]) (3: [<some response>]) (2: [<some response>]) (1: [<some response>])

tasks queu

```
def run udf_locals(udf name,inputs):
                                                          class TasksQueue:
  ids events=[]
                                                             def push(node,task):
  for node in nodes:
    task=Task(ExecUDF, (udf_name,inputs))
                                                               completion event=threading.Event()
    (id, event)=tasks queue.push(node, task)
                                                               task id=task counter.get next id()
    ids events.append((id, event))
                                                               self.tasks queue.push( (task id, node, completion event, task) )
  result=∏
                                                             return ( task id, completion event)
  for id, event in ids events:
    event.wait(timeout=config.timeout)
       result.append(results[id])
  return result
                    [table name,table name,table name]
```

def exec_task(task_id, node, event, task):
 response= ## http request to node, with parameters from task ## AWAITED
 results[task_id]=response
 event.set()

```
while(!tasks_queue.empty()):
    (id,node, event, task)=tasks_queue.pop()
    threading.Thread( target=exec_task, args=(id, nodes, event, task) )
...
```

```
results

(6: [table_name])
(5: [table_name])
(4: [table_name])
(3: [ <some response> ])
(2: [ <some response> ])
(1: [ <some response> ])
```

tasks queu