CSCI/ECEN 5673: Distributed Systems

Programming Assignment 1

Collaborated with Shree Krishna Subburaj

The goal of this assignment is to implement clock synchronization in distributed systems as well as gain experience with using ONC RPC and understand the overhead RPC incurs. Data Collection across versions and scenarios:

1. V1: UDP
2. V2: RPC

Scenarios

1. Localhost
2. LAN
3. Different Geographic Location Tokyo

Data format in **second.nanosecond**:

T0,Ts,T1

T0= The epoch time of the client before sending the request

T1= The epoch time of the client after receiving the response

Ts= The epoch time of the server upon receiving the request.

We've assumed that the difference between T2(time when the request reaches the server) and T3(the time when the response leaves the server) is negligible such that T2=T3=Ts

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pylab
```

In [2]:
```
def GetLatency(filename):
    time_data = open(filename,"r")
    count = 0
    latency_list = []
    while True:
        count += 1
        latency_secs = latency_nsecs = 0
        line = time_data.readline()
        if not line:
            break
        time_line = line.split(",")
        t_1_secs = int(time_line[0].split(".")[0])
        t_1_nsecs = int(time_line[0].split(".")[1])
        t_s_secs = int(time_line[1].split(".")[0])
        t_s_nsecs = int(time_line[1].split(".")[1])
        t_2_secs = int(time_line[2].split(".")[0])
        t_2_nsecs = int(time_line[2].split(".")[1])
        if ((t_2_nsecs - t_1_nsecs) < 0):
            latency_secs = t_2_secs - t_1_secs - 1
            latency_nsecs = t_2_nsecs - t_1_nsecs + 1000000000
        else:
            latency_secs = t_2_secs - t_1_secs
            latency_nsecs = t_2_nsecs - t_1_nsecs
        latency_list.append(latency_secs*1000000000 + latency_nsecs)
    return latency_list
```

Offset and Delay formula

$o_i = T_s - \frac{T_1+T_0}{2}$

$d_i = T_1 - T_0$

In [3]:
```
def GetDelayOffset(filename):
    time_data = open(filename,"r")
    count = 0
    delay_list = []
    offset_list = []
    while True:
        count += 1
        line = time_data.readline()
        if not line:
            break
        time_line = line.split(",")
        t_1_secs = int(time_line[0].split(".")[0])
        t_1_nsecs = int(time_line[0].split(".")[1])
        t_s_secs = int(time_line[1].split(".")[0])
        t_s_nsecs = int(time_line[1].split(".")[1])
        t_2_secs = int(time_line[2].split(".")[0])
        t_2_nsecs = int(time_line[2].split(".")[1])
        t_s = t_s_secs*1000000000 + t_s_nsecs
        t_1 = t_1_secs*1000000000 + t_1_nsecs
        t_2 = t_2_secs*1000000000 + t_2_nsecs
        delay_list.append(t_2 - t_1)
        offset_list.append((2*t_s - t_1 - t_2)/2)
    return (delay_list, offset_list)
```

Christans Clock Estimate Formula:

$estimate\ T_{new} = T_s + \frac{T_1 - T_0}{2}$

In [4]:
```
def GetClockDiff(filename):
    time_data = open(filename,"r")
    count = 0
    diff = []
    while True:
        count += 1
        line = time_data.readline()
        if not line:
            break
        time_line = line.split(",")
        t_1_secs = int(time_line[0].split(".")[0])
        t_1_nsecs = int(time_line[0].split(".")[1])
        t_s_secs = int(time_line[1].split(".")[0])
        t_s_nsecs = int(time_line[1].split(".")[1])
        t_2_secs = int(time_line[2].split(".")[0])
        t_2_nsecs = int(time_line[2].split(".")[1])
        t_s = t_s_secs*1000000000 + t_s_nsecs
        t_1 = t_1_secs*1000000000 + t_1_nsecs
        t_2 = t_2_secs*1000000000 + t_2_nsecs
        estimate = t_s + (t_2-t_1)/2
        diff.append(t_2 - estimate)
    return diff
```

In [5]:
```
def GetTMin(filename):
    time_data = open(filename,"r")
    count = 0
    t_min = float('inf')
    while True:
        count += 1
        line = time_data.readline()
        if not line:
            break
        time_line = line.split(",")
        t_1_secs = int(time_line[0].split(".")[0])
        t_1_nsecs = int(time_line[0].split(".")[1])
        t_s_secs = int(time_line[1].split(".")[0])
        t_s_nsecs = int(time_line[1].split(".")[1])
        t_2_secs = int(time_line[2].split(".")[0])
        t_2_nsecs = int(time_line[2].split(".")[1])
        t_s = t_s_secs*1000000000 + t_s_nsecs
        t_1 = t_1_secs*1000000000 + t_1_nsecs
        t_2 = t_2_secs*1000000000 + t_2_nsecs
        t_min = min(abs(t_s-t_1), abs(t_2-t_1), t_min)
    return t_min
```

Christans Clock Error Bounds Formula:

$Errorbounds\ = \pm(\frac{T_1 - T_0}{2} - T_{min})$

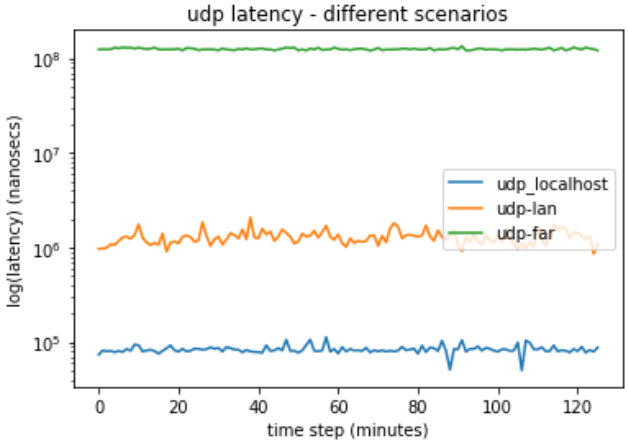
In [6]:
```
def GetErrorBound(filename, t_min):
    time_data = open(filename,"r")
    count = 0
    error_bounds = []
    while True:
        count += 1
        line = time_data.readline()
        if not line:
            break
        time_line = line.split(",")
        t_1_secs = int(time_line[0].split(".")[0])
        t_1_nsecs = int(time_line[0].split(".")[1])
        t_s_secs = int(time_line[1].split(".")[0])
        t_s_nsecs = int(time_line[1].split(".")[1])
        t_2_secs = int(time_line[2].split(".")[0])
        t_2_nsecs = int(time_line[2].split(".")[1])
        t_s = t_s_secs*1000000000 + t_s_nsecs
        t_1 = t_1_secs*1000000000 + t_1_nsecs
        t_2 = t_2_secs*1000000000 + t_2_nsecs
        error_bounds.append((t_2-t_1)/2 - t_min)
    return np.mean(error_bounds)
```

Question 1

Compute roundtrip latencies along with average and standard deviation for each scenario, and plot them in a graph. Provide an analysis of your results in terms of why there is a variation in latencies, which ones you expect to be more accurate, etc.

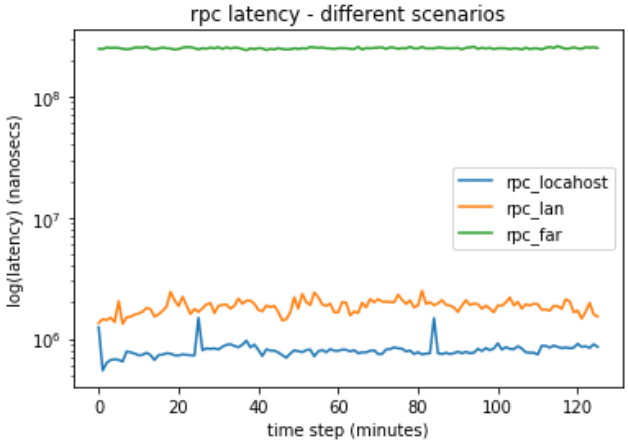
In [7]:
```
files_v1 = ["udp_localhost//udp_localhost.txt", "udp-lan//udp_lan.txt", "udp-fa
r//udp_far.txt"]
print("UDP :: \n","-"*70)
for file in files_v1:
    print(file.split("//")[0], " ::")
    latency_list = GetLatency(file)
    print("Round trip latency mean : ", np.mean(latency_list))
    print("Round trip latency std deviation : ", np.std(latency_list))
    plt.title('udp latency - different scenarios')
    plt.yscale('log')
    plt.xlabel('time step (minutes)')
    plt.ylabel('log(latency) (nanosecs)')
    plt.plot(latency_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP ::
 ----------------------------------------------------------------------
udp_localhost  ::
Round trip latency mean :  83596.03174603175
Round trip latency std deviation :  7898.850681854309
udp-lan  ::
Round trip latency mean :  1285351.134920635
Round trip latency std deviation :  209754.83768832698
udp-far  ::
Round trip latency mean :  125355141.78571428
Round trip latency std deviation :  2637809.1931893555
```



In [8]:
```
files_v2 = ["rpc_locahost//rpc_localhost.txt", "rpc_lan//rpc_lan.txt", "rpc_far
//rpc_far.txt"]
print("RPC :: \n","-"*70)
for file in files_v2:
    print(file.split("//")[0], " ::")
    latency_list = GetLatency(file)
    print("Round trip latency mean : ", np.mean(latency_list))
    print("Round trip latency std deviation : ", np.std(latency_list))
    plt.title('rpc latency - different scenarios')
    plt.yscale('log')
    plt.xlabel('time step (minutes)')
    plt.ylabel('log(latency) (nanosecs)')
    plt.plot(latency_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
RPC ::
 ----------------------------------------------------------------------
rpc_locahost  ::
Round trip latency mean :  807674.4682539683
Round trip latency std deviation :  113669.79892267783
rpc_lan  ::
Round trip latency mean :  1854724.3095238095
Round trip latency std deviation :  227311.04870815453
rpc_far  ::
Round trip latency mean :  251106498.76984128
Round trip latency std deviation :  3562368.874666962
```



Analysis

From the above graph we note that, the latency for localhost is minimum, whereas the latency for a different geographical location is maximum.

latency(localhost) < latency(LAN) < latency(Tokyo)

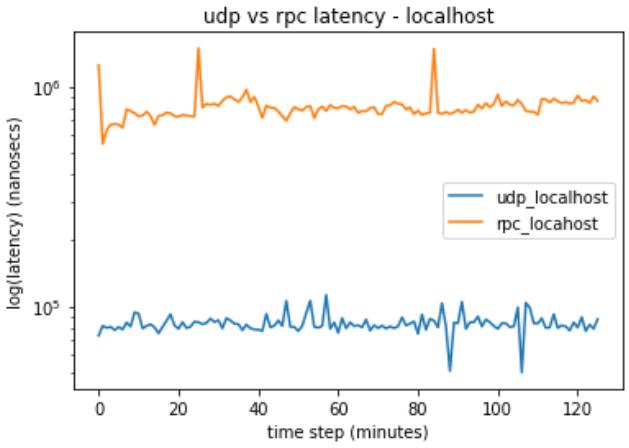
The network latency is directly propotional to the distance between the client and server
This is an expected result since the network latency for localhost is almost zero whereas the packet must travel across multiple hops to get from the client(USA) to the server(Tokyo). The latency for LAN falls between these two scenarios.

Question 2

How much performance overhead does RPC incur under different scenarios?
Provide an explanation of this overhead. Your answer must be based on the data you have collected.

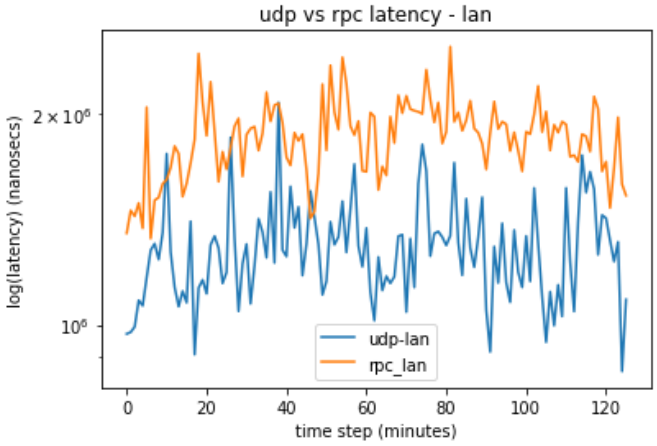
In [9]:
```
files = ["udp_localhost//udp_localhost.txt", "rpc_locahost//rpc_localhost.txt"]
print("UDP RPC localhost comparison:: \n","-"*70)
for file in files:
    print(file.split("//")[0], " ::")
    latency_list = GetLatency(file)
    print("Round trip latency mean : ", np.mean(latency_list))
    print("Round trip latency std deviation : ", np.std(latency_list))
    plt.title('udp vs rpc latency - localhost')
    plt.yscale('log')
    plt.xlabel('time step (minutes)')
    plt.ylabel('log(latency) (nanosecs)')
    plt.plot(latency_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP RPC localhost comparison::
 ----------------------------------------------------------------------
udp_localhost  ::
Round trip latency mean :  83596.03174603175
Round trip latency std deviation :  7898.850681854309
rpc_locahost  ::
Round trip latency mean :  807674.4682539683
Round trip latency std deviation :  113669.79892267783
```



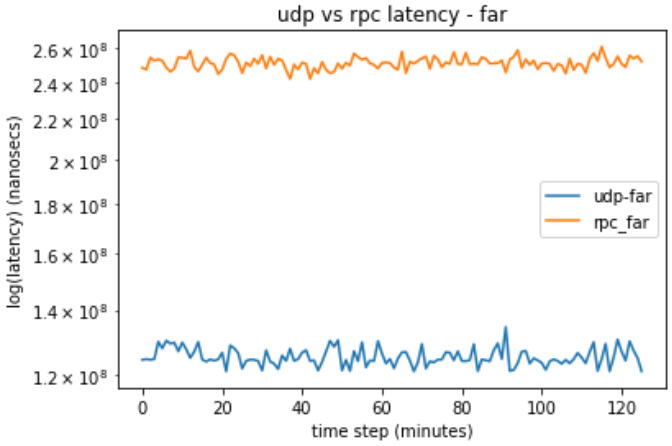
In [10]:
```
files = ["udp-lan//udp_lan.txt", "rpc_lan//rpc_lan.txt"]
print("UDP RPC localhost comparison:: \n","-"*70)
for file in files:
    print(file.split("//")[0], " ::")
    latency_list = GetLatency(file)
    print("Round trip latency mean : ", np.mean(latency_list))
    print("Round trip latency std deviation : ", np.std(latency_list))
    plt.title('udp vs rpc latency - lan')
    plt.yscale('log')
    plt.xlabel('time step (minutes)')
    plt.ylabel('log(latency) (nanosecs)')
    plt.plot(latency_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP RPC localhost comparison::
 ----------------------------------------------------------------------
udp-lan  ::
Round trip latency mean :  1285351.134920635
Round trip latency std deviation :  209754.83768832698
rpc_lan  ::
Round trip latency mean :  1854724.3095238095
Round trip latency std deviation :  227311.04870815453
```



In [11]:
```
files = ["udp-far//udp_far.txt", "rpc_far//rpc_far.txt"]
print("UDP RPC localhost comparison:: \n","-"*70)
for file in files:
    print(file.split("//")[0], " ::")
    latency_list = GetLatency(file)
    print("Round trip latency mean : ", np.mean(latency_list))
    print("Round trip latency std deviation : ", np.std(latency_list))
    plt.title('udp vs rpc latency - far')
    plt.yscale('log')
    plt.xlabel('time step (minutes)')
    plt.ylabel('log(latency) (nanosecs)')
    plt.plot(latency_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP RPC localhost comparison::
 ----------------------------------------------------------------------
udp-far  ::
Round trip latency mean :  125355141.78571428
Round trip latency std deviation :  2637809.1931893555
rpc_far  ::
Round trip latency mean :  251106498.76984128
Round trip latency std deviation :  3562368.874666962
```



Analysis

From the above graphs, we conclude that RPC incurs an performance overhead compared to raw UDP. We attribute this performance overhead to the marshalling and unmarshalling of data across client server stubs as well as the portmapper service discovery that the client initiates to locate the port associated with the service(Program,Version) hosted on the server.

Mean performance difference between RPC and UDP for localhost is :: $807674 - 83596 = 724078$ns
Mean performance difference between RPC and UDP for LAN is :: $1854724 - 1285351 = 569373$ns
Mean performance difference between RPC and UDP for different continent is ::
$251106498 - 125355141 = 125751357$ns

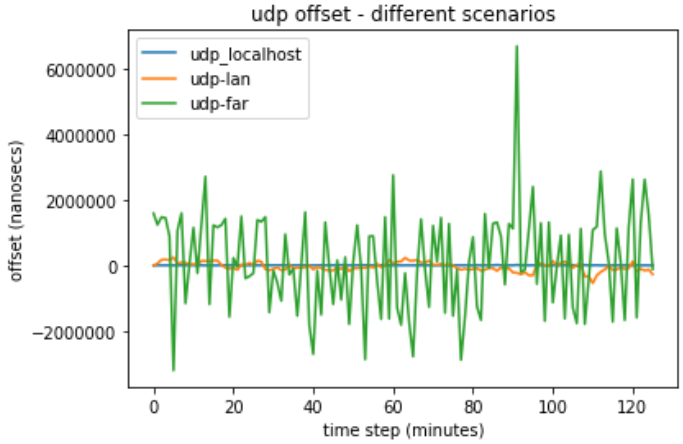
Question 3

Compute the offset (oi) and delay (di) for each of the measurements for each scenario using the
NTP formula and plot them in a graph (x-axis: measurement #; y-axis: oi or di).
Provide an analysis of your results.
What difference do you see in your estimates between the two versions of your program?
Based on your data, can you justify the statement:
The shorter and more symmetric the round-trip time is, the more accurate the estimate of the current time.

In [12]:
```
files_v1 = ["udp_localhost//udp_localhost.txt", "udp-lan//udp_lan.txt", "udp-fa
r//udp_far.txt"]
print("UDP :: \n","-"*70)
for file in files_v1:
    delay_list, offset_list = GetDelayOffset(file)
    #fig = plt.figure()
    #ax = fig.add_subplot(2, 1, 1)
    #line, = ax.plot(offset_list, color='blue', lw=2)
    #ax.set_yscale('log')
    #pylab.show()

    plt.title('udp offset - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('offset (nanosecs)')
    plt.plot(offset_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP ::
 ----------------------------------------------------------------------
```



The above graph represents the time offset between the client and server using raw UDP.

We observe that the offset depends on the formula: $o_i = T_s - \frac{T_1+T_0}{2}$

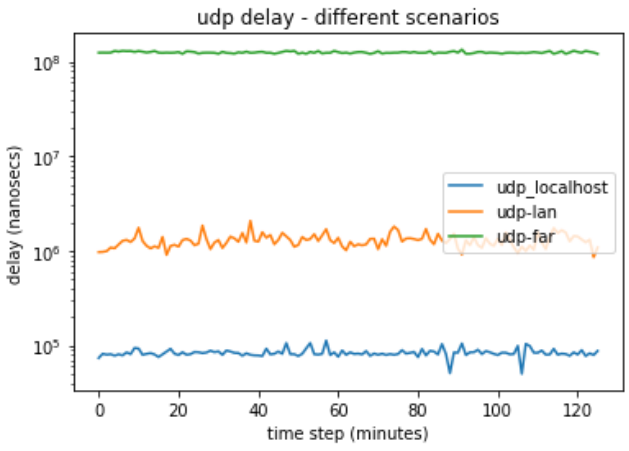
While udp-localhost and udp-lan have low variance, udp-far (Tokyo) shows high variance due to the delay ($d = T_1 - T_0$) in packet transfer across continents. We also observe that the offset varies between positive and negative values. We attribute this behaviour to the way udp packets are handled between the server and client. Since the client is aware of the port on which the service is hosted, the client initiates the time request immediately. Now when we have THREE cases:

1. Positive offset: When the packet suffers delay from client to server, the time at which the server invokes the clock_gettime() function to fetch the time is delayed after the point $\frac{T_1+T_0}{2}$ (ie closer to T_1). Hence we obtain a positive offset.
2. Negative offset: When the packet reaches the server quickly, the server invokes clock_gettime() to create the response packet. Now if the response packet suffers delay from the server to client, T_s occurs before the point $\frac{T_1+T_0}{2}$. Hence we obtain a negative delay.
3. Zero offset: For symmetric packet transfer ie(the time taken from the client to server is equal to the time taken from server to client), $T_s = \frac{T_1+T_0}{2}$

In [13]:
```
files_v1 = ["udp_localhost//udp_localhost.txt", "udp-lan//udp_lan.txt", "udp-far//udp_far.txt"]
print("UDP :: \n","-"*70)
for file in files_v1:
    delay_list, offset_list = GetDelayOffset(file)
    #fig = plt.figure()
    #ax = fig.add_subplot(2, 1, 1)
    #line, = ax.plot(delay_list, color='blue', lw=2)
    #ax.set_yscale('log')
    plt.yscale('log')

    plt.title('udp delay - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('delay (nanosecs)')
    plt.plot(delay_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
    #pylab.show()
```

```
UDP ::
 ----------------------------------------------------------------------
```

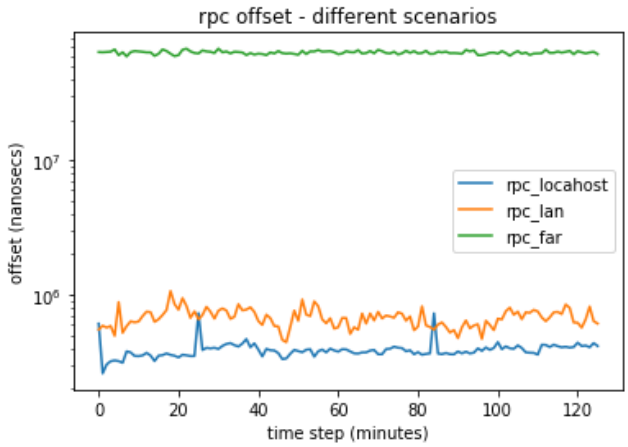


We observe that the delay graph is the same at the latency graph

In [14]:
```
files_v2 = ["rpc_locahost//rpc_localhost.txt", "rpc_lan//rpc_lan.txt", "rpc_far
//rpc_far.txt"]#, "tcp_rpclocal.txt"]
print("RPC :: \n","-"*70)
for file in files_v2:
    delay_list, offset_list = GetDelayOffset(file)
    #fig = plt.figure()
    #ax = fig.add_subplot(2, 1, 1)
    #line, = ax.plot(offset_list, color='blue', lw=2)
    #ax.set_yscale('log')
    #pylab.show()
    plt.yscale('log')

    plt.title('rpc offset - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('offset (nanosecs)')
    plt.plot(offset_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
RPC ::
 ----------------------------------------------------------------------
```



In the case of RPC, when we calculate the time offset between the client and server, we observe that the offset in almost all cases is positive. We recognize that the offset between a client and server can be either positive, negative or zero. However in our case, since we run the client and server programs on Cloud VM instances we can confidently say that both the client and server machines hold accurate UTC time.

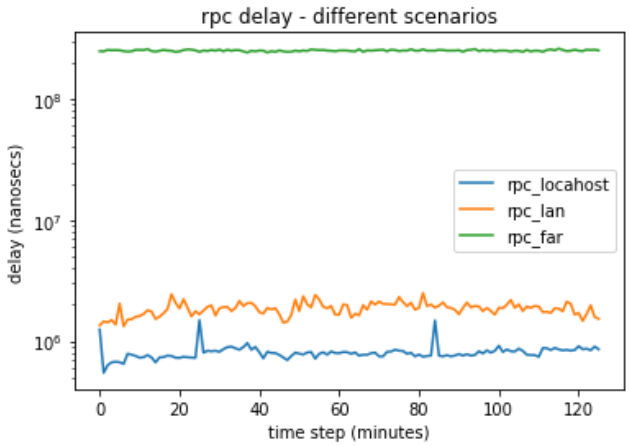
In such a scenario we attribute the positive offset in the RPC case to the the additional overhead that RPC requires to setup the connection. Other than marshalling of data at the client stub, the client is unaware of the port on which the service is hosted. In order to get the correct port number associated with the invoked function (Program, Version), the client stub first contacts the portmapper service running on the server machine at port 111.

This additional delay required to obtain the corresponding port for the program invoked, pushes the T_s after the point $\frac{T_1+T_0}{2}$ (ie toward T_1) in all time steps.

In [15]:
```
files_v2 = ["rpc_locahost//rpc_localhost.txt", "rpc_lan//rpc_lan.txt", "rpc_far
//rpc_far.txt"]
print("RPC :: \n","-"*70)
for file in files_v2:
    delay_list, offset_list = GetDelayOffset(file)
    plt.yscale('log')

    plt.title('rpc delay - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('delay (nanosecs)')
    plt.plot(delay_list, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
RPC ::
 ----------------------------------------------------------------------
```



We observe that the delay graph is the same at the latency graph

Based on the above data, we can say that the symmetric nature of packet transfer provides accurate estimates of the server time, since the offset is directly proportional to difference between the the point when the server invokes the clock_gettime() function T_s and the midpoint of the latency $\frac{T_1+T_0}{2}$.

Now if the packet takes exactly $\frac{T_1+T_0}{2}$ seconds from the client to server and once again $\frac{T_1+T_0}{2}$ seconds from the server to client, the offset calculated will be the exact difference between ther server time and client time. This way we get a more accurate estimate of the server time. We also prefer short round-trip latencies since this reduces the delay factor which is the error bounds for the offset. In the event the delay is negligible (localhost), the offset calculated will have a high degree of accuracy and thereby yeild an accurate time estimate.

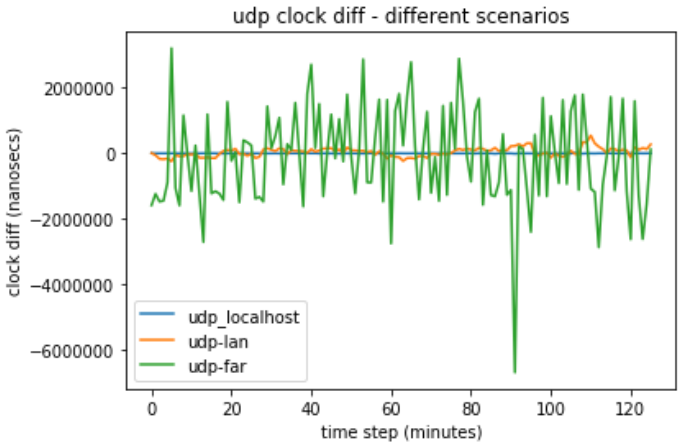
We see this behaviour in the case of UDP-localhost where the delay is almost negligible and the offset is symmetric about the x-axis. Whereas in the case of RPC, since the offset is usually postive ie asymmetric offset and the delay is considerable due to performance overhead(marshalling and service discovery), we obtain a low degree of accuracy for the time estimate.

Question 4

Compute server clock time estimate using the Cristian's clock synchronization algorithm,
and plot the difference between the local clock and the estimated server clock values for each scenario.
Based on your observations, what is a reasonable estimate of absolute minimum latency between the two machines you used for experiments for different scenarios.
Using this estimate, calculate the error bounds for the synchronized time.

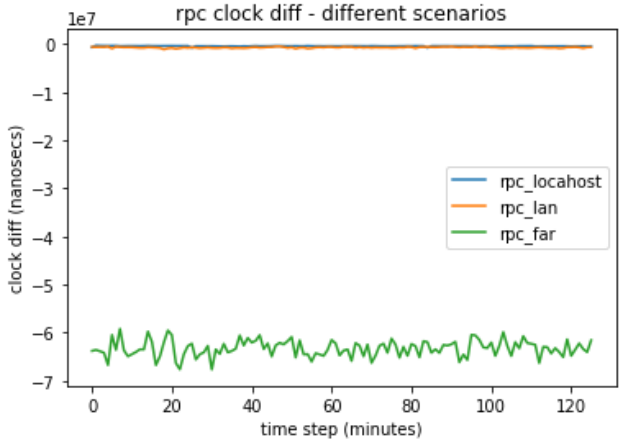
In [16]:
```
files_v1 = ["udp_localhost//udp_localhost.txt", "udp-lan//udp_lan.txt", "udp-far//udp_far.txt"]
print("UDP :: \n","-"*70)
for file in files_v1:
    diff = GetClockDiff(file)
    plt.title('udp clock diff - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('clock diff (nanosecs)')
    plt.plot(diff, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
UDP ::
 ----------------------------------------------------------------------
```



In [17]:
```
files_v2 = ["rpc_locahost//rpc_localhost.txt", "rpc_lan//rpc_lan.txt", "rpc_far
//rpc_far.txt"]
print("RPC :: \n","-"*70)
for file in files_v2:
    diff = GetClockDiff(file)
    plt.title('rpc clock diff - different scenarios')
    plt.xlabel('time step (minutes)')
    plt.ylabel('clock diff (nanosecs)')
    plt.plot(diff, label = file.split("//")[0])
    plt.legend(loc = 'best')
```

```
RPC ::
 ----------------------------------------------------------------------
```



Analysis

The difference between client and server clock values is calculated as:

$$T_1 - (T_s + \frac{T_1 - T_0}{2}) = \frac{T_0 + T_1}{2} - T_s$$

The minimum latency T_{min} was calculated by finding the minimum transmission delay to and from the server over 120 minutes for each of the scenarios. Following this, the error bounds for each scenario was calculated as $\pm(\frac{T_1 - T_0}{2} - T_{min})$

In [18]:
```
files_v2 = ["rpc_locahost//rpc_localhost.txt", "rpc_lan//rpc_lan.txt", "rpc_far
//rpc_far.txt"]
print("RPC :: \n","-"*70)
for file in files_v2:
    t_min = GetTMin(file)
    print(file.split("//")[0])
    print("T min ", " :: ", t_min)
    error_bound = GetErrorBound(file, t_min)
    print("Error bounds are ", " :: (-", abs(error_bound), ", +", abs(error_bou
nd), ")")
```

```
RPC ::
 ----------------------------------------------------------------------
rpc_locahost
T min   ::  533658
Error bounds are   :: (- 129820.76587301587 , + 129820.76587301587 )
rpc_lan
T min   ::  1168691
Error bounds are   :: (- 241328.84523809524 , + 241328.84523809524 )
rpc_far
T min   ::  181487574
Error bounds are   :: (- 55934324.615079366 , + 55934324.615079366 )
```

In [19]:
```
files_v1 = ["udp_localhost//udp_localhost.txt", "udp-lan//udp_lan.txt", "udp-fa
r//udp_far.txt"]
print("UDP :: \n","-"*70)
for file in files_v1:
    t_min = GetTMin(file)
    print(file.split("//")[0])
    print("T min ", " :: ", t_min)
    error_bound = GetErrorBound(file, t_min)
    print("Error bounds are ", " :: (-", abs(error_bound), ", +", abs(error_bou
nd), ")")
```

```
UDP ::
 ----------------------------------------------------------------------
udp_localhost
T min   ::  34583
Error bounds are   :: (- 7215.015873015873 , + 7215.015873015873 )
udp-lan
T min   ::  205778
Error bounds are   :: (- 436897.56746031746 , + 436897.56746031746 )
udp-far
T min   ::  60396472
Error bounds are   :: (- 2281098.8928571427 , + 2281098.8928571427 )
```