Improving Indoor Localization using Filtering

Giuseppe D'Ambrosio
Dipartimento di Informatica
Università degli Studi di Salerno, Salerno, Italy
gdambrosio@unisa.it

1 Introduction

Indoor localization regards the technologies and methods to obtain information associated with a particular position within buildings. The necessity to track people and objects is getting increased interest in academia and commercial areas for several reasons. The major growth factors are the increased number of smartphone-connected devices and location-aware technologies, as well as the pervasive presence of sensors and smart objects, enabling the integration of indoor localization in countless applications.

The techniques used to estimate the position exploit signal characteristics analysis, but working in an enclosed space makes the estimation challenging since the presence of obstacles, walls, and even human bodies influence the signal increasing the interference [1].

One of the most common measures used in Indoor Localization is the Received Signal Strength Indicator (RSSI). The RSSI value indicates the power signal strength of a received radio signal. The higher the value, the smaller the distance between the sender and the receiver. RSSI has been widely used because it does not require extra hardware to be computed and is easy to implement because it can be used with common communication protocols like Bluetooth and Wi-Fi. However, a significant drawback is a high vulnerability to interference, a very common situation in indoor environments.

Since RSSI is based on radio waves transmission, its value is highly influenced by absorption, interference, and diffraction effects. Even small fluctuations can produce meaningful changes in position estimation and prediction, lowering their precision, reliability, and efficiency [2]. Therefore, the use of RSSI for indoor localization requires the application of special filters to make RSSI amplitude lower.

In this work, we apply different mathematical filtering functions to reduce the noise within an RSSI set of values to smooth the signal, improve accuracy, and compare the results. Several works proved that filtering techniques could relevantly enhance the efficiency and effectiveness of prediction when used on training data [2].

In literature, different filters exist, and the choice of the most appropriate to adopt should be made based on the specific requirements. Indeed each filtering solution has peculiar characteristics in terms of computational complexity and precision improvement. We applied filtering algorithms based on grey system theory, Kalman filter, Fourier transforms, and particle filters.

1.1 Data set

The data set used consists of the RSSI values collected by seven BLE-equipped beacons located into different points of a limited area of a building for twenty minutes. Two smartphones continuously send BLE packets which RSSI is used to locate them inside the room [3]. The data set contains information about the beacon's name, a location status representing one of the possible beacon locations in the form of a string, a timestamp registered in milliseconds, and an RSSI value for each smartphone.

2 Preliminary Phase

In this phase, we analyze the shape of the data set to understand if some data preparation process is needed.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv('../dataset/raw_data.csv')
```

[2]:
```
df
```

[2]:

	name	locationStatus	timestamp	rssiOne	rssiTwo
0	FF:FF:C0:21:F1:31	INSIDE	1551367495147	-90	-90
1	FF:FF:C0:21:EF:BA	INSIDE	1551367496681	-93	-93
2	FF:FF:C0:21:EF:1A	INSIDE	1551367495114	-88	-88
3	FF:FF:C0:21:EF:BF	INSIDE	1551367497508	-82	-82
4	FF:FF:C0:21:F1:31	INSIDE	1551367496906	-94	-94
...	...	...	...	...	...
13579	FF:FF:C2:1D:9B:71	OUTSIDE	1551368693080	-93	-93
13580	FF:FF:C2:1D:9B:52	OUTSIDE	1551368694572	-80	-80
13581	FF:FF:C0:22:3D:C1	OUTSIDE	1551368694032	-89	-89
13582	FF:FF:C2:1D:9B:54	OUTSIDE	1551368694210	-78	-78
13583	FF:FF:C2:1D:9B:71	OUTSIDE	1551368694011	-91	-91

[13584 rows x 5 columns]

[3]:
```
df.isna().any()
```

[3]:
```
name              False
locationStatus    False
timestamp         False
rssiOne           False
rssiTwo           False
dtype: bool
```

The data set has 5 column, 13.584 rows and does not contains null values.

2.1 Data Preparation

2.1.1 Timestamp

The timestamp needs to be transformed from milliseconds form to a standard format that can be used as the index. Specifically, we apply the structure based on hours, minutes, seconds, and milliseconds (`hh:mm:ss.ss`).

[4]:
```
df['Time'] = pd.to_datetime(df['timestamp'], unit='ms').dt.strftime('%H:%M:%S.%f')
```

2.1.2 Data cleaning

In this part, we remove the columns that are not required for our scope. Specifically, the name of the beacon, the location status, and the timestamp in milliseconds.

[5]:
```
df = df.drop(columns=['name', 'locationStatus', 'timestamp'])
df
```

[5]:

	rssiOne	rssiTwo	Time
0	-90	-90	15:24:55.147000
1	-93	-93	15:24:56.681000
2	-88	-88	15:24:55.114000
3	-82	-82	15:24:57.508000
4	-94	-94	15:24:56.906000
...	...	...	...
13579	-93	-93	15:44:53.080000
13580	-80	-80	15:44:54.572000
13581	-89	-89	15:44:54.032000
13582	-78	-78	15:44:54.210000
13583	-91	-91	15:44:54.011000

[13584 rows x 3 columns]

[6]:
```
df.describe()
```

[6]:

	rssiOne	rssiTwo
count	13584.000000	13584.000000
mean	-83.904299	-83.904299
std	7.609678	7.609678
min	-105.000000	-105.000000
25%	-90.000000	-90.000000
50%	-85.000000	-85.000000
75%	-78.000000	-78.000000
max	-56.000000	-56.000000

Exploring the data set, we notice that the two columns referring to the RSSI registered from each smartphone seem to have the same values. We perform so further investigation to check that.

[7]:
```
df['Diff'] = df['rssiOne'] - df['rssiTwo']
df.loc[(df['Diff'] !=0)]
```

[7]:
```
Empty DataFrame
Columns: [rssiOne, rssiTwo, Time, Diff]
Index: []
```

The investigation confirms that there are no differences between the values registered by the two smartphones. Therefore, we can drop one column and reshape the data set to enhance readability and usability.

[8]:
```
df = df.drop(columns=['rssiTwo', 'Diff'])
df.rename(columns={'rssiOne':'rssi'}, inplace=True)
df
```

[8]:

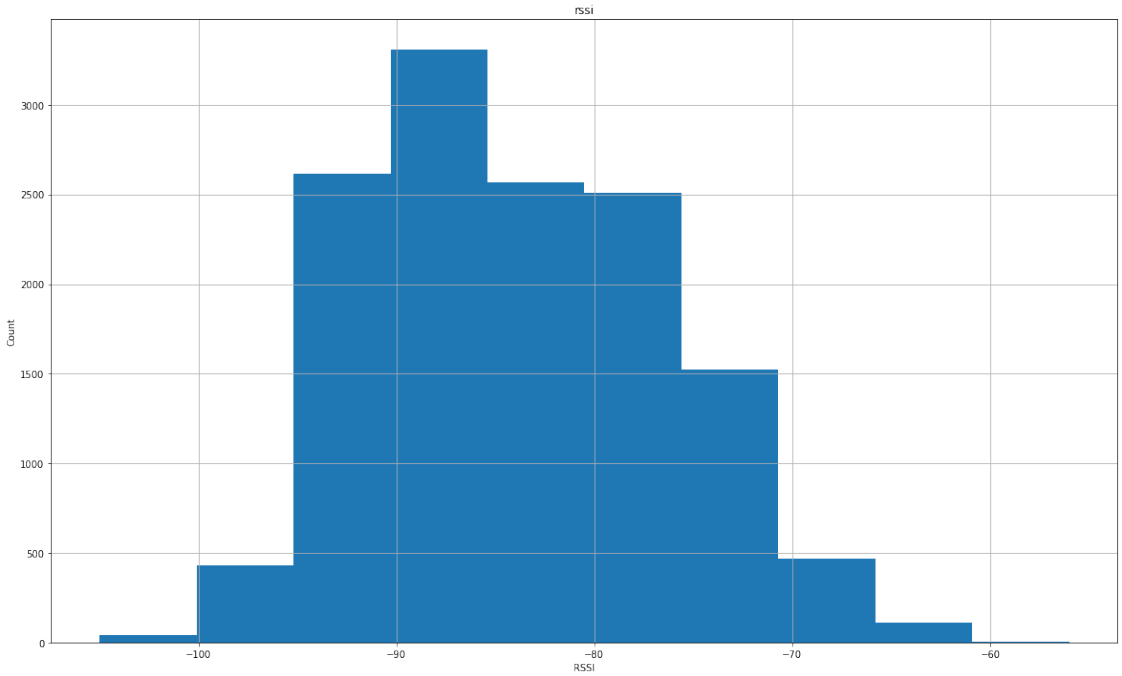
	rssi	Time
0	-90	15:24:55.147000
1	-93	15:24:56.681000
2	-88	15:24:55.114000
3	-82	15:24:57.508000
4	-94	15:24:56.906000
...	...	...
13579	-93	15:44:53.080000
13580	-80	15:44:54.572000
13581	-89	15:44:54.032000
13582	-78	15:44:54.210000
13583	-91	15:44:54.011000

[13584 rows x 2 columns]

2.2 Plotting

We create some basic plots to further understand the data set.

[9]:
```
histCount = df.hist(column='rssi', figsize=(20, 12))
for ax in histCount.flatten():
    ax.set_xlabel('RSSI')
    ax.set_ylabel('Count')
```



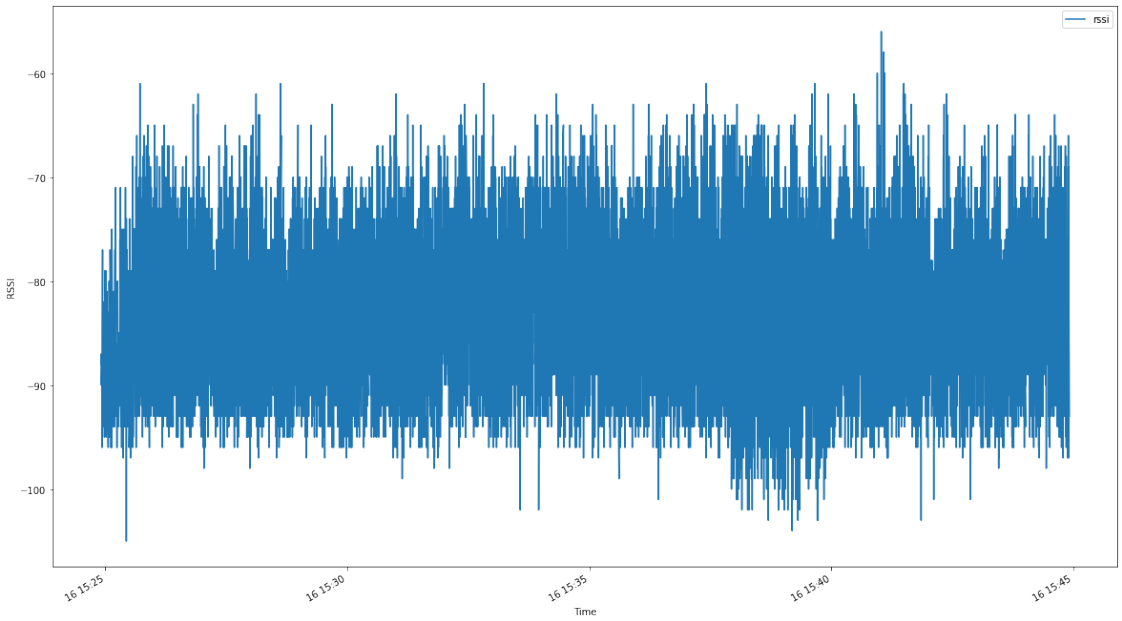
[10]:
```
df.describe()
```

[10]:
```
              rssi
count  13584.000000
mean     -83.904299
std        7.609678
min     -105.000000
25%      -90.000000
50%      -85.000000
75%      -78.000000
max      -56.000000
```

The table and the figure show that most of the signals received are on the lower end of the spectrum, i.e., the beacon gets the majority of signals with low power intensity. The max signal strength observed is -55 *dBm* (decibel milliwatt), which is considered a high enough value for most real work applications. Moreover, the third percentile has a value of -78 *dBm* reporting an acceptable coverage.

[11]:
```
df['Time'] = df['Time'].apply(pd.Timestamp)
df.plot(x='Time',xlabel='Time', y='rssi', ylabel='RSSI', figsize=(20, 12))
```

[11]:
```
<AxesSubplot:xlabel='Time', ylabel='RSSI'>
```



The plot shows a high fluctuation in the signal revealing the presence of a lot of noise, as expected in an indoor environment. Therefore we need to reduce the interference using techniques to smooth the signal. Different possible methods can be used, and the optimal solution highly depends on the specific context and the degree of noise present.

[12]:
```
df.to_csv('../dataset/clean_data.csv', index=False)
```

3 Resampling

A method to reduce high-frequency noise is resampling. Resampling methods refer to the process of changing the frequency of some data to produce a data set with a different cadence. The data set analyzed has several RSSI values for each second. However, this granularity is not necessary for localization purposes since people cannot completely change position in such a short time. Therefore, we can reduce some of the interference by lowering the frequency of each measurement.

3.1 Downsampling

The process of decreasing the frequency of a time series is called downsampling. Two main aspects are fundamental in this process: the **frequency** to which resample the data and the summary statistics to group the data, i.e., an **aggregate function**.

[13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv('../dataset/clean_data.csv')
df['Time'] = df['Time'].apply(pd.Timestamp)
```

3.1.1 Frequency

The first decision is the resolution of the resample, i.e., the new frequency of the measurements. Since the data set contains RSSI values from a smartphone carried by someone, an interval of few seconds might be suitable. We will try with a **one second** frequency and a **two seconds** frequency.

3.1.2 Aggregate function

The second decision regards the aggregation function. There are different options like *sum*, *mean*, *median*, but in the context of indoor localization **mean** and **median** seem the most appropriate.

One second frequency Line 1 shows the resampling method to produce a new time series with a one-second frequency. The RSSI value of each is computed taking the **means** over each second interval.

[15]:
```
1 mean_df_one = df.resample('1S', on='Time').mean()
2 mean_df_one
```

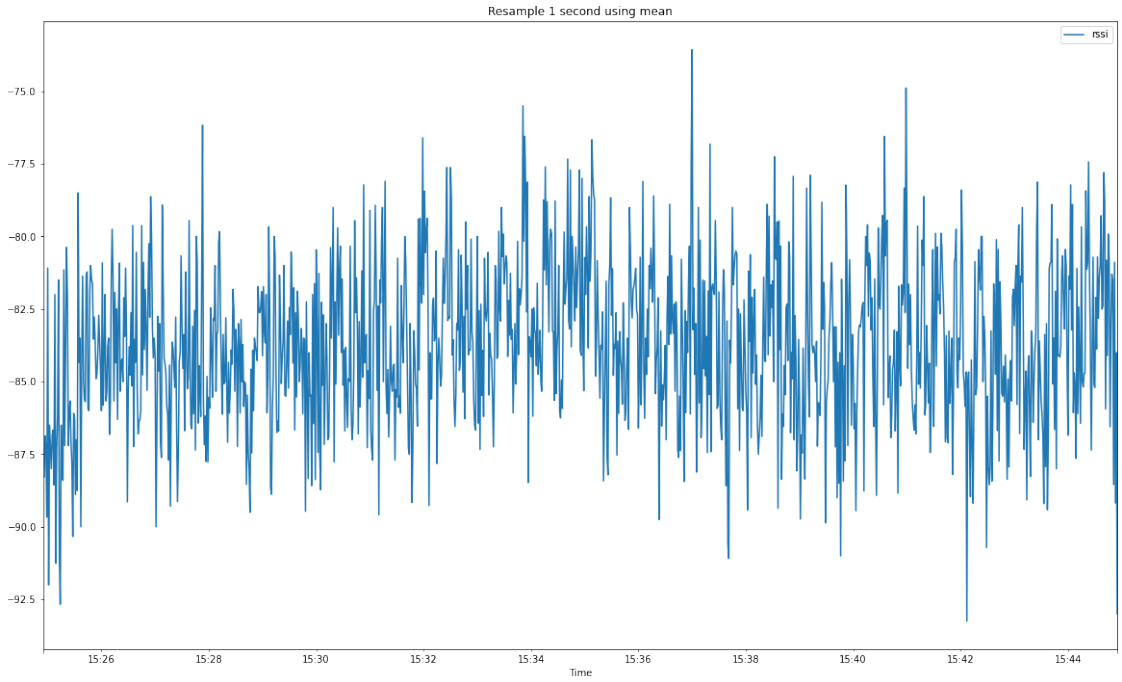
[15]:
```
                         rssi
Time
2021-07-16 15:24:55 -88.250000
2021-07-16 15:24:56 -88.285714
2021-07-16 15:24:57 -86.875000
2021-07-16 15:24:58 -87.250000
2021-07-16 15:24:59 -89.666667
...                         ...
2021-07-16 15:44:51 -88.545455
2021-07-16 15:44:52 -80.888889
2021-07-16 15:44:53 -89.181818
2021-07-16 15:44:54 -84.000000
2021-07-16 15:44:55 -93.000000

[1201 rows x 1 columns]
```

It is interesting to notice that the number of rows is reduced from 13.584 to 1201, showing that the initial data set contains an average of 11 RSSI values for each second.

[16]:
```
1 mean_df_one.plot(title='Resample 1 second using mean', figsize=(20, 12))
```

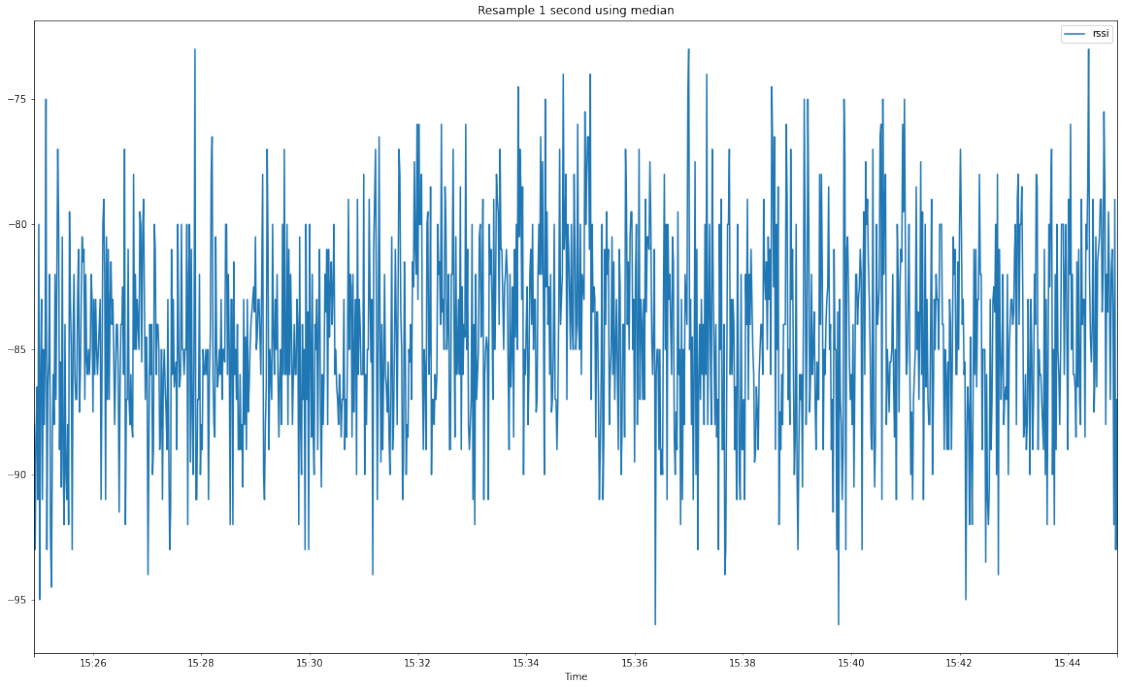
[16]:
```
<AxesSubplot:title={'center':'Resample 1 second using mean'}, xlabel='Time'>
```



Line 1 shows the use of the resampling method to produce a new time series with a one-second frequency. The RSSI value of each interval is computed taking the **medians** over each second window.

[17]:
```
1  median_df_one = df.resample('1S', on='Time').median()
2  median_df_one.plot(title='Resample 1 second using median', figsize=(20, 12))
```

[17]: `<AxesSubplot:title={'center':'Resample 1 second using median'}, xlabel='Time'>`



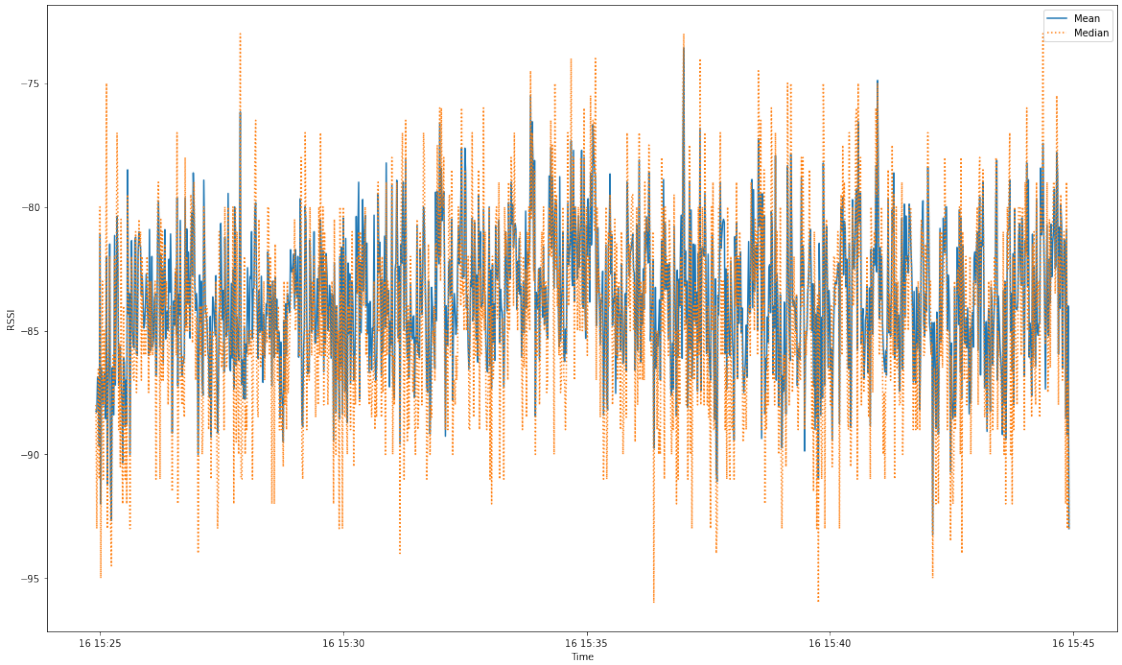
We compare the two downsampled data obtained using a line chart.

[18]:
```
plt.figure(figsize=(20, 12))

plt.plot(mean_df_one['rssi'], label='Mean')
plt.plot(median_df_one['rssi'], linestyle='dotted',  label='Median')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[18]:
```
Text(0, 0.5, 'RSSI')
```

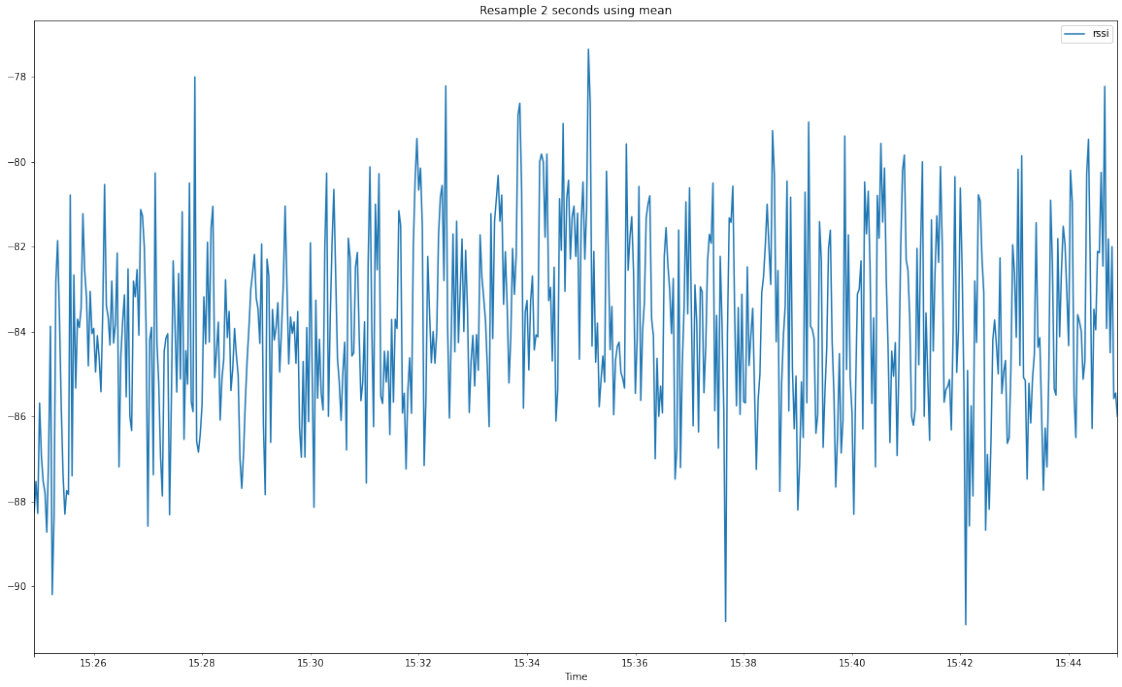


The comparison shows that using the mean as the aggregate function is more effective in reducing the interference. Specifically, the downsample based on the median still contains high-frequency noise.

Two seconds frequency Line 1 shows the resampling method to produce a new time series with a two-second frequency. The RSSI value of each is computed taking the **means** over an interval of two seconds.

[19]:
```
1  mean_df_two = df.resample('2S', on='Time').mean()
2  mean_df_two.plot(title='Resample 2 seconds using mean', figsize=(20, 12))
```

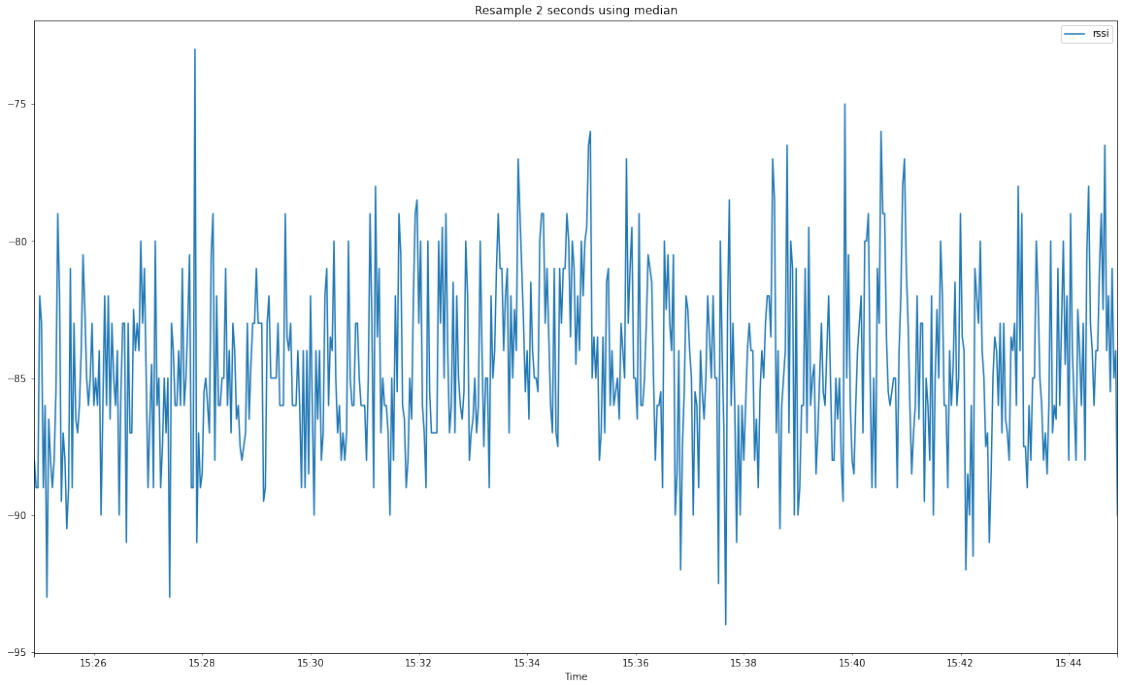
[19]:
```
<AxesSubplot:title={'center':'Resample 2 seconds using mean'}, xlabel='Time'>
```



Line 1 shows the use of the resampling method to produce a new time series with a two-second frequency. The RSSI value of each interval is computed taking the **medians** over a window of two seconds.

[20]:
```
1  median_df_two = df.resample('2S', on='Time').median()
2  median_df_two.plot(title='Resample 2 seconds using median', figsize=(20, 12))
```

[20]: `<AxesSubplot:title={'center':'Resample 2 seconds using median'}, xlabel='Time'>`



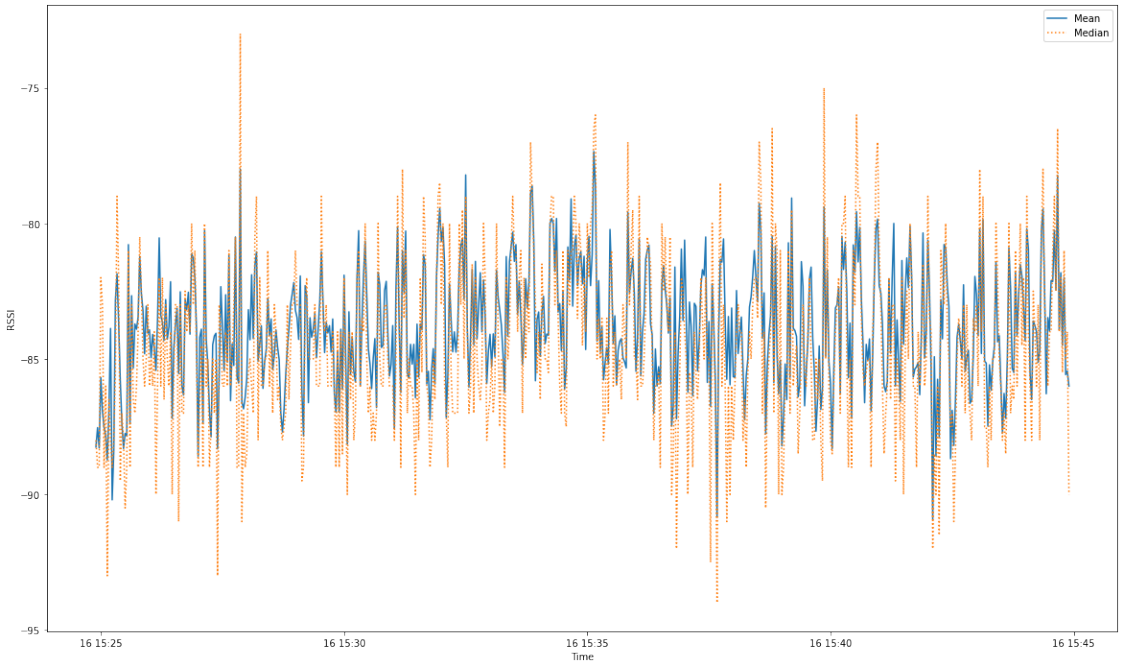
We compare the two downsampled data obtained using a line chart.

[21]:
```
plt.figure(figsize=(20, 12))

plt.plot(mean_df_two['rssi'], label='Mean')
plt.plot(median_df_two['rssi'], linestyle='dotted', label='Median')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[21]:
```
Text(0, 0.5, 'RSSI')
```



As noticed before, the downsample based on the mean function provides a signal with less interference and a more smoothed line than the median-based downsample.

3.1.3 Results

Indoor localization applications usually require a precision that allows us to use both the one-second and the two seconds frequency. However, for clarity purposes, we choose to use the **two seconds frequency** since it produces more readable graphs thanks to a data set with fewer rows. Moreover, based on the results obtained, we decide to use the downsample applying the **mean** as the aggregate function.

[22]:
```
1  mean_df_two
```

[22]:
```
                          rssi
Time
2021-07-16 15:24:54 -88.250000
2021-07-16 15:24:56 -87.533333
2021-07-16 15:24:58 -88.285714
2021-07-16 15:25:00 -85.684211
2021-07-16 15:25:02 -86.928571
...                         ...
2021-07-16 15:44:46 -84.500000
2021-07-16 15:44:48 -82.000000
2021-07-16 15:44:50 -85.578947
2021-07-16 15:44:52 -85.450000
2021-07-16 15:44:54 -86.000000
```

```
[601 rows x 1 columns]
```

[23]:
```
mean_df_two.to_csv('../dataset/resample_mean.csv', index=True)
```

4 Filtering

In this section, we describe and apply the filters selected to the data set obtained. Specifically, we present a brief explanation of the theory behind each filter, the related function implementation, and an evaluation of their performance. Finally, a comparison between the different filtered values is proposed.

[24]:
```
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df = pd.read_csv('../dataset/resample_mean.csv')
```

4.1 Grey Filter

Grey filters are based on the grey system theory, an interdisciplinary scientific area introduced by Julong Deng at the Huazhong University of Science and Technology in 1982. In system theory, a system can be defined with a color representing the amount of clear information about it. A completely unknown system is defined as a black system, while a white system is a completely known one. Similarly, systems with both known and unknown information are referred to as grey systems, representing the most common case since there are always some uncertainties in the real world. In this context, we define the grey models as models that can estimate the behavior of an unknown system only using a limited amount of data. Precisely, they can predict the values of future data based only on a set of the most recent data using a curve fitting approach [4].

The grey filter function implemented exploits a grey module that filters the RSSI values based on a series of values monitored in the past. Specifically, given the set of RSSI values measured $R_0 = r_0(1), ..., r_0(n)$, where $r_0(i)$ is the RSSI value at the discrete time i, $R_1 = r_1, ..., r_1(n)$ is calculated, Line 19-20, as following:

$$r_1(i) = \sum_{j=1}^{i} r_0(j)$$

Then, Line 23-25, a and u are determined from the Grey Model discrete differential equation of the first order:

$$\frac{dr_1(i)}{di} + ar_1(i) = u$$

It is used to obtain the predicted RSSI value $pr(i)$ at discrete time i, Line 27-39, according to the following grey model prediction function:

$$pr(i) = \left(r_1(1) - \frac{u}{a}\right) e^{-ai} + \frac{u}{a}$$

The filtered RSSI values obtained depend on the window size selected with the `N` argument referring to the number of actual RSSI values $r_0(i)$ considered in the grey model. A greater size produces more regular values, and also, the series is slower in following the time evolution of the actual sequence [2].

[25]:
```
# Grey filter taking in input the signal to filter, the time index and the windows size
# Return the filtered signal

def grey_filter(signal, index, N):
    filtered_signal = []

    # iterates on the entire signal, taking steps by window size
    for j in range(0, np.shape(signal)[0], N):

        # just in case we are at signal final and N samples are not available
        N = np.minimum(N, np.shape(signal)[0]-j)

        # saves in R_0 signal values of corresponding window size
        R_0 = np.zeros(N)
        R_0[:] = signal[j:j+N]

        # calculates R_1
        R_1 = []
        for i in range(N):
            R_1.append((np.cumsum(R_0[0:i+1]))[i])

        # calculates grey filter solution
        B = (np.matrix([np.ones((N-1)), np.ones((N-1))])).T
        for k in range(N-1):
            B[k, 0] = -0.5 * (R_1[k+1] + R_1[k])

        X_n = np.matrix(np.asarray(R_0[1:])).T
        _ = np.matmul(np.linalg.pinv(
                np.matmul(B.T, B)),
                      (np.matmul(B.T, X_n)))

        a = _[0, 0]
        u = _[1, 0]

        # updates predicted signal with window calculations
        X_ = R_0[0]
        filtered_signal.append(X_)
        for i in range(1, N):
```

```
            filtered_signal.append((((R_0[0] - u/a) * np.exp(-a * (i - 1)))*(1- np.exp(a))))

    # transforms the data into dataframe with the time index
    df_filtered = pd.DataFrame(filtered_signal, columns=['rssi'])
    df_filtered = df_filtered.join(index)

    return df_filtered
```

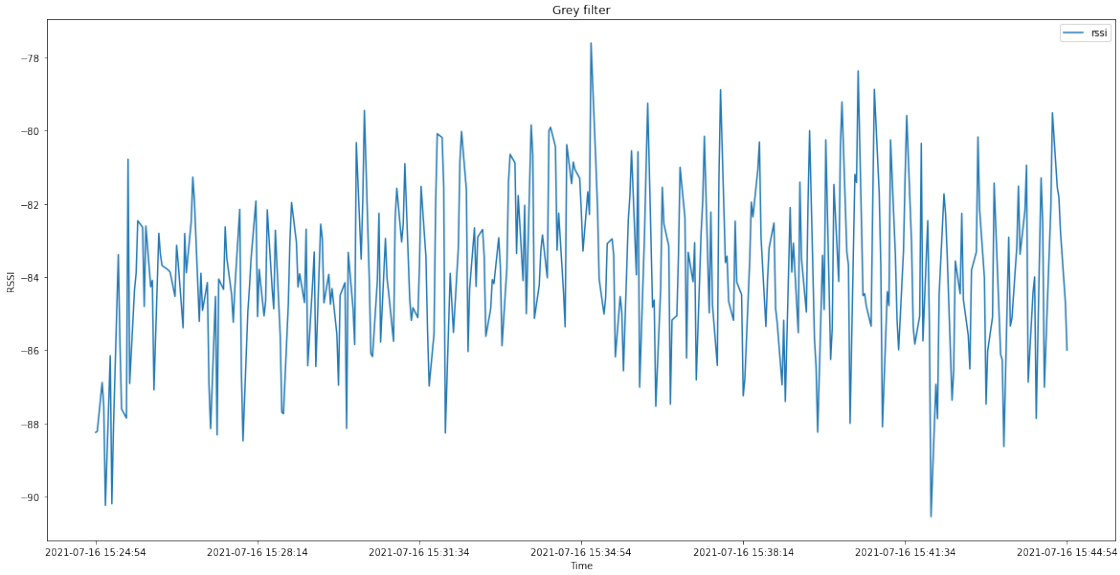
[26]:
```
grey_filtered = grey_filter(df['rssi'], df['Time'], N=5)
grey_filtered.head()
```

[26]:

	rssi	Time
0	-88.250000	2021-07-16 15:24:54
1	-88.218564	2021-07-16 15:24:56
2	-87.771991	2021-07-16 15:24:58
3	-87.327679	2021-07-16 15:25:00
4	-86.885616	2021-07-16 15:25:02

[27]:
```
grey_filtered.plot(x='Time',xlabel='Time', y='rssi', ylabel='RSSI',title='Grey filter', figsize=(20, 10))
```

[27]:
```
<AxesSubplot:title={'center':'Grey filter'}, xlabel='Time', ylabel='RSSI'>
```



We compare the filtered values obtained with the grey filter function with the raw data.

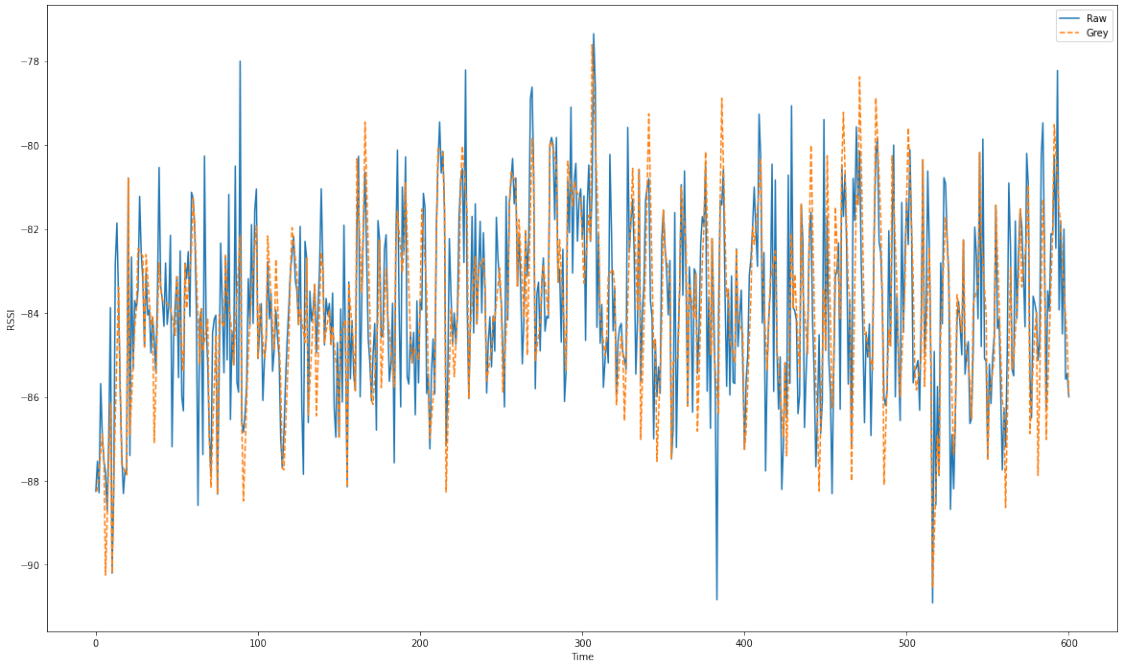
[28]:
```
plt.figure(figsize=(20, 12))

```

```
plt.plot(df['rssi'], label='Raw')
plt.plot(grey_filtered['rssi'], label='Grey', linestyle='dashed')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[28]: `Text(0, 0.5, 'RSSI')`



The figure shows that the grey filter is able to reduce the fluctuation, especially when there are high or low peaks. However, the filter is sensitive to quick changes in data since the line still has significant fluctuations and even some new ones.

4.2 Kalman Filter

The Kalman filter was invented by Rudolf Emil Kálmán at the Research Institute for Advanced Study in 1960 to solve the discrete-data linear filtering problems in a mathematically optimal way. The first use of the Kalman filter was on the Apollo missions, and since then, it has been used in an enormous variety of domains. The relative simplicity and robust nature of the filter have ensured the success of the Kalman filter.

The Kalman filter is essentially a set of mathematical equations that implement an optimal predictor-corrector type estimator. The term optimal is used because the function minimizes the estimated error covariance when some presumed conditions are met.

The Kalman filter exploits the target dynamics, i.e., its time evolution, to remove the effects of noise [5].

The Kalman filter function implemented filters the RSSI values using its time evolution, represented as a combination of signal noise and maximum signal evolving. A linear stochastic equation models the RSSI evolution assuming that noise is independent and has a normal probability distribution. The algorithm minimizes the noise in two phases, implemented in the `kalman_block` function. First, a predictor performs the next RSSI estimation, Line 5-6. Then a corrector improves the estimation by exploiting the current RSSI measurement, Line 8-10. The function is iterated within the `kalman_filter` function, Line 23-30, which takes in input the filter parameters. The value passed influences the filter's mitigation capability and the computational time [2].

[29]:
```
1  # Kalman filter taking in input the signal to filter, the time index and the␣
   ↪filter parameters
2  # Return the filtered signal
3
4  def kalman_block(x, P, s, A, H, Q, R):
5      x_mean = A * x + np.random.normal(0, Q, 1)
6      P_mean = A * P * A + Q
7
8      K = P_mean * H * (1 / (H * P_mean * H + R))
9      x = x_mean + K * (s - H * x_mean)
10     P = (1 - K * H) * P_mean
11
12     return x, P
13
14 def kalman_filter(signal, index, A, H, Q, R):
15     filtered_signal = []
16     # takes first value as first filter prediction
17     x = signal[0]
18     # sets first covariance state value to zero
19     P = 0
20
21     filtered_signal.append(x)
22     # iterates on the entire signal, except the first element
23     for j, s in enumerate(signal[1:]):
24         # calculates next state prediction
25         # x: previous mean state
26         # P: previous covariance state
27         # s: current observation
28         x, P = kalman_block(x, P, s, A, H, Q, R)
29         # updates predicted signal with this step calculation
30         filtered_signal.append(x[0])
31
32     df_filtered = pd.DataFrame(filtered_signal, columns=['rssi'])
33     df_filtered = df_filtered.join(index)
34
35     return df_filtered
```

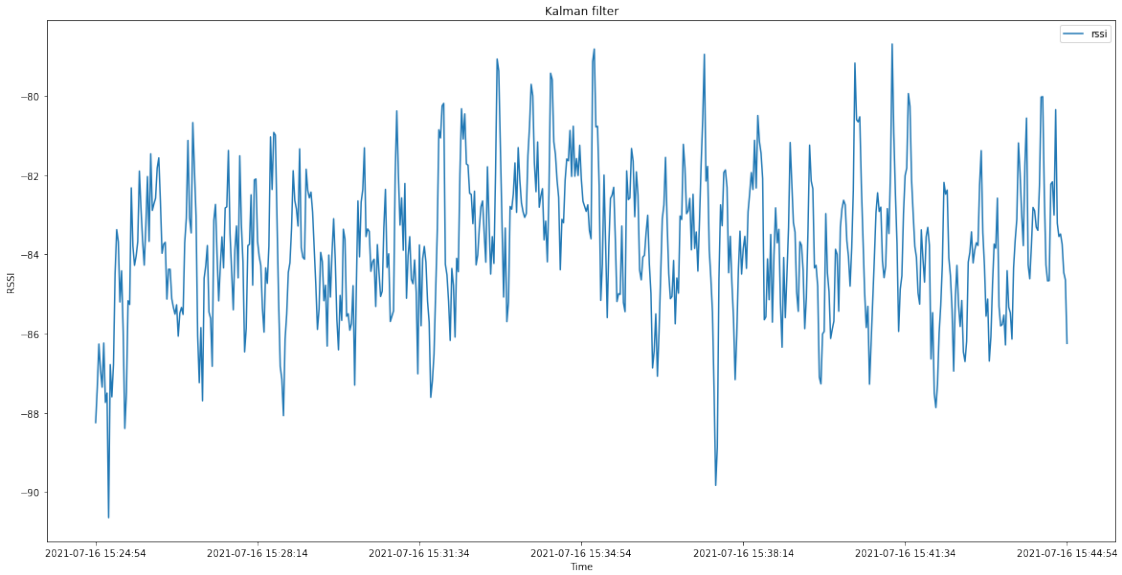
[30]:
```
kalman_filtered = kalman_filter(df['rssi'], df['Time'],  A=1, H=1, Q=1.6, R=6)
kalman_filtered.head()
```

[30]:

	rssi	Time
0	-88.250000	2021-07-16 15:24:54
1	-88.783282	2021-07-16 15:24:56
2	-90.224466	2021-07-16 15:24:58
3	-87.353689	2021-07-16 15:25:00
4	-86.125237	2021-07-16 15:25:02

[31]:
```
kalman_filtered.plot(x='Time',xlabel='Time', y='rssi', ylabel='RSSI',␣
↪title='Kalman filter', figsize=(20, 10))
```

[31]: <AxesSubplot:title={'center':'Kalman filter'}, xlabel='Time', ylabel='RSSI'>



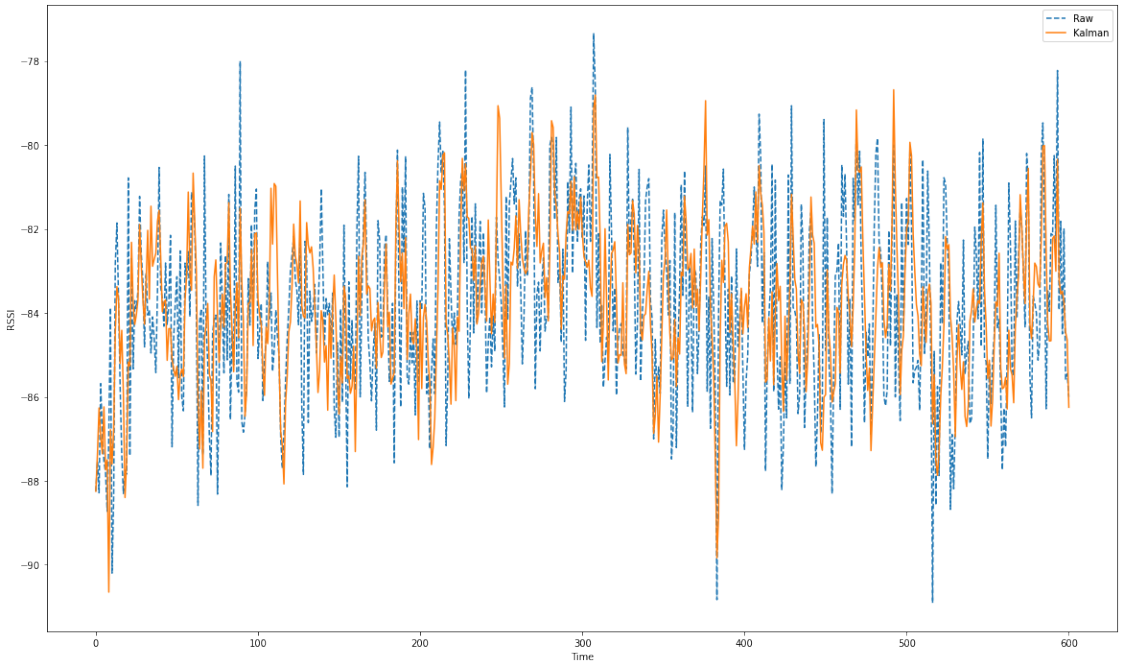
We compare the filtered values obtained with the Kalman filter function with the raw data.

[32]:
```
plt.figure(figsize=(20, 12))

plt.plot(df['rssi'], linestyle='dashed', label='Raw')
plt.plot(kalman_filtered['rssi'], label='Kalman')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[32]: Text(0, 0.5, 'RSSI')



The Kalman filter provides a good fluctuations mitigation and is able to follow the time evolution without amplifying many rapid changes in the values.

4.3 Fourier filter

The Fourier filter is a filtering function used in signal processing based on manipulating specific frequency components of a signal exploiting the Fourier transform. A Fourier transform is a mathematical transform that decomposes functions depending on space or time into functions depending on spatial or temporal frequency. However, the Fourier transform requires a considerable number of operations making the computation prohibitively expensive. The Fast Fourier transform addresses this problem, reducing the complexity introducing a memory portion within the algorithm.

The Fourier filter function extracts from the set of previous values R_0, the same defined in the grey filter, Section 4.1, a coefficient set representing the RSSI sequence in the frequency domain in a time window of duration $(R_0 size) * (RSSI sampling period)$ using `numpy` Fast Fourier Transform, Line 17. The function takes into input the window size `N`, and the `M` number of sample of Fourier filter to preserve, Line 21-23. The parameters values influence the fluctuation mitigation performed and the computational time [2].

[33]:
```
1  # Fourier filter taking in input the signal to filter, the time index, the␣
   ↪windows size
2  # and the number of sample to preserve
3  # Return the filtered signal
4
5  def fft_filter(signal, index, N, M):
```

```
    filtered_signal = []

    for j in range(0, np.shape(signal)[0], N):
        # just in case we are at signal final and N samples are not avail
        N = np.minimum(N, np.shape(signal)[0] - j)

        # saves in R_0 signal values of corresponding window size
        R_0 = np.zeros(N)
        R_0[:] = signal[j:j+N]

        # fft of signal window
        R_0_fft = np.fft.fft(R_0)

        # it keeps M samples of fft and sets the rest to zero
        # remembers fft symmetry
        for k in range(int(N / 2)):
            R_0_fft[M+k] = 0
            R_0_fft[-1-M-k] = 0

        # inverse fft
        R_0_ifft = np.fft.ifft(R_0_fft)

        # updates predicted signal with this window calculation
        for i in range(0, N):
            filtered_signal.append(R_0_ifft[i])

    df_filtered = pd.DataFrame(filtered_signal, columns=['rssi'])
    df_filtered = df_filtered.join(index)

    return df_filtered
```

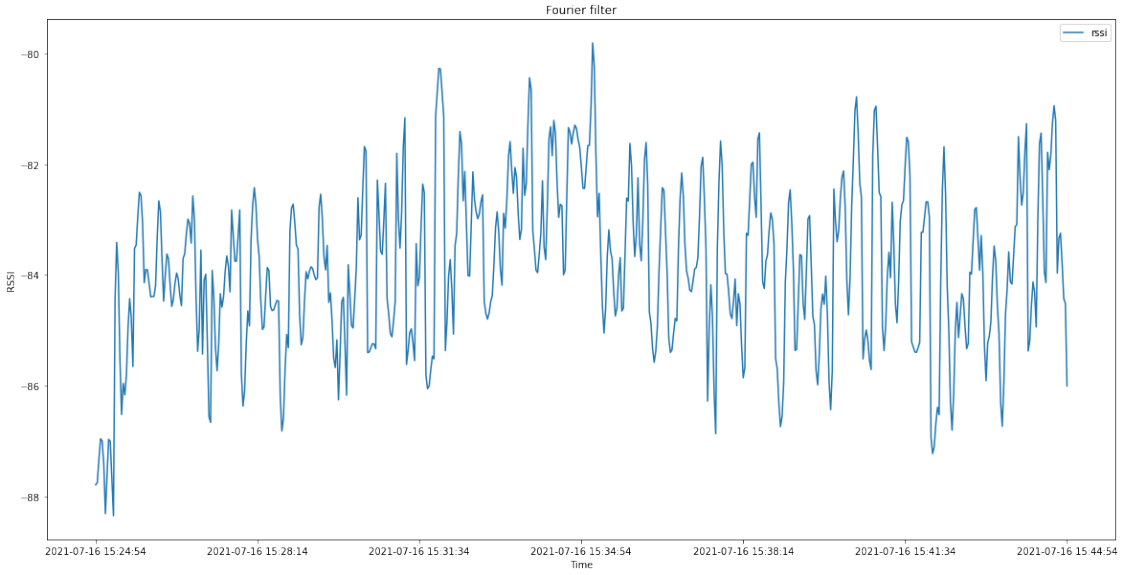
[34]:
```
fft_filtered = fft_filter(df['rssi'], df['Time'], N=6, M=2)
fft_filtered.head()
```

[34]:

	rssi	Time
0	-87.784524-0.195887j	2021-07-16 15:24:54
1	-87.746502+0.261743j	2021-07-16 15:24:56
2	-87.331172+0.457630j	2021-07-16 15:24:58
3	-86.953864+0.195887j	2021-07-16 15:25:00
4	-86.991886-0.261743j	2021-07-16 15:25:02

[35]:
```
fft_filtered.plot(x='Time',xlabel='Time', y='rssi', ylabel='RSSI',␣
↪title='Fourier filter', figsize=(20, 10))
```

[35]:
```
<AxesSubplot:title={'center':'Fourier filter'}, xlabel='Time', ylabel='RSSI'>
```



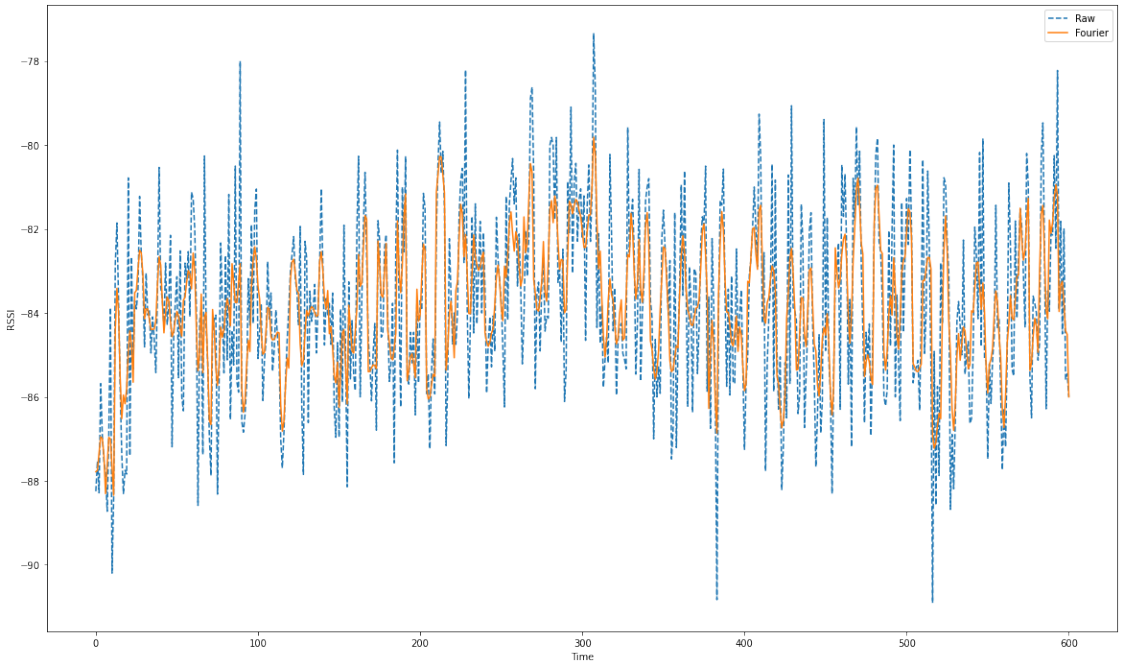
We compare the filtered values obtained with the Fourier filter function with the raw data.

[36]:
```
plt.figure(figsize=(20, 12))

plt.plot(df['rssi'], linestyle='dashed', label='Raw')
plt.plot(fft_filtered['rssi'], label='Fourier')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[36]:
```
Text(0, 0.5, 'RSSI')
```



The Fourier filter provides a significantly smoother line than the raw data reducing the fluctuations and following the time sequence without any amplification of the signal.

4.4 Particle filter

Particle filters are algorithms used to solve filtering problems arising in signal processing and Bayesian statistical inference. The term particle filters were first coined in 1996 by Del Moral in reference to mean-field interacting particle methods used in fluid mechanics since the beginning of the 1960s.

Particle filters use a set of particles, also called samples, to represent the posterior distribution of some stochastic process given noisy or partial observations. Each particle has a weight representing the probability of being sampled from the probability function. In the resampling step, the particles with negligible weights are replaced by new particles in the proximity of the particles with higher weights.

The Particle filter function computes, at each step, several possible filtered values for each measured RSSI and assigns a weight to each candidate value. When a new RSSI value is available, the filter chooses the most promising values and computes their average to obtain a new filtered value. The number of particles strongly influences the performance of the filter. A higher particles number allows the filter to follow the time series better but increases the computational time [2].

[37]:
```
# Particle filter taking in input the signal to filter, the time index, the number of particles,
# and the filter parameters
# Return the filtered signal

```

```
def choose_particle(particles):
    prob_distribution = []

    # calculates sum of weights to normalize wheight vector in next step
    sum_weights = 0
    for p in particles:
        sum_weights += p['weight']

    for p in particles:
        prob_distribution.append(float(p['weight'] / sum_weights))

    # choose particle according to weights distribution
    a = np.random.choice(particles, 1, replace=False, p=prob_distribution)

    return a[0]['value'][0]


def particle_filter(signal, index, quant_particles, A, H, Q, R):

    filtered_signal = []
    # variation range of particles for initial step
    rang = 10
    # takes first value as first filter prediction
    x = signal[0]
    # sets first covariance state value to zero
    P = 0

    filtered_signal.append(x)
    # defines some needed constants in algorithm
    min_weight_to_consider = 0.07
    min_weight_to_split_particle = 5

    # iterates on the entire signal, except the first element
    for j, s in enumerate(signal[1:]):
        # sets variation range for first step sampling
        range_ = [filtered_signal[j-1] - rang,
        filtered_signal[j-1] + rang]

        particles = []
        # loop on all particles
        for particle in range(quant_particles):
            # sample particle value from variation range
            input = np.random.uniform(range_[0], range_[1])
            # particle weight
            weight = 1 / np.abs(input-x)

```

```
            # it only iterates on particles which weights are greater than␣
    ↪_min_weight_to_consider_
            if weight > min_weight_to_consider:
                # calculates next state prediction
                x_, P = kalman_block(input, P, s, A, H, Q, R)

                # prediction weight
                weight = 1 / np.abs(s - x_)
                particles.append({'value': x_, 'weight': weight})

                # for particles with greater weights, it creates other␣
    ↪particles in the 'neighborhood'
                if weight > min_weight_to_split_particle:

                    input = input + np.random.uniform(0, 5)
                    x_, P = kalman_block(input, P, s, A, H, Q, R)

                    weight = 1 / np.abs(s - x_)
                    particles.append({'value': x_, 'weight': weight})

        # chooses a particle, according to weight distribution
        x = choose_particle(particles)

        # updates predicted signal with this step calculation
        filtered_signal.append(x)

    df_filtered = pd.DataFrame(filtered_signal, columns=['rssi'])
    df_filtered = df_filtered.join(index)

    return df_filtered
```

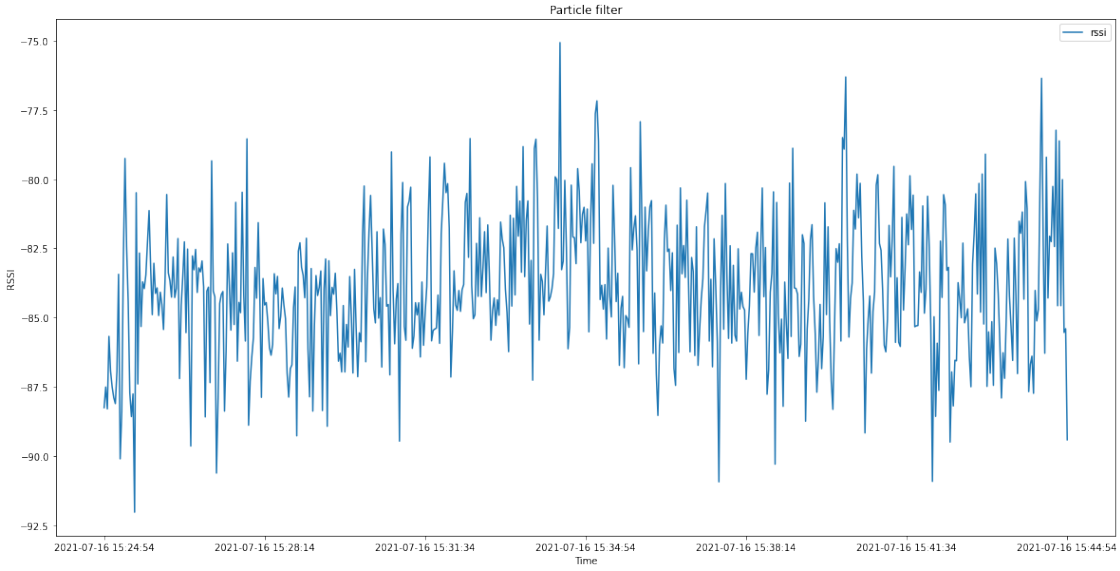
[38]:
```
particle_filtered = particle_filter(df['rssi'], df['Time'],␣
    ↪quant_particles=1000, A=1, H=1, Q=1.6, R=6)
particle_filtered.head()
```

[38]:

	rssi	Time
0	-88.250000	2021-07-16 15:24:54
1	-87.539481	2021-07-16 15:24:56
2	-94.872273	2021-07-16 15:24:58
3	-84.852620	2021-07-16 15:25:00
4	-87.292022	2021-07-16 15:25:02

[39]:
```
particle_filtered.plot(x='Time',xlabel='Time', y='rssi', ylabel='RSSI',␣
    ↪title='Particle filter', figsize=(20, 10))
```

[39]: <AxesSubplot:title={'center':'Particle filter'}, xlabel='Time', ylabel='RSSI'>



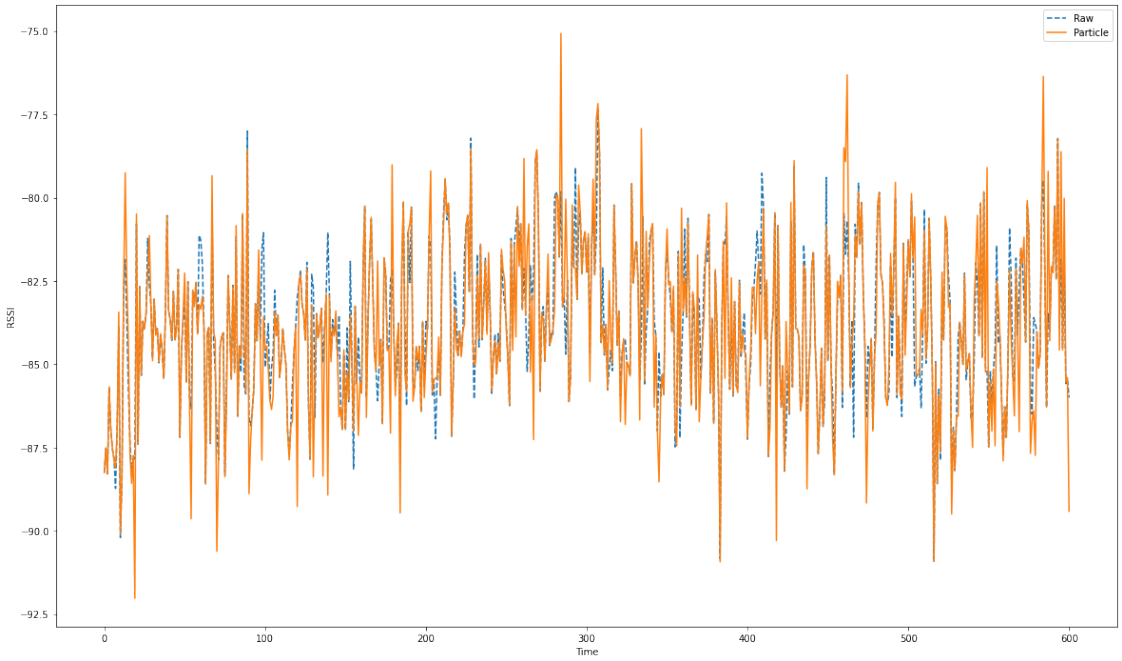
We compare the filtered values obtained with the particle filter function with the raw data

[40]:
```
plt.figure(figsize=(20, 12))

plt.plot(df['rssi'], linestyle='dashed', label='Raw')
plt.plot(particle_filtered['rssi'], label='Particle')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[40]:
```
Text(0, 0.5, 'RSSI')
```



The figure shows that the Particle filter is not able to mitigate the signal fluctuations. It introduces peaks when the signal rapidly changes and is also the slowest to compute. Even tuning the parameter, the performance does not improve.

4.5 Filters comparison

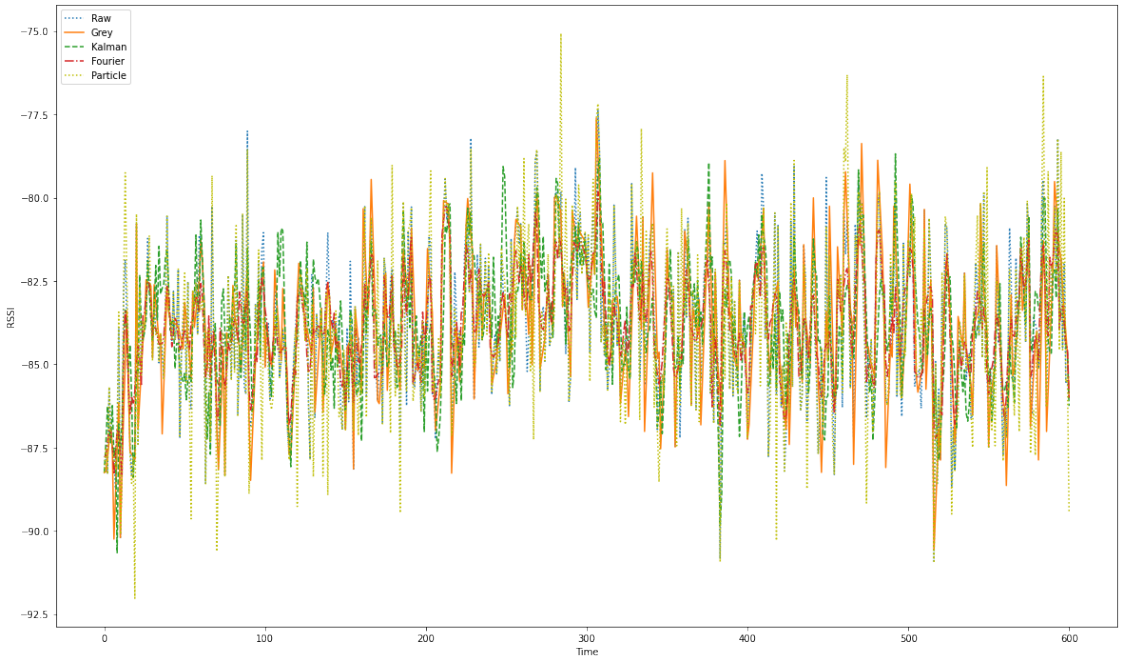
We compare the data obtained from each filter implemented.

[41]:
```
plt.figure(figsize=(20, 12))

plt.plot(df['rssi'], linestyle='dotted', label='Raw')
plt.plot(grey_filtered['rssi'],  label='Grey')
plt.plot(kalman_filtered['rssi'], ls='dashed', label='Kalman')
plt.plot(fft_filtered['rssi'], ls='dashdot',label='Fourier')
plt.plot(particle_filtered['rssi'], ls='dotted', color='y', label='Particle')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[41]:
```
Text(0, 0.5, 'RSSI')
```



The comparison confirms the comments made before. The Kalman and Fourier filters are the most promising, being able to smooth quite well the signal. The grey filter is able to mitigate the fluctuations in most cases but sometimes tends to introduce a new one when the signal has rapid changes. Finally, the particle filter performs the worst showing more fluctuations than the raw data.

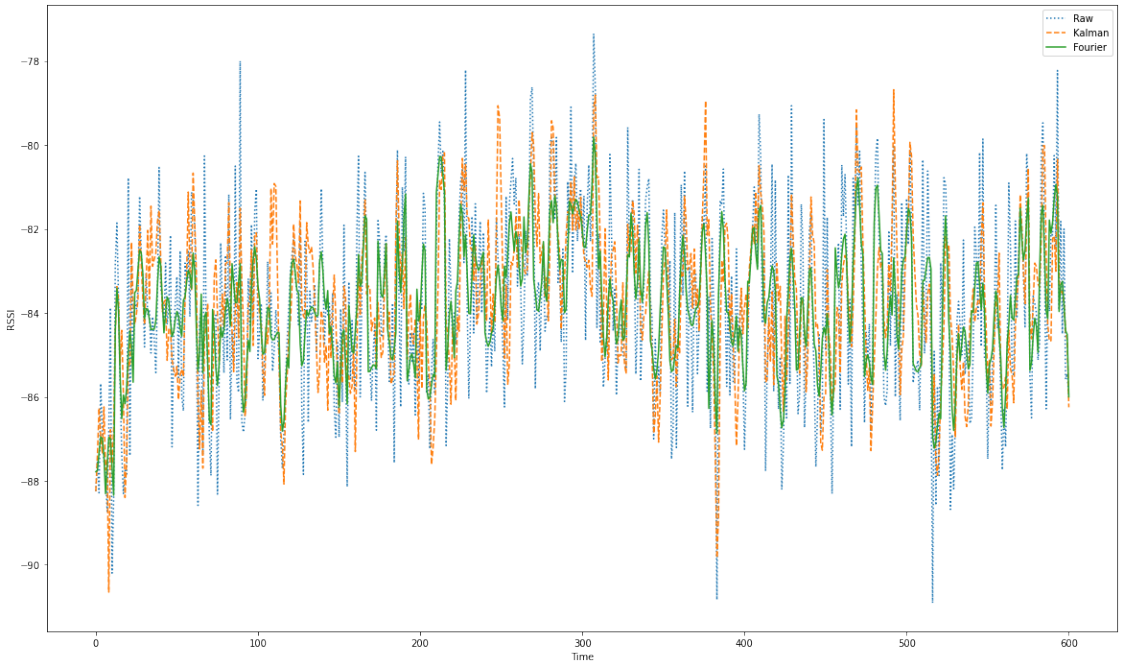
We perform another comparison, including only the Kalman and the Fourier filters and dropping the less effective ones.

[42]:
```
plt.figure(figsize=(20, 12))

plt.plot(df['rssi'], ls='dotted', label='Raw')
plt.plot(kalman_filtered['rssi'], linestyle='dashed', label='Kalman')
plt.plot(fft_filtered['rssi'], linestyle='solid', label='Fourier')

plt.legend()
plt.xlabel('Time')
plt.ylabel('RSSI')
```

[42]:
```
Text(0, 0.5, 'RSSI')
```



The Kalman and the Fourier filters provide the best results. The Kalman algorithm is able to follow the actual RSSI value better, while the Fourier grants the smoothest line. The choice of which filters to use depends on the application requirements and the context.

References

[1] H. Liu, H. Darabi, P. Banerjee, and J. Liu, "Survey of wireless indoor positioning techniques and systems," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 37, no. 6, pp. 1067–1080, 2007.

[2] P. Bellavista, A. Corradi, and C. Giannelli, "Evaluating filtering strategies for decentralized handover prediction in the wireless internet," *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pp. 167–174, 2006.

[3] A. Mussina and S. Aubakirov, "Rssi based bluetooth low energy indoor positioning," *2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–4, 2018.

[4] E. Kayacan, B. Ulutas, and O. Kaynak, "Grey system theory-based models in time series prediction," *Expert Systems with Applications*, vol. 37, pp. 1784–1789, 03 2010.

[5] M. Laaraiedh, "Implementation of kalman filter with python language," *The Python Papers*, 2012.