연구 보고서

작성자	김한호	작성일자	2021.08.08.

1. 연구 계획

- Hyperledger Fabric 매개변수 변경하며 TPS 측정값 확인
- 측정값을 이용한 수학적 분석 방향성 모색
- 데이터 생성 시간에 따른 실험 계획 설계

2. 논문 연구 진행

```
rounds:
  - label: get-asset-evaluate-100-100-100
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        create_sizes:
          - 100
        noSetup: false
        assets: 100
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-100-100-200
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        create_sizes:
          - 100
        noSetup: false
        assets: 200
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-100-100-300
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        create_sizes:
          - 100
        noSetup: false
        assets: 300
        byteSize: 100
        consensus: false
```

체인코드 getAssset 함수를 이용하여 자산(Asset)의 개수를 변화하여 성능 측정을 하는 과정을 실행함.

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
get-asset-evaluate-100-100-100	100	0	72.7	1.39	0.08	0.52	68.6
get-asset-evaluate-100-100-200	100	0	93.2	0.77	0.04	0.32	89.9
get-asset-evaluate-100-100-300	100	0	102.1	0.17	0.01	0.06	100.8

처리량이 도출되는 것을 확인함.

```
name: fixed-asset-test
description: >-
  This is a test yaml for the existing fixed-asset benchmarks for diff txNumber
  tps
workers:
  type: local
  number: 2
rounds:
  - label: get-asset-evaluate-100-100-100
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        create_sizes:
          - 100
        noSetup: false
        assets: 100
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-100-200-100
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 200
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        noSetup: true
        assets: 100
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-100-300-100
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 300
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        noSetup: true
        assets: 100
        byteSize: 100
        consensus: false
```

체인코드 getAssset 함수를 이용하여 고정속도로 입력 트랜잭션 변화하여 성능 측정을 하는 과정을 실행함.

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
get-asset-evaluate-100-100-100	100	0	105.5	0.86	0.10	0.39	95.6
get-asset-evaluate-100-200-100	100	0	127.1	0.78	0.06	0.50	96.5
get-asset-evaluate-100-300-100	100	0	143.9	0.69	0.03	0.37	121.4

처리량이 도출되는 것을 확인함.

```
name: fixed-asset-test
description: >-
  This is a test yaml for the existing fixed-asset benchmarks for diff txNumber
  tps
workers:
  type: local
  number: 2
rounds:
  - label: get-asset-evaluate-100-100-100
    chaincodeID: fixed-asset
    txNumber: 100
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        create_sizes:
          - 100
        noSetup: false
        assets: 100
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-200-100-100
    chaincodeID: fixed-asset
    txNumber: 200
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        noSetup: true
        assets: 100
        byteSize: 100
        consensus: false
  - label: get-asset-evaluate-300-100-100
    chaincodeID: fixed-asset
    txNumber: 300
    rateControl:
      type: fixed-rate
      opts:
        tps: 100
    workload:
      module: benchmarks/api/fabric/lib/get-asset.js
      arguments:
        chaincodeID: fixed-asset
        noSetup: true
        assets: 100
        byteSize: 100
        consensus: false
```

각 라운드에서 실행할 트랜잭션 번호를 다르게 설정하여 성능측정을 함.

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
get-asset-evaluate-100-100-100	100	0	101.4	0.99	0.06	0.44	95.8
get-asset-evaluate-200-100-100	200	0	100.7	0.08	0.01	0.02	100.3
get-asset-evaluate-300-100-100	300	0	100.0	0.14	0.01	0.04	99.2

처리량이 나오는 것을 확인함.

	private-da	encrypt	function	asset	tps	txNum	bytesize	send_rate	max_laten	min_laten	avg_latenc	time_taker	throughpu
e1	0	0	get	60	100	100	100	101.2	0.09	0.01	0.02	11.692	100.2
e2	0	0	get	70	100	100	100	94.6	0.18	0.01	0.02	11.965	92.8
e3	0	0	get	80	100	100	100	101.2	0.1	0.01	0.02	11.997	99.3
e4	0	0	get	90	100	100	100	102	0.17	0.01	0.04	11.724	101.2
e5	0	0	get	100	100	100	100	100.8	0.05	0.01	0.02	11.799	99.5
e6	0	0	get	200	100	100	100	101	0.47	0.01	0.2	16.178	90.6
e7	0	0	get	300	100	100	100	95.4	0.42	0.01	0.2	20.837	91.2
e8	0	0	get	400	100	100	100	101.3	0.1	0.01	0.03	24.989	100.7
e9	0	0	get	500	100	100	100	101.6	0.33	0.05	0.19	29.374	96.4
e10	0	0	get	600	100	100	100	101.2	0.13	0.01	0.04	33.834	95.4
e11	0	0	get	700	100	100	100	94.7	0.79	0.01	0.43	40.729	77.9
e12	0	0	get	800	100	100	100	98.5	0.24	0.04	0.16	42.961	94.9
e13	0	0	get	900	100	100	100	101.1	0.91	0.01	0.62	47.194	74.5
e14	0	0	get	1000	100	100	100	89.9	0.7	0.02	0.41	51.522	84
e15	0	0	get	2000	100	100	100	97.3	0.32	0.01	0.2	95.502	95.7
e16	0	0	get	3000	100	100	100	101.1	0.9	0.18	0.54	141.47	74.4
e17	0	0	get	4000	100	100	100	97.6	0.87	0.04	0.41	184.473	85.5
e18	0	0	get	5000	100	100	100	100.2	0.55	0.04	0.33	228.917	88.3
e19	0	0	get	6000	100	100	100	101.1	0.8	0.02	0.46	273.547	77.6
e20	0	0	get	7000	100	100	100	101	0.83	0.05	0.46	319.393	78.7
e21	0	0	get	8000	100	100	100	101.3	0.56	0.01	0.35	363.438	90.1
e22	0	0	get	9000	100	100	100	82.8	0.97	0.01	0.51	430.381	68.8
e23	0	0	get	10000	100	100	100	101.2	0.39	0.01	0.23	454.583	92.7
e24	0	0	get	20000	100	100	100	101.1	0.38	0.07	0.21	896.256	88.9
e25	0	0	get	30000	100	100	100	92.6	0.95	0.04	0.65	1351.124	66.1
e26	0	0	get	40000	100	100	100	98.3	0.12	0.01	0.03	1803.38	92.9
e27	0	0	get	50000	100	100	100	101.1	0.59	0.01	0.35	2521.729	81.4
e28	0	0	get	60000	100	100	100	101.2	0.59	0.01	0.21	2665.183	85.5
e29	0	0	get	70000	100	100	100	101.1	0.14	0.01	0.04	3110.535	100.4

수학적 분석과 앞으로 데이터를 얻는 시간을 예측하기 위해 getAsset의 기본 설정을 적용하여 asset 데이터를 변화하여 70만까지 늘여가며 걸린 시간과 처리량을 구하여 데이터 분석을 적용하기로 계획함.

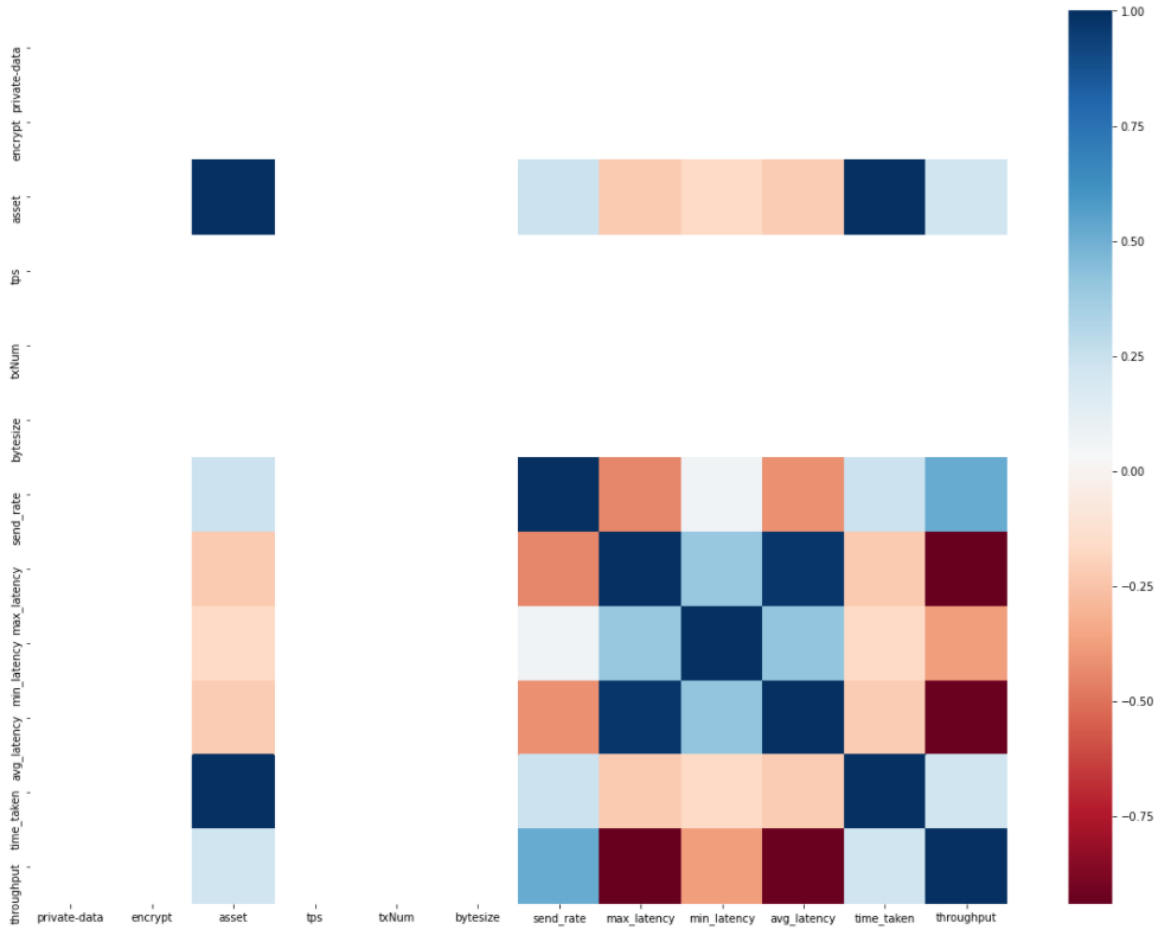
In [15]:
```
get_df = pd.read_csv('./datasets/getasset.csv', index_col=0)
```

In [17]:
```
get_df
```

Out[17]:

	private-data	encrypt	function	asset	tps	txNum	bytesize	send_rate	max_latency	min_latency	avg_latency	time_taken	throughput
e1	0	0	get	60	100	100	100	101.2	0.09	0.01	0.02	11.692	100.2
e2	0	0	get	70	100	100	100	94.6	0.18	0.01	0.02	11.965	92.8
e3	0	0	get	80	100	100	100	101.2	0.10	0.01	0.02	11.997	99.3
e4	0	0	get	90	100	100	100	102.0	0.17	0.01	0.04	11.724	101.2
e5	0	0	get	100	100	100	100	100.8	0.05	0.01	0.02	11.799	99.5
e6	0	0	get	200	100	100	100	101.0	0.47	0.01	0.20	16.178	90.6
e7	0	0	get	300	100	100	100	95.4	0.42	0.01	0.20	20.837	91.2
e8	0	0	get	400	100	100	100	101.3	0.10	0.01	0.03	24.989	100.7
e9	0	0	get	500	100	100	100	101.6	0.33	0.05	0.19	29.374	96.4
e10	0	0	get	600	100	100	100	101.2	0.13	0.01	0.04	33.834	95.4
e11	0	0	get	700	100	100	100	94.7	0.79	0.01	0.43	40.729	77.9
e12	0	0	get	800	100	100	100	98.5	0.24	0.04	0.16	42.961	94.9
e13	0	0	get	900	100	100	100	101.1	0.91	0.01	0.62	47.194	74.5
e14	0	0	get	1000	100	100	100	89.9	0.70	0.02	0.41	51.522	84.0
e15	0	0	get	2000	100	100	100	97.3	0.32	0.01	0.20	95.502	95.7
e16	0	0	get	3000	100	100	100	101.1	0.90	0.18	0.54	141.470	74.4
e17	0	0	get	4000	100	100	100	97.6	0.87	0.04	0.41	184.473	85.5
e18	0	0	get	5000	100	100	100	100.2	0.55	0.04	0.33	228.917	88.3
e19	0	0	get	6000	100	100	100	101.1	0.80	0.02	0.46	273.547	77.6
e20	0	0	get	7000	100	100	100	101.0	0.83	0.05	0.46	319.393	78.7
e21	0	0	get	8000	100	100	100	101.3	0.56	0.01	0.35	363.438	90.1
e22	0	0	get	9000	100	100	100	82.8	0.97	0.01	0.51	430.381	68.8
e23	0	0	get	10000	100	100	100	101.2	0.39	0.01	0.23	454.583	92.7

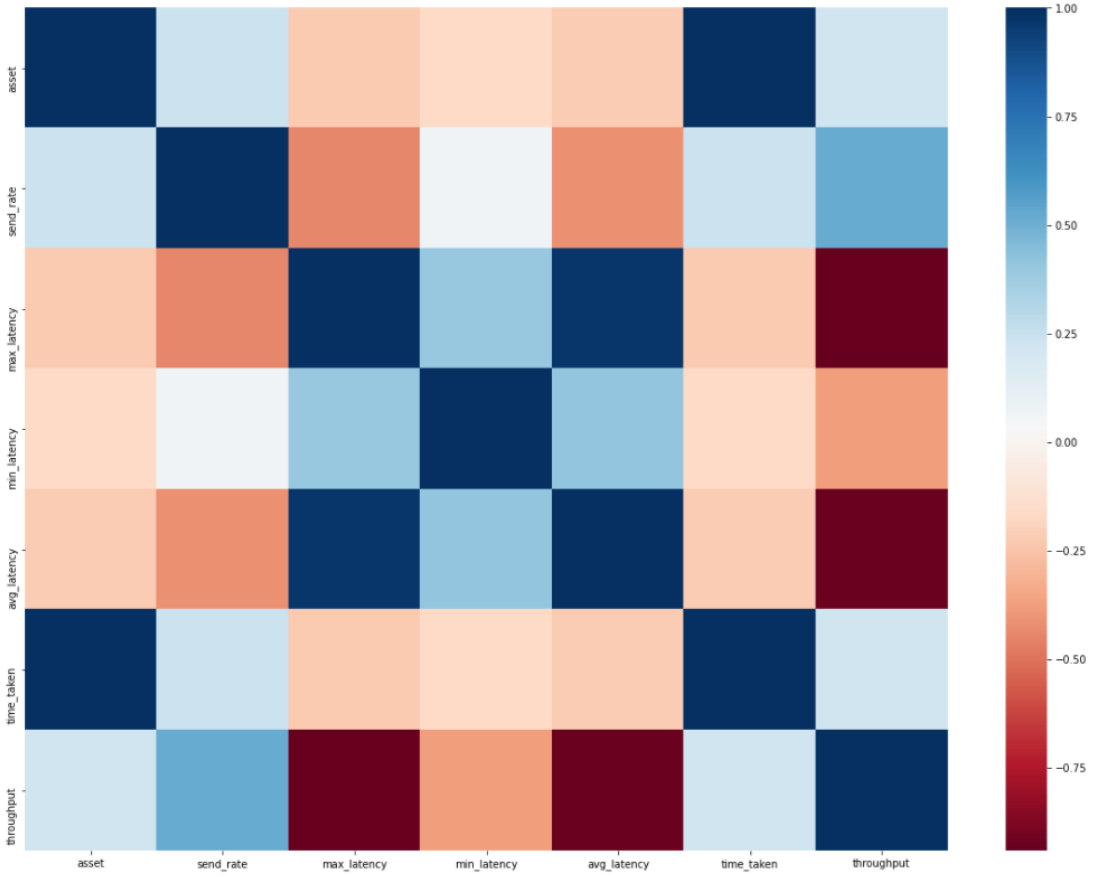
데이터 값을 csv파일에서 읽어와서 데이터 프레임으로 변환함.



각 요소를 히트맵을 이용하여 연관성을 판단하여 관련 없는 요소를 파악함.

In [21]:
```
refine_df = get_df.drop(['function','private-data','encrypt','tps', 'txNum', 'bytesize'], axis=1)
```

연관성이 없다고 파악한 요소열을 제거함.



요소에 따른 관계가 음의 연관성이 클수록 적색이 진해지고 양의 관계가 있다고 판단되는 것에 파란색이 진해지고 있음을 확인할 수 있었음.

	asset	send_rate	max_latency	min_latency	avg_latency	time_taken	throughput
e1	60	101.2	0.09	0.01	0.02	11.692	100.2
e2	70	94.6	0.18	0.01	0.02	11.965	92.8
e3	80	101.2	0.10	0.01	0.02	11.997	99.3
e4	90	102.0	0.17	0.01	0.04	11.724	101.2
e5	100	100.8	0.05	0.01	0.02	11.799	99.5
e6	200	101.0	0.47	0.01	0.20	16.178	90.6
e7	300	95.4	0.42	0.01	0.20	20.837	91.2
e8	400	101.3	0.10	0.01	0.03	24.989	100.7
e9	500	101.6	0.33	0.05	0.19	29.374	96.4
e10	600	101.2	0.13	0.01	0.04	33.834	95.4
e11	700	94.7	0.79	0.01	0.43	40.729	77.9
e12	800	98.5	0.24	0.04	0.16	42.961	94.9
e13	900	101.1	0.91	0.01	0.62	47.194	74.5
e14	1000	89.9	0.70	0.02	0.41	51.522	84.0
e15	2000	97.3	0.32	0.01	0.20	95.502	95.7
e16	3000	101.1	0.90	0.18	0.54	141.470	74.4
e17	4000	97.6	0.87	0.04	0.41	184.473	85.5
e18	5000	100.2	0.55	0.04	0.33	228.917	88.3
e19	6000	101.1	0.80	0.02	0.46	273.547	77.6
e20	7000	101.0	0.83	0.05	0.46	319.393	78.7
e21	8000	101.3	0.56	0.01	0.35	363.438	90.1
e22	9000	82.8	0.97	0.01	0.51	430.381	68.8
e23	10000	101.2	0.39	0.01	0.23	454.583	92.7
e24	20000	101.1	0.38	0.07	0.21	896.256	88.9
e25	30000	92.6	0.95	0.04	0.65	1351.124	66.1
e26	40000	98.3	0.12	0.01	0.03	1803.380	92.9
e27	50000	101.1	0.59	0.01	0.35	2521.729	81.4
e28	60000	101.2	0.59	0.01	0.21	2665.183	85.5
e29	70000	101.1	0.14	0.01	0.04	3110.535	100.4
e30	80000	100.9	0.17	0.01	0.05	3557.037	96.4
e31	90000	101.1	0.11	0.01	0.03	4003.348	100.0
e32	100000	100.2	0.16	0.01	0.05	4447.643	94.7

연관성이 없는 요소열을 제거한 데이터 프레임을 확인함.

In [26]:
```
X = refine_df.iloc[:, 0:-1]
y = refine_df.iloc[:, -1]
```

In [27]:
```
X
```

Out[27]:

	asset	send_rate	max_latency	min_latency	avg_latency	time_taken
e1	60	101.2	0.09	0.01	0.02	11.692
e2	70	94.6	0.18	0.01	0.02	11.965
e3	80	101.2	0.10	0.01	0.02	11.997
e4	90	102.0	0.17	0.01	0.04	11.724
e5	100	100.8	0.05	0.01	0.02	11.799
e6	200	101.0	0.47	0.01	0.20	16.178
e7	300	95.4	0.42	0.01	0.20	20.837
e8	400	101.3	0.10	0.01	0.03	24.989
e9	500	101.6	0.33	0.05	0.19	29.374
e10	600	101.2	0.13	0.01	0.04	33.834
e11	700	94.7	0.79	0.01	0.43	40.729
e12	800	98.5	0.24	0.04	0.16	42.961
e13	900	101.1	0.91	0.01	0.62	47.194
e14	1000	89.9	0.70	0.02	0.41	51.522
e15	2000	97.3	0.32	0.01	0.20	95.502
e16	3000	101.1	0.90	0.18	0.54	141.470
e17	4000	97.6	0.87	0.04	0.41	184.473
e18	5000	100.2	0.55	0.04	0.33	228.917
e19	6000	101.1	0.80	0.02	0.46	273.547
e20	7000	101.0	0.83	0.05	0.46	319.393
e21	8000	101.3	0.56	0.01	0.35	363.438
e22	9000	82.8	0.97	0.01	0.51	430.381
e23	10000	101.2	0.39	0.01	0.23	454.583
e24	20000	101.1	0.38	0.07	0.21	896.256
e25	30000	92.6	0.95	0.04	0.65	1351.124
e26	40000	98.3	0.12	0.01	0.03	1803.380

In [28]:
```
y
```

Out[28]:

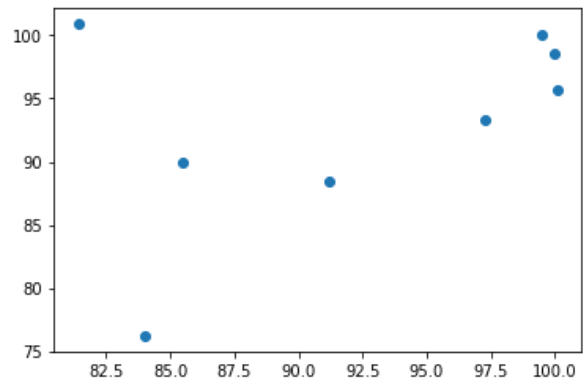
```
e1     100.2
e2      92.8
e3      99.3
e4     101.2
e5      99.5
e6      90.6
e7      91.2
e8     100.7
e9      96.4
e10     95.4
e11     77.9
e12     94.9
e13     74.5
e14     84.0
e15     95.7
e16     74.4
e17     85.5
e18     88.3
e19     77.6
e20     78.7
e21     90.1
e22     68.8
e23     92.7
e24     88.9
e25     66.1
e26     92.9
e27     81.4
e28     85.5
e29    100.4
e30     96.4
e31    100.0
e32     94.7
e33     86.6
e34     97.3
e35     96.1
e36     88.9
e37    100.1
e38     95.0
Name: throughput,
```

함수에 입력되는 X와 결과값 Y로 나눠서 저장함.

```
In [13]: X = refine_df.iloc[:, 0:-1]
         y = refine_df.iloc[:, -1]

In [17]: from sklearn.linear_model import LinearRegression
         from sklearn.model_selection import train_test_split

In [18]: X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=42)
         lnr_clf = LinearRegression()
         lnr_clf.fit(X_train, y_train)
Out[18]: LinearRegression()

In [19]: print('Intercept: ₩n', lnr_clf.intercept_)
         print('Coefficients: ₩n', lnr_clf.coef_)

         Intercept:
          33.679998659159324
         Coefficients:
          [-2.27470485e-03  6.65779945e-01 -6.05593192e+00 -1.97109441e+01
          -3.18201486e+01  5.09782249e-02]

In [23]: from sklearn.metrics import mean_squared_error, r2_score

In [24]: lnr_clf.score(X_train, y_train)
         lnr_clf_pred = lnr_clf.predict(X_test)
         mse = mean_squared_error(y_test, lnr_clf_pred)
         rmse = np.sqrt(mse)
         print('MSE: {0:.3f}, RMSE :{1:.3f}' .format(mse, rmse))
         print('Variance score : {0:.3f}' .format(r2_score(y_test,lnr_clf_pred)))

         MSE: 63.228, RMSE :7.952
         Variance score : -0.171
```

수학적 분석의 방향성을 모색하는 것이 실험의 방향성이므로 다변량 선형 회귀(Mutilivariate Linear Regression)을 적용하여 상수항(Intercept)과 계수(Coefficient)를 도출함.

```
In [68]: plt.scatter(y_test, lnr_clf_pred)
Out[68]: <matplotlib.collections.PathCollection at 0x19c3d5ed9d0>
```




```
In [ ]:
```

모델의 정확성 확인을 위해 MSE와 RMSE의 점수와 R2_SCORE를 확인함. MSE

값과 RMSE 값이 정확성이 높을수록 작은 값이 나와야 하나 실험에서 높은 값이 나와 모델의 정확성이 낮은 것을 확인할 수 있음. R2는 1에 가까울수록 모델의 정확성이 높다. R2 점수도 양의 값을 넘어서는 음수를 나타내고 있으므로 모델의 정확성이 높지 않은 것을 확인할 수 있음.

In [17]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

In [140]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.39,random_state=42)
lnr_clf = LinearRegression(fit_intercept=True, normalize=True, n_jobs=None)
lnr_clf.fit(X_train, y_train)
lnr_clf_pred = lnr_clf.predict(X_test)
```

In [141]:
```
print('Intercept: ₩n', lnr_clf.intercept_)
print('Coefficients: ₩n', lnr_clf.coef_)
```

```
Intercept:
 59.510105274688584
Coefficients:
 [ 6.38708739e-05  4.33908528e-01 -4.02566349e+01 -5.42185027e+01
  2.39701900e+01 -1.48178305e-03]
```

In [142]:
```
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix, f1_score
```

In [143]:
```
def get_clf_eval(y_test, pred):
    mse = mean_squared_error(y_test, lnr_clf_pred)
    rmse = np.sqrt(mse)
    print('MSE: {0:.3f}, RMSE :{1:.3f}' .format(mse, rmse))
    print('Variance score : {0:.3f}' .format(r2_score(y_test,lnr_clf_pred)))
```

In [144]:
```
get_clf_eval(y_test, lnr_clf_pred)
```

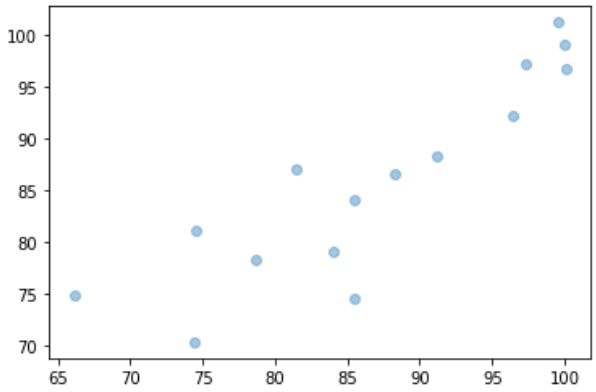
```
MSE: 23.948, RMSE :4.894
Variance score : 0.772
```

In [145]:
```
list(zip(X.columns, lnr_clf.coef_))
```

Out[145]:
```
[('asset', 6.387087390953729e-05),
 ('send_rate', 0.4339085283193433),
 ('max_latency', -40.25663494608417),
 ('min_latency', -54.21850269246551),
 ('avg_latency', 23.970189985045177),
 ('time_taken', -0.0014817830533482596)]
```

데이터 값이 적기 때문에 테스트셋과 훈련셋 분리 값을 변경하여 데이터 정확성을 높이는 방법을 적용하기로 함. 테스트 사이즈를 0.2에서 0.39로 변경시키면서 MSE와 RMSE 값이 줄어들고 R2_SCORE 점수도 0.772로 정확성이 높아진 것을 확인할 수 있음. 위의 상수항(Intercept)와 계수(Coeffient)를 통해 77.2%정도 설명할 수 있음. 데이터 값을 늘이고 private-data, 암호화 실험과 get, create, update, delete를 함수를 적용하는 데이터 값을 도출하면 더 정교한 수학 모델을 만들 수 있을 것으로 보임.

```
In [146]: score=r2_score(y_test,lnr_clf_pred)
          print('r2 socre is ',score)
          print('mean_sqrd_error is==',mean_squared_error(y_test,lnr_clf_pred))
          print('root_mean_squared error of is==',np.sqrt(mean_squared_error(y_test,lnr_clf_pred)))

          r2 socre is  0.7720554439303311
          mean_sqrd_error is== 23.94809820153922
          root_mean_squared error of is== 4.893679413441304

In [147]: accuracy = lnr_clf.score(X_test, y_test)
          print("Linear Regression test file accuracy:"+str(accuracy))

          Linear Regression test file accuracy:0.7720554439303311

In [148]: %matplotlib inline
          import matplotlib.pyplot as plt

In [149]: plt.scatter(y_test, lnr_clf_pred, alpha=0.4)
Out[149]: <matplotlib.collections.PathCollection at 0x19c3c6e5970>
```



정확성을 판단하기 위해 산점도 그래프를 그려서 확인해보면 분산이 있지만 선형모델로 괜찮은 수학적 분석을 할 수 있지 않을까 예상됨.

$$Th = 59.510105274688584 + 6.387087390953729e-05 \times A + 0.4339085283193433 \times Sr - 40.25663494608417 \times Mx - 54.21850269246551 \times mx + 23.970189985045177 \times Avgx - 0.0014817830533482596 \times Tx$$

$$Th = I + \alpha A + \beta Sr + \gamma Mx + \delta mx + \varepsilon Avgx + \zeta Tx$$

다변량 선형회귀로 77.2% 정확도로 설명하는 수식을 도출함. 상수항은 I로 정의함. Asset의 미지수는 A로 정의하고 계수는 α로 변환하여 αA로 변환함. SendRate는 미지수는 Sr로 정의하고 계수는 β로 변환함. MaxLatency는 미지수는 Mx로 정의하고 계수는 γ로 변환함. MinLatency는 미지수는 mx로 계수는 δ로 변환함. AvgLatency는 미지수는 Avgx로 계수는 ε으로 변환함. TimeTaken은 미지수는 Tx로 계수는 ζ로 변환함. 결과로 도출되는 산출량은 Th로 변환함. 수학적 분석을 할 수

있는 방안을 모색하여 데이터를 통해 수식을 도출하여 결과 값을 설명할 수 있는 수학식을 도출하는 방도를 실행함.

e30	80000	100.9	0.17	0.01	0.05	3557.037	96.4
e31	90000	101.1	0.11	0.01	0.03	4003.348	100.0
e32	100000	100.2	0.16	0.01	0.05	4447.643	94.7
e33	200000	101.1	0.51	0.04	0.21	8861.158	86.6
e34	300000	101.3	0.17	0.01	0.07	13301.483	97.3
e35	400000	101.1	0.15	0.01	0.04	17787.274	96.1
e36	500000	101.2	0.38	0.01	0.20	22289.659	88.9
e37	600000	101.6	0.18	0.01	0.09	26744.380	100.1
e38	700000	101.6	0.33	0.01	0.19	31233.005	95.0

실험을 진행할 실험계획 표를 작성함.

	Create				Update				Delete				Get			
	Assets	tXNumber	Tps	byteSize	Assets	tXNumber	Tps	byteSize	Assets	tXNumber	Tps	byteSize	Assets	tXNumber	Tps	byteSize
Public Data/Normal Data																
Public Data/Encrypt Data																
Private Data/Normal Data																
Private Data/Encrypt Data																

총 64가지의 실험진행이 필요함. 실험값을 변화시킬 Asset, txNumber, Tps, ByteSize의 변환되면서 진행시 더 많은 스크립트를 작성하여 진행할 것으로 예상됨. 지금 진행한 실험은 Get-Asset/PublicData-NormalData에 있는 64가지 경우 중 한 셀을 진행하였음. txNumber, Tps, ByteSize도 Asset과 동일하게 변화한다고 가정하면 30만 Asset의 경우 13301.483초가 걸려 약 3시간 40분이 걸린다. 16일 걸려서 데이터를 구하려고 하는 경우 4가지의 실험이 종료되야 하며 32일 정도 걸려서 결과를 산출한다고 하면 2가지 실험이 진행되어야 한다. 64일 진행한다고 계획하면 하루에 하나의 실험만 진행하면 된다. Asset 10만의 경우 1시간 14분 정도 걸림. 1-9, 10-90, 100-900, 1000-9000, 10000-100000까지 시간을 단위수를 대

략 계산해본 결과 7시간 40분정도 걸리는 것으로 예측됨. 하루 3단위의 실험 값을 소화할 수 있을 것으로 예측됨. 22일 정도 걸리는 것으로 계획하고 8월 말까지 데이터 결과를 실행하는 것으로 계획하기로 함. 아직 진행하지 않은 private-data/Encrypt-Data의 경우에는 컴퓨터에서 실험을 진행하는 중에 해결하는 것으로 진행하기로 함.
3. 차주 계획
- 계획한 실험 계획에 따라 데이터 산출