大阪市立大学
OSAKA CITY UNIVERSITY

情報指向ネットワークにおけるプログラマビリティ拡張のプロトタイプ作製

電子情報通信学会 ICN 研究会　2019.12

山澤 慶太*，浅井 貴大*，阿多 信吾**

* 大阪市立大学 工学部 電気情報工学科

** 大阪市立大学 大学院工学研究科 電子情報系専攻

研究背景

- ICNの対象
  - ストリーミングのような動的コンテンツにも適用
  - データ加工処理の要求といった適用も今後必要に
- ICNの持つプログラマビリティを活用
  - 静的コンテンツを要求するのと同じようにして、計算（ファンクション）を実行
  - 小さな処理を組み合わせて複雑で大規模な処理
- IoTとの組み合わせ
  - IoTデバイスの貧弱な計算能力をネットワークが補う
- 目標：ICN上でどれほどプログラマビリティを実現できるか？

目的

- プログラマブルネットワークとしての情報指向ネットワークの実現
  - API 呼び出し ⇄ Interest / Data 通信
    - 動的なコンテンツ取得
    - 計算資源の乏しいデバイスからも実行可能
  - ネットワーク層による最適化
    - アプリの開発者は下階層にICNを使うだけで恩恵
  - 分散型のアーキテクチャ
    - ノード数を増やすだけで水平スケーリング
  - サーバレスなアプリケーション
    - ネットワークに流せば自動で冗長化　耐障害性◎

課題

- ユーザビリティ
  - ICNの専門知識が必要
    - 学習コストの高さ
  - アプリケーション開発者への負担
    - コード記述量の増加
- ファンクションモビリティ
  - キャッシュできるのは実行結果のコンテンツのみ
    - 中間ノードの計算資源を利用できない
    - 必要なのはメカニズムのキャッシュ

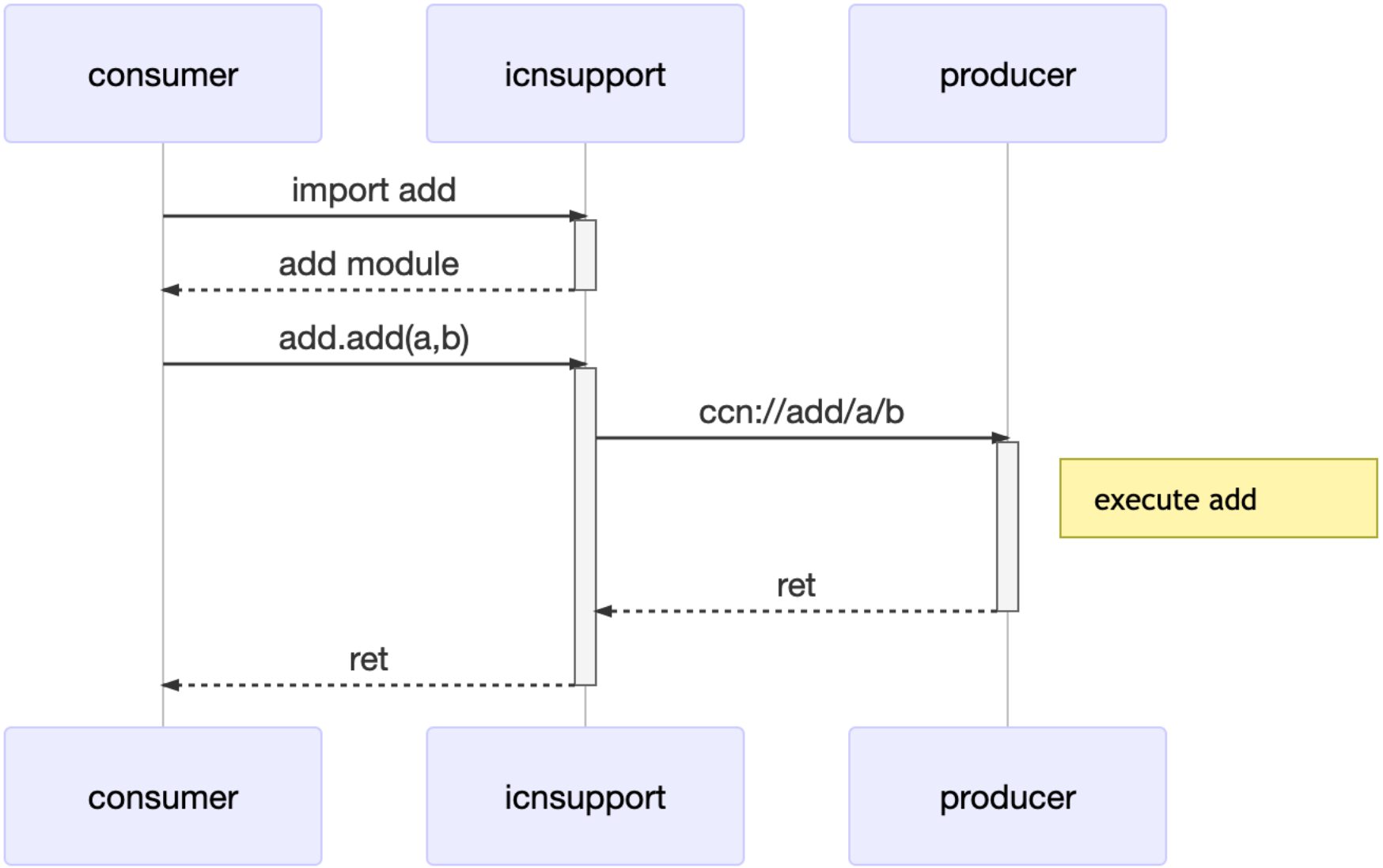
解決法

- Pythonアプリケーションにおける簡便なICNサポートの実現手法
  - ユーザビリティの課題を解決
  - Pythonの関数をファンクションとして使用
  - 将来的には全てのモジュールを使用可能に
- ファンクションキャッシュを行うアーキテクチャの開発
  - ファンクションモビリティの課題を解決
  - 中間ノードでファンクションのキャッシュと実行
  - 分散コンピューティング、エッジコンピューティングの実現

これまでの研究

- Pythonアプリケーションにおける簡便なICNサポートの実現手法を提示
  - “import icnsupport”の一行を挿入する
    - 以降，モジュールがインポートされた際に，ICNで利用できるかを調べ，可能なら自動的にICN呼び出しに
  - インポートフックを利用
  - 開発したプログラム
    - consumer.py（アプリケーション）
    - icnsupport モジュール
    - producer.py
  - 具体例の紹介
    - add モジュールを例

これまでの研究


consumer
icnsupport
producer
import add
add module
add.add(a,b)
ccn://add/a/b
execute add
ret
ret
consumer
icnsupport
producer

これまでの研究

consumer.py

```python
from time import time
import random

import add

param1 = random.randrange(10)
param2 = random.randrange(10)
ret = add.add(param1, param2)
print(ret)
```

consumer.py

```python
from time import time
import random

import icnsupport
import add

param1 = random.randrange(10)
param2 = random.randrange(10)
ret = add.add(param1, param2)
print(ret)
```

add.py

```python
def add(param1,param2):
    return int(param1)+int(param2)

```

```
$ python producer.py add.py
```

```
$ python consumer.py
```

icnsupportの目標

- アプリケーション上で，全てのモジュールのメソッドをICN呼び出し可能にする
  - まだまだ利用範囲が限られている
  - 特に改善すべき制約
    - 単一モジュールから単一関数
      - モジュールとメソッドが同名のものしか不可能
        - add.py から add メソッドしか呼び出せない
    - 型情報がない
      - str 型のみしか送受信できない
    - パケットサイズを上回るデータの送受信

具体的な目標

consumer.py

```
1  import random
2  import icnsupport
3
4  import calc
5  import make_list
6
7  param1 = random.randrange(10)
8  param2 = random.randrange(10)
9  ret_add = calc.add(param1, param2)
10 ret_sub = calc.sub(param1, param2)
11 ret_list = make_list.square()
12 print(ret_add, ret_sub)
13 print(type(ret_add))
14 print(len(ret_list))
15 print(type(ret_list))
```

calc.py

```
1 def add(param1, param2):
2     ret = int(param1)+int(param2)
3     return ret
4
5 def sub(param1, param2):
6     ret = int(param1)-int(param2)
7     return ret
```

make_list.py

```
1 def square():
2     ret = [i**2 for i in range(10000)]
3     return ret
```

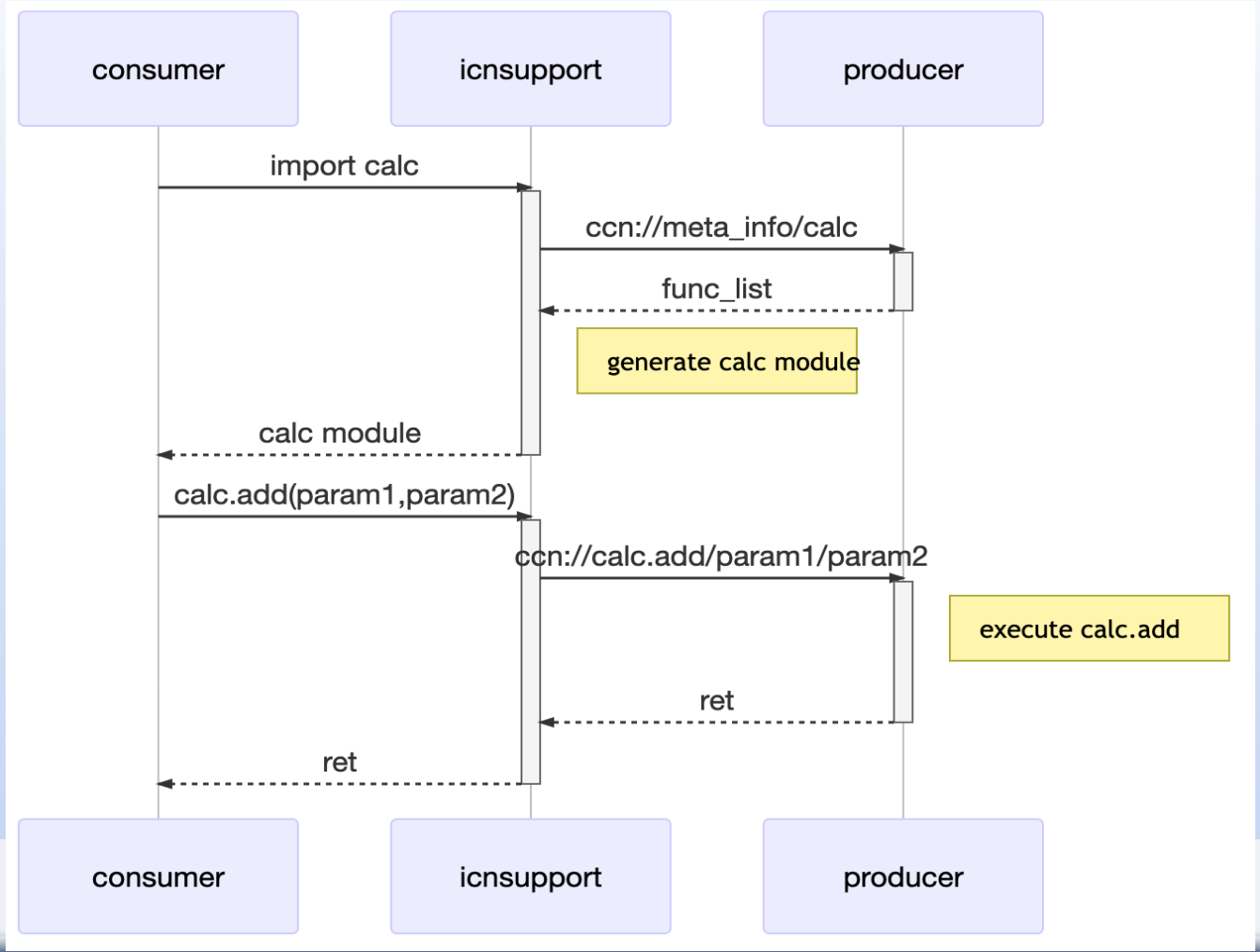
単一モジュールから単一関数の制約

- モジュールと関数が同一名のものに限られる
  - インポートフックで得られる情報はモジュール名のみ
    - そのモジュール内に，どのようなメソッドが存在するかが分からない
  - icnsupport内の対応すべきコード

```
class IcnModuleLoader:
  def load_module(self, fullname):
    m = imp.new_module(fullname)
    m.__loader__ = self
    setattr(m, fullname, (lambda *args: invoke_icn(fullname, args)))
    sys.modules.setdefault(fullname, m)
    return m
```

単一モジュールから単一関数の制約

consumerへレジストされた関数名の通知



単一モジュールから単一関数の制約

モジュール生成部分のコード

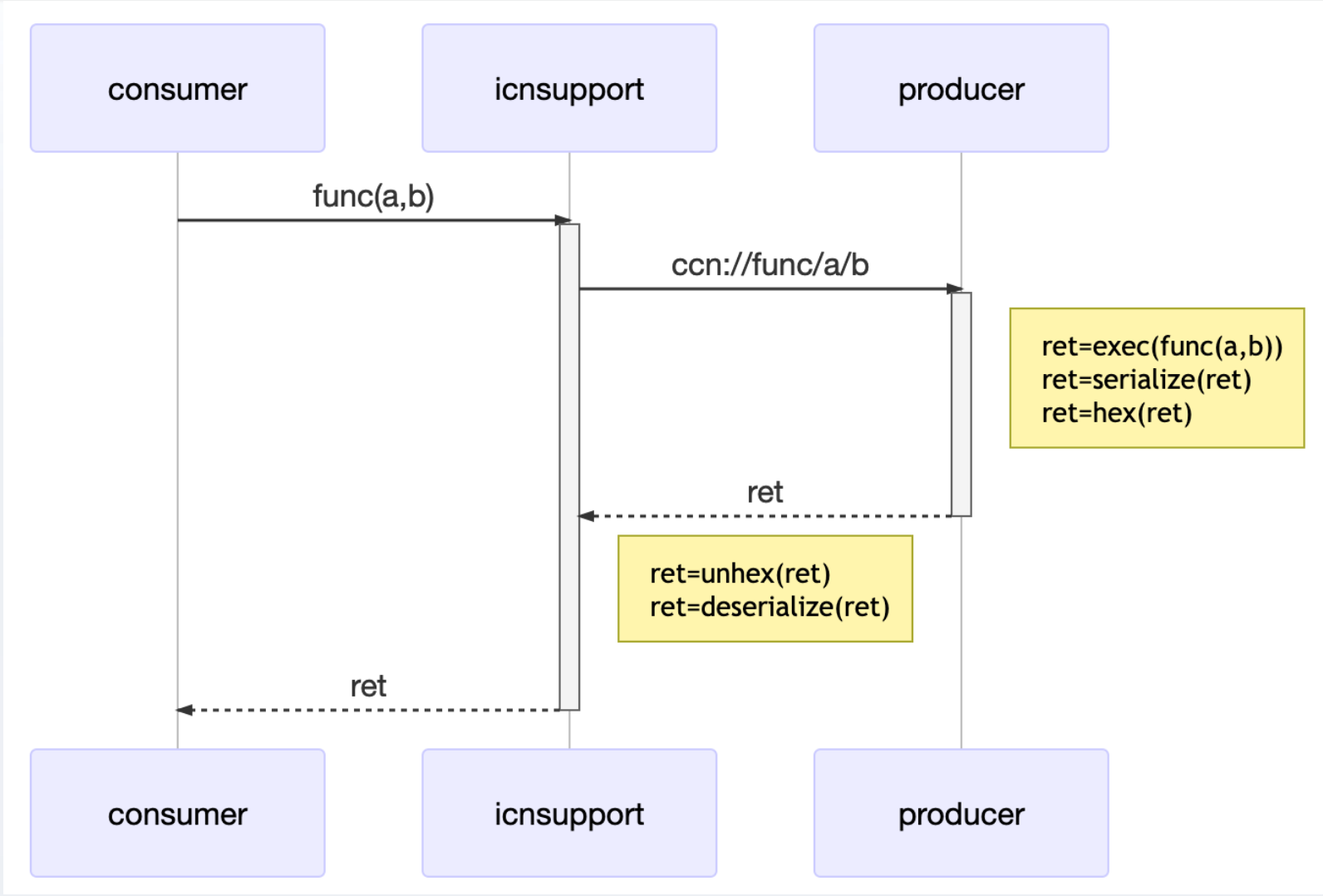
```
class IcnModuleLoader():
    def __init__(self, func_list, handle):
        self.func_list = func_list
        self.handle = handle
    def load_module(self, fullname):
        module = imp.new_module(fullname)
        module.__loader__ = self
        for attr in self.func_list:
            funcname = fullname+'.'+attr
            setattr(module, attr, lambda *args,name=funcname:
invoke_icn(name, args))
        sys.modules.setdefault(fullname, module)
        return 0
```

- consumer.py内で例えば，calc.add()が呼び出されたら，ccn://calc.addへのICN呼び出しへ

戻り値の型に対する対応

- 元々送受信可能なのはstr型
  - データの型が分からない
  - データをシリアライズすることで，型情報も合わせて送信する
  - 解決方法
    - producer側でデータをシリアライズ
    - そのバイナリを16進数文字列に変換して送信
    - icnsupport側で16進数文字列をバイナリに変換
    - バイナリをデシリアライズする
  - シリアライズにはpickleモジュールを利用
  - base64はbytes型のため16進数文字列に変換

戻り値の型に対する対応



動作様子

```
root@producer:~/programs# python producer.py calc.py
[cefpyco] Configure directory is /usr/local/cefore
[+] Register function from calc ... add
[+] Register function from calc ... sub
Info: Succeeded in receiving Interest packet with name 'ccn:/func_meta/calc' (#chunk: 0)
Info: Succeeded in receiving Interest packet with name 'ccn:/calc.add/5/9' (#chunk: 0)
[+]Execute Function
14

[+]Serialize data
b'\x80\x03K\x0e.'

Send Data:80034b0e2e
^C^C^C^CTraceback (most recent call last):
```

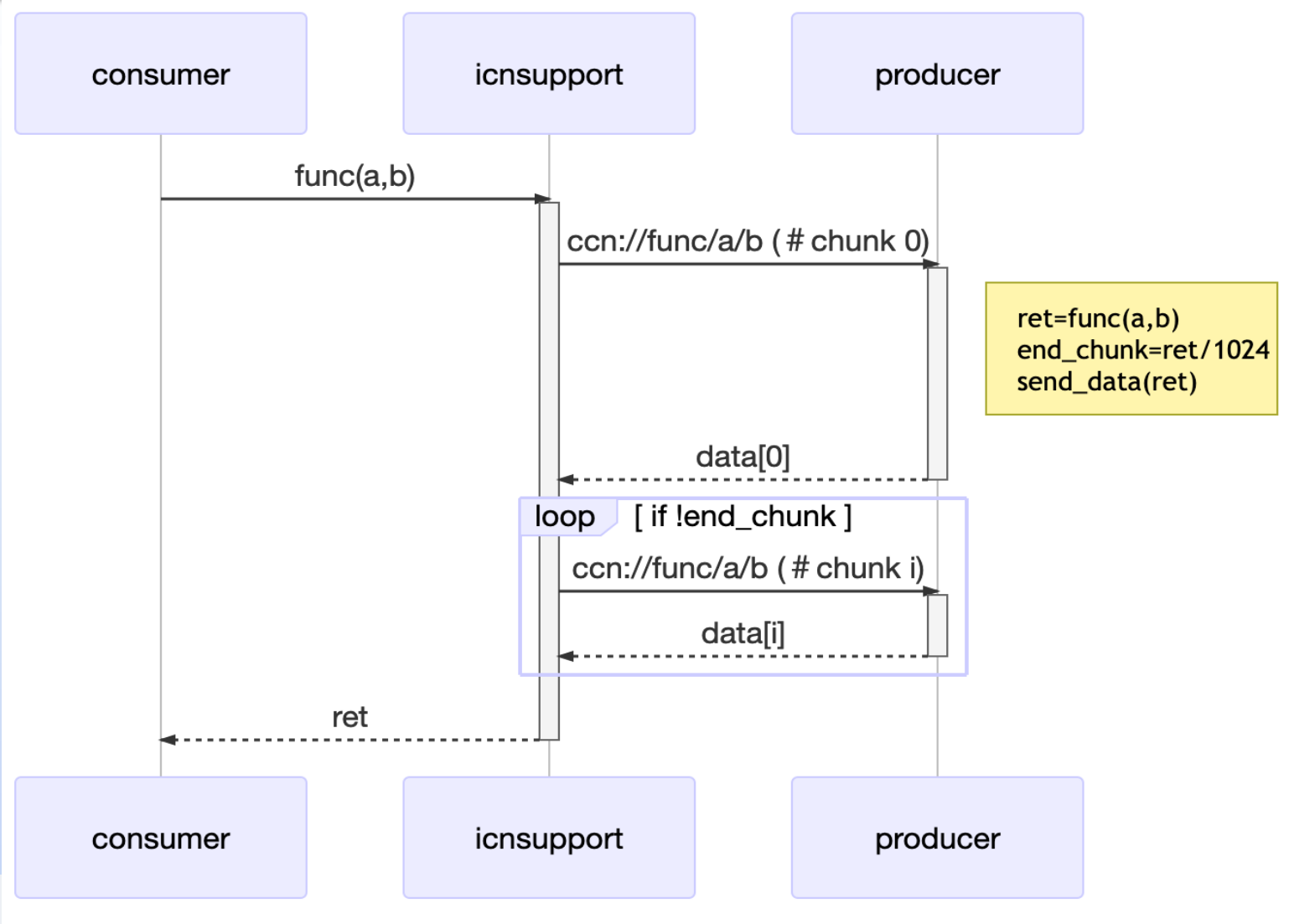
```
root@consumer:~/programs# python consumer.py
[cefpyco] Configure directory is /usr/local/cefore
[cefpyco] Configure directory is /usr/local/cefore
Info: Succeeded in receiving Data packet with name 'ccn:/calc.add/5/9' (#chunk: 0) and p
ayload '80034b0e2e' (10 Bytes)

[+] Serialized Data
b'\x80\x03K\x0e.'
[+] Deserialize Data
14
<class 'int'>
root@consumer:~/programs#
```

パケットサイズを上回るデータの送受信

- サイズが大きいデータを送受信
  - icnsupportとproducerで自動的に行わせる
- Dataのチャンク数
  - producer側は?
    - send_data()の第四引数で指定可能
  - icnsupport側は？
    - send_data()の第四引数の値を，CcnPacketInfo.end_chunkから得られる

パケットサイズを上回るデータの送受信



パケットサイズを上回るデータの送受信

producerの一部抜粋

```
ret = exec_func(info, modules, func_list)
serialized_ret = pickle.dumps(ret)
data = divide_data(serialized_ret)
send_data(handle, info, data)
```

データ送信部分

```
while True:
    if info.is_succeeded and info.chunk_num == i:
        handle.send_data(info.name, data[i].hex(), i,len(data))
        i += 1
    if i >= len(data):
        break
    info = handle.receive()
```

パケットサイズを上回るデータの送受信

- icnsupportの一部抜粋(受信部分)

```
data = bytes("".encode())
while True:
    handle.send_interest(name, chunk_num)
    info = handle.receive()
    if info.is_succeeded and info.name == name:
        data += bytes.fromhex(info.payload)
        chunk_num += 1
        if chunk_num == info.end_chunk_num:
            break
return data
```

- consumerにreturnする部分

```
data = send_interest(name, args)
ret = pickle.loads(data)
return ret
```

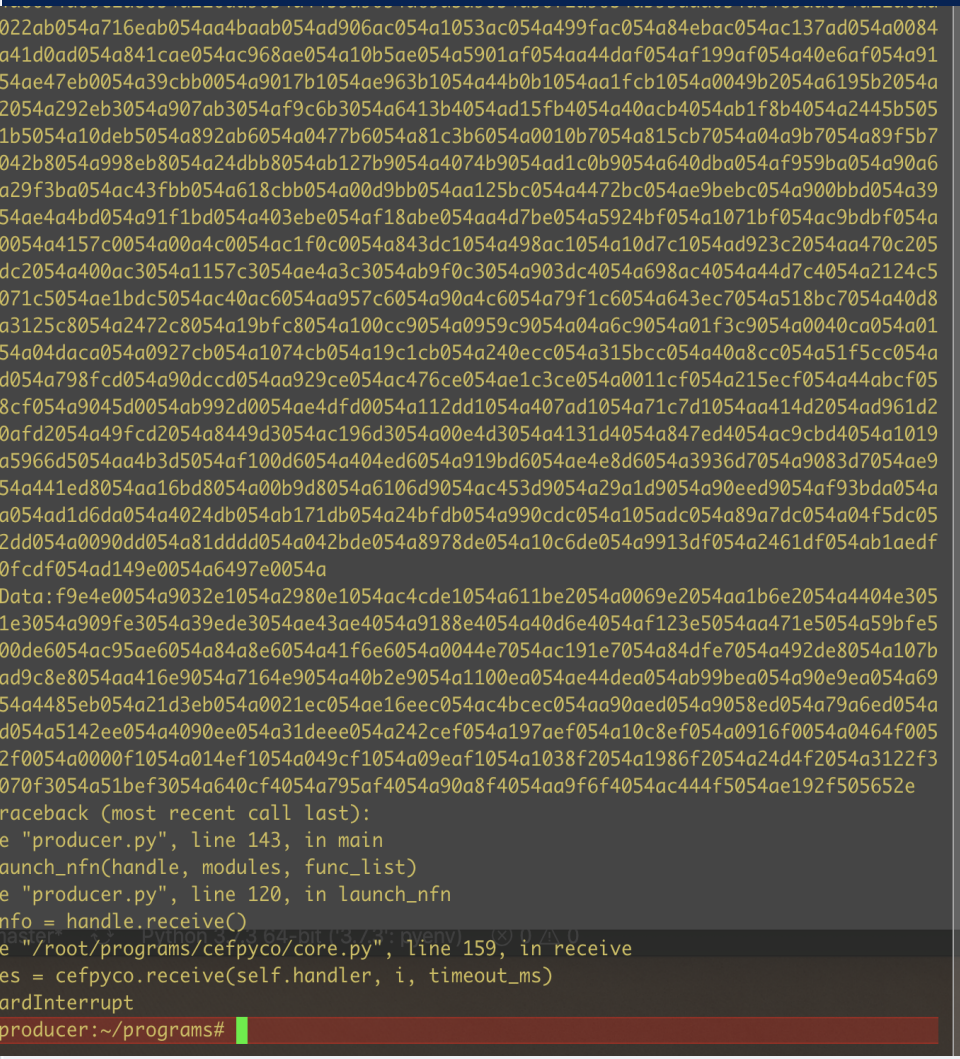
環境構築

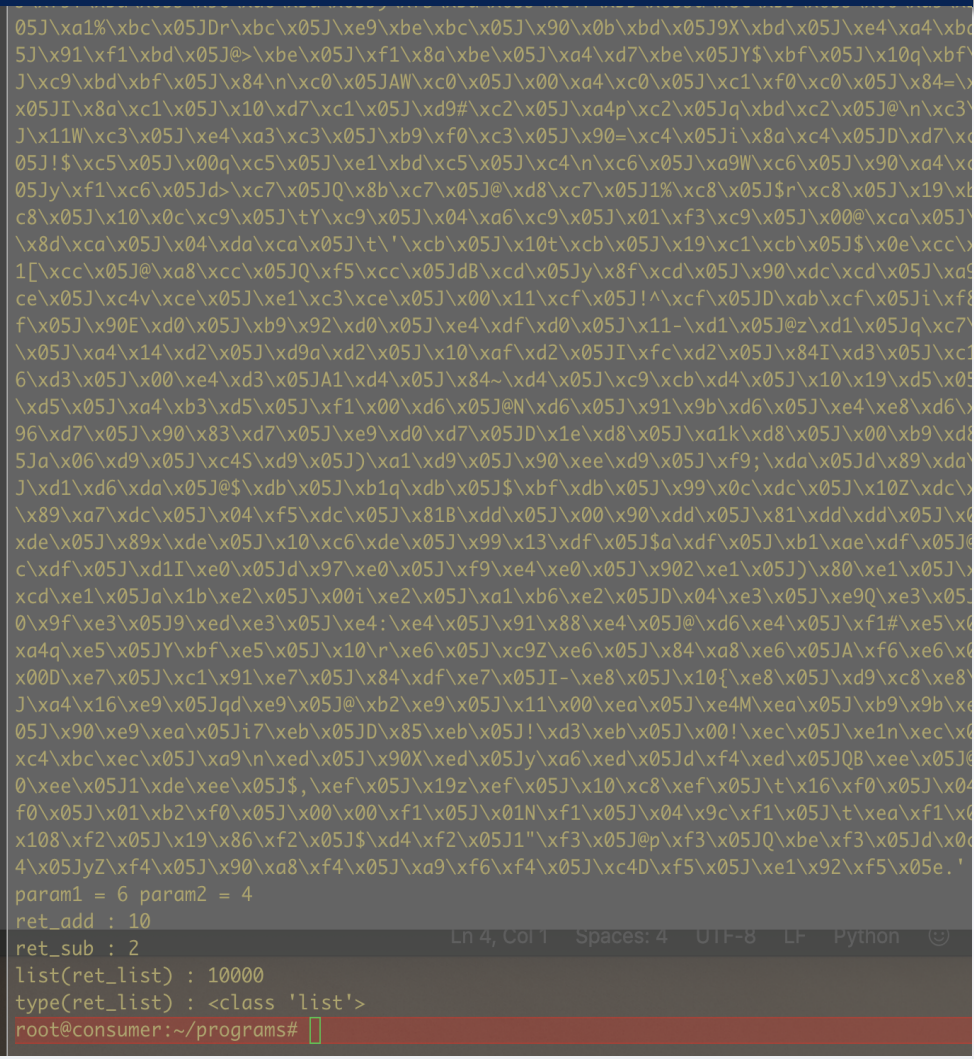
- 動作環境
  - Dockerのコンテナ
    - consumer,producerは1ノードずつ
  - Cefore 0.8.1/cefpyco 0.3.7
  - Python 3.7.4
- 実行プログラム
  - consumer.py(アプリケーション)
  - icnsupport(consumer上のモジュール)
  - producer.py
  - calc.py, make_list.py(producer上で動作)

動作検証

- consumer.pyの実行
  - 複数モジュール，複数メソッドが呼び出されるか
    - calc.py，make_list.pyモジュールをimport
    - calc.add()，calc.sub()，make_list.square()で確認
  - str以外の型を受け取れるか
    - ビルトインのtype()関数で確認
  - データサイズが大きいものが送受信できるか
    - 返されるリストの長さは4*10000(byte)
    - 自動的にマルチチャンクになる

デモ

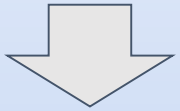

raceback (most recent call last):
e "producer.py", line 143, in main
aunch_nfn(handle, modules, func_list)
e "producer.py", line 120, in launch_nfn
nfo = handle.receive()
e "/root/programs/cefpyco/core.py", line 159, in receive
es = cefpyco.receive(self.handler, i, timeout_ms)
ardInterrupt
producer:~/programs#


param1 = 6 param2 = 4
ret_add : 10
ret_sub : 2
list(ret_list) : 10000
type(ret_list) : <class 'list'>
root@consumer:~/programs#

ファンクションキャッシュ

- **ICN でファンクションの実行を要求する拡張**
  - しかし、キャッシュできるのは計算の実行結果のみ
    中間ノードの計算リソースを利用する事はできない

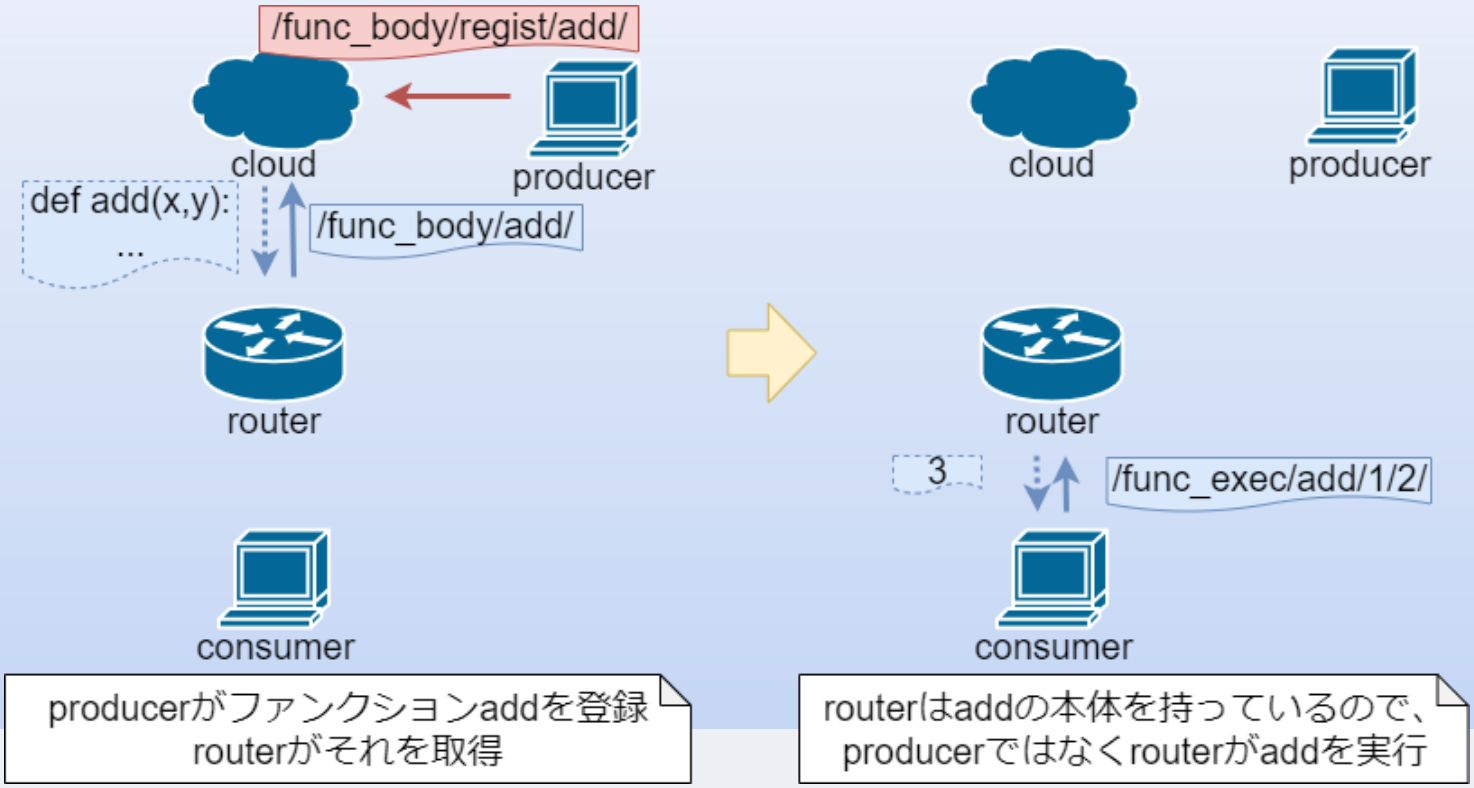
- **中間ノードによる最適化の効果**
  - 計算を行うノードを分散する⇛負荷分散
  - エッジに近いノードが計算を行う⇛応答時間の短縮



- 中間ノードでファンクションキャッシュができるアーキを開発

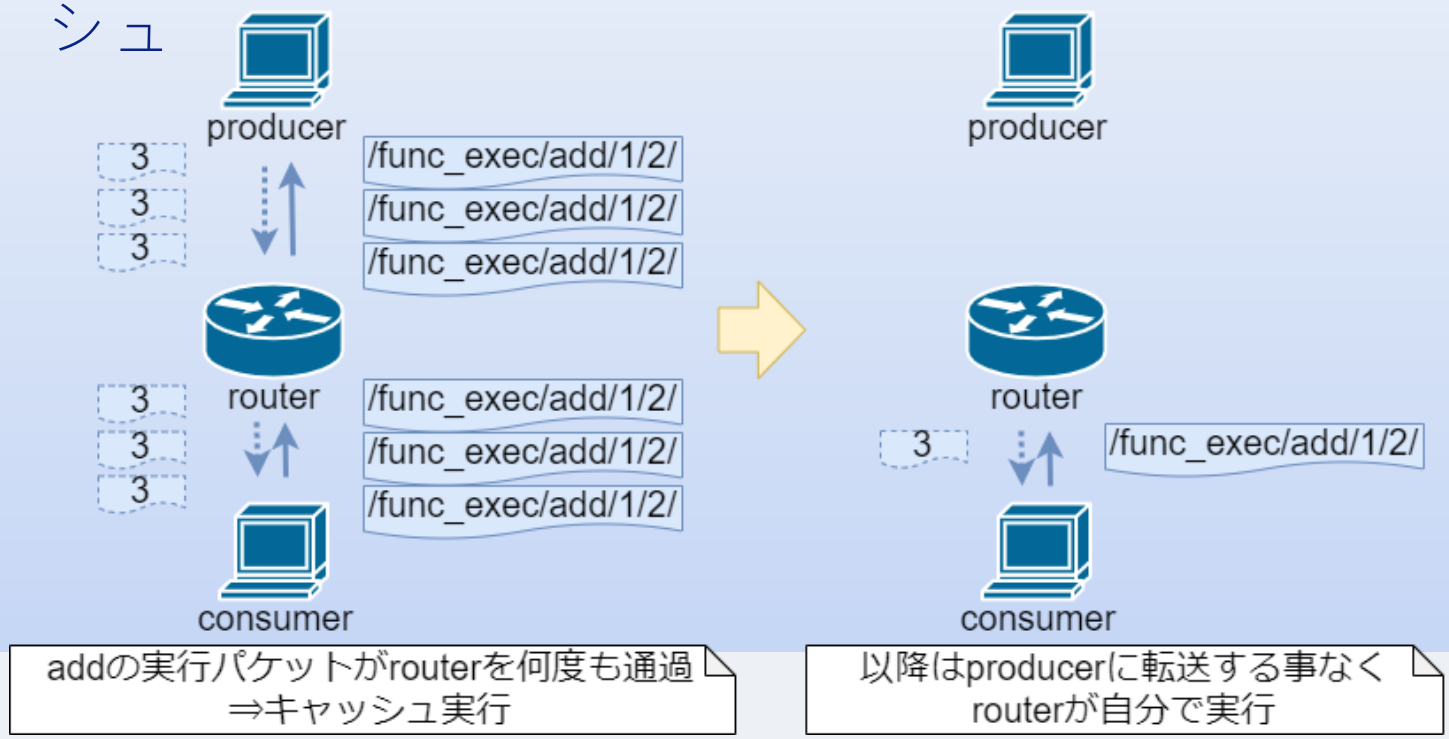
動作原理 (1)

- 中間ノードがファンクションを実行するためには、ファンクションのプログラム本体が必要
  - ファンクション名に加えプログラムもレポジトリへ



動作原理 (2)

- **キャッシュしたいのは、要求の多いファンクション**
  - ルータを通過するファンクション実行要求パケットを監視
  - 実行回数をカウント、一定の閾値を超えたらキャッシュ



追加する仕様

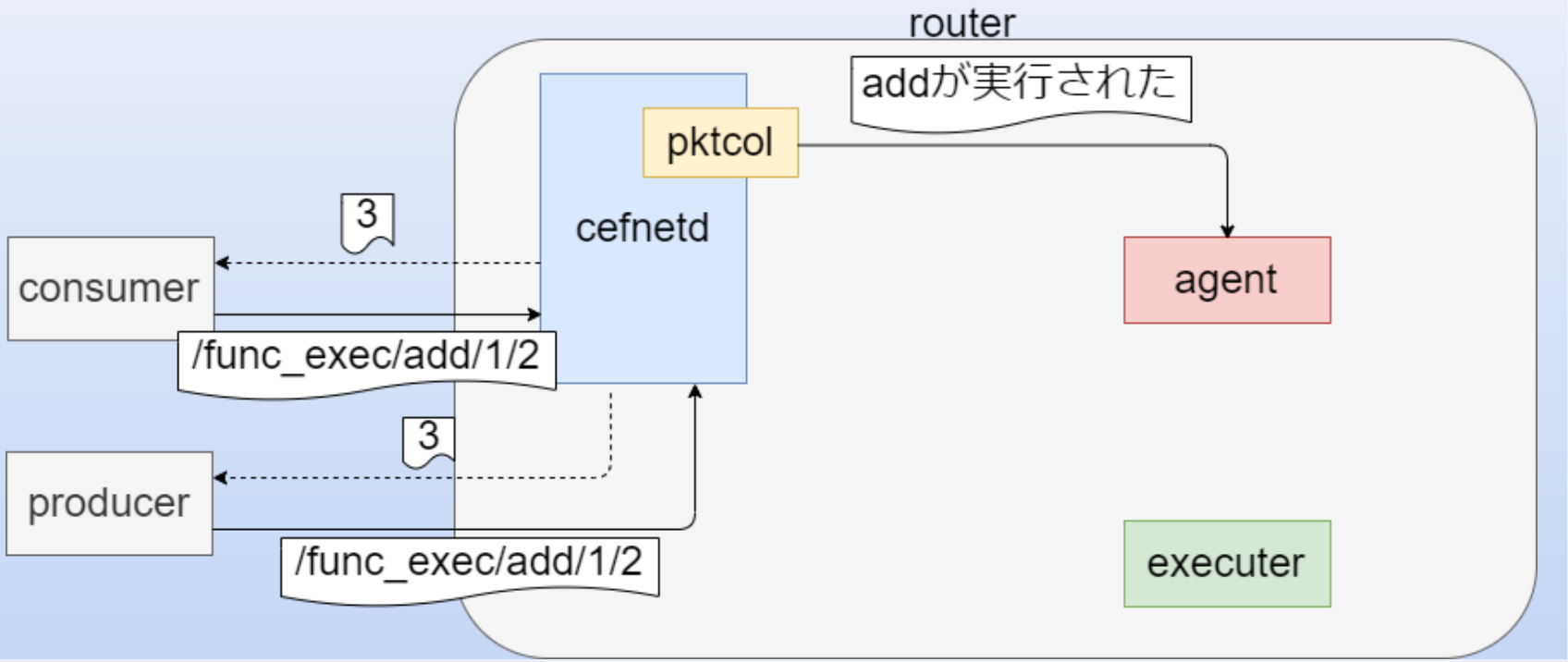
- **ファンクション本体の取得要求をできるようにする**
  - ccn:/func_body/regist/ で登録
  - ccn:/func_body/ で取得
- **実行回数を数えるために、ファンクションの実行とそれ以外のパケットを区別**
  - ccn:/func_exec/ というプリフィクスの付いたパケットを実行要求とする
  - フォワーディングするパケットを監視する変更を cefnetd に加える

システムの構成

- **3つのコンポーネントをルータに配置**
  - *agent*
    - システムの中心となる制御プログラム
  - *pktcol (*packet collector)
    - cefnetd 内を通過するパケットを監視する
  - *executer*
    - ファンクションの実行を行う

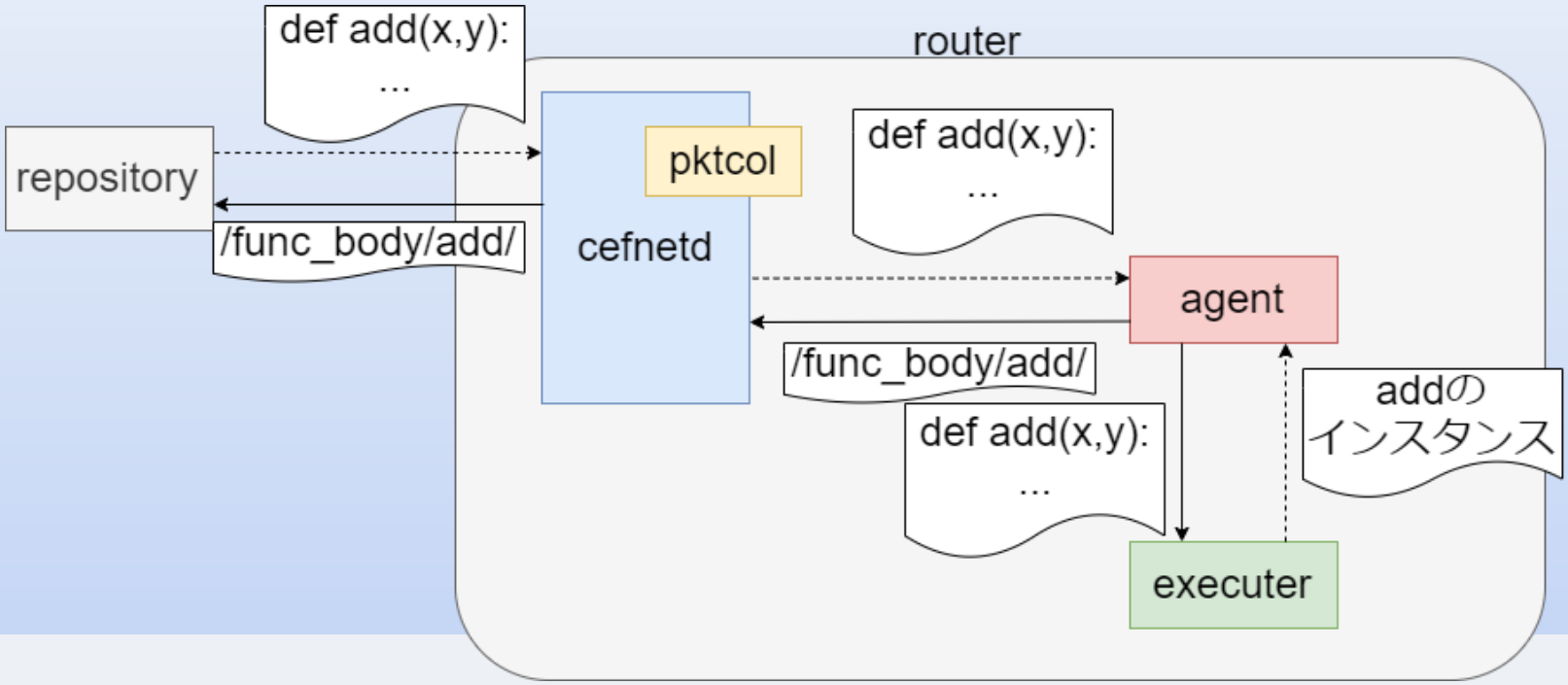
パケットの監視

- **pktcol が cefnetd 内を通過する *ccn:/func_exec/* で始まるパケットを監視**
  - agent に該当するパケットに関する情報を渡す



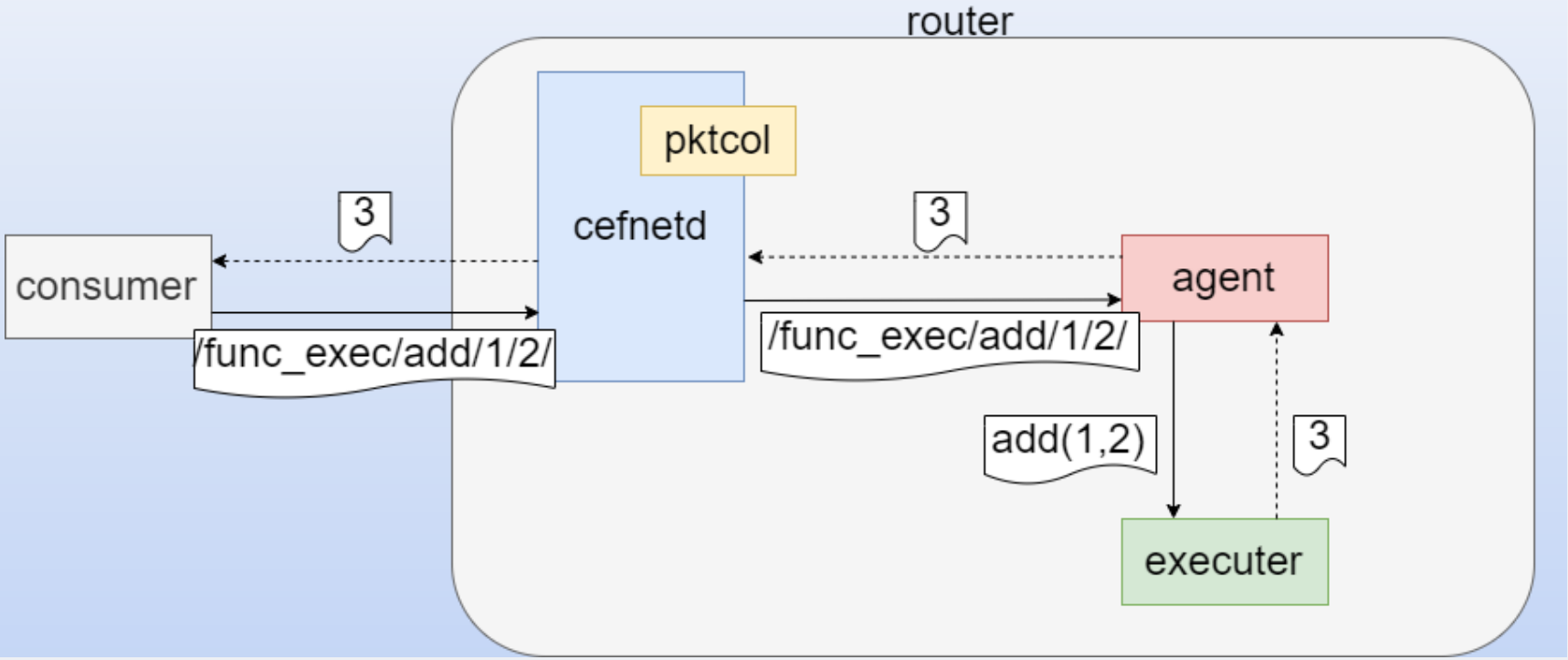
キャッシュの実行

- 一定回数実行パケットが通過すると、agent がレポジトリからファンクションを取得
  - 取得したものは executer へ渡され、インスタンス（実行のためのハンドル）を作成して agent に返す

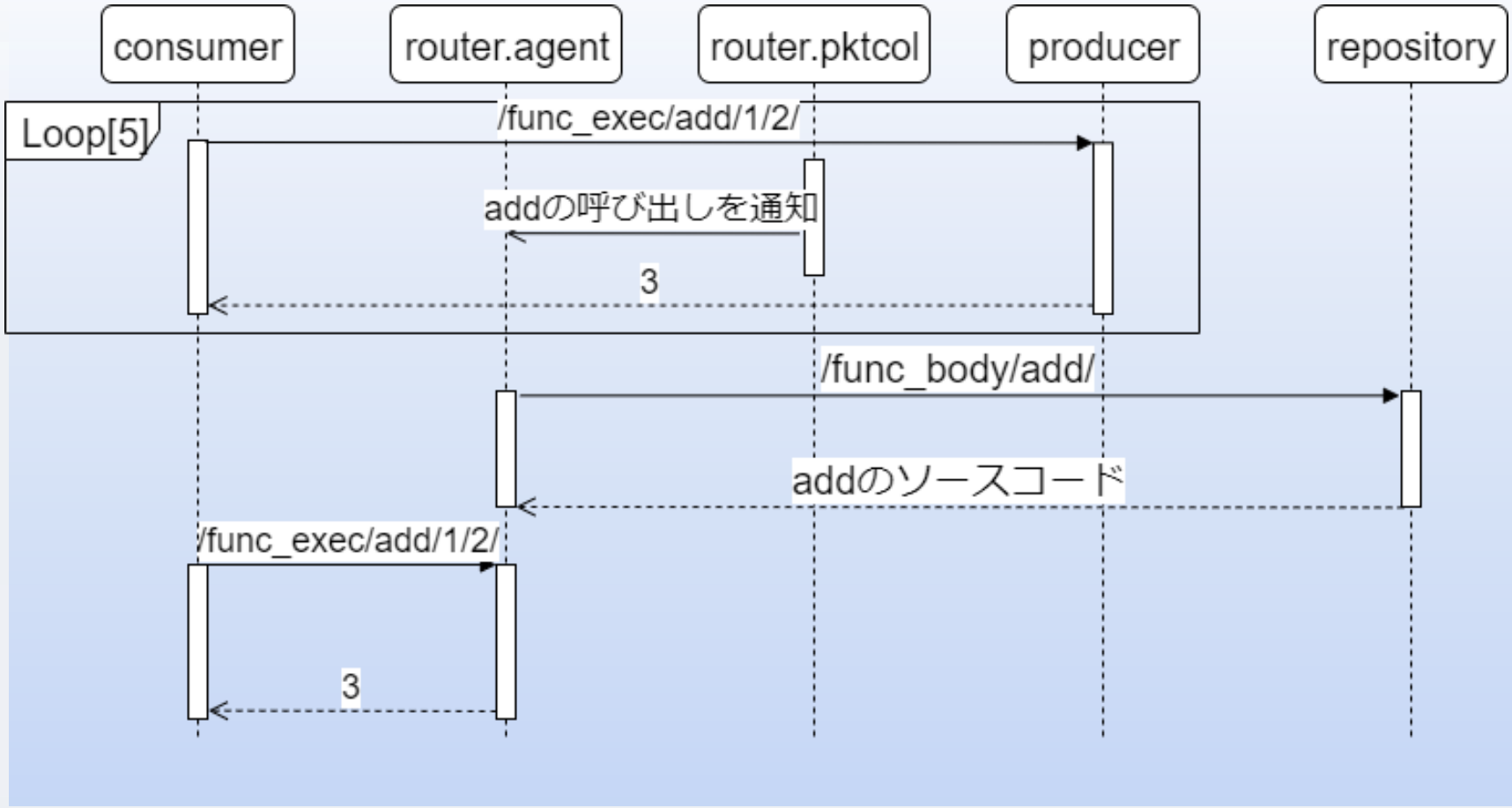


ルータによるファンクション実行

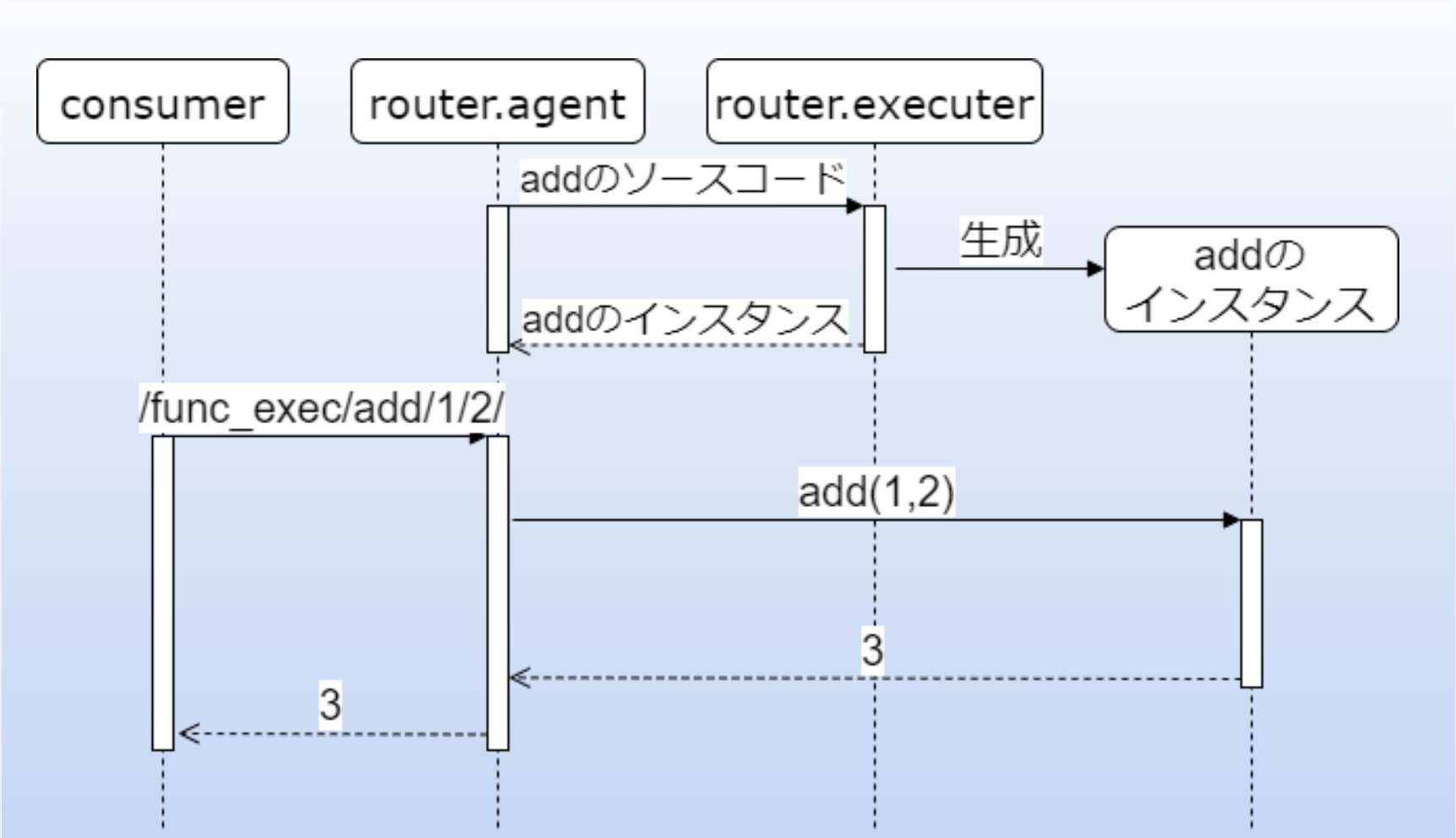
- キャッシュが完了すると、ルータがファンクションを実行できる
  - agent がインスタンスを通じて executer で実行



キャッシュ実行のシーケンス図



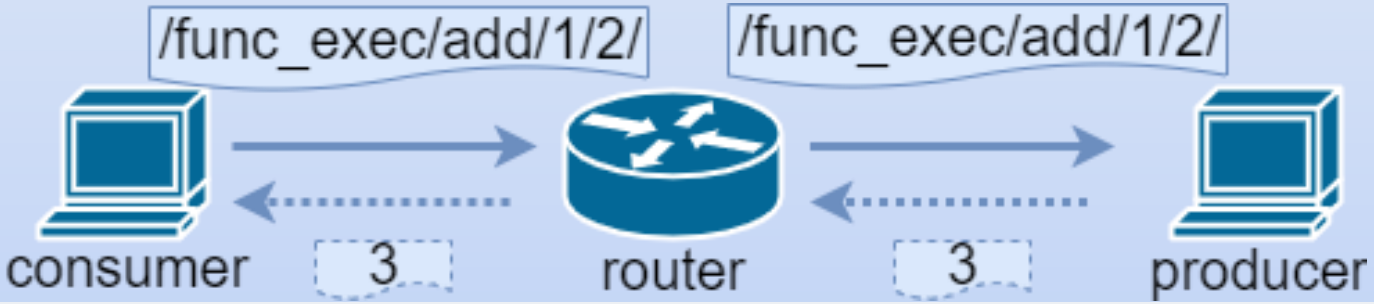
ファンクション実行要求への返答のシーケンス図



動作確認 (1)

- 動作環境
  - Ubuntu 18.04.3 LTS
    - VMWare Workstation 15 Player on Windows10
  - Cefore 0.8.1/cefpyco 0.3.7 のカスタム版
  - Cefore-Emu 0.1.0
    - Cefore が提供するエミュレータ
    - Mininet-CCNx を元にしている

- 確認に使用するネットワーク



動作確認 (2)

- 条件
  - producer がファンクション add を提供
    - 2つの整数を加算して返す
  - ファンクション本体は Python プログラム
  - router 上でキャッシュシステムを動作
  - 5回実行要求があるとキャッシュ
- 確かめること
  - router–producer 間の接続を切断して、consumer が add の実行結果を取得できるかどうか
  - router がキャッシュしていれば結果が返ってくるはず
- ネットワーク操作コマンドの実行には Cefore-Emu のスクリプトを用いる

動作確認 (3)

- 初期状態での実行結果を確認する
- script_uncached.txt

```
consumer cefgetfile ccn:/func_exec/add/1/10 -f linked.txt
link router producer down
consumer cefgetfile ccn:/func_exec/add/2/20 -f ¥
unlinked.txt
```

- linked.txt の中身は"11"
- unlinked.txt は生成されない（受信失敗）

動作確認 (4)

- キャッシュ後の動作を確認する
- script_cached.txt

```
consumer cefgetfile ccn:/func_exec/add/1/10
consumer cefgetfile ccn:/func_exec/add/2/20
consumer cefgetfile ccn:/func_exec/add/3/30
consumer cefgetfile ccn:/func_exec/add/4/40
consumer cefgetfile ccn:/func_exec/add/5/50
link router producer down
consumer cefgetfile ccn:/func_exec/add/6/60 -f cached.txt
```

- cached.txt の中身は ″66″
  - add を router が実行して返している

まとめと課題

- ICNのプログラマビリティを向上させるファンクション実行システムの改良
  - icnsupport の機能拡張
  - ファンクションキャッシュ機能の追加
- 課題
  - ファンクションへの引数の制約
    - マルチチャンク、型情報の送受信
  - 基底クラス型以外のデータの送受信
  - キャッシュの判断を行う戦略の評価
    - 一定回数実行後キャッシュという戦略の妥当性
    - キャッシュ戦略の評価方法