

Документация MicroServer.Debugger

1. Описание

«MicroServer.Debugger» — это утилита для получения отладочных данных из «Microservices.MicroServer». Данная программа реализует все отладочные методы сервера, но не предназначена для полноценного участия в работе микро-сервисной системы (программа позволяет мониторить данные на сервере, но не модифицировать их).

2. Начало работы

Для начала работы отладочной программы требуются адреса GET-JOB и POST-JOB. После ввода адресов, отладочная программа пытается подключиться к серверу, чтобы убедиться в корректности введенных данных. Если один из адресов будет введен некорректно, программа выдаст ошибку и описание этой ошибки, что должно помочь в устранении проблемы.

Если вы ранее уже вводили адреса, программа их сохраняет в файлы в своей рабочей директории. Это избавить Вас от необходимости вводить адреса при каждом запуске. Однако, если вы введёте адрес неправильно, или же Вам нужно будет ввести другой адрес, Вы можете удалить файлы в папке программы с именем «GET» и «POST», либо же отредактировать их содержимое.

После успешного выполнения проверки адресов в консоли будет выведено сообщение «SERVER VALIDATED SUCCESSFULLY».

ЗАМЕТКА: после выполнения перезапуска сервера **не требуется** перезапускать данную программу. Отладочная программа не проверяет работает сервер или нет после первичной проверки адресов, так что сервер может и не работать во время работы отладчика, однако, если сервер не будет активен во время выполнения команды отладчика, отладчик выдаст ошибку.

Важно! Использование этой программы имеет смысл только в том случае, если используется отладочная редакция сервера. В ином случае почти все команды, предоставляемые этой программой, будут недоступны.

3. Вывод данных

Все успешные результаты выполнения команд, в ходе которых отладочная программа получает полезные данные от сервера, записываются в файлы с именем, соответствующем имени команды, расположенным в рабочей папке программы (обычно, рядом с исполняемым файлом).

В файл сохраняются «сырые» данные, то есть непосредственно те, которые пришли с сервера, поэтому при создании парсера структур из файлов вывода, используйте документацию MicroServer.

4. Отладочные команды

Программа отладки (читайте «мониторинга») реализует следующие запросы сервера микро-сервисов:

1. MicroServer.25367be645.ExternalStatus
2. MicroServer.25367be645.DebugEdition.getPendings
3. MicroServer.25367be645.DebugEdition.getLocallyAvailibleTypes
4. MicroServer.25367be645.DebugEdition.getTypesStatistic
5. MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot
6. MicroServer.25367be645.DebugEdition.retriveGetHistory
7. MicroServer.25367be645.DebugEdition.retrivePostHistory

Как можно заметить, запрос под номером 1 не является эксклюзивом отладочной редакции сервера, а, значит, может быть выполнен и для основной редакции сервера.

Список доступных команд отладки программы (как можно заметить, данный список сопоставлен с предыдущим по номерам элементов):

1. **exstat** – получить статусы переполнения/недобора «внешних» типов (т.е. хранящихся в том числе и во внешнем хранилище).

2. **pend** – получить список требований, попадания пакетов под которые ожидают полупившие GET-запросов в момент выполнения команды.

3. **loctp** – получить список доступных локально типов (выводит просто имена типов).

4. **tystat** – получить список имён типов с количеством пакетов этого типа как во внутреннем, так и во внешнем хранилищах.

5. **isnap** – получить **полный** снимок внутреннего хранилища (получает все пакеты целиком, *но на экран выводит только отображаемые данные, а полный вариант сохраняет в файл*). Использовать эту команду следует только тогда, когда функционала других недостаточно, так как для передачи данных всего хранилища может потребоваться некоторое время.

6. **gethist** – получить список пакетов, поступивших на сервер за промежуток с момента последнего выполнения этой команды до момента текущего выполнения этой команды (получает все пакеты целиком, а также требования, под которые был подобран пакет и время приёма, *но на экран выводит только отображаемые данные, а полный вариант сохраняет в файл*). Если за промежуток между выполнениями этой команды на сервере скопится больше 512 пакетов для данного отладочного списка, этот список будет урезан путём удаления первых 128 пакетов.

7. **posthist** – получить список пакетов, полученных сервисами с сервера за промежуток с момента последнего выполнения этой команды до момента текущего выполнения этой команды (получает все пакеты целиком и время отправки, *но на экран выводит только отображаемые данные, а полный вариант сохраняет в файл*). Если за промежуток между выполнениями этой команды на сервере скопится больше 512 пакетов для данного отладочного списка, этот список будет урезан путём удаления первых 128 пакетов.

5. Обслуживающие команды отладчика

- **flush** – удалить из файлов все данные, которые были получены **не за** текущий сеанс. Данная команда может быть вызвана любой момент работы отладчика и не тронет данные, полученные за текущий сеанс. Полезна для уменьшения размера

файлов сохранения выводов команд (чаще всего находятся в папке с программой), так как со временем размер этих файлов возрастает.

- **validate** – выполнить тест сервера (как при запуске отладчика). Это может быть полезно, если, например, нужно убедиться в том, что сервер ещё работает.
- **exit** и **help** не нуждаются в дополнительном описании.

6. Примеры работы отладчика

Вот как в целом выглядит работа с отладчиком:

```

==== MiscoServer services communication debugger ===
<> GET url was loaded from GET.txt
<> POST url was loaded from POST.txt
== SERVER VALIDATED SUCCESSFULLY ==

<> Enter the debugger command: pyns
The command is not recognized. Enter "help" for help.

<> Enter the debugger command: help
-- Each command saves the result of its execution to a file in the working directory --
exstat - display the list of externals statuses
pend - display the list of pending GET connections
loctp - display the list of locally available types
tystat - display the count of jobs for each type
isnap - display the snapshot of local storage
gethist - display the list of processed GET-requests
posthist - display the list of processed POST-requests
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command:

<> Enter the debugger command: help
-- Each command saves the result of its execution to a file in the working directory --
exstat - display the list of externals statuses
pend - display the list of pending GET connections
loctp - display the list of locally available types
tystat - display the count of jobs for each type
isnap - display the snapshot of local storage
gethist - display the list of processed GET-requests
posthist - display the list of processed POST-requests
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command: exstat
<nothing to show>

<> Enter the debugger command: pend
<nothing to show>

<> Enter the debugger command: loctp
<nothing to show>

<> Enter the debugger command: tystat
<nothing to show>

<> Enter the debugger command: isnap
<nothing to show>

<> Enter the debugger command:

```

В верхнем снимке вверху видно, что адреса GET-JOB и POST-JOB были автоматически взяты из файлов (см. «начало работы»). Также здесь будет уместно упомянуть, что программа мониторинга не различает верхний/нижний регистры букв и игнорирует пробельные символы до и после команды.

Теперь стоит рассмотреть выводы команд `posthist` и `gethist`:

```
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command: exstat
<nothing to show> mean that there is nothing to show

<> Enter the debugger command: pend
<nothing to show>

<> Enter the debugger command: loctp
<nothing to show>

<> Enter the debugger command: tystat
<nothing to show>

<> Enter the debugger command: isnap
<nothing to show>

1 <> Enter the debugger command: 2 posthist 3
21:40:42.7 Type<text> MicroServer.DebugEdition.bc18e76bd30a0d773deb201cd66bd725367be645.TEST; 4 hidden-id<text> 5 id725367be645; Cont
ent: <text> 6 bc18e76bd30a0d773deb201cd66bd725367be645

<> Enter the debugger command: 7 gethist 8
21:40:42.7 Requested type<text> 9 MicroServer.DebugEdition.bc18e76bd30a0d773deb201cd66bd725367be645.TEST; Requested id<text> 10 null;
Found type 11 [SAME]; Found hidden-id<text> 12 id725367be645; Content: 13 <text> 14 bc18e76bd30a0d773deb201cd66bd725367be645

<> Enter the debugger command:
```

- Начать стоит с posthist, где (номера красных рамок):

1 – время поступления пакета на сервер (более точное время записано в файл)

2 – описание представления имени типа: в данном случае – текст, но, если бы тип был непечатной строкой, в поле 3 были бы выведены байты в hex-формате, а в поле 2 – «(hex)».

3 – имя типа в некотором представлении (в виде строки в данном случае). Пакет с таким типом отладчик шлёт на сервер при валидации.

4 – означает, что id у пакета скрытый

5 – по аналогии с 2, но по отношению к полю 6

6 – id, который имеет пришедший пакет (хоть он и скрыт, но ОН ЕСТЬ)

7 – по аналогии с 5 и 2, но немного больше значений: может быть не только «(text)» и «(hex)», но и «[boolean]», и «[NULL]», «JSON.Object» или даже «[...]Length=>» (после знака равенства идёт размер массива) и др. Всё это многообразие связано с тем, что поле content структуры basic content может содержать всё что угодно (в рамках дозволенного форматом JSON), но далеко не всё можно отобразить в терминале, не замусорив его полностью. Поэтому иногда поле 8 может отсутствовать, а иногда текстом показан фактически не текст («[boolean]», например). По скобкам можно понять, с чем имеется дело т.к. *тексту всегда предшествуют круглые скобки*. Есть случай, когда отсутствует поле 8: это происходит если content содержит число.

8 – в данном случае в этом поле находится текст. Именно такой текст шлёт отладчик на сервер для дальнейшей сверки (после получения).

- Переходим к gethist, где (номера красных рамок):

9 – время отправки пакета с сервера клиенту. Кажется, что оно совпадает с 1, но это не так, просто точности выводимого на экран числа недостаточно, чтобы сделать такой вывод, однако, сохранённый в файл результат имеет максимальную точность.

10 – по аналогии с 5 и 2, но по отношению к 11

11 – это **запрошенный** тип (требование), то есть тот, что клиент (отладчик) запросил у сервера. Он совпадает с 3.

12 – по аналогии с 11, 5 и 2, но по отношению к 13

13 – это **запрошенный** id (требование), который равен «null» (строке), а строка «null» означает, что клиенту не важен id пакета (см. документацию MicroServer: самый конец 4 раздела «Протокол связи с MicroServer»). Если бы в запросе (здесь) был указан id 6, то пакет не был бы выдан, так как хоть id пакета и запрошенный совпадают, id пакета скрыт, а, значит, недоступен для запросов.

14 – специальное слово, которое означает, что тип выданного пакета совпадает с запрошенным типом (пакет был получен по типу).

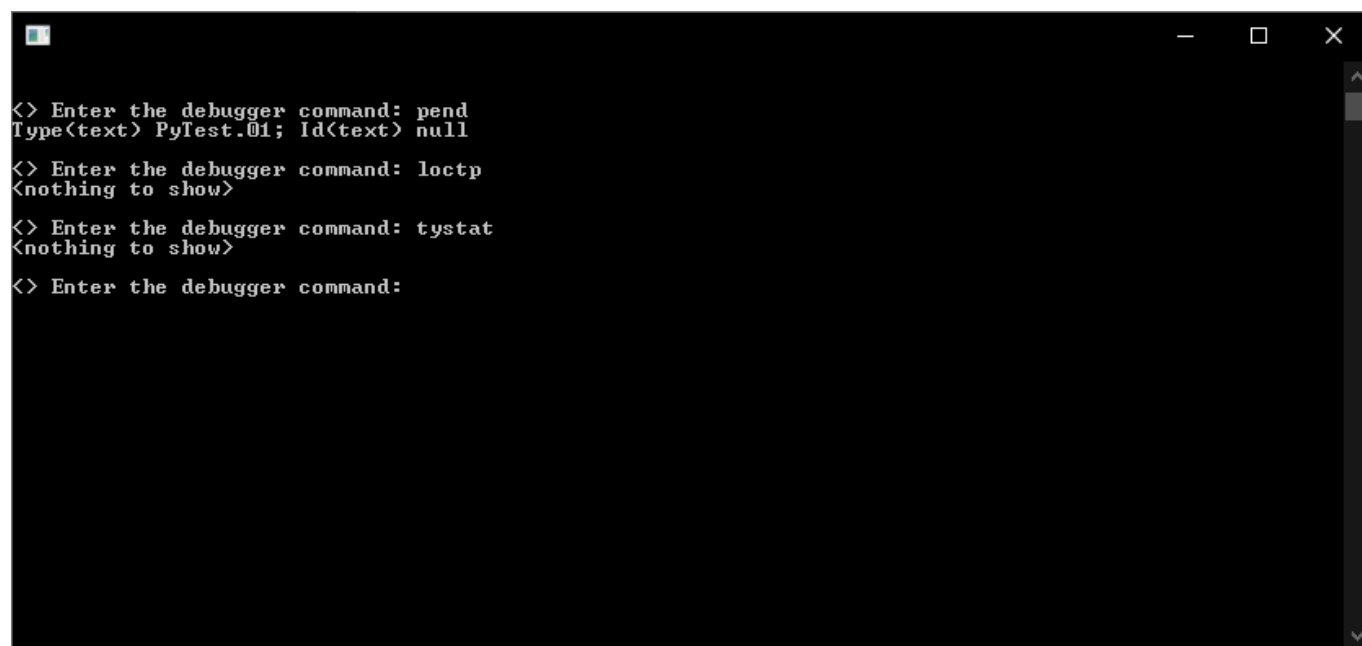
15 – по сути, повтор пункта 4.

16 – повтор пункта 5 (А чего ещё ожидать? Пакет-то тот же самый!)

17, 18, 19 – повторы пунктов 6, 7 и 8 соответственно т.к. речь идёт об одном и том же пакете.

Следует отметить, что posthist и gethist при правильном использовании являются очень мощными средствами отладки коммуникаций сервисов: с их помощью можно точно узнать, когда какой пакет поступил (и с каким содержимым), по какому запросу и во сколько он ушёл с сервера.

- Теперь приведём примеры «состояний» данных сервера, более близких к реальным:



```
<> Enter the debugger command: pend
Type<text> PyTest.01; Id<text> null

<> Enter the debugger command: loctp
<nothing to show>

<> Enter the debugger command: tystat
<nothing to show>

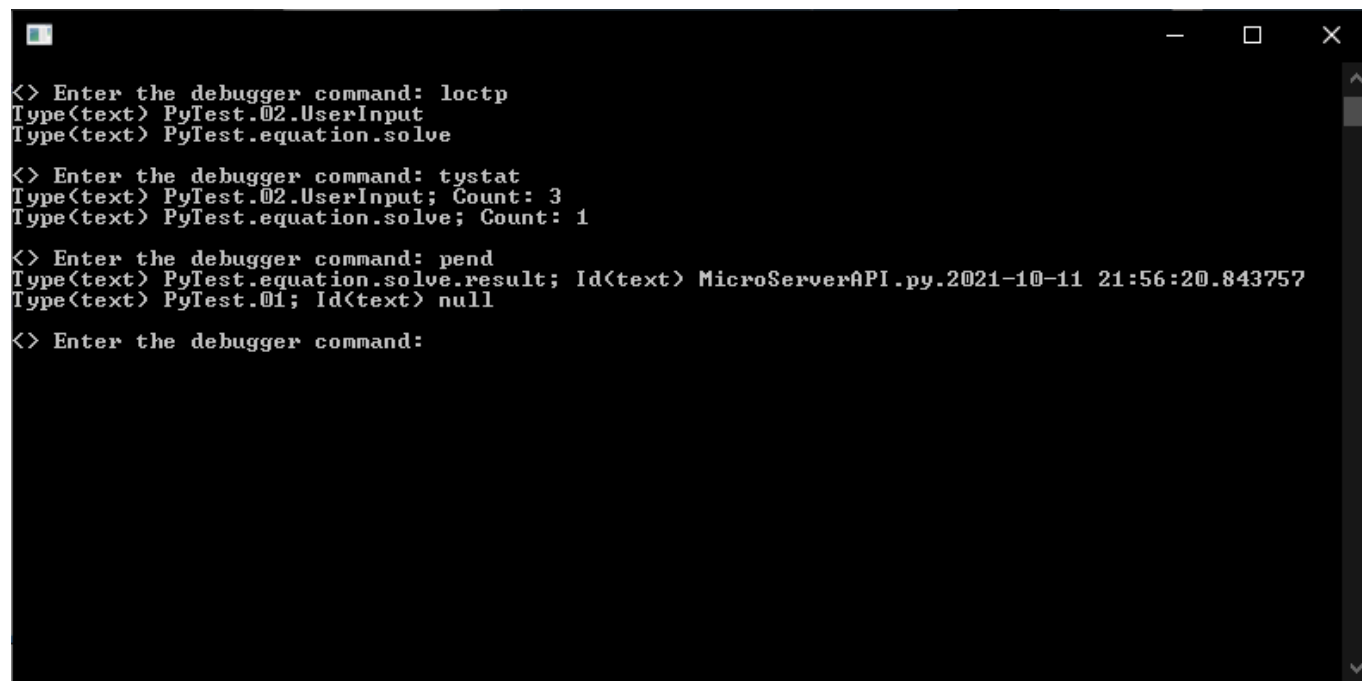
<> Enter the debugger command:
```

Как можно видеть, на сервере не хранится ни одного пакета, но есть один запрос (в реальности запросов может быть в десятки раз больше). Это означает, что микро-сервисная система ожидает поступления данных откуда-то извне (возможно, от пользователя). После того, как какой-то сервис отправит пакет на обработку, начнётся работа, а пока что все сервисы просто ждут.

Вывод команды «pend» содержит информацию о каждом активном в момент выполнения команды запросе: пакет какого типа и с каким id ожидает сервис. В примере видно, что сервису не важен id пакета, но ему важен тип пакета (тип не «null»). Для данного конкретного сервиса сервер ждёт пакет с типом «PyTest.01» и любым id (возможно, даже скрытым).

Обратите внимание, следом за «pend» была выполнена команда «loctp», хотя она менее информативна, чем «tystat». Это связано с тем, что запрос «loctp» выполняется быстрее, чем «tystat», что может быть заметно, когда на сервере хранится много различных типов (больше 500, например, и часть их них – во внешнем хранилище). Если сначала неизвестно, сколько типов хранится, лучше выполнить «loctp».

- А как понять, что что-то идёт «не так»? В данном случае рассмотрим «реактивную» систему, т.е. такую, которая сама по себе ничего не делает, а реагирует на внешние данные.



```
<> Enter the debugger command: loctp
Type<text> PyTest.02.UserInput
Type<text> PyTest.equation.solve

<> Enter the debugger command: tystat
Type<text> PyTest.02.UserInput; Count: 3
Type<text> PyTest.equation.solve; Count: 1

<> Enter the debugger command: pend
Type<text> PyTest.equation.solve.result; Id<text> MicroServerAPI.py.2021-10-11 21:56:20.843757
Type<text> PyTest.01; Id<text> null

<> Enter the debugger command:
```

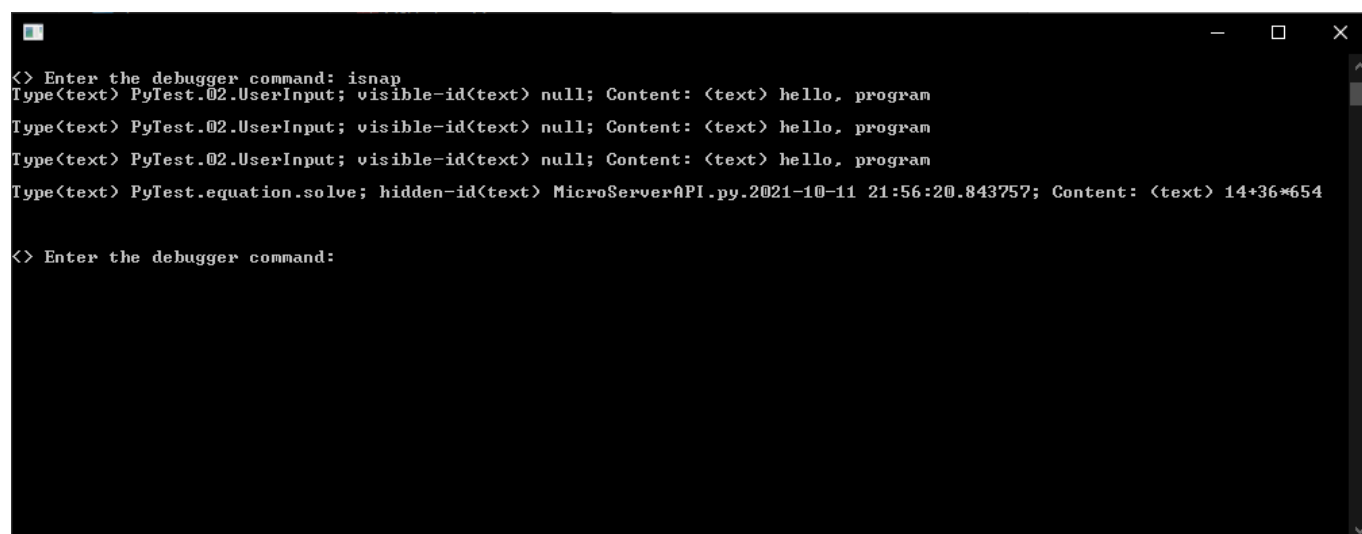
После выполнения «loctp» видим, что типов не 500 и даже не 100. Однако, допустим, у нас всего 4 сервиса. Если это не рендер-сервисы (или ещё что нагруженное), то довольно необычно, что удалось поймать ситуацию, где аж 2 типа в данный момент на сервере. Допустим, повезло.

Выполняем команду «loctp» и видим уже что-то подозрительное: пакетов типа «PyTest.02.UserInput» уже аж 3 штуки, что для «реактивной» системы в большинстве случаев означает, что какой-то из её сервисов «упал» и больше не обрабатывает данные, поэтому-то уже успело накопиться на обработку 3 пакета. Если так пойдёт дальше, то число пакетов может превысить тысячи и с зависимости от характеристик хоста сервера в тот или иной момент привести к «падению» сервера. Здесь можно было бы использовать «костыль» в виде внешнего хранилища, которое забирало бы излишки пакетов и сохраняло где-то (в особые случаях, вроде этого, внешнее хранилище могло бы начать удалять переизбыток пакетов). Также видим тип «PyTest.equation.solve», пакетов которого 1 штука. Что ж, хотя бы не три...

Теперь неплохо бы проверить запросы: выполняем «pend» и видим, что активны 2 запроса, один из которых – «PyTest.01» – ждёт данные на обработку (сервис исправен), а другой, видимо, от сервиса, сделавшего функциональный запрос: он послал данные типа «PyTest.equation.solve» на обработку, и теперь ожидает результата обработки данных с типом «PyTest.equation.solve.result» и id «MicroServerAPI.py.2021-10-11 21:56:20.843757». Если выполнить «isnap» в данной конфигурации, можно точно знать, так ли это. Можно выполнить команду «pend» ещё раз (через секунд 2-5-10 после первого выполнения, например), и если в выводе снова будет присутствовать «PyTest.equation.solve.result» с

тем же самым id, значит сервис по обработке данных типа «PyTest.equation.solve» тоже «упал»...

Чтож, выполним команду «isnap»:



```
<> Enter the debugger command: isnap
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.equation.solve; hidden-id<text> MicroServerAPI.py.2021-10-11 21:56:20.843757; Content: <text> 14+36*654

<> Enter the debugger command:
```

Догадки подтверждаются: во-первых, пакет с типом «PyTest.equation.solve» действительно является частью функционального запроса т.к. у него есть id, к тому же скрытый, во-вторых, видим, что этот пакет всё ещё не ушёл в обработку (см. id – он всё тот же), а, значит, сервис, обрабатывающий эти пакеты явно не в порядке.

И да, возможно что-то не в порядке в работающем сервисе, ведь он отдаёт выражение (14+36*654) с именем типа, содержащим слово «equation» (переводится «уравнение»), так что, возможно, проблема кроется даже не в уже упавших сервисах (во всяком случае, не только в них), а в работающем на данный момент, и ждущем результат обработки своих данных. Этот сервис либо поставил пакету неправильный тип (в типе уравнение, а не выражение), либо вставил неправильные данные в пакет. Аналогичный конвейерный неправильный (вероятно) пакет в прошлом и мог стать причиной падения другого сервиса.

В лучших традициях детектива виновным оказался тот, кого подозревали меньше...

Ещё не ясно, зачем нужны «PyTest.02.UserInput». Судя по названию, это какие-то входные данные. Сервис, их генерирующий, вероятно, работает (иначе почему он не упал после 1 пакета?). Здесь возможно много комбинаций (впрочем, как и в предыдущем выводе), так что нужны дополнительные данные.

И, обратите внимание, все выводы сделаны без какой-либо информации о функционале системы: только исходя из количества сервисов (четыре), информации из отладчика и знания того, что система «реактивная». Конечно, эти «выводы» – это, скорее, «догадки» (так как нет данных о функционале системы, о том, какие сервисы за что отвечали и др. – возможно, системе просто нужно больше работающих экземпляров сервисов), но и такой результат вполне неплох (с минимальными-то знаниями о системе).

Сейчас впору использовать тяжёлую артиллерию (команду «gethist»), чтобы проследить историю получения пакетов с сервера, но... я не буду этого делать, так как я не организовывал красивое падение системы, а написал заглушку для демонстрации работы с отладчиком. Зато пример красивый :)

В реальной ситуации данных будет значительно больше (так как информации о сервисах, как минимум, будет больше), так что и размышления будут более предметными (меньше догадок и «дедуктивного метода»).