HITACHI
Inspire the Next

Graceful Shutdown
(and other development support functions)

27 January 2020

Kunihiko Toumura

Research and Development Group
Hitachi, Ltd.

HITACHI
Inspire the Next

Contents

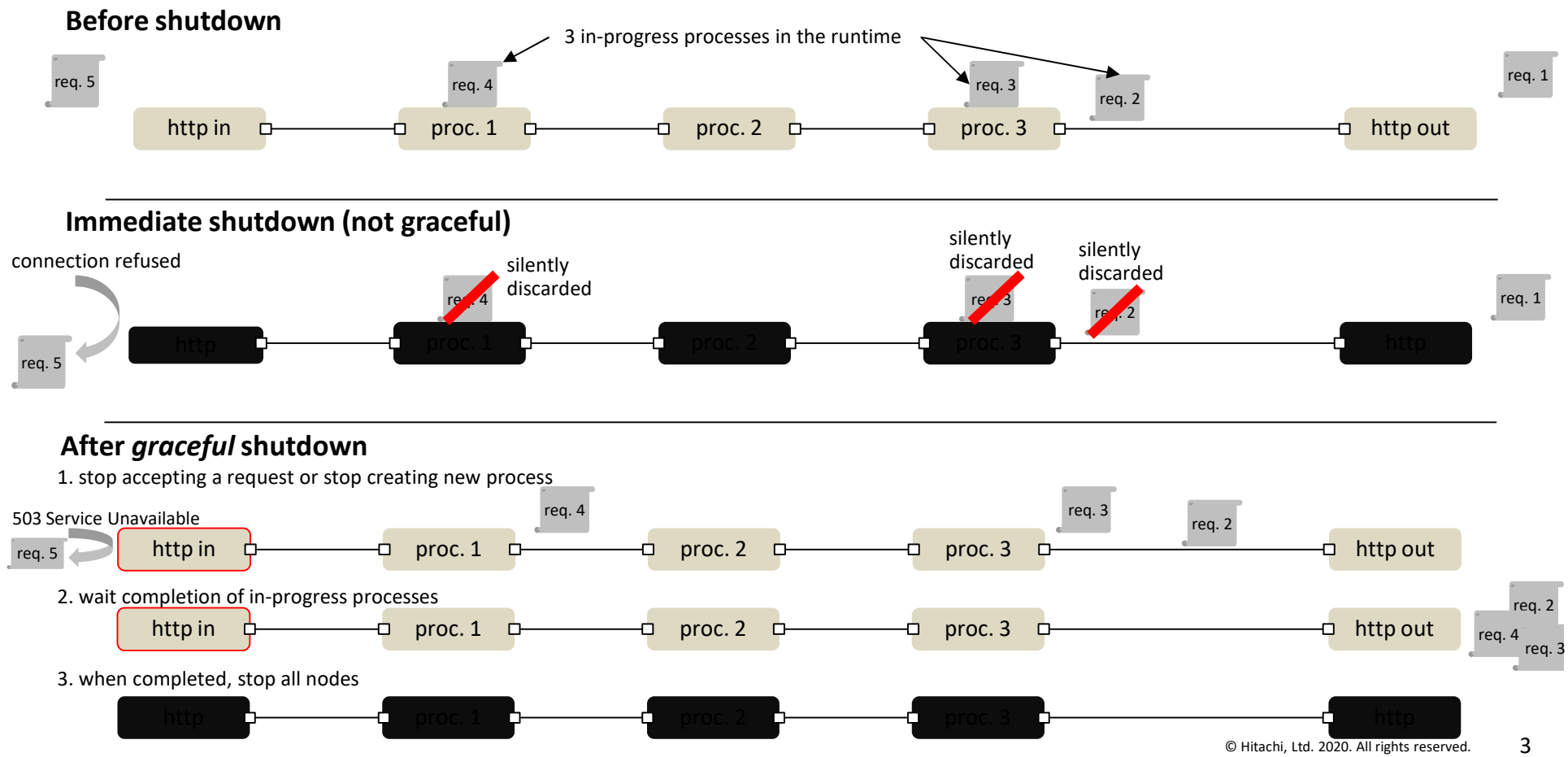
1-1. Introduction

HITACHI
Inspire the Next

- When use Node-RED as a Web API server, occasionally we need to shutdown it *gracefully*:
  - reject further incoming requests.
  - reply response to on-going sessions.
  - shutdown connection with depended services (flush write-cache to database, etc.)
  - shutdown related services (temporally created services to execute some part of flows)
  - …
- To make shutdown gracefully, it is necessary to design new APIs.
  - Additional shutdown phase to stop creating new jobs.
  - Mechanism for check existence of in-progress process.

1-2. Use case: Web API endpoint on Node-RED

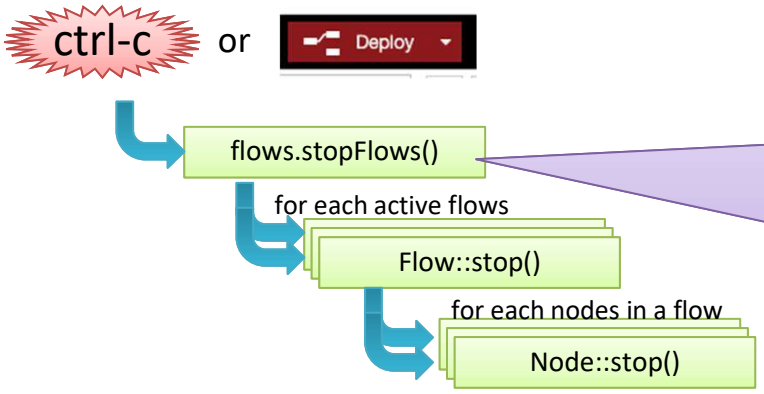
HITACHI
Inspire the Next



2-1. Design of Graceful Shutdown

HITACHI
Inspire the Next

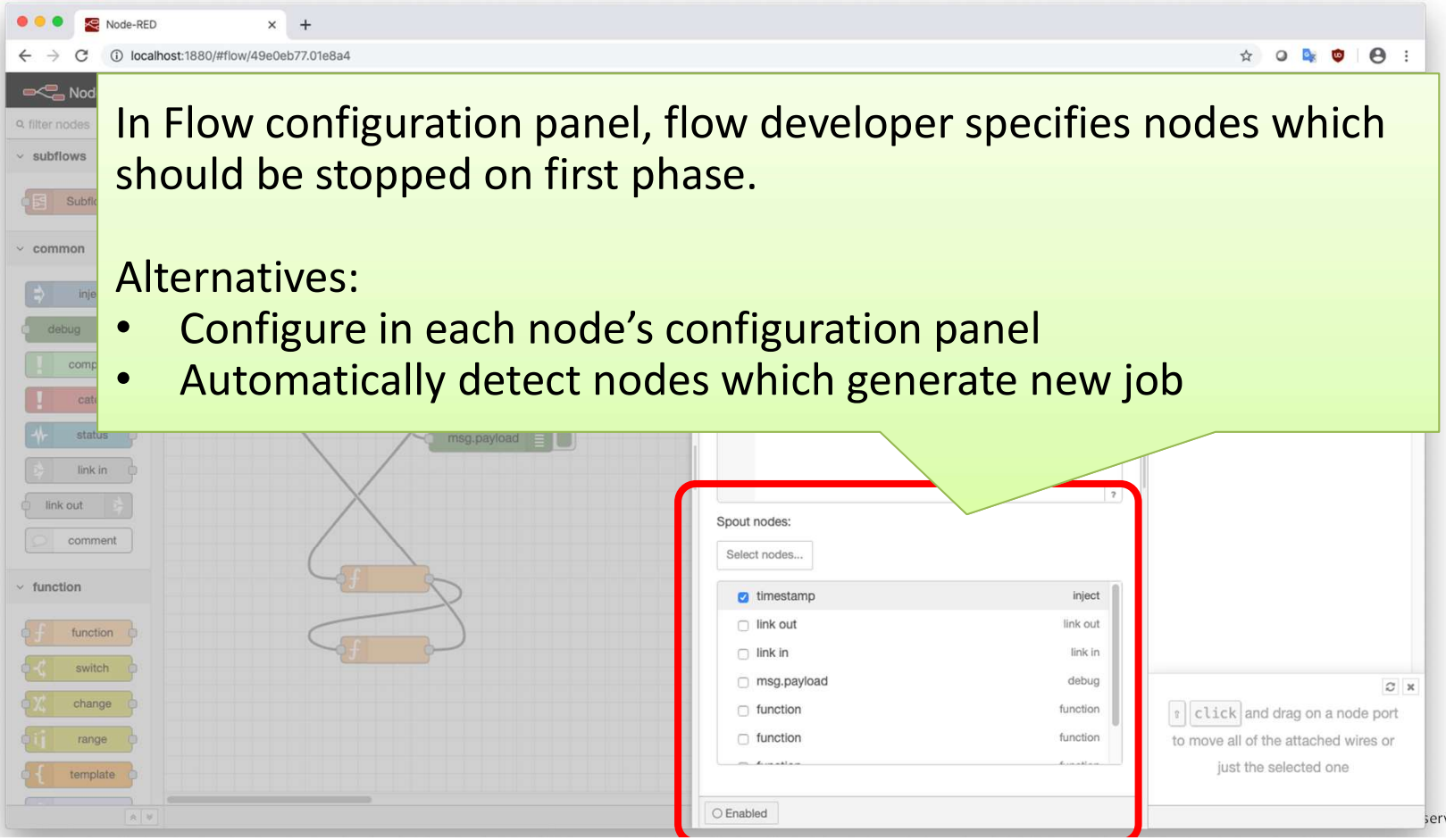
- Flow developer specifies which nodes should stop first, and how long the runtime waits before final shutdown process, in a flow configuration panel.
- When the runtime receives SIGINT (or, restarted by flow deployment), the runtime call ‘close’ handler of designated nodes, wait for graceful period or completion of all in-progress process, and then call ‘close’ handler of other nodes.



Split this function into three phases:
1. stop only nodes which specified at flow configuration panel
2. wait for *gracefulPeriod* (default: 10 seconds), or completion of in-progress process.
3. stop other nodes

2-2. User Interface: Editor

HITACHI
Inspire the Next



In Flow configuration panel, flow developer specifies nodes which should be stopped on first phase.

Alternatives:

- Configure in each node's configuration panel
- Automatically detect nodes which generate new job

2-3. User Interface: Configurations in settings.js

HITACHI
Inspire the Next

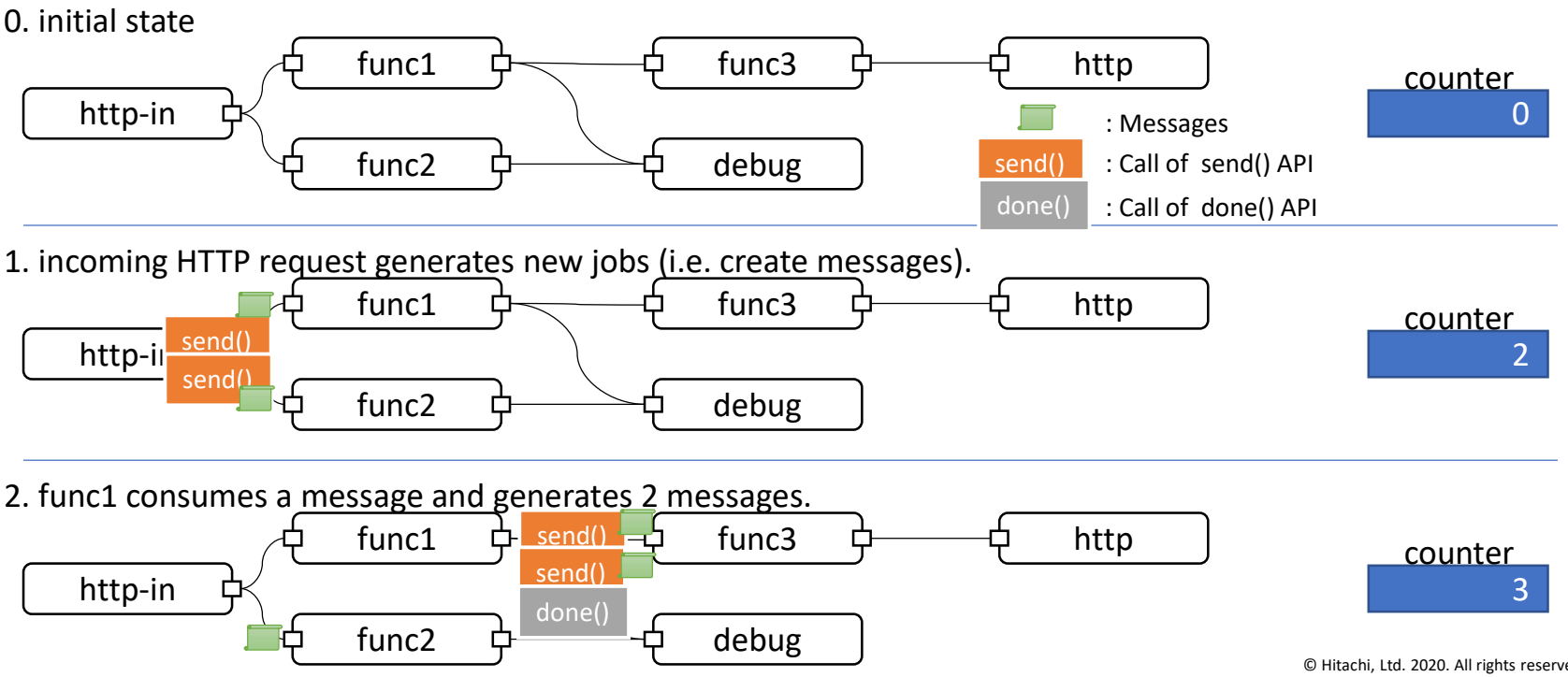
In settings.js, a flow user can enable/disable Graceful Shutdown function and set the timeout period for Graceful Shutdown process:

```
…
        gracefulShutdown:       true,
        gracefulPeriod:         10000,
…
```

2-4. In-progress process detection (1/2)

HITACHI
Inspire the Next

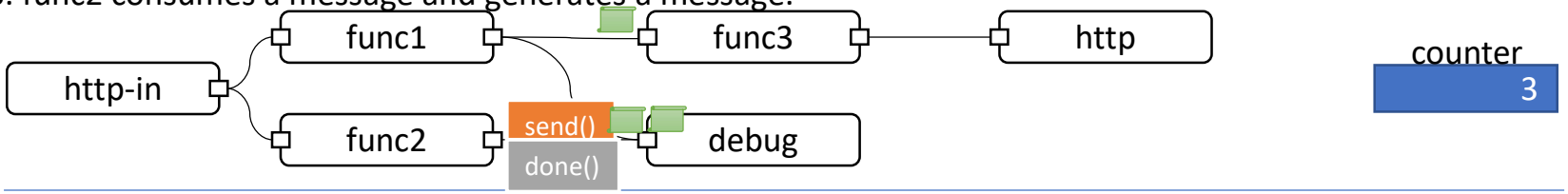
To detect an existence of in-progress process, we use send() and done() to count messages. When counter is zero, no in-progress process is existed.



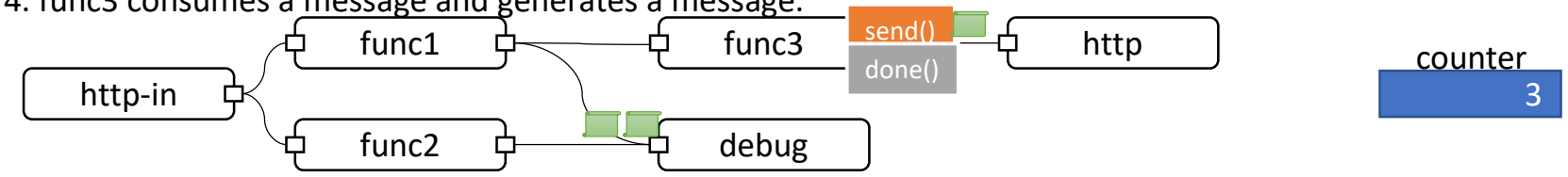
2-5. In-progress process detection (2/2)

HITACHI
Inspire the Next

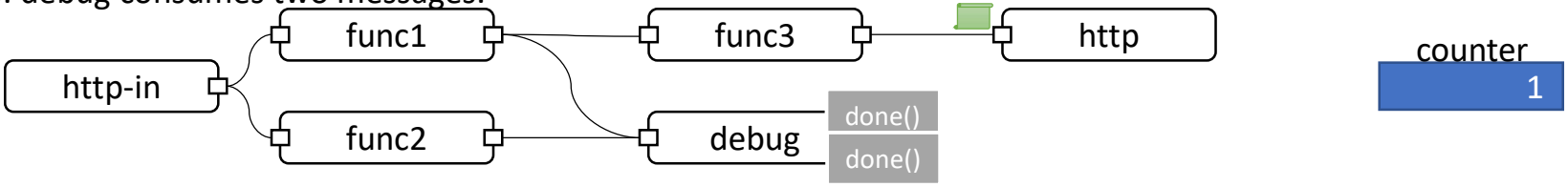
3. func2 consumes a message and generates a message.



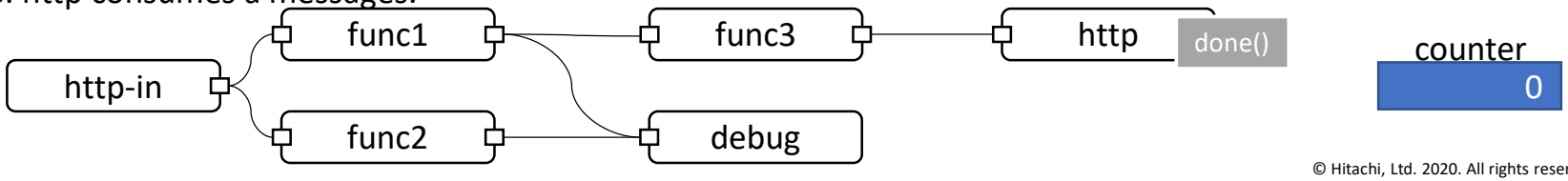
4. func3 consumes a message and generates a message.



5. debug consumes two messages.



6. http consumes a messages.



2-6. When the flow contains the nodes which don't use send()/done()?

HITACHI
Inspire the Next

Because the Graceful Shutdown logic fully depends on the proper usage of send()/done(), the flow can not be shutdown gracefully.

1. Wait for timeout despite of no in-progress process
   - This is annoying, but not harmful.
2. Premature shutdown despite of in-process processes
   - This is something harmful, but not worse than ordinal shutdown

In current implementation, the runtime warns that graceful shutdown may fail when graceful shutdown is enabled, and the flow contains the nodes which don't use send()/done.

```
21 Jan 09:34:24 - [info] [flow:49e0eb77.01e8a4] Graceful shutdown may fail because some node does not support Node Messaging API.
```

2-7. Demo

HITACHI
Inspire the Next

3. Work plan for this week

HITACHI
Inspire the Next

- Test and brush-up of shutdown logic
- Review of Node Messaging API support in core nodes.
- UI design for Graceful Shutdown

4-1. Other development support functions: Message tracing

HITACHI
Inspire the Next

Use case: In debugging, we want to trace the messages…

Currently, we use Debug nodes to assure a process in some node is finished.
But…

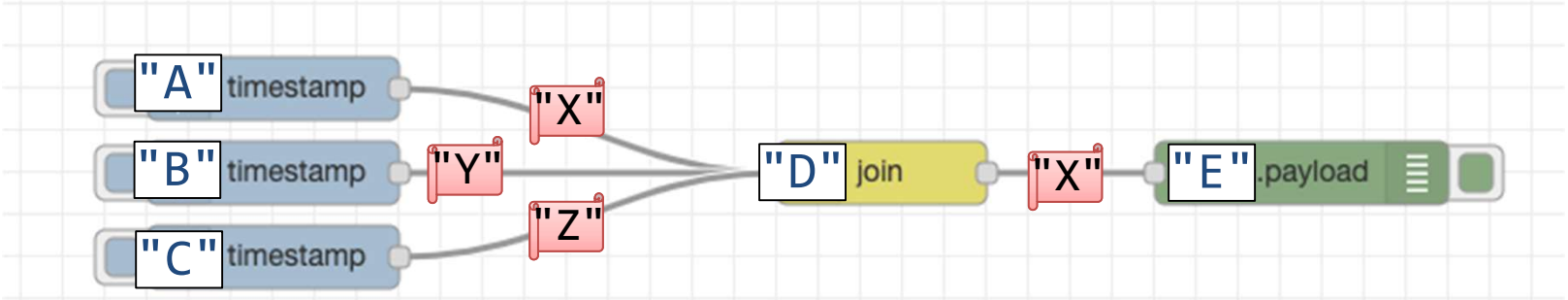
- we can't know the message is received and start processing in some node.
- Inserting Debug nodes is cumbersome process, and readability of process become worse.

Also, by enabling metric log, we can see message IDs that is sent/received, but it is difficult to analyze a causal relationship between messages.

4-2. Example of message tracing and causal relationship

HITACHI
Inspire the Next

Node “D” receives message “X”, run join process, and then emit message to node “E”, but, had messages “Y” and “Z” been processed, or still queued in node “D”?




```
14:21:51 - [metric] {"level":99,"nodeid":"B","event":"node.inject.receive","msgid":"Y","timestamp":1579670511861}
14:21:51 - [metric] {"level":99,"nodeid":"B","event":"node.inject.send","msgid":"Y","timestamp":1579670511862}
14:21:51 - [metric] {"level":99,"nodeid":"D","event":"node.join.receive","msgid":"Y","timestamp":1579670511863}
14:21:53 - [metric] {"level":99,"nodeid":"C","event":"node.inject.receive","msgid":"Z","timestamp":1579670513507}
14:21:53 - [metric] {"level":99,"nodeid":"C","event":"node.inject.send","msgid":"Z","timestamp":1579670513508}
14:21:53 - [metric] {"level":99,"nodeid":"D","event":"node.join.receive","msgid":"Z","timestamp":1579670513508}
14:21:56 - [metric] {"level":99,"nodeid":"A","event":"node.inject.receive","msgid":"X","timestamp":1579670516424}
14:21:56 - [metric] {"level":99,"nodeid":"A","event":"node.inject.send","msgid":"X","timestamp":1579670516424}
14:21:56 - [metric] {"level":99,"nodeid":"D","event":"node.join.receive","msgid":"X","timestamp":1579670516424}
14:21:56 - [metric] {"level":99,"nodeid":"D","event":"node.join.send","msgid":"X","timestamp":1579670516425}
14:21:56 - [metric] {"level":99,"nodeid":"E","event":"node.debug.receive","msgid":"X","timestamp":1579670516425}
```

4-3. Message tracing in other languages

HITACHI
Inspire the Next

In other languages:

- Breakpoint insertion and stepwise execution (for interactive debugging)
- Emit trace log for message receiving and sending (for batch analysis)

4-4. Possible enhancement to Node-RED

HITACHI
Inspire the Next

Breakpoint:

- When a breakpoint node receive a message, the entire flow stops execution, then it enters “stepwise execution” mode.
- Inspecting messages/contexts in the runtime
- Skip to completion of node processing, then inspect the emitting message.

Log enhancement:

- Emit log message when message processing in each node is completed (i.e., when done() called).
- Introduce a debug flag for each node
  - In debug mode, the node emit log message for each reception and sending message.