



Graceful Shutdown and Timeout Handling

2 July 2019

Kunihiko Tsumura

Research & Development Group
Hitachi, Ltd.

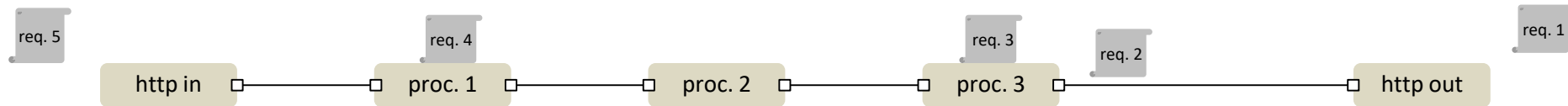
Kazumi Yoshida

Hitachi Solutions, Ltd.

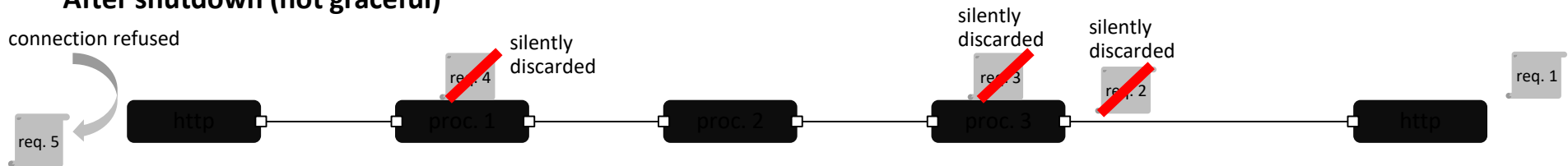
- When use Node-RED as a Web API server, occasionally we need to shutdown it *gracefully*:
 - reject further incoming requests.
 - reply response to on-going sessions.
 - shutdown connection with depended services (flush write-cache to database, etc.)
 - shutdown related services (temporally created services to execute some part of flows)
 - ...
- To make shutdown gracefully, it is necessary to design new APIs.
 - Additional phases to 'close' each nodes.
 - Timeout handling in message processing in node.

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

Before shutdown

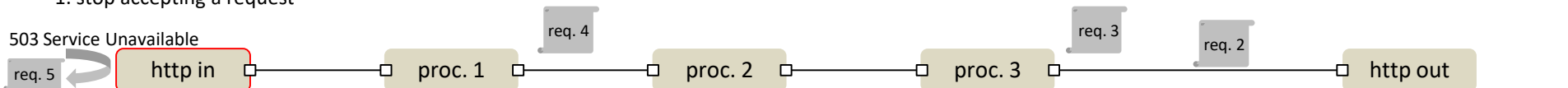


After shutdown (not graceful)

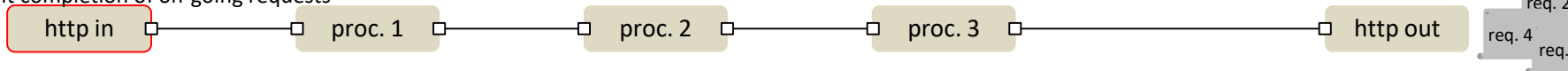


After *graceful* shutdown

1. stop accepting a request



2. wait completion of on-going requests

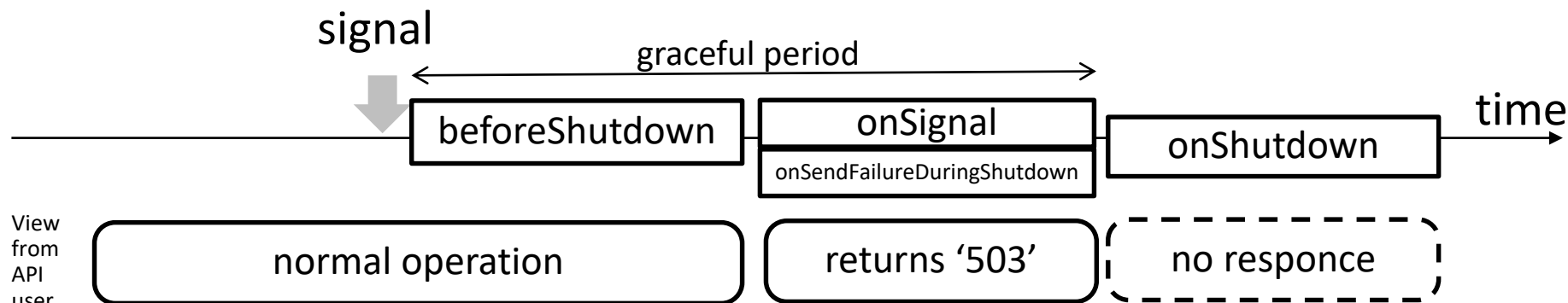


2. when completed, stop all processes



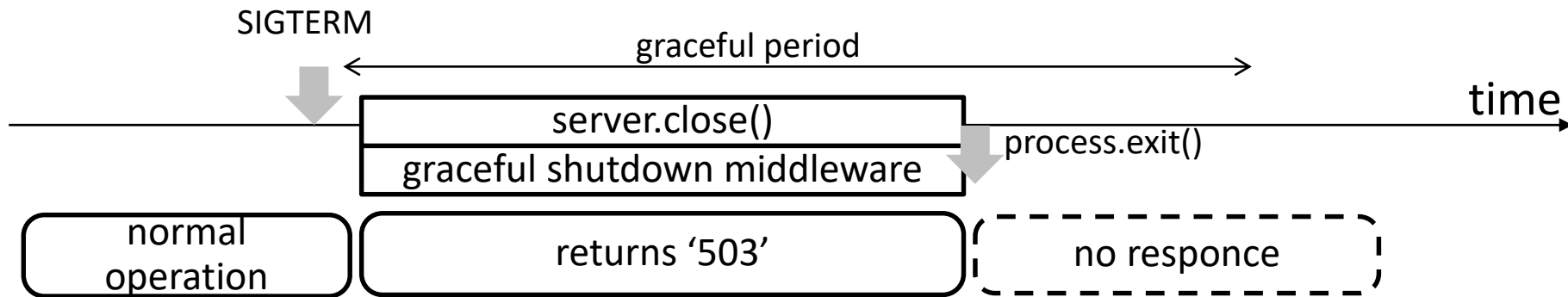
3-1. Approaches in other systems: @godaddy/terminus

- Graceful shutdown for any Node.js HTTP applications:
 - `beforeShutdown()`: called before the HTTP server starts its shutdown.
 - `onSignal()`: cleanup function.
 - `onShutdown()`: called right before exiting.
 - `onSendFailureDuringShutdown()`: called before sending each 503 during shutdowns



3-2. Approaches in other systems: express-graceful-shutdown

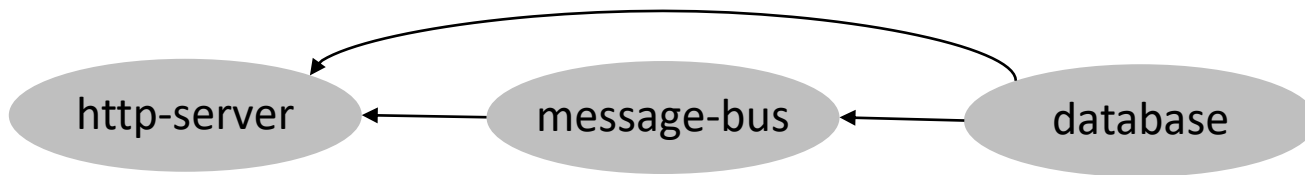
- Use as a middleware of Express.js.
- It ensures that Express returns correctly with a 503 during shutdown



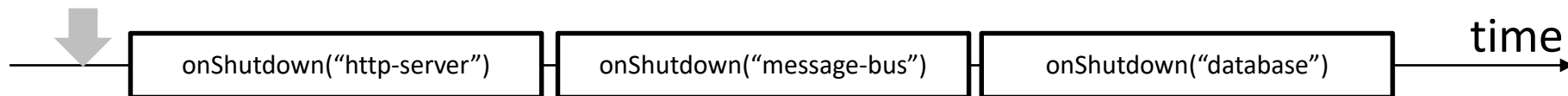
3-3. Approaches in other systems: node-graceful-shutdown

- More generalized approach. Create dependency graph of function and wait for finishing depending function during shutdown

```
onShutdown("database", ["http-server", "message-bus"], async function () {  
    // Shut down code for "database";  
    // ONLY AFTER "http-server" and "message-bus" are completed.  
});
```

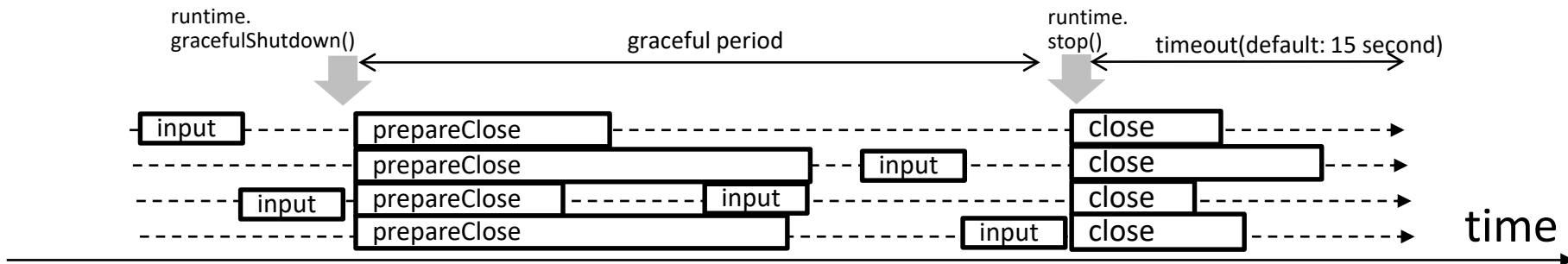


Note: No timeout support



4. Approach in Node-RED

- Insert a graceful shutdown phase in front of current runtime stop process (runtime.stop()):
 - Stop accepting new request
 - HTTP request at *http in* node, generate message in *inject* node, etc.
 - To signal this, we should create a new event type: e.g. *'prepareClose'*
 - `node.on('prepareClose', function (...) { ... });` /* details are to be determined */
 - Wait until all prepareClose functions and all message processing finished
 - Considering (infinite) loops in flow, we should handle timeout of graceful period.
 - Then, initiate ordinal stopping process (runtime.stop()).



- To ensure completing all message processing on nodes, we should handle timeout for message processing function. Details are described from next slide.

<Background>

We received the following request:

Node-RED should timeout and respond in case of no responding in a communication, etc. instead of waiting indefinitely. Examples below:

- case 1: Executing SQL statements and queries take a long time.
- case 2: No response returned when sending a request to a web service with "http-request" node.
- case 3: The flows between "http-in" and "http-out" nodes take a long time.

Apps which sent a request to Node-RED would be kept waiting. Desirably, "http-out" node replies an error after the elapse of a certain period of time.

<Current situation>

In the case 2, "http-request" node has timeout feature. We can define the timeout value in the setting.json.

We are considering the processing ways to support timeout based on Nishiyama-san's idea which is described in the following URL.

Node Messaging API: <https://github.com/node-red/node-red/wiki/Design%3A-Node-Messaging-API>

5-2. Timeout Feature: Points to Note

```
module.exports = function(RED) {  
  function aNode(config) {  
    RED.nodes.createNode(this, config);  
    var node = this;  
    node.on('input', function(message) {  
      message.payload = message.payload.aNode();  
      var a = syncFunc();  
      node.send(message);  
    });  
  }  
  RED.nodes.registerType("a node", aNode);  
}
```

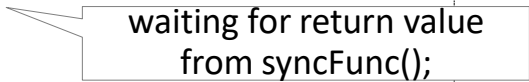


figure 1

- 1.syncFunc() might return a value after timeout.
- 2.Maintaining consistencies among node's behaviours.
- 3.Existing nodes must have the timeout function without any enhancements.

We are considering using setTimeout() function, and getting the timeout value in the setting.js.
After timeout, automatically node.error() will be called behind a node.

5-3. Timeout Feature: Approach

In the figure 2, a concrete node inherits Node class, a listener function for timeout could be defined in Node or a concrete node. To avoid changing existing node, it should be defined in Node class.

```
Node.prototype.receive = function (message) {  
  var self = this;  
  setTimeout(function() {  
    process.nextTick(function() {  
      self.emit("timeout", message);  
    });  
  }, 3000);  
  self.on("timeout", function(message) {  
    self.error(message);  
  });  
  this.emit("input", message);  
};
```

Node: code1

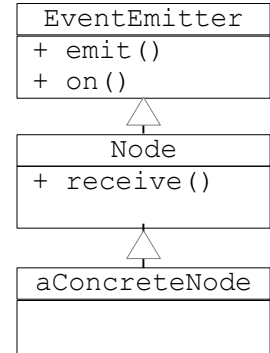


figure2

We must design the default timeout behaviour but, there is an issue. In the case the previous node calls `send()` function multiple times, automatically "on" is called same times. Once the timeout event occurs, all listener functions will be called. We need something to control calling listener functions. An idea is to use unique event names.

```
Node.prototype.receive = function (message) {  
  var self = this;  
  setTimeout(function() {  
    process.nextTick(function() {  
      self.emit('timeout'+self.count, message);  
    });  
  }, 3000);  
  self.on('timeout'+self.count, function(message) {  
    self.error(message);  
  });  
  self.count ++;  
};
```

Node: code2

example:
unique event name by counter
(counter < 2**53)

event number also have to be
defined by `setMaxListeners(val)`;

5-4. Timeout Feature: Problem - When should the timer be cancelled?

Once called "setTimeout()" function, the listener function registered always is called after timeout value passes. When should the timer be cancelled?

In the listener function, we can define a lot of callback functions like below. We are considering what the basis of an end of "input" event caused by send() call is.

aConcreteNode:

```
node.on('input', function(msg) {
  msg.payload = msg.payload.aNode();
  fs.readFile("hoge.txt", function (err, hoge) {
    fs.readFile("fuga.txt", function (err, fuga) {
      fs.readFile("foo.txt", function (err, foo) {
        fs.readFile("bar.txt", function (err, bar) {
          console.log(hoge + fuga + foo + bar);
        });
      });
    });
  });
});
```

Also we think that the basis is very important in "Graceful Shutdown" too. To solve this issue, we can introduce "done()" api, but this approach makes developers, who provide nodes, modify their codes.

5-5. Timeout Feature: Problem - How to know the end of processing?

If there is no callback functions in the listener function for "input" event in a concrete node, it's not difficult to cancel timeout by getting a listener function for "input" event and calling the function directly.

Node:

```
Node.prototype.receive = function (message) {  
  var self = this;  
  var id = setTimeout(function() {  
    process.nextTick(function() {  
      self.emit('timeout'+self.count, message);  
    });  
  }, 3000);  
  self.on('timeout'+self.count, function(message) {  
    console.log('timeout0: ' + message);  
  });  
  self.on('input'+self.count, function(message, id) {  
    var funcs = self.listeners('input');  
    console.log("start");  
    funcs[0](message);  
    console.log("end");  
    clearTimeout(id);  
  });  
  this.emit('input'+self.count, message, id);  
  self.count ++;
```

here:
cancel timeout

We are considering the best way how to know the end of a node's processing.

- How should a Function node indicate it can be timed-out?
- Default timeout value:
 - Should it be possible for individual nodes to be given a custom timeout value?
 - How does the user provide that value?
 - A node-level API to give its value to override the default.
- A careful design would be to have one setTimeout active per node that checks for the next thing to timeout.

7. Implementation Plan of Graceful Shutdown

- Initial discussion: 7/2-5
- Design note and prototype implementation: September
- Merge to core: December

END



Graceful Shutdown and Timeout Handling

2 July 2019

Kunihiko Toumura

Research & Development Group
Hitachi, Ltd.

Kazumi Yoshida

Hitachi Solutions, Ltd.

HITACHI
Inspire the Next 