knowledge++;
By Eric J. Bruno
if (success == true) {
(index.html)

(http://twitter.com/progwithreason) (http://facebook.com/programmingwithreason)

CoAP Messaging in Depth

Eric J. Bruno

April 2016

CoAP is a protocol not unlike HTTP or REST communication where the messages generally fall into the category of GET, POST, PUT, and DELETE. CoAP is also more conversational by nature since it's request/response driven, as opposed to the publish/subscribe nature of MQTT (for details, see the article on MQTT in this issue).

At a lower level, CoAP messages are sent and received over UDP, which by nature is unreliable, so a basic reliability scheme is built into CoAP's communication protocol on top of UDP. For added security, messages can be sent using Datagram Transport Layer Security (DTLS) protocol instead of UDP. In either case, each CoAP message needs to fit into a single UDP/DTLS datagram packet. Further, CoAP supports datagram messaging over IPv4 and IPv6 networks and variants such as 6LoWPAN – see Figure 1.

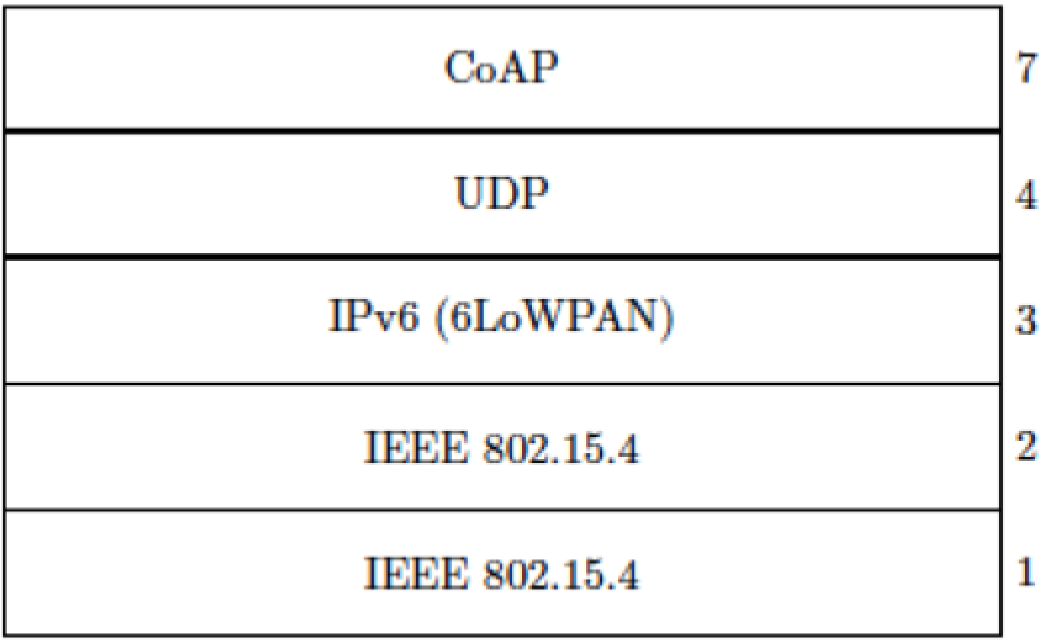


Figure 1 - The COAP messaging network communication layers.

Although unicast UDP is used for the request/reply CoAP protocol, multicast UDP messaging is used to support CoAP device/sensor discovery. CoAP clients and servers support a special "All CoAP Nodes" multicast address, with port 5683, to discover other CoAP servers and their shared resources.

Basic CoAP Message Model

All message exchanges in CoAP are similar to HTTP, but with some key differences. For instance, with CoAP all interchanges are asynchronous and use a datagram transport such as UDP or DTLS. Optional reliability is built into the message exchange using a timeout and retransmission protocol based on random, increasing back-off timers with eventual timeout. Figure 2 below shows CoAP's logical two-layer approach to messaging.

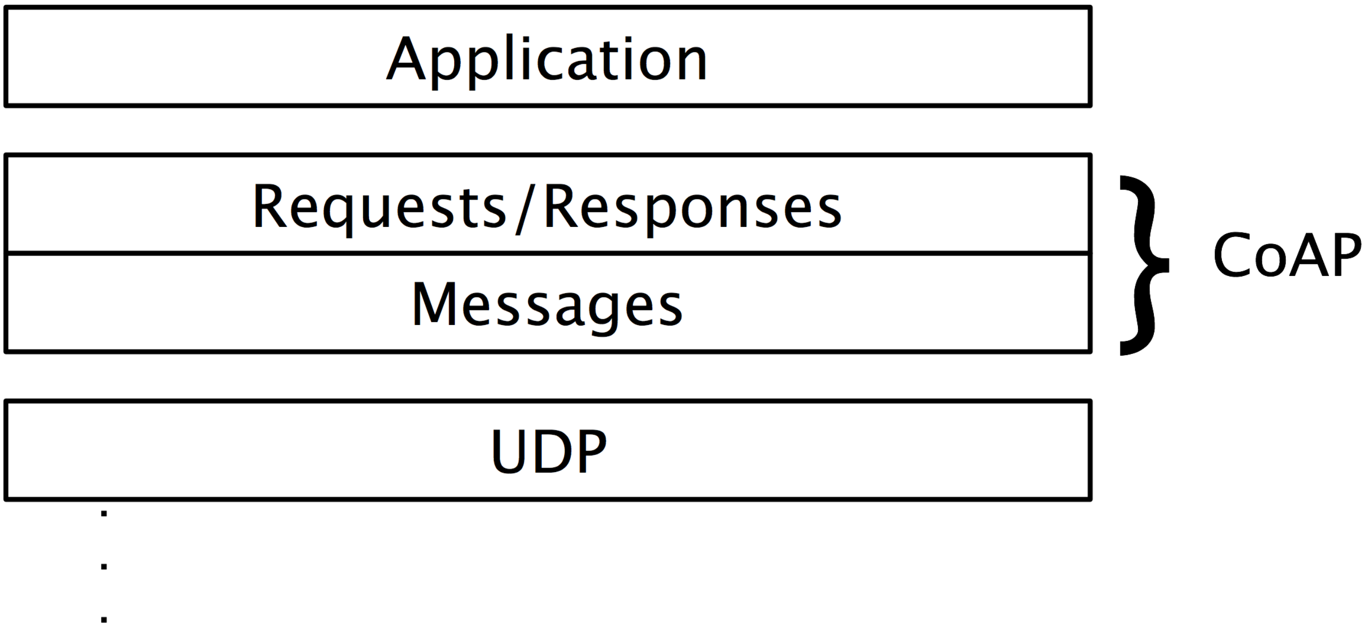


Figure 2 - The CoAP two-layer messaging approach.

All CoAP messages include a four-byte fixed length header followed by, depending upon the message type, an optional header portion and payload. Further, each message includes a 16-bit message ID used to link requests to their accompanying acknowledgements (ACK) or error statuses where applicable. A non-confirmable (NON) message—one that doesn't require confirmation or, hence, reliability—such as those coming in a constant flow of sensor measurements, for example, are simply sent from client to server with no ACK (see Figure 3). If the recipient is unable to process the message for any reason, it may reply with a reset message (RST) to indicate this.

In short, message IDs are unique values that are used to identify duplicate messages, and match confirmable messages (CON) to acknowledgements (ACK). Tokens are unique values used to match requests to responses.

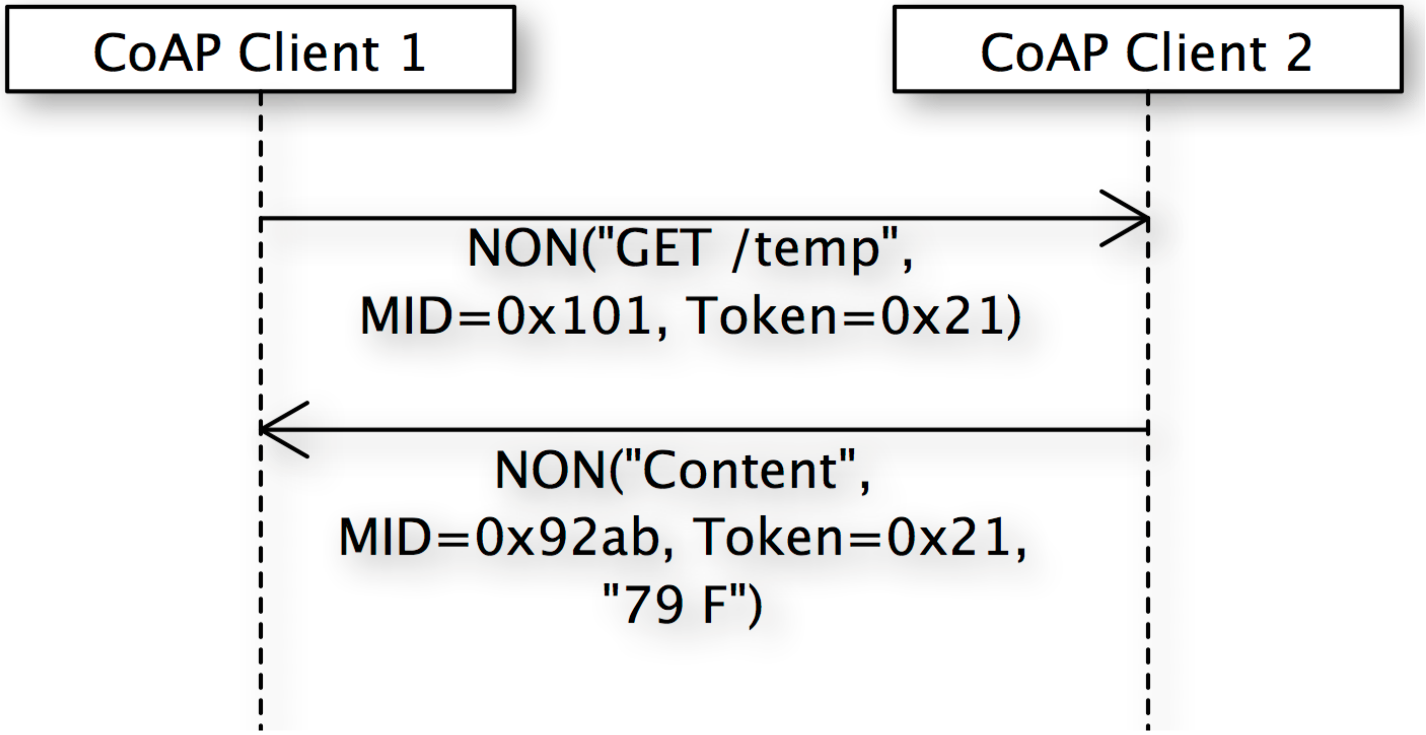


Figure 3 - A non-confirmable (NON) request message and response.

In this example client 1 sends a non-confirmable GET request for a temperature reading to client 2. A unique message ID (0x101) and token (0x21) are provided. The message ID is useful to detect message duplication (more on that later). For a request/response message exchange, the token must match throughout the entire message exchange. Therefore, in this example, the message IDs will be unique for both the request and response messages, but the token (0x21) will be the same for both.

In the case of a NON message, either the request or the response may be lost. For some types of message exchanges, this may be acceptable. For those that are not, CoAP supports confirmable messages (CON), as shown in Figure 4.

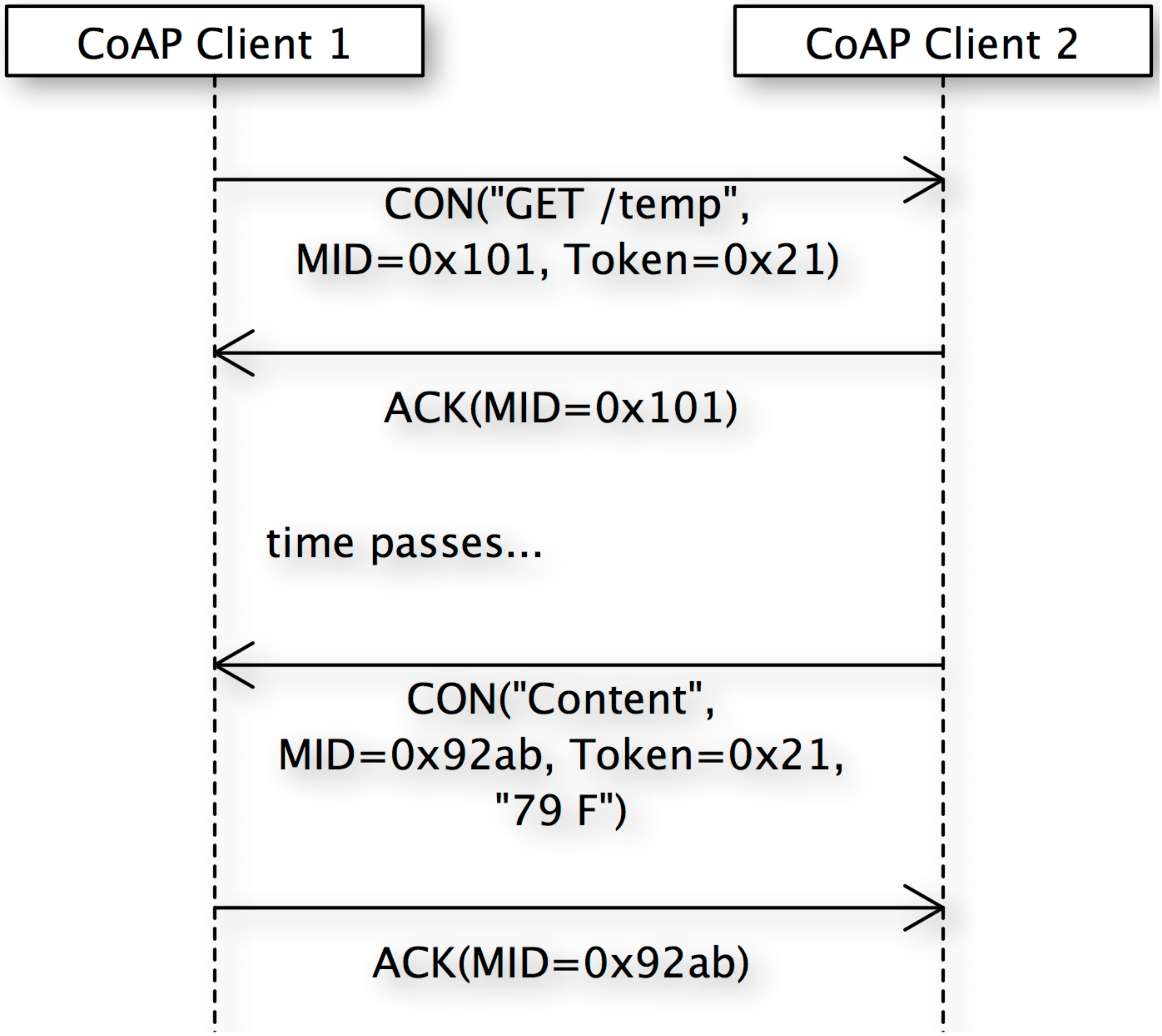


Figure 4 - A confirmable (CON) request message and response.

In this example client 1 sends a confirmable GET request for a temperature reading to client 2. A unique message ID (0x101) and token (0x21) are provided. For a request/response message exchange, the token must match throughout the message exchange. When client 2 receives the CON request, it sends an acknowledgement (ACK) message containing the same message ID as the request. Some time later, client 2 sends a CON response message containing the requested data and new message ID, but the same token as in the CON request from client 1. To confirm that client 1 received the data, it sends an ACK back to client 2 containing the same message ID as in the response (0x92ab), and the exchange is complete.

Piggybacking

As a shortcut, and to reduce message traffic and processing overhead, CoAP supports the concept of piggybacking. With this, the response data to a request can be included with (piggybacked on) the ACK message (see Figure 5).

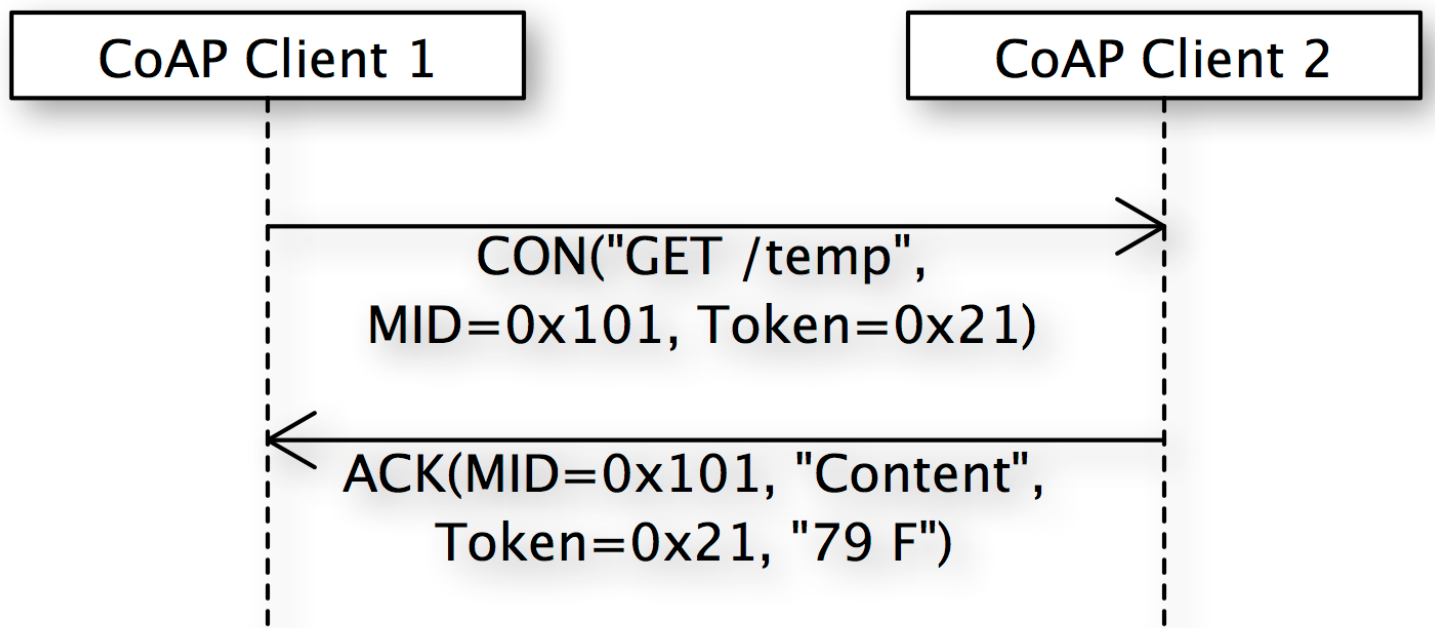


Figure 5 - CON response content can be piggybacked onto ACK messages.

The CON request is identical to the previous exchange. However, when client 2 sends the ACK, it also includes the response data. When client 1 receives this ACK with data, the CON message exchange is complete.

Dealing with Lost Messages

Since UDP is unreliable, the CoAP protocol specifies a way to detect lost messages and then retransmit them. For instance, in any of the CON examples above, if an ACK for the initial CON request was not received, after a timeout period the CON request will be resent with the identical message ID and token (see Figure 6).

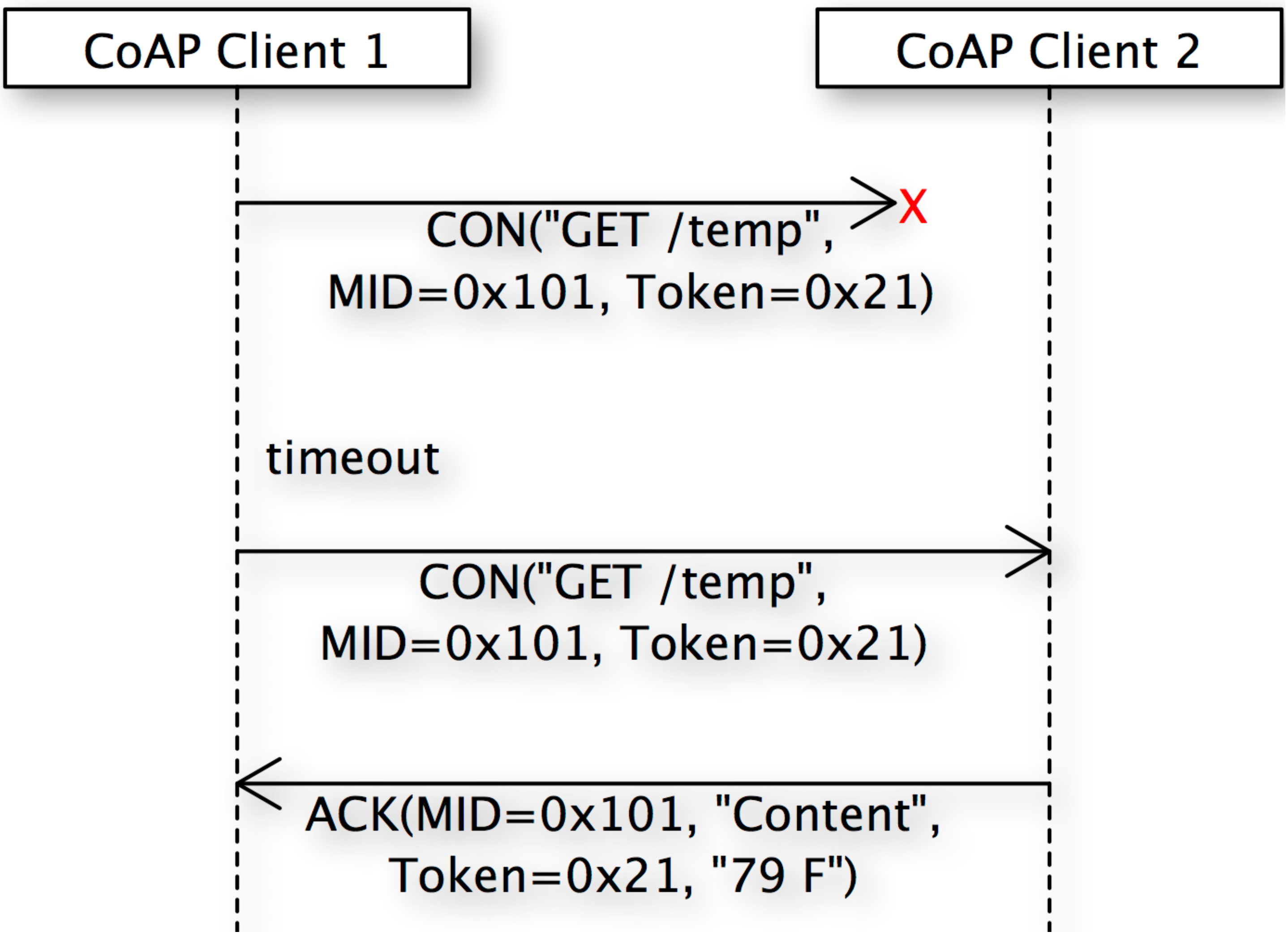

CoAP Client 1
CoAP Client 2
X
CON("GET /temp", MID=0x101, Token=0x21)
timeout
CON("GET /temp", MID=0x101, Token=0x21)
ACK(MID=0x101, "Content", Token=0x21, "79 F")

Figure 6 - CON request message is lost, and resent.

In the case where the original CON request arrived at client 2, but where the CON ACK was lost instead, client 1 will again timeout and resend the original request, as shown in Figure 7.

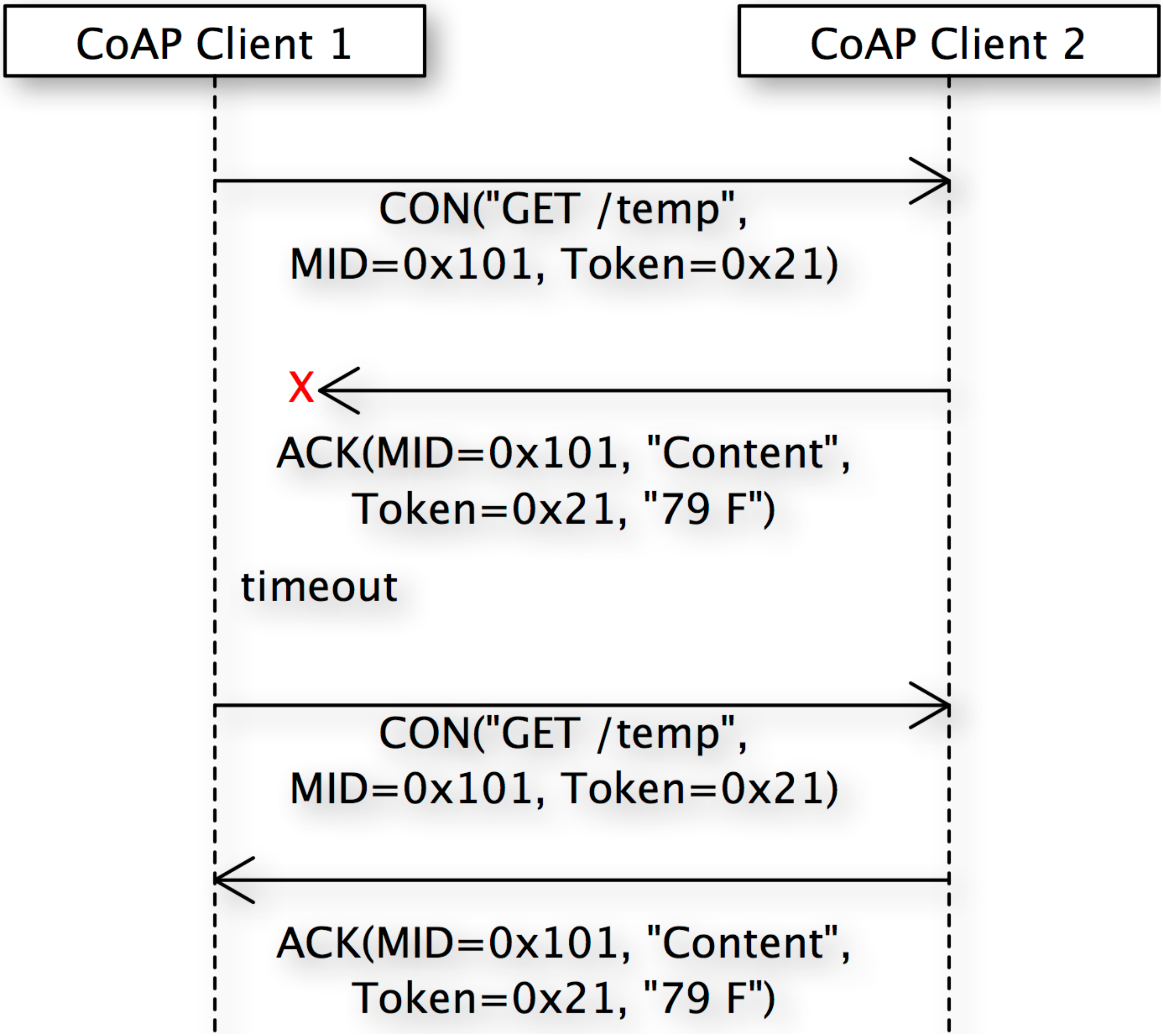

CoAP Client 1
CoAP Client 2
CON("GET /temp", MID=0x101, Token=0x21)
X
ACK(MID=0x101, "Content", Token=0x21, "79 F")
timeout
CON("GET /temp", MID=0x101, Token=0x21)
ACK(MID=0x101, "Content", Token=0x21, "79 F")

Figure 7 - CON request arrives, but ACK is lost.

For the scenario where piggybacking is not used, and an explicit CON response is sent, the failure scenario may be slightly different (see Figure 8).

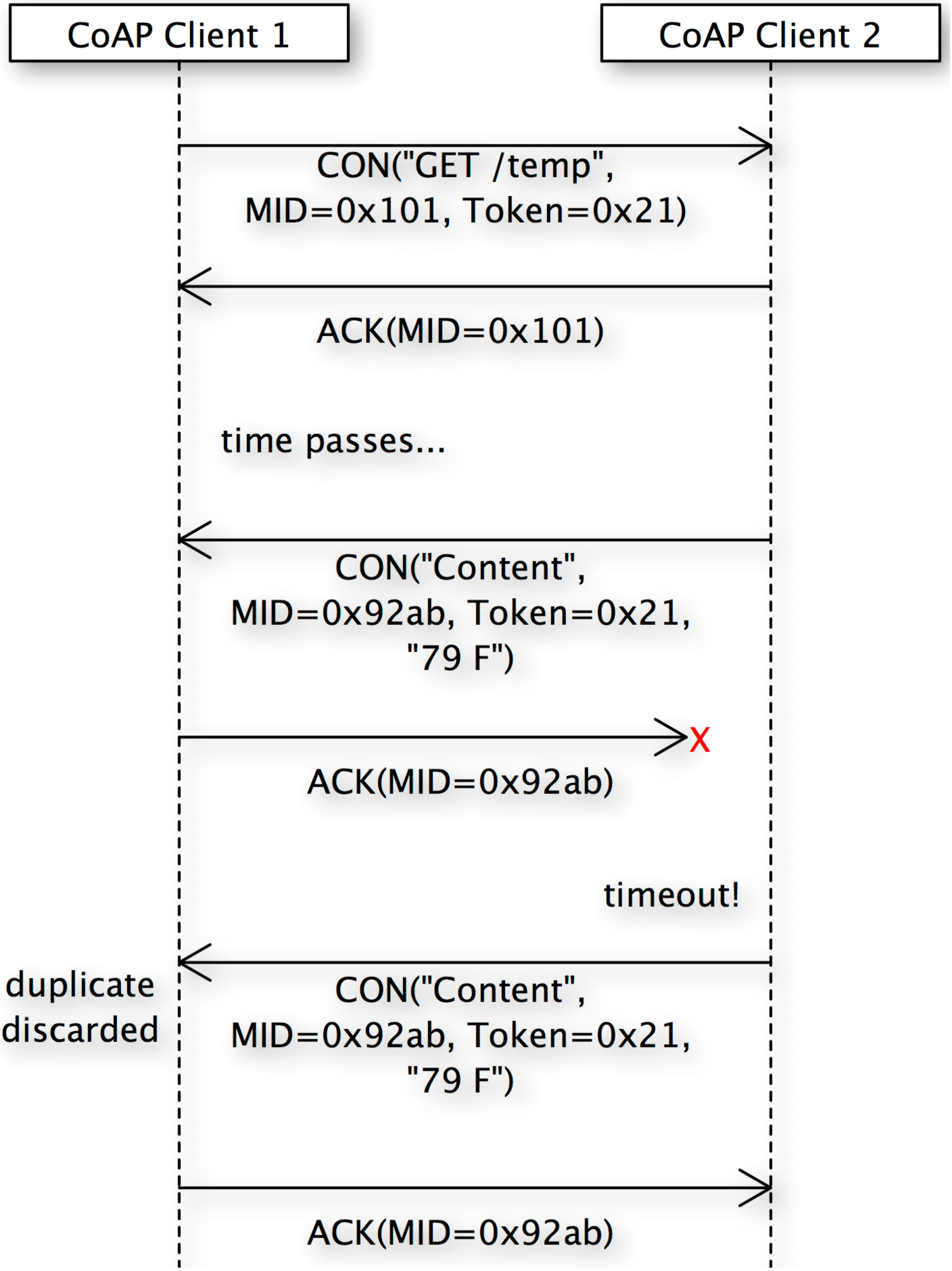

CoAP Client 1
CoAP Client 2
CON("GET /temp", MID=0x101, Token=0x21)
ACK(MID=0x101)
time passes...
CON("Content", MID=0x92ab, Token=0x21, "79 F")
X
ACK(MID=0x92ab)
timeout!
duplicate discarded
CON("Content", MID=0x92ab, Token=0x21, "79 F")
ACK(MID=0x92ab)

Figure 8 - CON request, ACK, and response arrives, but final ACK is lost.

In this scenario, the CON request, its ACK, and its CON response are all received. However, client 1's ACK back to client 2 (which would otherwise complete the exchange) is lost. After a timeout, client 2 assumes its CON response was lost, resends it, which client 1 receives and detects as a duplicate. Client 1 ignores the response but resends the ACK, completing the CON message exchange.

Message Retransmission Timeout Interval

When any CON message is sent, an ACK must be received by a random time interval somewhere between the ACK_TIMEOUT and (ACK_TIMEOUT * ACK_RANDOM_FACTOR) calculated value. If an ACK is not received in time, the sender retransmits the CON message at exponentially increasing intervals, until it receives an acknowledgement (or a Reset message), or it runs out of attempts. This is defined as the MAX_RETRANSMIT value.

In summary, each time a message is retransmitted, its transmit counter is incremented and its wait time is doubled. See the CoAP specification for a detailed explanation of the algorithm. Figure 9 below summarizes all of the possible CoAP message exchange scenarios discussed (and possibly some more).

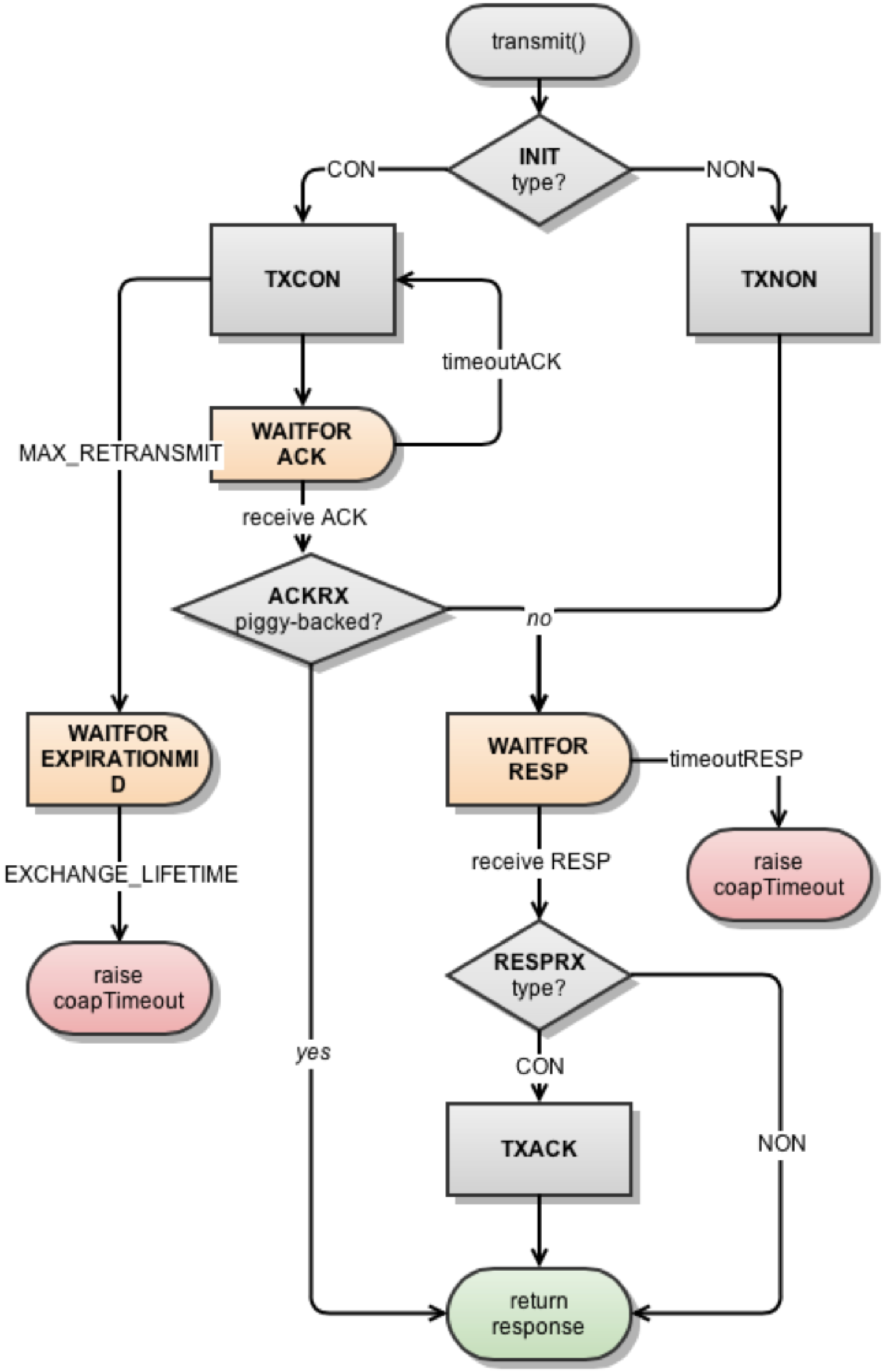

transmit()
INIT type?
CON
NON
TXCON
TXNON
timeoutACK
WAITFOR ACK
MAX_RETRANSMIT
receive ACK
ACKRX piggy-backed?
no
yes
WAITFOR EXPIRATIONMID
EXCHANGE_LIFETIME
raise coapTimeout
WAITFOR RESP
timeoutRESP
raise coapTimeout
receive RESP
RESPRX type?
CON
NON
TXACK
return response

Figure 9 - CoAP message model summary diagram - Image credit: https://openwsn.atlassian.net/wiki/display/OW/CoAP.

Whereas the messaging discussed so far has been viewed from the unicast UDP perspective, CoAP does support multicast messaging. In this paradigm, requests can be sent to groups of servers reachable at a UDP multicast address, where all or some of the servers listening may respond to a single request.

CoAP Security Overview

CoAP by default supports what is known as "NoSec" mode, where no security, encryption, or authentication is performed. As mentioned previously, for secure communications, CoAP messaging can be performed over DTLS instead of UDP. Additionally, CoAP provides support for key-based certificates and access control lists for authentication and authorization.

The CoAP URI

The CoAP URI scheme uses coap (for communication over UDP) and coaps (for communication over DTLS) to locate resources, which are organized hierarchically. The format is as such:

`URI = "coap://[:][?]"`
or…
`URI = "coaps://[:][?]"`

Example: `coap://example.com:5683/~sensors/readings.xml`

CoAP Method Definitions

CoAP messaging is analogous, but not equivalent, to HTTP communication. For example, both CoAP and HTTP support the same four basic commands: GET, POST, PUT, and DELETE, but the semantics of the commands vary slightly. For instance, many of the success and error codes are different between the two. Let's examine the CoAP method definitions closer now.

CoAP GET

As with REST, this method retrieves a representation for the information that corresponds to the resource within the URI at the point in time the request is made. As with HTTP, the request can include an Accept option that suggests the preferred content of the response.

The GET response status codes are: 2.03 for a valid request, 2.05 for valid with content, 4.05 for not allowed. The code in Listing 1 is an example of a CoAP GET from a Java client using the Eclipse Calfornium CoAP library (https://www.eclipse.org/californium) .

```
import java.net.URI;
import java.net.URISyntaxException;
import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.Utils;

public class Coap_get_client {
  public static void main(String args[])
      throws URISyntaxException {
    URI uri = new URI("coap://10.0.1.97:5683/temp");

    // make synchronous get call
    CoapClient client = new CoapClient(uri);
    CoapResponse response = client.get();

    if ( response != null ) {
      System.out.println(response.getCode());
      System.out.println(response.getOptions());
      System.out.println(response.getResponseText());

      System.out.println("\nDETAILED RESPONSE:");

      // Calfornium class provides response details:
      System.out.println(Utils.prettyPrint(response));
    }
  }
}
```

Listing 1 - Java code to make a CON GET for temperature data

The call to `CoapClient.get()` is synchronous (via the Californium library implementation), and will wait to return a code and payload (if successful). You can use the `response.getPayload()` to retrieve the payload as a byte array—`byte[]`—or use the `getResponseText()` shortcut as shown in this example. Also shown in this listing is the call to retrieve the return code and any optional parameters. Finally, the Californium library provides a utility class with a method, `prettyPrint()`, which shows you all of the response details as shown in Listing 2.

```
2.05
{"Content-Format":"text/plain"}
70 degrees F

ADVANCED:
==[ CoAP Response ]============================================
MID    : 53522
Token  : d09b7c5ede
Type   : ACK
Status : 2.05
Options: {"Content-Format":"text/plain"}
Payload: 12 Bytes
---------------------------------------------------------------
70 degrees F
```

Listing 2 - The output of the CoAP GET request for the current temperature, with complete response details.

In this case, the return code was 2.05 indicating that the response was sent as part of the CoAP ACK message. Also, the payload in the response to the `/temp` request was the text “70 degrees F”. If an error had occurred instead, the return code would have indicated the error and no payload would have been provided.

To contrast, the code in Listing 3 is for an asynchronous CoAP GET request, where execution continues after the request is sent. The response will be processed some time later via a callback class that’s supplied.

```
class AsynchListener implements CoapHandler {
    @Override
    public void onLoad(CoapResponse response) {
        String content = response.getResponseText();
        System.out.println("onLoad: " + content);
    }

    @Override
    public void onError() {
        System.err.println("Error");
    }
}

// ...
AsynchListener asynchListener = new AsynchListener();
client.get( asynchListener );
```

Listing 3 - An asynchronous CoAP GET request, with callback to handle the response.

In this example, after the GET request is made, the response will arrive asynchronously via the `onLoad` method within the `AsynchListener` class, which extends the Calfornium library’s `CoapHandler` callback class.

Similar CoAP GET code in C++ is shown in Listing 4.

```
CoapPDU *pdu = new CoapPDU();
pdu->setType(CoapPDU::COAP_CONFIRMABLE);
pdu->setCode(CoapPDU::COAP_GET);
pdu->setToken((uint8_t*)token, tokenLength);
pdu->setMessageID(0x0005);
pdu->setURI((char*)"temp",4);

// send packet
send( sockfd, pdu->getPDUPointer(),
      pdu->getPDULength(), 0);

// receive packet
recvfrom( sockfd, &buffer, BUF_LEN, 0,
          (sockaddr*)&recvAddr, &recvAddrLen);

CoapPDU *recvPDU = new CoapPDU((uint8_t*)buffer, ret);
```

Listing 4 - C++ CoAP GET code using the cantcoap library.

The “server” code that runs on the temperature device handles the GET request. Alternatively, a device connected to the temperature sensor that’s able to handle CoAP communication—such as an IoT gateway—would run this code. The C++ code in Listing 5 shows a sample CoAP server to handle temperature requests. This application uses the cantcoap C/C++ library (https://github.com/staropram/cantcoap) .

```
struct addrinfo* bindAddr;
int ret = setupAddress( "http://10.0.1.97",
                        "5683",
                        &bindAddr,
                        SOCK_DGRAM, AF_INET);

if (ret != 0) {
    return -1;
}

// setup socket
int sockfd = socket(bindAddr->ai_family,
                    bindAddr->ai_socktype,
                    bindAddr->ai_protocol);

bind(sockfd, bindAddr->ai_addr, bindAddr->ai_addrlen);

// buffers for UDP and URIs
char buffer[BUF_LEN];
char uriBuffer[URI_BUF_LEN];

// storage for the receive address object
struct sockaddr_storage recvAddr;
socklen_t recvAddrLen = sizeof(struct sockaddr_storage);

// reuse the same PDU
CoapPDU* recvPDU =
  new CoapPDU((uint8_t*)buffer,
              BUF_LEN, BUF_LEN);

// block and handle requests
while (1) {
    // receive packet
    recvfrom( sockfd, &buffer, BUF_LEN, 0,
             (sockaddr*)&recvAddr, &recvAddrLen);

    recvPDU->setPDULength(ret);
    if (recvPDU->validate() != 1) {
        // Malformed CoAP packet
        continue;
    }

    // send response
    sendResponse(recvPDU,
                 sockfd,
                 &recvAddr,
                 getCurrentTemp() );
}
```

Listing 5 - A C++ CoAP listener that responds to requests for the current temperature.

The first portion of the code is to setup the socket binding and listening. The second portion begins the CoAP specific code, where the protocol data unit (PDU) class, `CoapPDU`, is instantiated to handle requests. Finally, the application enters a loop where it waits for network messages, validates each message as a CoAP message, and sends the appropriate response back to the sender—the current temperature in this case. If your CoAP listener handles multiple URI requests (for different resources, say barometric pressure and humidity for example), you can branch off the URI by calling `CoapPDU::getURI()`.

Although verbose, the code to send the response is relatively straightforward, as shown in Listing 6.

```
int sendResponse(CoapPDU* request,
                 int sockfd,
                 struct sockaddr_storage* recvFrom,
                 char* payload) {
    socklen_t addrLen = sizeof(struct sockaddr_in);
    if (recvFrom->ss_family == AF_INET6) {
        addrLen = sizeof (struct sockaddr_in6);
    }

    // prepare response
    CoapPDU* response = new CoapPDU();
    response->setVersion(1);
    response->setMessageID(request->getMessageID());
    response->setToken(request->getTokenPointer(),
                       request->getTokenLength());

    // Determine CoAP request
    switch (request->getCode()) {
        case CoapPDU::COAP_GET:
            response->setCode(
                CoapPDU::COAP_CONTENT);
            response->setContentFormat(
                CoapPDU::COAP_CONTENT_FORMAT_TEXT_PLAIN);
            response->setPayload(
                (uint8_t*)payload, strlen(payload));
            break;
        case CoapPDU::COAP_POST:
            response->setCode(CoapPDU::COAP_CREATED);
            // handle POST here...
            break;
        case CoapPDU::COAP_PUT:
            response->setCode(CoapPDU::COAP_CHANGED);
            // handle PUT here...
            break;
        case CoapPDU::COAP_DELETE:
            response->setCode(CoapPDU::COAP_DELETED);
            // handle DELETE here...
            response->setPayload((uint8_t*) "DELETE OK", 9);
            break;
    }

    // type
    switch (request->getType()) {
        case CoapPDU::COAP_CONFIRMABLE:
        case CoapPDU::COAP_NON_CONFIRMABLE:
            // send the response as part of the ACK
            response->setType(
                CoapPDU::COAP_ACKNOWLEDGEMENT);
            break;
        case CoapPDU::COAP_ACKNOWLEDGEMENT:
        case CoapPDU::COAP_RESET:
        default:
            return -1;
            break;
    };

    // send the network message
    ssize_t sent = sendto(
            sockfd,
            response->getPDUPointer(),
            response->getPDULength(),
            0,
            (sockaddr*) recvFrom,
            addrLen
            );

    return sent;
}
```

Listing 6 - Sending the appropriate CoAP response to match the resource request and type.

Some code is for network communication, but the rest is CoAP specific, such as determining the type of request (CON versus NON, and GET, POST, PUT, DELETE). In this example, the response type is set to ACK and the response payload is piggybacked onto it.

It’s possible to write CoAP code in Python as well, using a Python library such as CoAPthon. The code in Listing 7 is a Python client that requests the current temperature from a CoAP server application.

```
#!/usr/bin/env python
from Queue import Queue
import getopt
import random
import sys
import threading
from coapthon import defines
from coapthon.client.coap import CoAP
from coapthon.client.helperclient import HelperClient
from coapthon.messages.message import Message
from coapthon.messages.request import Request
from coapthon.utils import parse_uri

client = None

def client_callback(response):
    print "Callback"

def main():  # pragma: no cover
    global client

    host = "localhost"
    port = 5683
    path = "temp"
    client = HelperClient(server=(host, port))

    response = client.get(path)
    print response.pretty_print()
    client.stop()
    sys.exit(0)

if <i>name</i> == '<i>main</i>':  # pragma: no cover
    main()
```

Listing 7 - CoAP Python client that requests the current temperature.

One of the strengths of CoAP is in how it defines the wire protocol very carefully. This enables interoperability between different CoAP library implementations, even across platforms and languages.

CoAP POST

A CoAP POST request asks the recipient to process the representation enclosed *within* the request. The actual function performed by the POST is dependent on the target. It typically results in the target resource being updated or a new resource being created if it doesn't yet exist. If the target content was created, the response should include the new URI to it.

The code in Listing 8 shows a CoAP POST request being made to supply XML to the CoAP server resource named "data".

```
CoapClient client = new CoapClient("coap://10.0.1.97:5683/data");
CoapResponse resp = client.post( "<data>this is data</data>",
                                 MediaTypeRegistry.TEXT_XML );
```

Listing 8 - Making a CoAP POST request is similar to GET, with form data provided.

In this example, the XML sent to the CoAP server resource named “data” would be processed in the C++ `switch` statement for type `CoapPDU::COAP_POST` (see previous Listing 6). Handling this in Java (see Listing 9) is straightforward as well.

```
public class MyCoapServer extends CoapResource {
  public static void main(String[] args) {
    CoapServer server = new CoapServer();
    server.add(new MyCoapServer("data"));
    server.start();
  }

  public MyCoapServer(String name) {
    super(name);
  }

  @Override
  public void handleGET(CoapExchange exchange) {
    // ...
  }

  @Override
  public void handlePOST(CoapExchange exchange) {
    exchange.accept();
    int format = exchange.getRequestOptions().getContentFormat();
    if (format == MediaTypeRegistry.TEXT_XML) {
        String xml = exchange.getRequestText();
        String responseTxt = "Received XML: '" + xml + "'";
        System.out.println(responseTxt);
        exchange.respond(CREATED, responseTxt);

    }
    else if (format == MediaTypeRegistry.TEXT_PLAIN) {
        // ...
        String plain = exchange.getRequestText();
        String responseTxt = "Received text: '" + plain + "'";
        System.out.println(responseTxt);
        exchange.respond(CREATED, responseTxt );
    }
    else {
        // ...
        byte[] bytes = exchange.getRequestPayload();
        System.out.println("Received bytes: " + bytes);
        exchange.respond(CREATED);
    }
  }
  // ...
}
```

Listing 9 - Handling CoAP GET and POST requests in Java.

POST status codes: 2.01 if the target was created, 2.04 if the target exists and was updated, 2.02 if the target exists and was deleted, and 4.05 for not allowed.

CoAP PUT

This method is similar to POST, with a subtle difference for when the target resource doesn't exist. PUT results in the target resource being updated if it exists, but if it doesn't exist, the target has the option to create it or not. If the target content was created, the response should include the new URI to it.

PUT status codes: 2.01 if the target was created, 2.04 if the target exists and was updated, and 4.05 for not allowed.

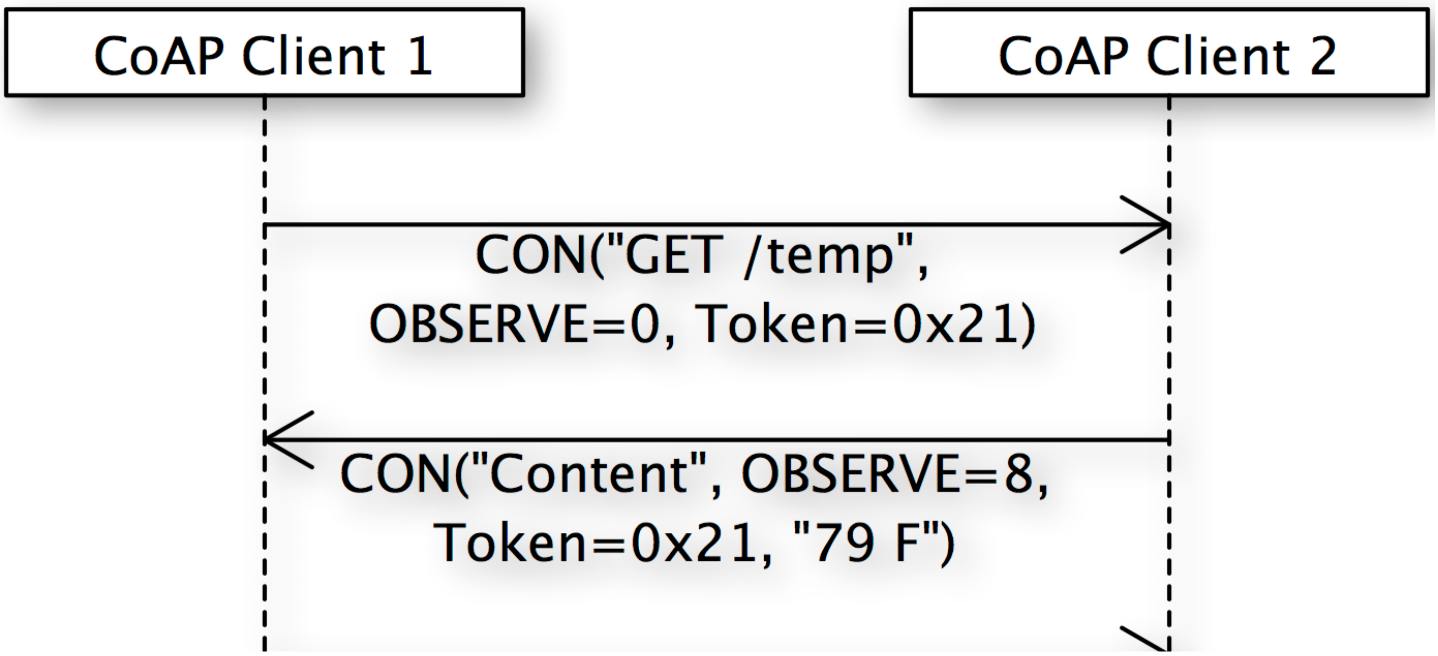
CoAP DELETE

The DELETE method requests that the resource identified by the target resource URI be deleted.

DELETE status codes: 2.02 if the target was deleted, and 4.05 for not allowed.

The Subject-Observer Pattern

The CoAP protocol facilitates extensions to the basic REST-like request/response protocol. One example is the CoAP extension for observing resources via the observer design pattern (http://tools.ietf.org/pdf/draft-ietf-core-observe-03.pdf) . In this scenario, let's say a client is interested in the changes to a target resource over time. To register interest in updates to a resource without explicitly requesting or polling for it continuously, which wastes time and resources, the CoAP client simply adds an OBSERVE entry in the request header with the value of 0 indicating a registration request. The Observer extension to CoAP supports confirmable and non-confirmable registrations for resource updates. The sequence diagram in Figure 10 is a sample confirmable (CON) observer registration request with updates.



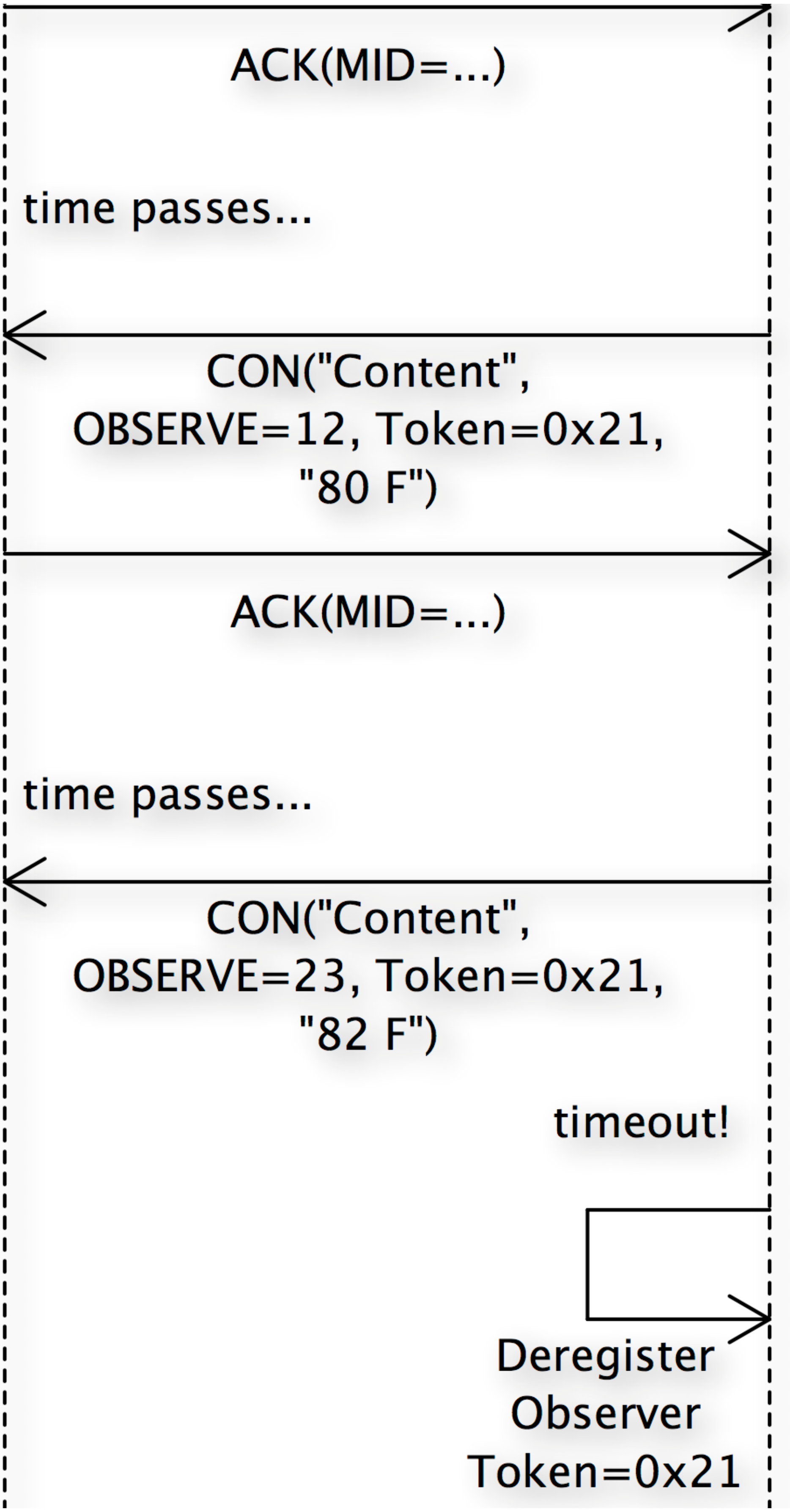

ACK(MID=...)
time passes...
CON("Content", OBSERVE=12, Token=0x21, "80 F")
ACK(MID=...)
time passes...
CON("Content", OBSERVE=23, Token=0x21, "82 F")
timeout!
Deregister Observer Token=0x21

Figure 10 - A CON request with observer registration.

First, client 1 makes a request to client 2 with the OBSERVE option in the header set to 0. This indicates a registration request (value must be 0). Client 2 sends an update with initial value immediately, echoing the token client 1 sent with the request, and will continue to use this token with each update. Since the request was CON message, each response with data requires an ACK. As time passes and the observed value changes, client 2 will send the new value in an update CON response. Again, these updates will contain the same token as the initial request, and client 1 must send an ACK. If an ACK is not sent, client 2 will deregister client 1 after the normal timeout period.

From a client application point of view, setting up the observer pattern is similar to an asynchronous resource request (see Listing 10).

```
class AsynchListener implements CoapHandler {
    @Override
    public void onLoad(CoapResponse response) {
        System.out.println( response.getResponseText() );
    }

    @Override
    public void onError() {
        System.err.println("Error");
    }
}
//...

CoapClient client =
    new CoapClient("coap://10.0.1.97:5683/temp");

// observer pattern uses asynchronous listener
AsynchListener asynchListener =
    new AsynchListener();

CoapObserveRelation observation =
    client.observe(asynchListener);

// User presses ENTER to exit
waitForKeypress();
System.out.println("Exiting...");

// cancel updates from the server resource
observation.proactiveCancel();
```

Listing 10 - CoAP client application implementing the observer listener pattern

As with an asynchronous GET request, the first step is to supply an asynchronous listener callback in the call to `CoapClient.observe()`. From that point onward, the callback class will receive updates as the data is changed according to the server resource (in this example, as the measured temperature changes).

The updates can be stopped by canceling the observation through the method `CoapObserveRelation.proactiveCancel()`. This method sends a RESET message to the server in response to the next update, essentially canceling the observation. At that point, the server will remove this client from the list of observers for the associated resource.

With the Californium library, on the server side, the observation is implemented by marking the resource as observable as shown in Listing 11.

```
public class Coap_server_observe extends CoapResource {
    public static void main(String[] args) {
        CoapServer server = new CoapServer();
        server.add(new Coap_server_observe("temp"));
        server.start();

        // schedule a periodic update task,
        // otherwise let events call changed() as needed
        Timer timer = new Timer();
        timer.schedule(new UpdateTask(), 0, 1000);
    }

    public Coap_server_observe(String name) {
        super(name);

        // enable observations
        setObservable(true);

        // configure the notification type to CONs
        setObserveType(Type.CON);

        // mark observable in the Link-Format
        getAttributes().setObservable();

        // schedule a periodic update timer
        // otherwise just call changed() as appropriate
        Timer timer = new Timer();
        timer.schedule(new UpdateTask(), 0, 1000);
    }
    // ...
```

Listing 11 - Marking a CoAP server resource as observable.

Calling the `CoapResource.changed()` method causes this Calfornium CoAP server to automatically send a subsequent response (an update) to the initial GET request for data on the observable resource (see Listing 12).

```
private class UpdateTask extends TimerTask {
    @Override
    public void run() {
        changed(); // notify all observers
    }
}

@Override
public void handleGET(CoapExchange exchange) {
    // the Max-Age value should match the update interval
    exchange.setMaxAge(1);
    exchange.respond("Current temperature: " +
                     getCurrentTemp() );
}
```

Listing 12 – The CoAP server resource timer task that causes a GET response to be sent to all observers periodically.

For each active observer, the `handleGET()` method will be called, where the latest data value (temperature in this example) will be acquired and sent in the response.

Device Discovery

CoAP supports dynamic device discovery, as it's intended to support changing networks of devices and sensors in the IoT world where human intervention isn't required. To discover another CoAP server, the client is required to either know about the resource ahead of time, or support Multicast CoAP via UDP messaging on a multicast address and port. Servers that wish to be discoverable must listen on the "All CoAP Nodes" multicast address and default port of 5683 (unless a different port is agreed upon in advance). A server can reply to requests at this address and port combination to let other clients or servers know of its existence and addressable URI.

The multicast "All CoAP Nodes" address is `224.0.1.187` for IPv4, and `FF05::FD` for IPv6. Sending a request for the CoAP resource directory name "`/.well-known/core`" should result in a reply from every reachable CoAP resource on the local network segment, so long as multicast is enabled and each server is listening on the multicast address (see Listing 13).

```
CoapServer server = ...
InetAddress addr = InetAddress.getByName("224.0.1.187");
bindToAddress = new InetSocketAddress(addr, COAP_PORT);
multicast = new CoapEndpoint(bindToAddress);
server.addEndpoint(multicast);
```

Listing 13 – Listening for "All CoAP Nodes" multicast requests.

Here, the multicast address is set as a `CoapEndpoint` to the Californium `CoapServer` object.

You can make a request to discover CoAP servers and their resources via a CoAP GET request such as that shown in Listing 14.

```
CoapClient client =
    new CoapClient("coap://FF05::FD:5683/.well-known/core");
client.useNONs();
CoapResponse response = client.get();
if ( response != null ) {
    System.out.println("Response code: " +
                       response.getCode());
    System.out.println("Resources: " +
                       response.getResponseText());
    System.out.println("Options: " +
                       response.getOptions());

    // get server’s IP address
    InetAddress addr = response.advanced().getSource();
    int port = response.advanced().getSourcePort();
    System.out.println("Source address: " +
                       addr + ":" + port);
}
```

Listing 14 - CoAP GET request to discover CoAP servers and resources on a local network.

It’s important to note that the request needs to be a NON GET request, hence the call to `client.useNONs()` in the second line. Additionally, making a request to the base URI `coap://FF05::FD:5683` yields basic information about the server (at least it does if it’s using the Californium library).

Dynamic resource discovery is useful when building a dynamic IoT application, where it’s not desired to hard-code or manually configure available CoAP servers and their resources. For instance, if you have a single controller application that allows all lights (or other appliances) within a room or building floor to be turned off and on together, you can use resource discovery to locate all available smart lighting devices. Using the results of the discovery, you can send CoAP commands to each smart lighting device to turn off and on as appropriate. If new lighting devices are added at some future date, the controller code will continue to work on all lighting devices without change.

CoAP Resource Directory

To better enable resource discovery for constrained devices—where some devices may be sleeping or non-communicative at times—the CoAP Resource Directory entity was defined (https://tools.ietf.org/html/draft-ietf-core-resource-directory) , which is still in draft form. This entity maintains descriptions of CoAP servers and resources within your distributed application. Specifically, devices can register themselves as servers, along with their resources, in a well-known resource directory node. CoAP client applications can subsequently refer to the resource directory to learn about resources as they become available, and then become part of the distributed application.

CoAP endpoints register themselves with the resource directory via its registration interface (see Figure 11). CoAP client applications then use the lookup and/or CoAP group interfaces—more on groups in the next sections—to learn about available resources.

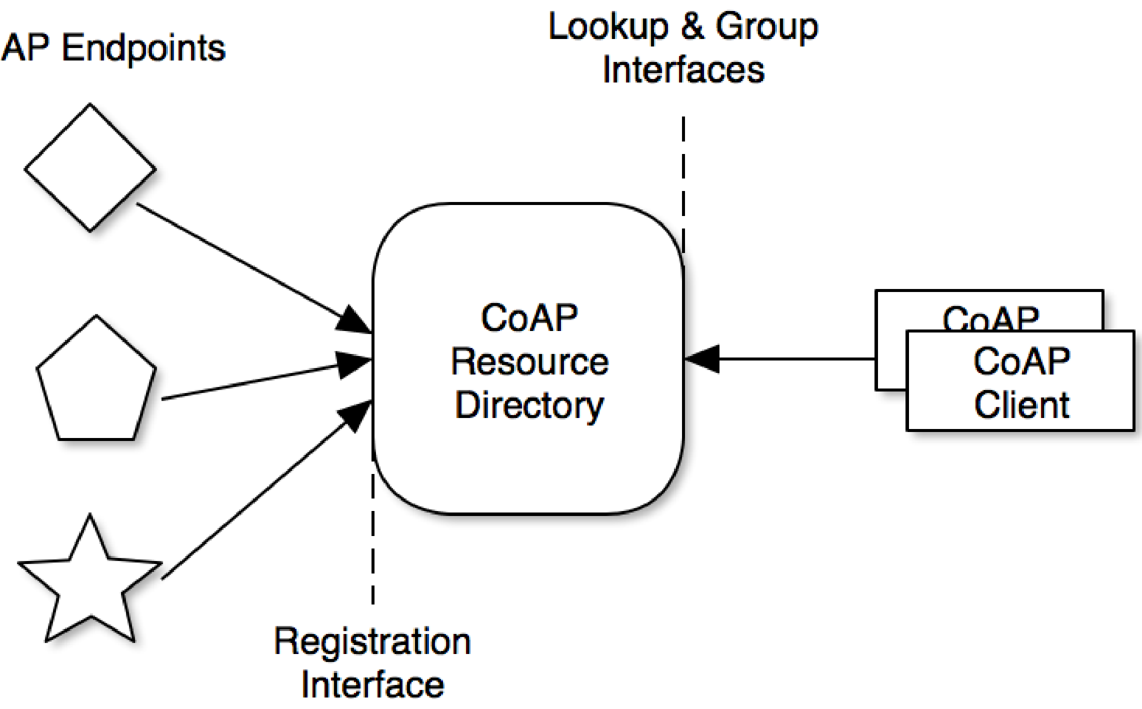


Figure 11 - CoAP Resource Directory interfaces.

Endoints register their resources by sending a POST request with the resource path (i.e. `/temp`), the endpoint name, and optional data such as a domain, the endpoint type, and other data. If successful, the response code will be 2.01, and a resource identifier will be returned (i.e. `/rd/1234`). This identifier can be used to access the CoAP resource through the resource directory server. For instance, the code in Listing 15 registers a pulse oximeter's *heartrate* and *oxygen saturation* telemetry resources.

```
String host = "coap://10.0.1.111/”;
CoapClient rd;

System.out.println("Registering resource: heartrate ");
rd = new CoapClient(host+”rd?ep=pulseoximeter/heartrate/");
resp = rd.post("</pulseoximeter/heartrate>;"
                + "ct=41;rt=\"Heartrate Resource\";"
                + "if=\"sensor\"",
               MediaTypeRegistry.APPLICATION_LINK_FORMAT);
System.out.println("--Response: " +
                   resp.getCode() + ", " +
                   resp.getOptions().getLocationString());

System.out.println("Registering resource: oxygen-saturation ");
rd = new CoapClient(host+”rd?ep=pulseoximeter/oxygen-saturation/");
resp = rd.post("</pulseoximeter/oxygen-saturation>;"
                + "ct=41;rt=\"Oxygen Saturation Resource\";"
                + "if=\"sensor\"",
               MediaTypeRegistry.APPLICATION_LINK_FORMAT);
System.out.println("--Response: " +
                   resp.getCode() + ", " +
                   resp.getOptions().getLocationString());
```

Listing 15 - CoAP client code to register CoAP endpoint resources with a resource directory server.

First, the code connects to a CoAP resource directory server running on node 10.0.1.111 (this is just an arbitrary address for this example). It connects using a resource endpoint name of `pulseoximeter/heartrate` since that's the resource it registers first. Next, a POST request is made using that endpoint, a URI path of `/pulseoximeter/heartrate`, a name of “Heartrate Resource”, and endpoint type “sensor”. The same is done for the other resource, `/pulseoximeter/oxygen-saturation`. When this executes successfully, you should see output similar to Listing 16.

```
Registering resource: heartrate
--Response: 2.01, /rd/pulseoximeter/heartrate
Registering resource: oxygen-saturation
--Response: 2.01, /rd/pulseoximeter/oxygen-saturation
```

Listing 16 - The result of a successful resource registration.

To further illustrate the results of registration, add code to make a resource discovery request to the resource directory server, as shown in Listing 17.

```
CoapClient q = new CoapClient("coap://10.0.1.111/.well-known/core");
CoapResponse resp = q.get();
System.out.println( "--Registered resources: " +
                    resp.getResponseText());
```

Listing 17 - Send a resource discovery request to the resource directory.

Adding this code both before the endpoint registration requests and after yields the complete set of output shown in Listing 18.

```
--Registered resources: </rd>;rt="core.rd",</rd-lookup>;rt="core.rd-lookup",</rd-lookup/d>,</rd-loo

Registering resource: heartrate
--Response: 2.01, /rd/pulseoximeter/heartrate
Registering resource: oxygen-saturation
--Response: 2.01, /rd/pulseoximeter/oxygen-saturation

--Registered resources: </rd>;rt="core.rd",</rd/pulseoximeter/heartrate/>,</rd/pulseoximeter/oxygen
```

Listing 18 - The results of endpoint registration.

Note the results from resource discovery request made after the endpoint registration POST requests now includes the two added pulse oximeter endpoint resources, as expected.

CoAP Resource Group Definitions

To further enable CoAP device group communication, the CoAP CoRE Working Group has defined the Group Communication for CoAP specification as RFC 7390 (https://tools.ietf.org/html/rfc7390) . This specification outlines methods to define and subsequently communicate with groups of devices.

For instance, for a CoAP server that supports rfc7390, a POST request to the resource `/coap-group` with a group name and index provided as form data would create a new CoAP group. An example is shown in Listing 19.

```
POST /coap-group
Content-Format: application/coap-group+json
{
  "n": "lights.floor1.example.com",
  "a": "[ff15::4200:f7fe:ed37:abcd]:1234"
}
```

Listing 19 - CoAP Group creation as a POST message.

If successful, the response will be similar to that shown in Listing 20.

```
2.01 Created
Location-Path: /coap-group/12
```

Listing 20 - The response for a POST request to create a new CoAP group.

Making a GET request to `/coap-group` will return JSON data indicating all members of the group. A sample response is shown in Listing 21.

```
2.05 Content
Content-Format: application/coap-group+json
{
  "8" : { "a": "[ff15::4200:f7fe:ed37:14ca]" },
  "11": { "n": "sensors.floor1.example.com",
          "a": "[ff15::4200:f7fe:ed37:25cb]" },
  "12": { "n": "lights.floor1.example.com",
          "a": "[ff15::4200:f7fe:ed37:abcd]:1234" }
}
```

Listing 21 - CoAP GET response for a CoAP group request.

Subsequently, you can make requests to a specific group, i.e. `/coap-group/12`, to control or read the status of the devices within that group as a whole.

Cross-Protocol Proxying (CoAP and HTTP)

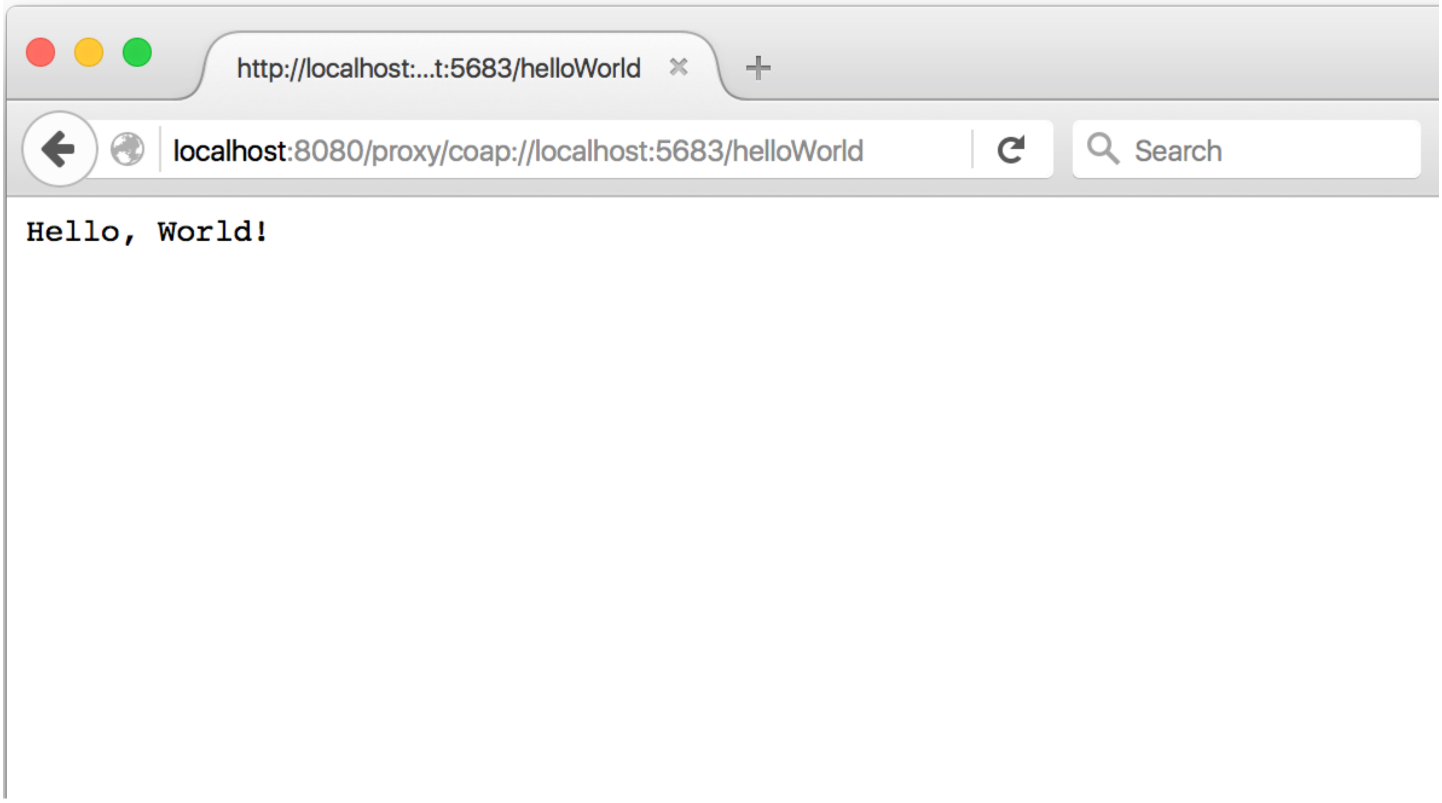
Because CoAP is a REST-based protocol based on the underlying HTTP implementation, it’s straightforward to map CoAP methods to HTTP. As such, it’s straightforward to proxy CoAP requests to and from HTTP to, for instance, allow CoAP clients to access resources available on an HTTP server, or allow HTTP clients (such as JavaScript code) to make requests to CoAP servers.

For instance, the code in Listing 22 is from the Californium sample code, which shows how to implement a simple CoAP-to-CoAP and CoAP-to-HTTP proxy.

```
private static class TargetResource extends CoapResource {
    private int counter = 0;

    public TargetResource(String name) {
        super(name);
    }

    @Override
    public void handleGET(CoapExchange exchange) {
        exchange.respond(
          "Response " + (++counter) +
          " from resource " + getName() );
    }
}

public coap_cross_proxy() throws IOException {
    ForwardingResource coap2coap =
      new ProxyCoapClientResource("coap2coap");
    ForwardingResource coap2http =
      new ProxyHttpClientResource("coap2http");

    // Create CoAP Server with proxy resources
    // from CoAP to CoAP and HTTP
    targetServerA = new CoapServer(8082);
    targetServerA.add(coap2coap);
    targetServerA.add(coap2http);
    targetServerA.start();

    ProxyHttpServer httpServer =
      new ProxyHttpServer(8080);
    httpServer.setProxyCoapResolver(
      new DirectProxyCoapResolver(coap2coap) );

    System.out.println(
      "CoAP resource \"target\" available over HTTP at: " +
      "http://localhost:8080/proxy/coap://localhost:PORT/target");
}
```

Listing 22 - A simple HTTP-to-CoAP Proxy from the Californium sample applications.

Running a resource named “helloWorld” as `coap://localhost:5683/helloWorld` and browsing to the proxy URL, as specified in the code, results in the payload displayed in the browser (see Figure 12).


http://localhost:...t:5683/helloWorld
localhost:8080/proxy/coap://localhost:5683/helloWorld
Search

```
Hello, World!
```

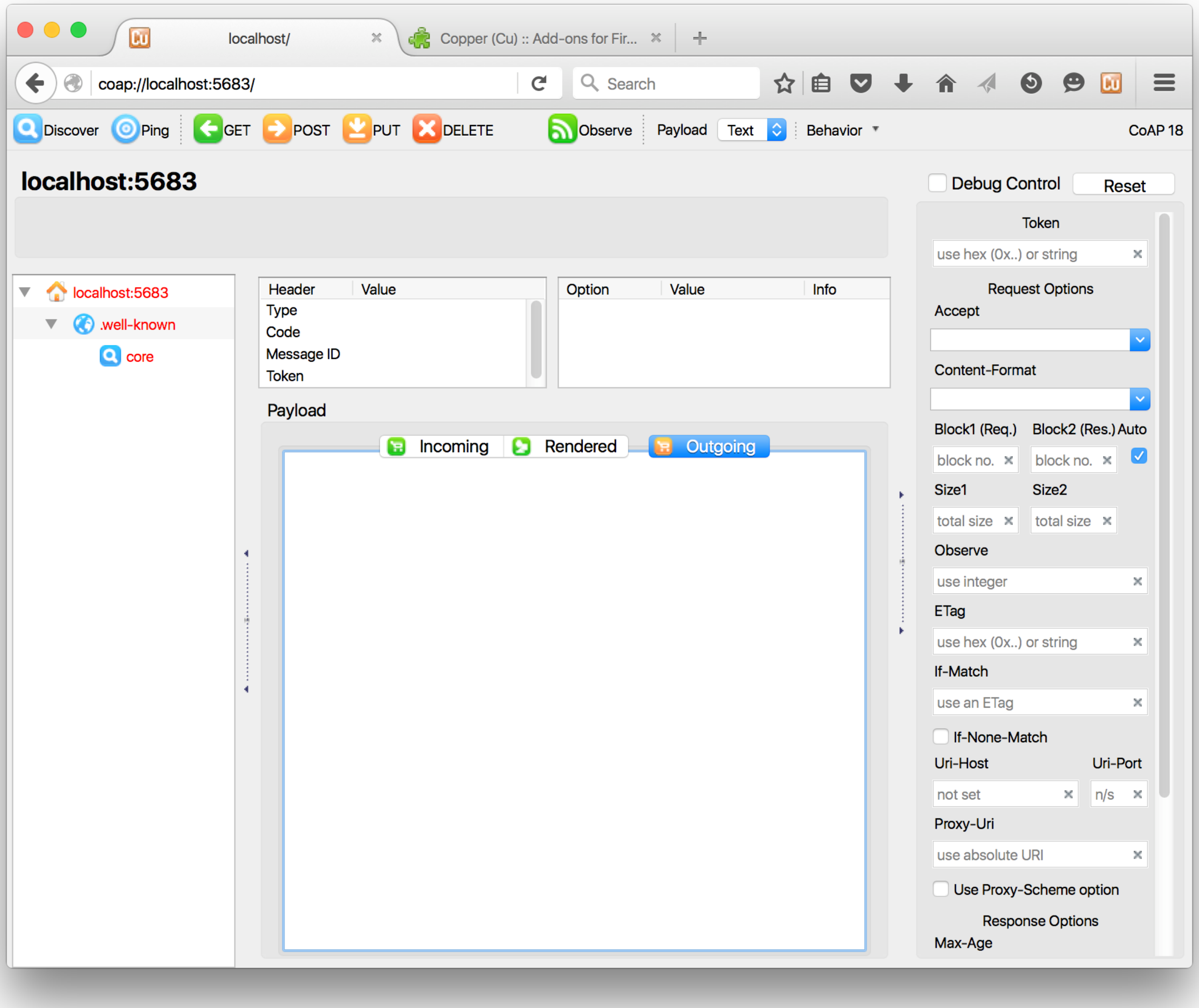
Figure 12 - The result of a CoAP-to-HTTP proxy: response text displayed in the browser

Further, CoAP can be proxied to other protocols, such as Session Initiation Protocol (SIP), Extensible Messaging and Presence Protocol (XMPP), and so on.

Copper for Firefox

Copper is Firefox plug-in (https://addons.mozilla.org/en-US/firefox/addon/copper-270430) that allows you to visually discover, explore, and interact with CoAP devices on your network. It was written by Matthias Kovatsch., and is a fantastic tool that you should use to help learn and debug CoAP endpoint programming.

Once Copper is installed, browse to the address of a CoAP node on your network. For instance, Figure 13 shows the result of typing `coap://localhost:5683` in the Firefox address bar.



coap://localhost:5683/

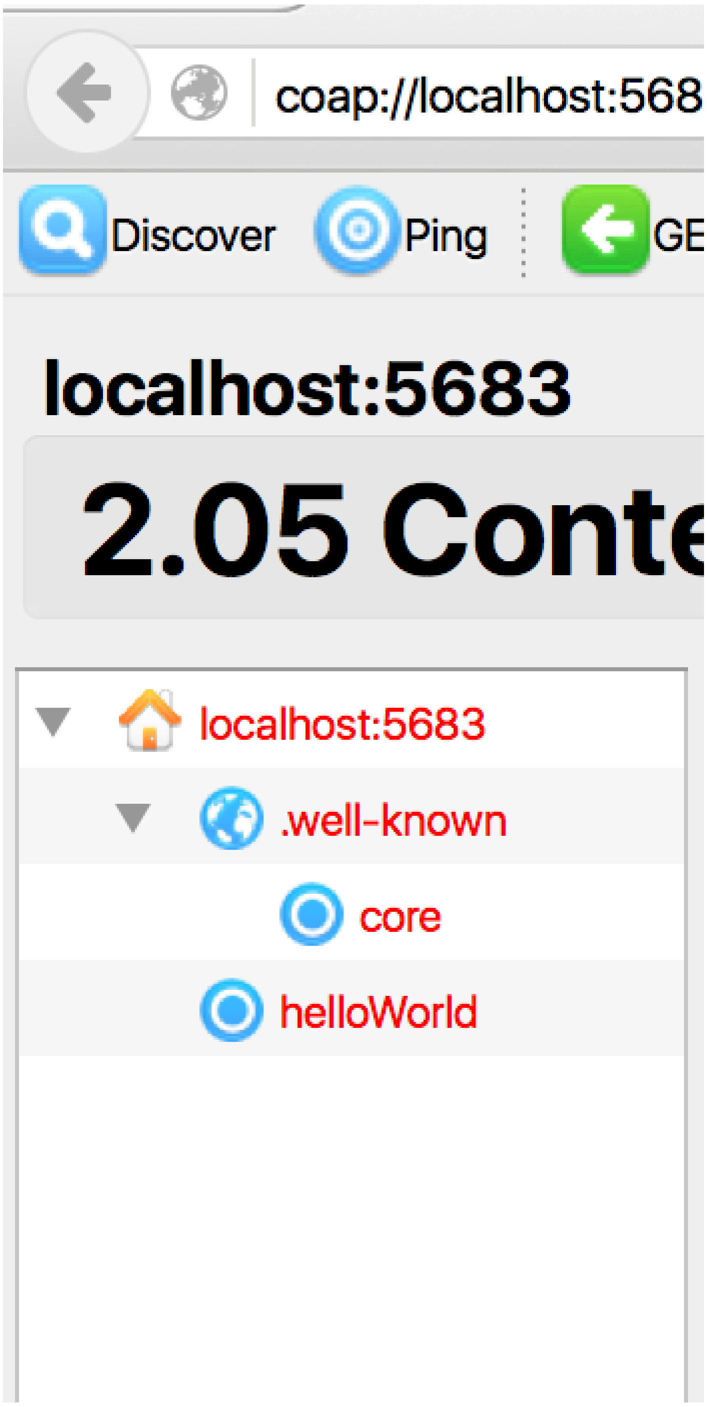
localhost:5683

Header	Value
Type	
Code	
Message ID	
Token	

Option	Value	Info

Figure 13 - The Copper Firefox add-on visually displays and interacts with CoAP instances.

Clicking on the *Discover* button in the upper left updates the list box on the left (see Figure 14) with the server's supported resources (in this case, a resource named *helloWorld*).



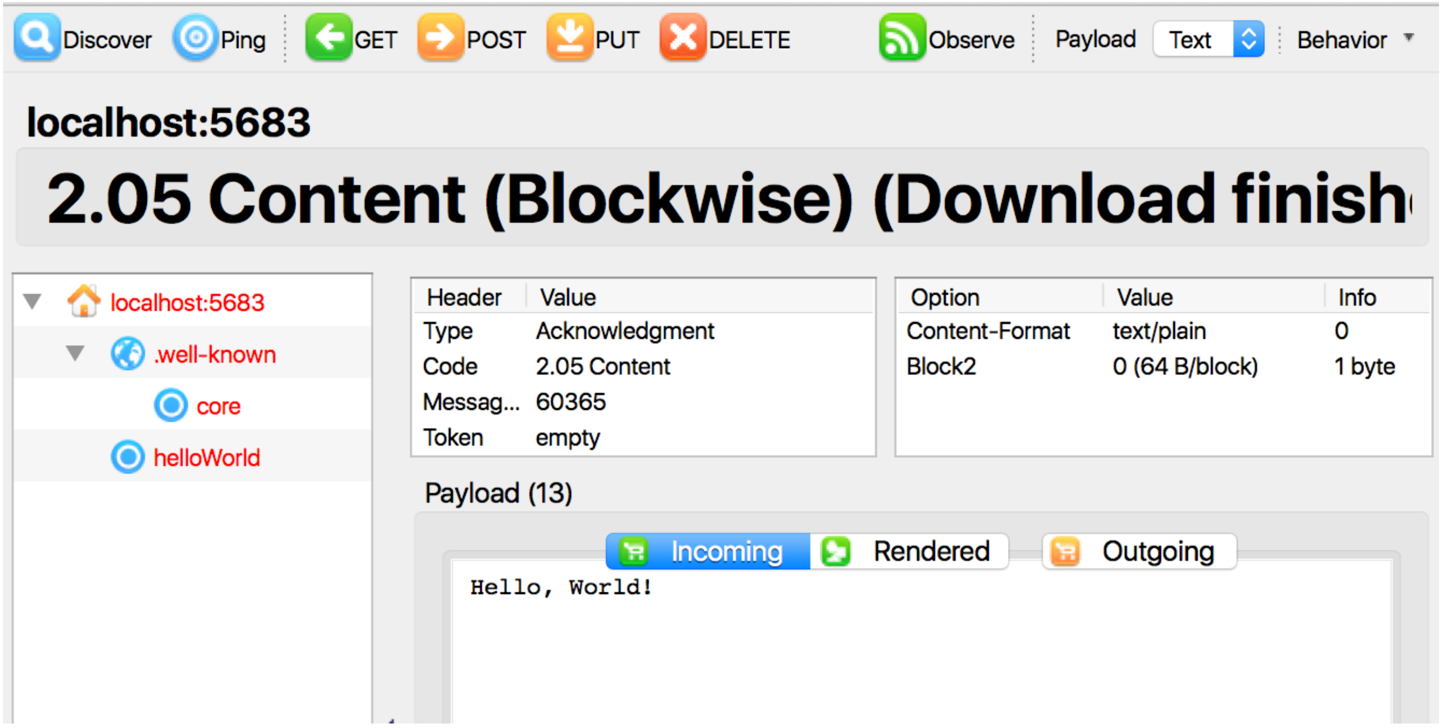
coap://localhost:568

localhost:5683

2.05 Conte

Figure 14 – Easily discover CoAP resources available at CoAP endpoints on your network.

Next, select the *helloWorld* resource and click the *Get* button at top. As a result a CoAP GET request is sent to the resource, and the results are displayed in the *Payload* text box, as shown in Figure 15.



localhost:5683

2.05 Content (Blockwise) (Download finish

Header	Value
Type	Acknowledgment
Code	2.05 Content
Messag...	60365
Token	empty

Option	Value	Info
Content-Format	text/plain	0
Block2	0 (64 B/block)	1 byte

Payload (13)

```
Hello, World!
```

Figure 15 - Easily make GET and POST requests and debug your CoAP resources.

You can interact by sending data to resources, discovering the resources available on your network, and using other built-in debugging tools without writing any CoAP client code.

CoAP Implementations

There are quite a few open source CoAP implementations for multiple programming languages. There are also some commercial implementations. The next few sections highlight some used for the code samples in this article.

Using Cantcoap for C/C++

Cantcoap is a C/C++ CoAP implementation used for some of the samples in this article. You can download it at https://github.com/staropram/cantcoap (https://github.com/staropram/cantcoap) , and the implementation is small enough that you can explore and understand it easily.

After downloading it, you need to build and install it. Unless you remove the build target named test in `Makefile`, you'll also need to download and build CUnit (https://sourceforge.net/projects/cunit/files/latest/download) . Next, inside `Makefile`, set `LIB_INSTALL` and `INCLUDE_INSTALL` to the following:

```
LIB_INSTALL=$(HOME)/cantcoap/lib
INCLUDE_INSTALL=$(HOME)/cantcoap/include
```

Next, create the directories themselves (lib and include) in the directories added above, and copy the header files via the following commands:

```
> cp dbg.h ~/cantcoap/include/ > cp nethelper.h ~/cantcoap/include
```

Finally, type, `make install` and you're ready to go.

Using Californium for Java

Californium is an Eclipse project (https://www.eclipse.org/californium) that can be downloaded from Github at https://github.com/eclipse/californium (https://github.com/eclipse/californium) . Make sure to get Copper for Firefox (https://www.eclipse.org/californium/#tools) as well.

You need Maven to build and install Californium. To use Maven, remember to set your `JAVA_HOME` environment variable, and then run the Maven command shown here:

```
> mvn clean install -DskipTests
```

Other Java CoAP Implementations

As an alternative to Californium, there are other open-source CoAP libraries available, such as nCoap and jCoap. You can download nCoap from https://github.com/okleine/nCoAP (https://github.com/okleine/nCoAP) and then install it just as you do Californium (via the same Maven command).

You can also download jCoap from WS4D's site (http://ws4d.org/ws4d-jcoap) . To install it, you can use same Maven or simply import it into Eclipse (or other IDE of your choice) and build it there.

Using CoAPthon for Python

To write Python CoAP code, use a CoAP library such as CoAPThon (https://github.com/Tanganelli/CoAPthon) , and then install it via the following command:

```
> sudo python setup.py install
```

Conclusion

CoAP is a simple yet powerful IoT communication specification that defines a lightweight wire protocol meant to work on constrained devices. As such, interoperability is easily achieved across devices, operating systems, and languages, making it flexible to use as well. Additionally, the fact that it closely resembles HTTP/REST communication makes it familiar and easy to understand.