View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Apollo

arXiv:1902.07009v1 [cs.DC] 19 Feb 2019

# Zest: REST over ZeroMQ

John Moore, Andrés Arcia-Moret, Poonam Yadav, Richard Mortier
*University of Cambridge*
Cambridge, UK

Anthony Brown, Derek McAuley, Andy Crabtree, Chris Greenhalgh
*University of Nottingham*
Nottingham, UK

Hamed Haddadi
*Imperial College*
London, UK

Yousef Amar
*Queen Mary University*
London, UK

***Abstract*—In this paper, we introduce Zest (REST over ZeroMQ), a middleware technology in support of an Internet of Things (IoT). Our work is influenced by the Constrained Application Protocol (CoAP) but emphasises systems that can support fine-grained access control to both resources and audit information, and can provide features such as asynchronous communication patterns between nodes. We achieve this by using a hybrid approach that combines a RESTful architecture with a variant of a publisher/subscriber topology that has enhanced routing support. The primary motivation for Zest is to provide inter-component communications in the Databox, but it is applicable in other contexts where tight control needs to be maintained over permitted communication patterns.**

## I. Introduction

The goal behind Zest is to utilise middleware to improve on the features offered by the Constrained Application Protocol (CoAP) [1], [2] by providing:

- Encryption as standard
- Access control through Macaroons
- Support for auditing communication across nodes
- Support for asynchronous communication between nodes

We chose to build our solution using ZeroMQ[1] because of its flexibility to support different topologies such as brokerless communication and for its simple abstraction over traditional TCP sockets. Other reasons we adopted ZeroMQ included its support for secure connections based on elliptic-curve cryptography and that it is well supported across a variety of platforms and programming languages. Zest forms the core protocol within the Databox project [3], which we envisage being instantiated in the form-factor of a set-top box or similar. All components are encapsulated as Docker containers.[2] Databox hosts third-party computations as *Apps*, while external devices such as sensors interface to the Databox via *Drivers* responsible for interacting with the external device through reads and writes to an associated *store*, a lightweight time-series database. Zest's requirements therefore are to support this highly controlled communication model, in a relatively resource-constrained environment, where operations must be logged for subsequent audit and where data transfers should be authenticated and protected in flight. The Databox communication model not only involves encryption across communication channels but also requires support for fine-grained access to resources. We therefore developed the Zest

[1]http://zeromq.org/
[2]https://docker.com/

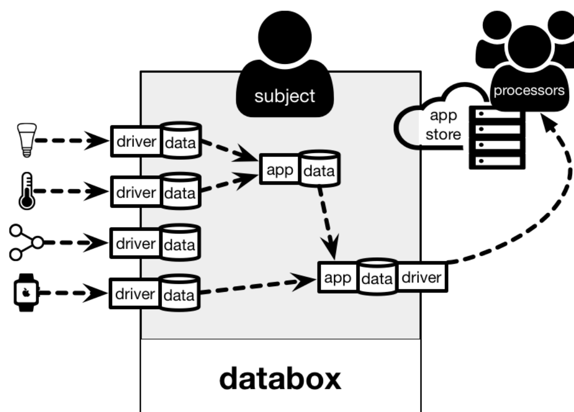


Fig. 1. Databox architecture.

protocol to support these features as standard, including support for audit information to be pushed to any App permitted to receive it. Data is isolated within Databox by enforcing that each Driver may write only to its own Store, and Apps must request permission on installation to be able to access a Store. If the user grants permission, the *App* receives a set of access tokens (formatted as Macaroons [4]) which it can subsequently present to the Store to verify its access is allowed. Data may only be communicated to a third-party service through an *Export Driver*, subject to the user granting appropriate permission when installing the *App*. Available Stores are registered in a HyperCat catalogue on installation, so they can be discovered by other Apps.[3] Each store provides a *RESTful* API supporting JSON, text and binary data across the Zest protocol. Underlying storage is implemented using the Irmin [5] system using a *git*-structured backend. This supports a key design goal of Databox, to provide accountability of data stored and accessed, by using the commit history of the git-based storage system to provide a detailed account of all mutations to data. In the remainder of this paper we discuss related work (§II), detail the Zest protocol (§III) and architecture (§IV), and conclude (§V).

## II. Related Work

A number of alternative technologies exist which can be used to support an Internet of Things, however, the baseline of

[3]https://hypercat.io/

our work corresponds to the Constrained Application Protocol (CoAP) for which the need for integration with enterprise infrastructure has recently motivated its development over TCP [6], [7]. The goal of the CoAP standard was to bring a *RESTful* experience to the Internet of Things. As such, its design targets low-end devices including micro-controllers and it runs over UDP to provide more light-weight communication. However, the decision to use UDP sometimes introduces complexity such as when operating over networks that use NAT or when required to handle large message payloads efficiently. Upgrading CoAP to TCP required changes to the protocol structure to, for example, to accommodate reliable communication. However, complexities remain. The CoAP standard supports the concept of observing a resource so that data can be pushed back to client nodes. This type of interaction is intended to be supported using WebSockets, but this requires supporting an additional protocol.

The basic concept of supporting a RESTful architecture using ZeroMQ is not new.[4] Novelty in Zest arises from adoption of CoAP's approach plus additional features introduced in §I. A number of technologies exist that do not adopt a RESTful approach but are suitable for building and deploying IoT solutions. Some make assumptions on the topology that can be used between communicating nodes and some place restrictions on the data format used within messages. For example, MQTT[5] is a publisher/subscriber messaging protocol where clients communicate via a server known as a broker. A client can connect as either a publisher, subscriber or both publisher and subscriber. Brokers are topic based. A publisher will write data for a specific topic which any subscriber of that topic will receive. Topics can be subscribed to on a hierarchical basis with optional wildcard filtering within the topic path. Security can be implemented over TCP using SSL/TLS.

Protocol Buffers[6] was created by Google to be used for machine-2-machine communication between their servers. Its binary on-the-wire format provided a more light-weight alternative to serialising text-based formats such as XML. Google also developed Cap'n Proto[7] as an improvement over Protocol Buffers in terms of its performance and also included the addition of RPC capabilities. The serialisation approach of both Protocol Buffers and Cap'n Proto is similar to that of Abstract Syntax Notation One (ASN.1) [8] where the protocol is specified in a platform independent language which needs to be parsed into target language code with library support that can generate a binary on-the-wire representation of the data. Binary protocols are efficient for machine-2-machine communication, especially when it comes to transferring numeric data, however, they enforce an on-the-wire format. Using RPC technologies you are not restricted to use a broker to mediate communications like MQTT but you still need to design suitable high-level modes of interaction that end-users or developers can understand. Choosing the correct technology depends very much on the use-case. For example, a traditional REST model is a good approach for database read and writes but is not well suited if a client needs to continually poll a resource to receive an update. Therefore, technologies such as CoAP and HTTP/2 [9] help address these issues by describing support to push data back to clients. The Zest protocol takes a more hybrid approach and borrows from traditional pub/sub technologies such as MQTT to deploy a broker to handle such use-cases. The following section will describe the Zest protocol in more detail.

[4]https://rfc.zeromq.org/spec:40/XRAP/

[5]http://mqtt.org/

[6]https://developers.google.com/protocol-buffers//

[7]https://capnproto.org/

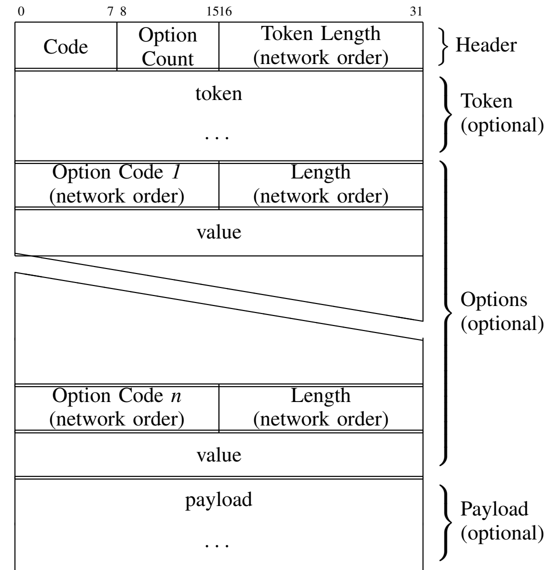


TABLE I
PROTOCOL STRUCTURE

## III. PROTOCOL

Table I shows the structure of a Zest message. A message must consist of at least a header, however, the token, options and payload are all optional and depend on the type of message being sent.

A header is made up of 32 bits. The first 8 bits of the header are used to set the type of message used being either a request message or a response message. Table II lists the possible request and response codes.

The following 8 bits of the header specify how many options have been encoded. Options are used to configure a message and provide extra information such as specify the content type of the payload if one exists. The first 16 bits specify which option is being encoded based on its value shown in table III.

Options are encoded in a tag, length, value sequence where the length refers to the number of bytes required to store the value. The type of value encoded varies depending on the option. For example, to encode a *content_format* option requires encoding an unsigned integer representing the content type such that 0 represents text, 42 represents binary and 50 represents JSON.

The Zest protocol supports POST, GET and DELETE to a specific endpoint specified through a path. As previously

| Type | Code | Meaning |
|---|---|---|
| REQ → | 1 | GET |
| REQ → | 2 | POST |
| REQ → | 4 | DELETE |
| RESP ← | 65 | Acknowledge (POST) |
| RESP ← | 66 | Acknowledge (DELETE) |
| RESP ← | 69 | Acknowledge with payload (GET/POST) |
| RESP ← | 128 | Bad request |
| RESP ← | 129 | Unauthorised |
| RESP ← | 134 | Not acceptable |
| RESP ← | 141 | Request entity too large |
| RESP ← | 143 | Unsupported content format |
| RESP ← | 160 | Internal server error |
| RESP ← | 163 | Service unavailable |

TABLE II
REQUEST AND RESPONSE CODES

| code | meaning | value/type |
|---|---|---|
| 3 | uri_host | string |
| 6 | observe | string set to 'data', 'audit' or 'notify' |
| 11 | uri_path | string |
| 12 | content_format | unsigned integer based on type |
| 14 | max_age | unsigned integer representing seconds |
| 2048 | public_key | string |

TABLE III
OPTION ENCODING

described, each method is configured by setting specific protocol options as shown in figure IV. The POST method is used to add data to a resource such as a database or provide data to update a dashboard. Unlike the CoAP standard, a POST method can not only acknowledge the status of the request but can also return a payload of data. A POST response message might contain an option specifying the content of its payload. A GET method is used to request data from a resource such as a database. The GET method has a variant to support observing a resource. This is used to obtain information suitable for logging and auditing interaction across a Zest deployment. Observing a resource sets up a connection to the router endpoint of a Zest node which remains active until a specific expiry period is reached as dictated by the *max_age* option. If the *max_age* option is not provided a default of 60 seconds is assumed. In addition, a *max_age* value of 0 indicates the connection should not expire. As with all method calls, their usage should be controlled through suitably minted access tokens. The response to a GET method always contains a payload so must have the *content_format* option set in the response. Finally, the DELETE method is used to remove data from a resource such as a database. Note that although the DELETE request will not contain any payload data it still requires the *content_format* option to be set to identify the type of data that will be deleted. A DELETE response has no options or payload so consists of a header only.

In addition to the standard REST protocol there is a meta-protocol which is delivered across the router endpoint of a Zest node. The meta-protocol is a simple character separated format delivered as the payload of a Zest acknowledgement to a notification or observation request (see IV for more

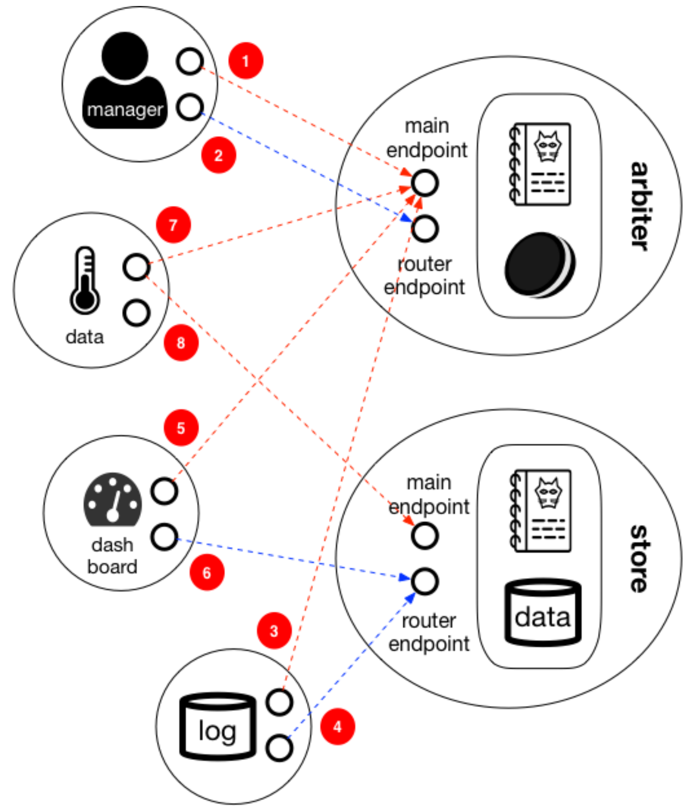


Fig. 2. Node interaction across a Zest deployment: ❶ Manager connects to Arbiter to set up permissions for minting tokens, ❷ Manager subscribes to Arbiter to receive audit information, ❸ Logger connects to Arbiter to request audit logging token for Store, ❹ Logger subscribes to Store using token to receive audit information, ❺ Dashboard connects to Arbiter to request data logging token for Store, ❻ Dashboard subscribes to Store using token to receive copy of data posted, ❼ Sensor connects to Arbiter to request token for posting data to Store, ❽ Sensor posts data to Store using token

details). Observations have 3 formats based on the request: data, audit, notify. Whereas a notification has a single format. For example, the result of observing data posted to path /kv/foo/bar might look like:

```
#timestamp #uri-path #content-format #data
1521554211213 /kv/foo/bar json {"room": "lounge", "value": 1}
```

Note that the first line contains a comment to represent the structure and is not included in the protocol. Further details and examples of the different formats are available online.[8]

## IV. ARCHITECTURE

Figure 2 depicts a simple Zest deployment where a number of client nodes interact with a server node (store) which has data storage and retrieval functionality. For example, ZestDB[9] is a storage node which provides both key/value and time-series functionality. Access to this node is controlled through tokens which need to be obtained from a privileged server node known as the arbiter. To bootstrap a Zest deployment requires a single privileged node depicted in Figure 2 as the manager

[8]https://github.com/me-box/zestdb/tree/master/docs#observation
[9]https://me-box.github.io/zestdb/

| | GET REQ | GET RESPONSE | POST REQ | POST RESP | DELETE REQ | DELETE RESP |
|---|---|---|---|---|---|---|
| *uri_path* | x | | x | | x | |
| *uri_host* | x | | x | | x | |
| *content_format* | x | x | x | + | x | |
| *observe* | + | | | | | |
| *max_age* | + | | | | | |
| *public_key* | | + | | | | |

TABLE IV
OPTIONS PER REQUEST OR RESPONSE: X IS MANDATORY, + IS OPTIONAL

node. The manager node is required to grant permissions on the arbiter node. This sets up which tokens can be minted and which nodes can request them. For example, at step ❸ the log node connects to the arbiter to request a token which it will later present to the store. The architecture is flexible as it can be connected up in different ways. The sensor node in this deployment is not receiving any logging data but it could request this from either the arbiter or store or both provided it had the required access token. Access tokens are implemented as Macaroons [4] which are similar to traditional bearer tokens but allow restrictions (caveats) to be added for delegation. A Zest deployment must support at least 3 caveats: the REST method required, the resource path to access and the target identity of the node which will receive the token. Each server node in a Zest deployment must also provide a HyperCat to describe its state. For example, a store will describe what data sources it currently contains, whereas an arbiter will describe which nodes have been granted permissions to request tokens.

*Observations and Notifications*

Observations are used for logging/auditing data flowing through nodes, whereas notifications are used to support asynchronous communication between nodes. A Zest node provides two endpoints. The main endpoint uses a request/reply protocol to support the RESTful interface and is used to set up communication with the router endpoint, The router endpoint supports a router/dealer topology in ZeroMQ which is responsible for pushing data back to clients. A router/dealer topology in ZeroMQ differs from a traditional publisher/subscriber topology in that it maintains an internal routing table to ensure data is only routed to a single client connection rather than broadcast to multiple subscribers. Figure 3 summarises the sequence involved to set up an observation or notification. During an observation GET request the key of the server (e.g. store) is returned to the client node to allow secure communication to take place. In addition, this exchange also provides the client node with its unique identity in the form of a UUID which it is required to present when connecting to the router endpoint. A notification GET request follows a similar sequence, however, instead of obtaining a UUID from the server it generates its own identity in the form of a callback path for notification responses. Notifications are used in conjunction with observations and support communication between two nodes interacting with a Zest store through a */notification/request* and */notification/response* path. Data is communicated through a store using a client/server paradigm

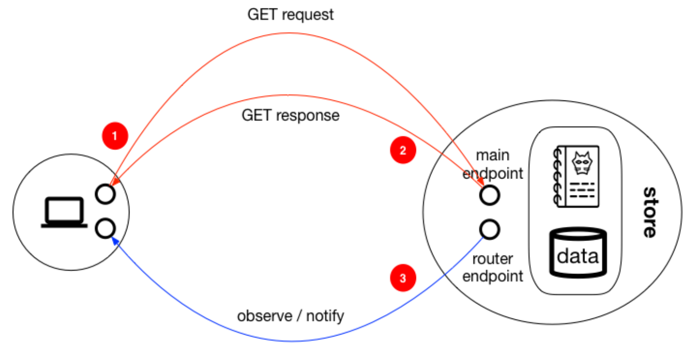


Fig. 3. Sequence to set up an observation or notification.

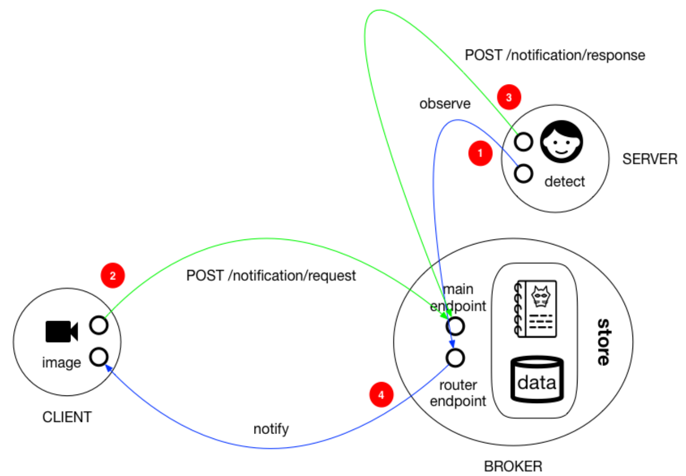


Fig. 4. Asynchronous communication with a client node talking to server node through a store acting as a broker.

with the store acting as a broker between the client and server nodes. A client node can issue a request to a server node which can obtain the necessary information from an observation to respond back asynchronously with the result. Within Databox this allows an application developer to build an app or driver which uses the services of an existing app. For example, there may be an existing image processing app which specialises in offering face detection algorithms and this app has made this service available within the Databox system. Figure 4 illustrates how a client capturing images is able to send them to a server node for facial recognition and receive back the results using the notification system. The intermediary node (store) acts as a broker between client and server nodes to

facilitate the communication. This takes place over predefined paths on the broker which means both client and server can be controlled through access tokens and have no direct way of communicating with one another. The sequence of steps involved could be summarised by the following interaction:

1) Server observes requests on broker path
   */notification/request/image_capture/**
2) Client POSTs image to broker path
   */notification/request/image_capture/001*
3) Server carries out image processing and POSTs result to broker path
   */notification/response/image_capture/001*
4) Client receives notification containing processed image through a previously established GET request to path
   */notification/response/image_capture/001*

Note, that steps 1 and 4 translate into the sequence previously described in figure 3. In this example, the server node is using a wildcard to accept requests from any path with the */image_capture/* prefix and therefore will accept the client request of */image_capture/001*. The Zest protocol does not enforce a particular naming scheme, so a client is responsible for generating a unique request path to ensure that a unique callback is generated, for example, by adding a UUID into the structure.

## V. CONCLUSION

In this paper we presented, Zest, a middleware solution used in the Databox project. A key design goal of Zest was to abstract complexities that exist implementing the CoAP protocol by building on top of ZeroMQ to support both a traditional REST interface together with additional high-level features such as brokering asynchronous communication between nodes. Access to all resources takes place across paths which can be controlled through a Macaroon minting process. This level of access control is valuable to IoT deployments such as Databox. The work presented in this paper is open-source and available for download at https://github.com/me-box/zestdb under an MIT license.

## VI. ACKNOWLEDGEMENTS

Work funded by EPSRC grants EP/N028260/1, EP/M001636/1 and EP/M02315X/1. Andrés Arcia-Moret is collaborating with Databox project under the support of Grant RG90413 NRAG/527.

## REFERENCES

[1] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, March 2012.

[2] M. Kovatsch, "CoAP for the Web of Things: From Tiny Resource-constrained Devices to the Web Browser," in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, ser. UbiComp '13 Adjunct. New York, NY, USA: ACM, 2013, pp. 1495–1504. [Online]. Available: http://doi.acm.org/10.1145/2494091.2497583

[3] R. Mortier, J. Zhao, J. Crowcroft, L. Wang, Q. Li, H. Haddadi, Y. Amar, A. Crabtree, J. Colley, T. Lodge, T. Brown, D. McAuley, and C. Greenhalgh, "Personal data management with the Databox: What's inside the box?" in *Proc. Cloud Assisted Networking workshop at ACM CoNEXT*, Dec. 12 2016.

[4] A. Birgisson, J. G. Politz, lfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud," in *Network and Distributed System Security Symposium*, 2014.

[5] T. Gazagnaire, A. Chaudhry, A. Madhavapeddy, R. Mortier, D. S. adn D. Sheets, G. Tsipenyuk, and J. Crowcroft, "Irmin: a branch-consistent distributed library database," in *OCaml User and Developer Workshop*, 2014.

[6] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor, "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets," Internet Requests for Comments, RFC Editor, RFC 8323, February 2018.

[7] C. Gomez, A. Arcia-Moret, and J. Crowcroft, "TCP in the Internet of Things: From Ostracism to Prominence," *IEEE Internet Computing*, vol. 22, no. 1, pp. 29–41, Jan 2018.

[8] International Telecommunication Union, "Specification of Abstract Syntax Notation One (ASN.1)," ITU-T Recommendation X.208, 1988.

[9] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," Internet Requests for Comments, RFC Editor, RFC 7540, May 2015, http://www.rfc-editor.org/rfc/rfc7540.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7540.txt