JxActinium: a runtime manager for secure REST-ful CoAP applications working over JXTA

Filippo Battaglia
Department of Electric, Electronic and Computer Engineering
University of Catania
filippo.battaglia@dieei.unict.it

Giancarlo Iannizzotto
Department of Cognitive Science, Education and Cultural Studies
University of Messina
ianni@unime.it

Lucia Lo Bello
Department of Electric, Electronic and Computer Engineering
University of Catania
lobello@unict.it

ABSTRACT

In the Representational State Transfer (REST) paradigm, which is commonly adopted in the IoT field, all the operations are performed on *resources* addressed by a URI. In order to simplify the deployment of REST-ful architectures, the *Actinium* runtime container, which allows remote clients to load, unload and run REST-ful applications written in Javascript (JS), was recently proposed. However, Actinium is based on the Constrained Application Protocol (CoAP) over the User Datagram Protocol (UDP) and inherits the limitations of both of them. For example, on hybrid networks made up of multiple subnets based on different transport protocols, CoAP requires proprietary solutions, such as *http-to-coap* gateways or *border-routers* to deliver packets across the different subnetworks. Moreover, CoAP lacks of support for secure group communications. This paper proposes JxActinium, a novel implementation of CoAP devised to work over the JXTA P2P protocol proposed by Sun Microsystem. JxActinium not only supports hybrid networks, but also allows to create nested networks in which some JS services are reserved only to the peers belonging to a given secure peergroup.

CCS Concepts

•Networks → Peer-to-peer protocols; •Computer systems organization → *Cloud computing;*

Keywords

REST, CoAP, Javascript, JXTA, Cloud computing

1. INTRODUCTION AND MOTIVATION

Internet of Things (IoT) is a paradigm referring to a scenario where all computers, palmtops, domestic appliances, sensors and actuators are interconnected through several physical technologies, as Ethernet, Powerline or wireless links (where the standard Wi-Fi/IEEE 802.11 is commonly used for computers and ZigBee/IEEE 802.15.4 for sensors).

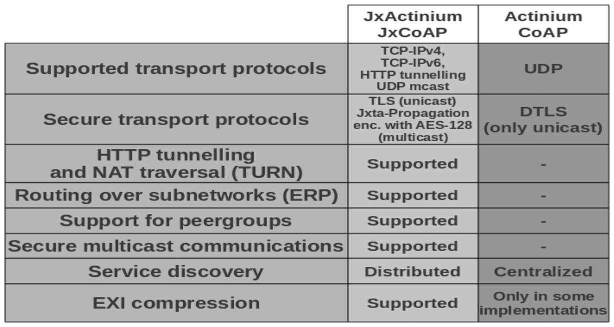
As the IoT paradigm can include the support for a great number of devices providing quite various functionalities, it would be advisable to find a standard approach for their interconnection and communication at the application level. One of the first solutions proposed is the adoption of Web services based on SOAP and XML, a paradigm that is widely exploited in industrial applications [3][18][19]. As an alternative, the use of the Representational State Transfer (REST) [10] paradigm, where the functionalities and the informations provided by devices and sensors are seen as *resources* that are addressed through an URI (Uniform Resource Identifier), was recently proposed. All the operations on these *loosely-coupled* architectures are performed through four HTTP methods (GET, POST, PUT and DELETE) referred to a definite resource, thus allowing the sensor interrogation through a client library or a browser. However, HTTP can be too heavy for the resource-limited devices commonly used in wireless sensor networks (WSN). As a result, a new solution, named CoAP (*Constrained Application Protocol*)[5][28], that offers compactness and low computational requirements, was recently proposed by the IETF CoRE Working Group. CoAP was designed to work over UDP, which is an unreliable transport protocol. As a result, for the transmission of large payloads CoAP implements its own retransmission mechanism at the application level, which foresees that messages can be set as *Confirmable* and retransmitted until the reception of the recipient's acknowledgement. Although it was initially designed for the embedded devices typically found in narrowband WSNs (IEEE 802.15.4), CoAP provides some features, such as compactness, low memory requirements and the support for resource observation, that can be useful also in the world of personal computers. As a consequence, several implementations were developed, also for the Java environment, such as jCOAP [15] and Californium [17].

Furthermore, a runtime-container, named *Actinium*[16], was recently proposed, that manages operations as loading or unloading of Javascript (JS) applications provided by remote nodes through a REST-ful API. The applications deployed by Actinium communicate with the clients using a binding module for JavaScript of the Californium library.

A limitation of CoAP is that it was designed for *homogeneous* networks in which every peer can directly send UDP packets to the other ones. A typical scenario consists of a wireless sensor network where the nodes communicate

SAC'16, April 4-8, 2016, Pisa, Italy

http://dx.doi.org/10.1145/2851613.2851808



	JxActinium JxCoAP	Actinium CoAP
Supported transport protocols	TCP-IPv4, TCP-IPv6, HTTP tunnelling UDP mcast	UDP
Secure transport protocols	TLS (unicast) Jxta-Propagation enc. with AES-128 (multicast)	DTLS (only unicast)
HTTP tunnelling and NAT traversal (TURN)	Supported	-
Routing over subnetworks (ERP)	Supported	-
Support for peergroups	Supported	-
Secure multicast communications	Supported	-
Service discovery	Distributed	Centralized
EXI compression	Supported	Only in some implementations

Figure 1: The features of JxActinium/JxCoAP and of the standard versions for UDP in comparison.

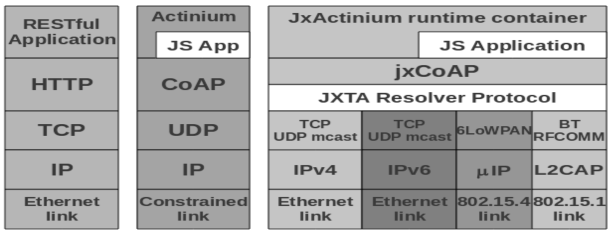


Figure 2: A comparison between HTTP/REST, CoAP and jxCoAP protocol stacks.

among themselves through 6LoWPAN [20]. In this case, a gateway between the WSN and the external Internet is mandatory. CoAP designers advice to exploit a *http-to-coap* gateway for this purpose, but this adds complexity to the network and may create issues (for example, with secure requests, see [7], Sec. 10.2).

Finally, if the network is made up of different subnets, connected through several link-layer protocols and bounded by firewalls, gateways or NAT that block the packet delivery, the solutions based on CoAP cannot work. Moreover, CoAP exploits a security scheme that is implemented at the transport level through DTLS (*Datagram Transport Layer Security*)[11], but this protocol secures only end-to-end connections, with no support for *multicast communications*[27].

The main contribution of this paper consists of a new implementation of the Actinium runtime container, named *jxActinium*, aimed to create a REST-ful heterogeneous service network where all the nodes can provide JS services that can be loaded, unloaded, executed and interrogated by the remote clients in a secure environment. In order to achieve this result, JxActinium exploits a new implementation of CoAP, named *jxCoAP*, that exploits a modified version of JXTA, a P2P protocol originally released by Sun Microsystem, as a replacement of UDP. The integration of CoAP and JXTA below the jxActinium software layer ensures several advantages (see Fig. 1):

- *jxActinium is agnostic about the underlying transport protocol.* JXTA can use several transport protocols (such as TCP-IPv4, TCP-IPv6, HTTP tunnelling) and is able to hide the transport protocol really used to the CoAP level. jxActinium can be easily extended to work over Bluetooth RFCOMM or ZigBee even without IP emulation, by adding the support for these protocols to JXTA (see Fig. 2).

- *jxCoAP can support hybrid networks, thus making transparent for the application layer the presence of a gateway or even of a chain of gateways at the transport or network levels.* JXTA supports the *Endpoint Routing Protocol* (ERP) in order to provide connections between peers belonging to subnetworks based on different communication technologies. Operations such as *address conversion* or *hop-by-hop delivery* are performed automatically by the JXTA level. Through ERP, each JXTA peer can automatically promote itself to the role of gateway when it is needed. No effort for gateway/proxy deployment and configuration is required to the developers of the applications. As a consequence, the computers belonging to an Ethernet-based subnet and the embedded devices belonging to a ZigBee-based WSN can communicate through jxCoAP requests without the need for an *http-to-coap gateway* working at the application level.

- *Improved group security.* The proposed architecture allows to define groups of privileged members sharing some services. The groups can be nested, thus creating a hierarchical deployment of peers. Moreover, within a group both unicast and multicast communications can be secured by encryption.

- *Support for multicast requests within the hybrid network.* Multicast requests are supported by CoAP [26], but they leverage on multicast UDP packets that can be blocked by a gateway that is outside the local network. Conversely, JXTA exploits a *propagation model* that can deliver a message to all the peers of the group even when no support for multicast packets is available at the transport level.

- *Support for HTTP tunneling and NAT traversal.* By leveraging on JXTA, jxCoAP supports HTTP tunnelling [9] and TURN (*Traversal Under Relay NAT*)[25] technologies, thus allowing communications through firewalls and NATs.

- *Distributed service discovery.* Each JxActinium client is able to discover the services provided by the other peers in the group by leveraging on the JXTA Discovery Protocol, thus achieving a fully scalable, distributed and fault-tolerant system for resource lookup. Conversely, the standard specifications of CoAP for UDP provide for a Resource Discovery server (RD) [32], which may be a point of failure and a bottleneck when a high number of clients joins the network at the same time.

This paper assumes that the links used in the heterogeneous network are based on wired or at least broadband wireless technologies as Wi-Fi. A typical scenario is, for instance, a smart home in which some computers communicate through an Ethernet router. Some of them are connected to domestic appliances via IP over HomePlug Powerline, while others are connected to some sensors via Wi-Fi. This group of devices could be, in turn, part of a larger secure peergroup encompassing all the smart homes in the building. The paper is organized as follows. Section II describes the JXTA features, while Section III deals with related work. Section IV presents the JxCoAP and JxActinium features, while Section V discusses the experimental results. Finally, Section VI concludes the paper and gives hints for future work.

2. JXTA OVERVIEW

Project JXTA [29] is a peer-to-peer (P2P) protocol developed by Sun Microsystem since 2001 which can be used in order to create a heterogeneous network that is seen by the application level as a uniform network. The current version of JXTA (2.7) is made up of six protocols (only the first two of them are mandatory) [1]: *Endpoint router protocol* (ERP), *Peer resolver protocol* (PRP), *Peer information protocol* (PIP), *Rendezvous protocol* (RP), *Pipe binding protocol* (PBP) and *Peer discovery protocol* (PDP).

JXTA supports NAT traversal and HTTP tunnelling, thus allowing the communication even through firewalls. Each node (named *peer*) is addressed by its name or by a 128-bit PeerID regardless of the technology used by the link-layer. JXTA supports a routing protocol that delivers messages through peers belonging to different subnets. The peers that are on the border between two subnets promote automatically themselves to the role of gateways. Moreover, JXTA allows to create *peergroups*, i.e., groups of privileged peers that can access a subset of the network resources. A further advantage is that the peergroups can be nested, thus allowing the creation of a hierarchical system of resource access. Each resource of the distributed network is provided through XML documents named *advertisements* (adv). The communications inside a peergroup can be secured in order to avoid the access by unauthorized peers. JXTA supports unicast connections through TLS and can be modified in order to support security in multicast propagation.

JXTA was devised so as to work regardless of the underlying transport protocol. It currently supports *Message Transport Bindings* (MTB) (also known as *Messengers*) for TCP-IPV4, TCP-IPV6, UDP (for multicast operations) and HTTP traversal. If in the future developers need to add the support for a new transport protocol (e.g., for ZigBee), they shall only write a new MTB able to manage such a protocol. JXTA makes easier the development of applications as they will continue to work without changes also when the support for new transport protocols is added.

JXTA provides 4 working modes for the peers: *adhoc*, *edge*, *rendezvous* (rdv), and *relay*. The *edge* mode is devised for reduced-resource devices, but requires an active connection to a rdv peer before starting. The *rendezvous peers* mantains in memory a register that stores all the informations available for the whole group [30]. The *adhoc* mode is suitable for tiny, *resource constrained* devices.

JXTA communications are based on the exchange of *Messages*. Each message is made up of some *namespaces*, which contain some *MessageElements*. Each MessageElement is characterized by a *name*, a *MIME type* and a *content*. Messages are transmitted between peers by the Resolver protocol (PRP), however this mechanism is *unreliable* and *connectionless*. Asynchronous and synchronous reliable communications are supported by JXTA *pipes* and *sockets*, but they are quite inefficient in terms of bandwidth utilisation [4].

The JXTA specifications 2.0 [1] provide for the concept of *JXTA service*, but they do not define a standard for service interaction. The document only states that a JXTA service is associated to a *module specification adv* (MSA) and to a *module implementation adv* (MIA), containing the name, the description, and the ID of an optional pipe that can be used to contact the service. As a consequence, the approach proposed in this work can be advantageous also for JXTA developers, as it uses the CoAP RESTful paradigm as an interaction model for accessing the functionalities of a JXTA service, thus making very easier the development of the source code providing the related functionalites.

3. RELATED WORKS

In [14] the use of XMPP (*Extensible Messaging and Presence Protocol*) was proposed in order to merge the world of personal computers and the world of sensors without the need for a gateway, by leveraging on the XMPP feature to support several transfer protocols via extension modules. Unfortunately, XMPP is unefficient for binary transfers (the contents are Base64-encoded) and therefore is unsuitable for narrowband networks. Conversely, JXTA supports *ByteArray MessageElements* for full-efficient binary transfer.

There are very few works testing CoAP over a protocol different than UDP or DTLS. In [22][21] the authors propose a bridging protocol aimed to support both uPNP (*Universal Plug and Play*) and ZigBee devices at the gateway level. The paper deals with a version of CoAP that was deployed over the ZigBee MAC layer and extended with new *method codes* in order to support uPNP. This solution still requires a *http-to-coap* gateway in order to allow communications towards the outside Internet. In [8] an architecture based on CoAP and on the Kademlia P2P network was proposed but it was only used to create a distributed registry for service discovery. In fact, CoAP still worked over UDP/IP, with all the limitations about gateways previously described.

In order to overcome the lack of support for *secure multicast communications* from DTLS, several solutions were proposed. For instance, implementing CoAP over multicast-IPSec [12] or sending to all the group members via DTLS some information aimed to generate a *Traffic Encryption Key* (TEK) to be used to decrypt the multicast packets [13]. Unfotunately, these methods can work only over IP and do not support the concept of *group hierarchy*. As a result, it is not possible to create nested groups so as to restrict the access to a child group only to the peers that are already members of the related parent group.

4. JXCOAP/JXACTINIUM

In this work in order to implement CoAP over JXTA the Californium library for Java [17] was modified. Californium exploits some modules, named *Connectors*, to interface with the supported transport protocols and currently includes a legacy module for UDP and a module named *Scandium* for DTLS. The first step of the work was therefore to create a JXTA Connector to send CoAP messages through the JXTA Peer Resolver Protocol (PRP).

The choice to develop JxCoAP over PRP was motivated by the lower bandwidth occupation compared to JXTA pipes and sockets. The potential unreliability of the PRP protocol is not an issue, because it can be overcome by the retransmission mechanism integrated in CoAP. Moreover, the unreliability of the PRP protocol means that message delivery is not guaranteed if the underlying protocols of the transport chain used by JXTA are not reliable. As a consequence, if jxCoAP works over TCP *Messengers* only, the channel is actually reliable and the CoAP retransmission mechanism is not needed and will not be used.

When the jxActinium server starts, it creates and binds a local instance of a *jxCoAP server* and waits for requests by other peers of the same group. The new jxCoAP server instance, in turn, initializes a new *JXTA low level service* (jxllservice), thus publishing its MSA advertisement in the peergroup (the following examples will assume that the JXTA name of the jxllservice is *jxSvcName*).

A new *listener* is registered in the Resolver Protocol, aimed to manage all *JXTA low-level messages* (jxllmessages) received by the peer. Next, the jxCoAP server is bound to the jxllservice. In the proposed architecture, a single node can run more jxActinium/jxCoAP server instances at the same time, by binding different jxllservices in different peergroups. In this way, a single physical server can support different subsets of peers, each one featuring different access rights to the resources.

The original Californium source code uses the *InetSocketAddress* Java objects as it supports only UDP, therefore it was replaced by the new *AbstractSocketAddress* object containing a *Type* field (0 for UDP, 1 for JXTA), the name of the *JXTA Service*, the *128-bit PeerID* of the target peer, and a 32-bit *SessionID*. This choice allows to support nodes using UDP or JXTA at the same time.

In the peripheral nodes, a new module named *jxCoAP client* provides access to the jxActinium/jxCoAP server. The operation consists of the following steps:

1. **Discovery of the MSA advertisement associated to the low-level JXTA service.** This can be done by every peer that has joined the peergroup through the Discovery Protocol. If the JXTA messages related to a peergroup are encrypted, only the nodes that legally joined the group will be able to access the functionalities provided by the jxCoAP server [31].

2. **Client initialization and binding.** As the communication between the jxCoAP server and the client is *stateless*, there is no real *connect()* operation between them. However, an initialization of the client is required anyway. During this step, a 32-bit SessionID is assigned to the current instance of the jxCoAP client object. Later the SessionID will be sent together with all messages transmitted to the server. Each remote node can run at the same time several jxCoAP clients bound to different jxllservices belonging to several peergroups. Moreover, it can even run more jxCoAP client instances bound to the same jxllservice (in such a case, the instances are named *virtual clients* and feature different SessionID values). The name of the bound jxllservice on the remote peer is registered in the local client instance.

3. **Request sending**. Each CoAP request is converted in binary format by Californium and encapsulated into the JXTA message. This consists of three *ByteArray MessageElements* containing the ID of the sender peer, the SessionID, and the CoAP request, all encoded in a binary format.

4. **Response receiving**. The jxCoAP server receives the request and notifies the bound jxActinium local server instance. The runtime container runs the target JS application on the basis of the associated jxURI, thus generating a CoAP response to be notified to the requester. The server is able to determine the node or virtual client to which the response must be sent using the tuple (PeerID, JXTA Service, SessionID) contained in the received JXTA message. The CoAP response is encoded in binary form, encapsulated in a *ByteArray MessageElement* of a new JXTA message, together with the name of the bound jxllservice and the SessionID of the target virtual client. The JXTA message is finally sent to the remote node.

The Connector for JXTA supports also *Group Communications* [26]. When a response must be sent via multicast, the *propagate()* method of the JXTA ERP Protocol is used, therefore the message is sent to all rdv peers of the group (listed in the *rdv peerview* [30]). For each of these, the message is resent to all the connected edge peers, so that no support for UDP multicast packets is needed.

4.1 The URI for jxActinium and jxCoAP

In the standard version of Actinium, all resources are addressed by an URI (*Uniform Resource Identifier*). This consists of 4 subelements: the *URI-scheme* (for example *coap://* or *coaps://*), the pair *URI-host:URI-port* (containing the address and port of the target server), the *URI-Path* (containing an identifier for the target resource) and the *URI-Query* (containing the query together with some extraparameters in the form *name=value*).

This scheme must be modified in order to support JXTA, as the pair *URI-host:URI-port* must be replaced by an element providing at least two information, i.e., the *PeerName* of the JXTA server to which the request must be sent (or its *PeerID*) and the name of the target *jxllservice*. Furthermore, the new kind of URI (named *jxURI*) must support *nested peergroups*, thus allowing to reach a node belonging to a *n-th level group* after (n-1) successful authentications (which require a *group name* and a *password* for each nested group). In the proposed architecture it is possible to use **long jxURIs** as the default solution to provide such informations. An example of long jxURI is the following:

$\texttt{jxcoap}://<\texttt{jxhost}>/\texttt{fibonacci?n}=\texttt{20}$

where $<\texttt{jxhost}>$ is:

$\texttt{PG}=\texttt{group1\&PWD}=\texttt{passw1/PG}=\texttt{group2\&PWD}=\texttt{passw2/}$

$\texttt{SVC}=\texttt{jxSvcName}$

or

$\texttt{PG}=\texttt{group1\&PWD}=\texttt{passw1/PG}=\texttt{group2\&PWD}=\texttt{passw2/}$

$\texttt{PEER}=\texttt{PeerName\&SVC}=\texttt{jxSvcName}$

This URI means that jxCoAP client will try to join the peergroup named *group1* using the password *passw1*. If the operation is successful, the client will look for the adv of the peergroup *group2* within the parent *group1* and will try to join such a group using the password *passw2*. Finally, the client will try to bind the jxllservice *jxSvcName* in *group2* and next it will GET the resource *fibonacci* passing the parameter $n = 20$. Using this kind of URI, it is possible to access resources belonging to several JXTA nested groups. The example assumes that all the groups use the *StringAuthentication* method in JXTA Membership Service [31]. In the first example, the client will use the JXTA PDP protocol to look for the MSA of the service *jxSvcName* (and thus the name and the ID of the peer server), whereas in the second one the URI is referred to a service instance provided by the specific peer named *PeerName*.

As long jxURI can be too verbose in some cases, jxCoAP supports also an alternative form, called **short jxURI**:

`jxcoap : //fibonacci?n = 20`

In this case, as the < `jxhost` > part is missing, the request will be considered as directed to the jxActinium/jxCoAP server instance that was bound to the jxCoAP client during initialization (see step 2 in Section 4). Using this URI the client tries to GET the resource *fibonacci* provided by the bound target server, thus passing the parameter *n=20*.

4.2 jxActinium

The Actinium RESTful API[16] is based on three resource trees : *install* (to modify or see the list of installed apps), *apps/appsconfigs* (to modify the configuration of installed apps) and *apps/running* (to send message to or receive messages from the running instances of an installed app).

Our implementation of jxActinium required minimal changes in the source code, that consist in replacing the CoAP version for UDP with the one for JXTA. The main difference is that the standard URIs now can be replaced by jxCoAP URIs. Moreover, as the interaction model provided by Actinium states that the apps can communicate only through REST requests, it is possible to make interoperable the nodes connected through UDP/DTLS and JXTA in a very simple way. The server can distinguish between the two types of requests by the URI scheme (*jxcoap://*, for JXTA, *coap://* for UDP or *coaps://* for DTLS). For example, when a peer of the group wishes to install on the server an app named *HelloWorld*, it sends a POST request to the jxActinium runtime container using the *long jxURI*:

`jxcoap : // < jxhost > /install?HelloWorld`

where <*jxhost*> is:

`PG = grp1&PWD = passw/PEER = pName&SVC = jxSvcName`

In alternative, if the jxCoAP client was previously bound to a reference server, the request can be sent to the following *short jxURI*: :

`jxcoap : //install?HelloWorld`

The request payload contains the source code of the JS app to install:

```
app.root.onget = function(request){
request.respond(2.05,"HelloWorld"); }
```

The server responds with a message confirming the installation. Now the client can start one or more instances of the app by sending a CoAP POST requests with the jxURI:

`jxcoap : // < jxhost > /install/HelloWorld?HelloW1`

or:

`jxcoap : //install/HelloWorld?HelloW1`

containing the name *HelloW1* of the new app instance.

4.3 Security

The security model adopted by JxActinium assumes that all the peers that were able to successfully join a peergroup must be considered as trusted and authorized to use all the resources provided by each jxCoAP server that is available in the group. However, as several jxCoAP servers can be instantiated on the same machine at the same time, it is possible to create several groups, each reserved to a subset of the peers and each managed by a single server instance, thus realizing the correct hierarchy in the access rights.

The real issue in this model is to ensure *secure group communications*, i.e., to avoid that unauthorized peers access the messages belonging to a group. This property can be achieved in a very simple way by leveraging on the features provided by JXTA 2.7. However, some changes in JXTA are anyway necessary, in order to add an efficient method for managing private and public keys related to each peer belonging to a group. This is because the specifications for JXTA 2.7 [31] do not define any Certification Authority aimed to store the public certificates of the group members.

In the networks based on JXTA, the creation of a new *child peergroup* is performed by an rdv peer (named *group owner*), which runs the method *newGroup()* and publishes the related peergroup adv (PGAdv) within the parent group. First, JXTA source code was modified so that the *child PGAdv* includes the PeerID and the X.509 certificate (public key) of the group owner.

Second, the methods *toWireExternal()* and *fromWireExternal()* were modified in order to add the support for an AES-128 encypher, thus encoding the output/input messages with a *Group Traffic Encryption Key* (GTEK) (this has the same role of the *Traffic Encryption Key* proposed in [13]). Therefore, only the peers that know the GTEK related to a group can decode the related messages and can perform successfully the join operation. This choice allows to secure the Jx-Actinium apps. In fact, as the runtime container is bound to a jxCoAP server, only the peers that successfully joined the same group can read and exploit the resources available in the group. About the management of the keys, two modes are available, i.e., *Preshared key* and *Certificate-based*. In the first one, the GTEK is known "a-priori" by the authorized nodes. This is suitable to preserve *data confidentiality* if the peer identity is secure.

The *Certificate-based* mode exploits the new version of PGAdv. When a new member tries to join a child group, it reads the PeerID and the X.509 Certificate of the group owner (from the PGAdv found in the parent group) and sends its own X.509 Certificate to the group owner, which in turn verifies if the X.509 cert of the new potential member is contained in its own *jxta Keystore* (which contains the certificates of all the members authorized to join the group). If the check is successful, both the group owner and the new member know their respective public keys and can establish a TLS (*Transport Layer Security*) connection that will be used to send to the new member the GTEK of the child group. Finally, the new member uses the received GTEK key to complete the join operation.

5. EXPERIMENTAL RESULTS

As JXTA is a protocol much heavier than UDP, it adds more overhead to the CoAP messages. Hence, the first issue is to determine if the larger overhead in bandwidth utilization and the longer latencies in resource retrieving remain still acceptable for the purposes of an IoT architecture aimed at applications such as home automation or sensor data gathering. As JXTA exploits several MessageElements that are encoded as XML documents, another point to clarify is the effectiveness of an XML-compressor to reduce the network overhead. For this reason, in this work JXTA 2.7 was modified to use *OpenEXI* [2], an open source implementation of the EXI+GZIP compression technology [24].

It is important to clarify that the aim here is not to demonstrate that the performance of jxCoAP is better than the one of UDP CoAP in the same conditions, as it is foreseeable that the UDP CoAP requests will experience lower latency times. Conversely, this work aims to show that the gap between the two protocols can be mantained within acceptable limits, taking also into account that JXTA offers more functionalities and overcomes several limitations of UDP CoAP.

5.1 Network overhead

The first esperiment measures the additional overhead added by the UDP header and by the structures of JXTA binary format [1] to the payload data transmitted on the wire. A set of $N = 100$ CoAP requests is sent to two CoAP servers (which use JXTA and UDP, respectively) in two consecutive trials. The overall number of bytes $\Phi_{n.trx.bytes}$ that are sent to the server ports 5683 (for UDP) and 9701 (for JXTA) is measured by Wireshark [23] (in the latter case, a filter for JXTA messages is used so as to take in account only jxCoAP messages). Next, a parameter named *message overhead ratio* β is defined as:

$$\beta = \frac{\Phi_{n.trx.bytes}}{N\Phi_{payload.size}} - 1 \tag{1}$$

where $\Phi_{payload.size}$ is the average size of the payloads encapsulated in the messages. As the standard version of CoAP is designed to work over UDP, if the payload is larger than 1024 bytes, CoAP automatically splits the message in several submessages, thus enabling the *block-wise transfer mode* [6]. This determines further overhead, but it is consistent with the choice of using a protocol with size-limited datagrams and of performing retransmission and reassembly of out-of-order packets at the application level. Conversely, as jxCoAP works over TCP, the block-wise transfer feature is disabled in the relevant tests.

The def.(1) considers the additional overhead due to the message segmentation in such conditions. If no block-wise transfer is used, $\Phi_{n.trx.bytes} = N\Phi_{msg.size}$ and the def.(1) can be rewritten as:

$$\gamma = (\beta + 1) = \frac{\Phi_{n.trx.bytes}}{N\Phi_{payload.size}} = \frac{\Phi_{msg.size}}{\Phi_{payload.size}} \tag{2}$$

where the *message-on-wire-vs-payload size ratio* $\gamma = (\beta + 1)$ is introduced. Moreover, the *additional message overhead* Δm can be defined as:

$$\begin{aligned} \Delta m &= \frac{\Phi_{n.trx.bytes}}{N} - \Phi_{payload.size} = \\ \Delta m &= \frac{(\beta+1)N\Phi_{payload.size}}{N} - \Phi_{payload.size} = \\ &\beta\Phi_{payload.size} \end{aligned} \tag{3}$$

If $\gamma < 1$ ($\beta < 0$) the average size of the message on the wire is smaller that the size of the payload (in such a case the message overhead Δm can be considered negative). This is typical of the cases in which the payload size is reduced by some compression algorithm. In such a case, the parameter $\Gamma = \gamma^{-1}$ can be considered as a measure of the *compression efficiency.* If $\Gamma > 1$, the average size of the message on the wire is smaller than the size of the payload.

The experiment is run in two trials. In the first one $N = 100$ CoAP requests are sent to two CoAP servers (based on JXTA and UDP, respectively) that provide a service named *SumOfBytesInTheArray*.

Table 1: Measured values for network overhead

nBytes or XML items	10	100	1000	10000
SumByteArray service				
$\Phi_{payload.size}$(B)	10	100	1000	10000
$\Phi_{avg.msg.size}$ (B)				
UDP CoAP	85	175	1075	11564
jxCoAP EXI+GZIP	841	931	1831	10831
jxCoAP without EXI	1031	1121	2021	11021
Msg-on-the-wire vs payload size ratio γ				
UDP CoAP	8.500	1.750	1.075	**1.156**
jxCoAP EXI+GZIP	84.100	9.310	1.831	**1.083**
jxCoAP without EXI	103.100	11.210	2.021	**1.102**
SumValuesInXMLDoc service				
$\Phi_{payload.size}$(B)	409	3795	39457	414097
$\Phi_{avg.msg.size}$ (B)				
UDP CoAP	483	4411	45516	478760
jxCoAP EXI+GZIP	1049	2391	13177	111845
jxCoAP without EXI	1408	4765	40528	416193
Msg-on-the-wire vs payload size ratio γ				
UDP CoAP	1.181	1.162	**1.154**	**1.156**
jxCoAP EXI+GZIP	2.564	**0.630**	**0.334**	**0.270**
jxCoAP without EXI	3.443	1.255	**1.027**	**1.005**

Each POST request contains a payload consisting of B=10, 100, 1000 or 10000 bytes that are randomly chosen. The servers compute the 32-bit sum of the values contained in the payload and send the result in the CoAP response.

The second trial consists of sending $N = 100$ POST-requests to a second service named *SumOfValuesInXMLDoc*. Each request includes a payload consisting of a XML document containing $B = 10, 100, 1000, 10000$ strings which represent randomly determined double precision float numbers. Each string is encapsulated in an XML element with an opening and closing tag. The servers reply to the requests by sending the sum of the values contained in the XML document. All trials are performed using the Californium library for UDP and the modified version for JXTA.

The Table 1 shows the values measured for the *message-vs-payload size ratio* γ. In the first trial, the byte array payload cannot be compressed, therefore EXI operates only on the JXTA MessageElements dedicated to the internal functions provided by the protocol. For small payloads, the γ values for UDP CoAP are better (smaller) than the same for Jx-CoAP. However, when $\Phi_{payload.size}$ increases, the difference becomes smaller and for ($\Phi_{payload.size} > 1000$) the γ value for UDP CoAP is even higher than one measured for Jx-CoAP without EXI (respectively 1.156 and 1.102). This is a consequence of the inefficiency of UDP CoAP in bandwidth utilisation when the *block-wise transfer mode* is enabled (i.e., with large payloads).

In the second trial, EXI operates both on the XML payload and on JXTA MessageElements. In such a case, Jx-CoAP+EXI can considerably outperform the standard version of UDP CoAP, even achieving a *compression efficiency* $\Gamma > 1$. Moreover, even disabling EXI, JxCoAP outperforms UDP CoAP when $\Phi_{payload.size} \geq 1000$, thus confirming the results of the first trial.

5.2 RTT measures

The second experiment compares the RTT (*Round Trip Time*) of the Actinium services over UDP, JXTA and JXTA secure peergroup. In the last case, a group secured by an AES-128 cryptographic scheme was deployed. The three implementations of the server are started and then four services (imple-

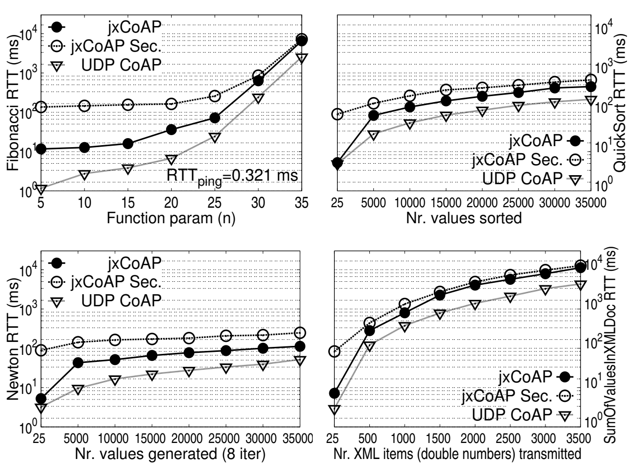


Figure 3: The average Round Trip Times (RTT) measured for some benchmarks.

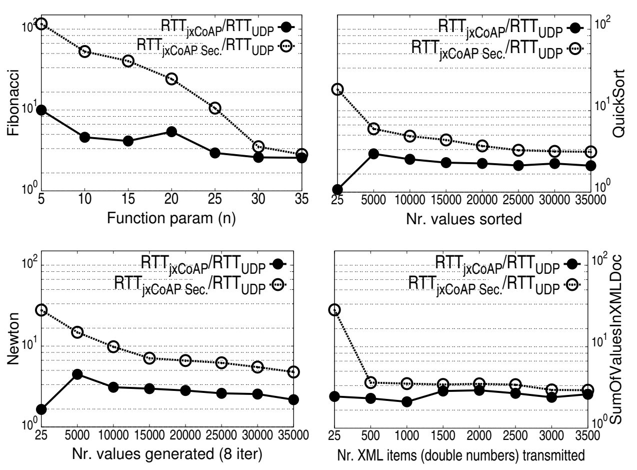


Figure 4: The RTT performance factors measured for some benchmarks.

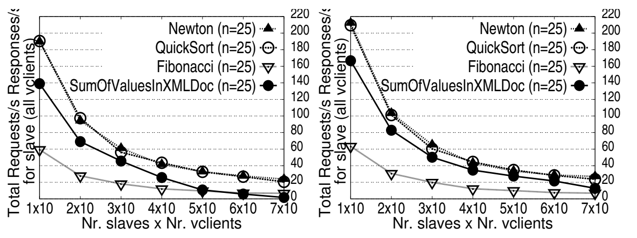


Figure 5: Scalability rates for JxActinium server. (a) EXI Enabled; (b) EXI Disabled.

mented in Javascript) are loaded by the REST-ful API: *Fibonacci*, *Quicksort*, *Newton SquareRoot* (these benchmarks are the same used in [16]) and *SumOfValuesInXMLDoc*. The first three benchmarks receive an integer n as input, contained in the payload of a POST request. The first service returns to the client the n-th number of the Fibonacci sequence. In the second benchmark, the server randomly generates n double precision float numbers and sorts them using the Quicksort algorithm; finally, a confirmation message is sent to the requesting client. In the third benchmark, the server internally generates n integer numbers and calculates iteratively (8 times) their square roots; once completed, the confirmation message is sent. The fourth benchmark, *SumOfValuesInXMLDoc*, is run using a XML document containing n double-precision float numbers.

The complexity classes of the services are respectively $O(n^2)$, $O(n \cdot log(n))$, $O(n)$ and $O(n)$. All tests were performed using PCs equipped with CPU Phenom II X6 3.0 Ghz. The server and the clients worked in single-core mode running the OS Linux KUbuntu 14.10 and the JVM v.1.7.0-71 released by Oracle. In JXTA benchmarks, the server was configured as *rendezvous* and the client was configured as *edge*. Moreover,the EXI compressor was enabled both for payload and for JXTA MessageElements. Fig. 3 shows the RTT measured for the 4 benchmarks. The same measures are reported in Fig. 4, in terms of *performance factor* (PF), where $PF_{jxcoap} = RTT_{jxcoap}/RTT_{udp.coap}$. The RTT values measured for the Actinium JS services over JXTA are higher than the ones measured over UDP. However, in several tests the RTT achieved by JxActinium is no more than 4 times the latency measured for UDP Actinium in the same conditions. Consequently, the proposed solution appears to be suitable for all tasks involving large payload transfers (as content retrieval or multisensor control through XML documents) or when low latencies are not needed (smart metering, home automation, IoT data gathering). The additional latency can be considered as an acceptable price if the support of hybrid networks is advantageous or mandatory for the considered scenario.

5.3 Scalability measures

The last experiment measures the scalability of the JxActinium server when multiple concurrent requests are issued by several *virtual clients*. A group made up of a rdv *master server* and $q = s * v \in [10..70]$ *virtual clients* (vclients) running over $s = [1..7]$ *slave* personal computers is deployed (each slave is configured as *edge* and runs 10 virtual clients). All systems are connected through wired Ethernet. The master server provides the 4 services *Fibonacci*, *Quicksort*, *Newton SquareRoot* and *SumOfValuesInXMLDoc*. Each virtual client (running in a dedicated thread) sends some jxCoAP requests for $T = 180s$ to the master server (using the input parameter $n = 25$) and then waits for the response before repeating the operation. Fig. 5 shows the response frequency measured for each slave (all tests are repeated enabling and disabling the EXI compression). The *Newton* and *Quicksort* benchmarks achieve the best performance when multiple concurrent requests are submitted to the server, whereas *Fibonacci* achieve the worst score (this is probably a consequence of the Actinium JS interpreter slowness in running recursive operations, a result already reported in [16]). Moreover, enabling EXI determines a slightly lower rate. Despite this, with or without EXI the JxActinium server provides a response rate higher than 50 resp/s using up to 30 virtual clients, for the *Quicksort*, *Newton* and *SumOfValuesInXMLDoc* services.

6. CONCLUSIONS

This paper proposed *jxActinium*, a REST-ful runtime container allowing the loading, unloading and the execution by remote clients of service applications written in Javascript.

JxActinium exploits a new version of CoAP, named *jxCoAP*, devised to work over JXTA overlay network. By leveraging on JXTA features, JxActinium and JxCoAP take advantage of the support for hybrid networks, HTTP tunnelling and nested peergroups, thus achieving a secure communication among IoT devices. Moreover, JxCoAP provides a secure delivery of multicast messages, thus overcoming a current limitation of the DTLS protocol.

The experimental results show that the JxCoAP message size can be mantained within acceptable limits, by using a XML compressor such as EXI. The measured latencies for JxActinium operations are often not larger than 4 times the values measured for the same operations using the standard version of Actinium for CoAP over UDP. This performance is suitable for applications as content retrieving, IoT data gathering, home automation and smart metering. Future works will address a compression protocol aimed to further improve the performance by reducing the required bandwidth and the latency, and a *http-to-jxcoap* gateway for the integration of JXTA services into the Web.

7. REFERENCES

[1] JXTA 2.0 Protocol Specifications. Available online: https://jxta.kenai.com/Specifications/JXTAProtocols2_0.pdf.

[2] OpenEXI. Online: http://openexi.sourceforge.net/.

[3] L. A. Amaral, R. T. Tiburski, et al. Cooperative middleware platform as a service for internet of things applications. In *Proc. of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 488–493, New York, NY, USA, 2015. ACM.

[4] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance Evaluation of JXTA Communication Layers. *Proc. of IEEE Int. Symposium on Cluster Computing and Grid*, 1:251–258, May 2005.

[5] C. Bormann, A. P. Castellani, et al. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, March 2012.

[6] C. Bortmann and Z. Shelby. Blockwise transfers in CoAP, 2013. draft-ietf-core-block-17.

[7] A. Castellani and S. Loreto. Guidelines for HTTP-to-COAP Mapping Implementations, July 2015. draft-ietf-core-http-mapping-07.

[8] S. Cirani, L. Davoli, et al. A scalable and self-configuring architecture for service discovery in the internet of things. *Internet of Things Journal, IEEE*, 1(5):508–521, Oct 2014.

[9] M. Crotti, M. Dusi, et al. Detecting HTTP tunnels with statistical mechanisms. In *IEEE Int. Conf. on Communications*, pages 6162–6168, June 2007.

[10] X. Feng, J. Shen, and Y. Fan. REST: An alternative to RPC for Web services architecture. *Proc. of First Int. Conf. on Future Information Networks*, Oct. 2009.

[11] J. Granjal, E. Monteiro, and J. S. Silva. Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues. *IEEE Communications Surveys and Tutorials*, 17(3):1294–1312, Jan. 2015.

[12] S. Kent. IP Encapsulating Security Payload (ESP), 1998. RFC 2406.

[13] S. Keoh, S. Kumar, et al. DTLS-based Multicast Security for Low-Power and Lossy Networks, Oct. 2012. draft-keoh-tls-multicast-security-00.

[14] M. Kirsche and R. Klauck. Unify to bridge gaps: Bringing XMPP into the Internet of Things. *Proc. of IEEE Int. Conf. on Pervasive Computing and Communications Workshops*, pages 455–458, 2012.

[15] B. Konieczek, M. Rethfeldt, and F. Golatowski. Real-Time Communication for the Internet of Things Using jCoAP. *Proc. of IEEE 18th Int. Symposium on Real-Time Distributed Computing*, Apr. 2015.

[16] M. Kovatsch, M. Lanter, and M. Duquennoy. Actinium: A RESTful runtime container for scriptable Internet of Things applications. *3rd Int. Conf. on the Internet of Things (IOT)*, pages 135–142, Oct. 2012.

[17] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable cloud services for the Internet of Things with CoAP. *Proc. of the 2014 Int. Conf. on the Internet of Things*, pages 1–6, Oct 2014.

[18] A. Lee and J. Lastra. Data aggregation at field device level for industrial ambient monitoring using web services. In *9th IEEE Int. Conf. on Industrial Informatics (INDIN), 2011*, pages 491–496, July 2011.

[19] L. Lo Bello, O. Mirabella, and N. Torrisi. Modelling and evaluating traceability systems in food manufacturing chains. *Proc. of Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 173–179, Jun 2004.

[20] X. Ma and W. Luo. The Analysis of 6LowPAN Technology. *Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, 1, Dec. 2008.

[21] J. Mitsugi and H. Hada. EPC based internet of things architecture. *Proc. of IEEE Int. Conf. of RFID Technologies and Applications*, pages 527–532, 2011.

[22] J. Mitsugi, S. Yonemura, and H. Hada. Bridging UPnP and ZigBee with CoAP: protocol and its performance evaluation. *Proc. of the Workshop on Internet of Things and Service Platforms*, 2011.

[23] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[24] D. Peintner, H. Kosch, and J. Heuer. Efficient XML Interchange for rich internet applications. *Proc. of IEEE Int. Conf. on Multimedia and Expo*, July 2009.

[25] R. Mahy, P. Matthews, J. Rosenberg, et al. RFC5766: Traversal Using Relays around NAT, Apr. 2010.

[26] A. Rahman, E. Dijk, et al. Group Communication for CoAP, March 2015. draft-ietf-core-groupcomm-25.

[27] S. H. Shaheen and M. Yousaf. Security Analysis of DTLS Structure and its Application to Secure Multicast Communication. *12th Int. Conf. on Frontiers of Information Technology*, Dec. 2014.

[28] Z. Shelby, K. Hartke, et al. The Constrained Application Protocol (CoAP), Jun 2014. RFC 7252.

[29] Sun Microsystem. Project JXTA, 2001. Available online: https://java.net/projects/jxta.

[30] B. Traversat, M. Abdelaziz, and E. Pouyoul. Project JXTA: A loosely-consistent DHT rendezvous walker. Sun Microsystem, 2001.

[31] J. Verstrynge. JXSE v2.7 The JXTA Java Standard Edition Implementation Programmer's Guide, March 2011. Online: https://jxse.kenai.com/Tutorials/.

[32] Z. Shelby, M. Koster, and C. Bormann. CoRE Resource Directory. Internet Draft., July 2015. Available online: https://datatracker.ietf.org/doc/draft-ietf-core-resource-directory/.