ORIGINAL ARTICLE

A middleware solution for integrating and exploring IoT and HPC capabilities

Leonardo de Souza Cimino[1,2] | José Estevão Eugênio de Resende[2] | Lucas Henrique Moreira Silva[2] | Samuel Queiroz Souza Rocha[2] | Matheus de Oliveira Correia[2] | Guilherme Souza Monteiro[2] | Gabriel Natã de Souza Fernandes[2] | Renan da Silva Moreira[4] | Junior Guilherme da Silva[4] | Andre Luiz Lins Aquino[3] | André Luís Barroso Almeida[1,4] | Joubert de Castro Lima[2]

[1]Departamento de Tecnologia da Informação, Instituto Federal de Minas Gerais, Congonhas, Minas Gerais, 36415-000, Brazil

[2]Instituto de Ciências Exatas e Biológicas - Departamento de Computação, Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais, 35400-000, Brazil

[3]Instituto de Computação, Universidade Federal de Alagoas, Maceió, Alagoas, 57480-000, Brazil

[4]Coordenadoria do Curso Técnico de Automação Industrial, Instituto Federal de Minas Gerais, Ouro Preto, Minas Gerais, 35400-000, Brazil

Correspondence
Leonardo de Souza Cimino, Departamento de Tecnologia da Informação, Instituto Federal de Minas Gerais, Congonhas, Minas Gerais, 36415-000, Brazil
Email: leonardo.cimino@ifmg.edu.br

Funding information

This work presents the JavaCá&Lá (JCL), a middleware used to help the implementation of sensing, actuating, processing, storage, context awareness and security services for distributed user-applications classified as IoT-HPC. This ubiquity is possible because JCL incorporates: i. a single API to program different device categories, precisely HPC, high-end, low-end, sink node, and sensor node devices; ii. the support for different programming models, specifically event-based, distributed shared memory and task-oriented models; iii. the interoperability of sensing, processing, storage and actuating services; iv. the integration with MQTT technology; and v. security, context awareness and actions services introduced through JCL API. Experimental evaluations demonstrated that JCL scales when doing the IoT-HPC services, which implies that it can manage IoT, HPC or IoT-HPC user-applications with few components. Additionally, we identify that customized JCL deployments become an alternative when Java-Android and vice-versa code conversion

is necessary. MQTT is faster than JCL HashMap sensing storage, but it is not distributed, so it cannot handle huge amount of sensing data. Based on these evaluations we show that JCL is a feasible middleware alternative to implement integrated IoT-HPC user-applications. A short example for monitoring moving objects, like pets, phones, person and so forth, exemplify JCL facilities for HPC and IoT development.

KEYWORDS
Internet of Things, High Performance Computing, Middleware

1 | INTRODUCTION

Big data [12], Internet of Things (IoT) [61] and elastic cloud services [89] are technologies that provide this new decentralized, dynamic and communication-intensive society. Based on El Baz et al. [23], the demand of an integration of IoT and High Performance Computing (HPC) exists and it will increase in the near future motivated by different applications, like smart buildings, smart cities or smart logistics. These new applications are requiring both alternatives in a single middleware solution. The integration imposes new challenges related with heterogeneity since both technologies must communicate and operate together. Other challenges related with fundamental IoT and HPC requirements, like deployment, code refactoring, performance, scheduling and fault tolerance, are also important and hard to be solved when an integration is mandatory.

Middleware are everywhere helping to reduce the complexity of application development [2]. The challenging issue is how to provide sufficient support and general high-level mechanisms using middleware for rapid development of distributed, parallel and embedded applications [15]. Existing IoT middleware do not detail how they implement distributed processing and storage, and HPC one do not detail how they provide sensing, actuating and context awareness issues. None of the middleware alternatives used to develop our solution put these services together in a single Application Programming Interface (API).

Based on these issues, the main problem stated in this work is: "*How to combine IoT capabilities, e.g., sensing, actuating, security, sensing data storage and context awareness services, with HPC ones, e.g., distributed general-purpose storage and processing, in a single middleware solution capable of running over multiple clusters, some of which have invalid IP addresses?*". To address an alternative solution for this problem, we present a middleware called JavaCá&Lá or just JCL[1], a middleware designed to combine IoT and HPC capabilities.

The device categories supported by JCL were defined by Perera et al. [60]: i. HPC devices, e.g., huge servers and workstations and cloud computing solutions; ii. High-end devices, composed of PCs and laptops; iii. Low-end computational devices, where smart-phones and tablets are included; iv. Sink node devices, i.e., devices with low or medium processing power and storage capacity (Raspberry Pi, Galileo, Cubieboard, and Onion single-board computers); v. Sensor node devices, i.e., devices with low processing power and storage capacity (Arduino and PIC microcontrolled boards). Categories i. and ii. were presented and evaluated in the first JCL contribution [3]. In the current work, we improve JCL by integrating the other three categories, allowing the implementation of IoT-HPC applications. Additionally, Perera et al. [60] define the miniaturized devices category, not considered here, composed of ultra-low

[1] JCL is available for download at http://www.javacaela.org/

processing and storage capabilities devices, like the Waspmote.

JCL API supports three programming models to control devices and sensors that can be combined to produce complex applications. The first one is the event-based programming model, following the publish-subscribe pattern [26]. Normally, external events, named contexts, start tasks, like sensing or actuating tasks, but also any general-purpose HPC tasks, so this programming model is the basis of the well-known sensing as a service [73], where IoT applications are called publishers and subscribers of common brokers. The second one is based on a distributed and shared memory address space (DSM) [65], where global variables and distributed Java maps are stored over clusters, enabling concurrent and thread-safe JCL tasks accesses. The last one is the task-oriented programming model [71], where ordinary Java classes members are invoked asynchronously and remotely, but also where context awareness, sensing, actuating and security services are supported, since they are modeled as tasks. Thus, JCL is the unique middleware solution modeling IoT services as ordinary HPC tasks started from a main code. The benefits include active sensing or actuating, and also a strategy to wait for contexts without requiring third-party IoT brokers. Precisely, a Future Java object[2] to wait for asynchronous computations is sufficient to wait for contexts to be reached, therefore very familiar for multi-thread Java development.

Based on these capabilities, JCL applications can be implemented in Android or Java, running over clusters of things composed of Java, Android and Arduino devices, including clusters with invalid IP addresses since in JCL these clusters are interconnected by super-peers [49, 5]. Additionally, the user-applications (`UAs`) can add actions to be started when a context is reached. In this case, an action can be a sensing, an actuator control or even a general-purpose computing task using, for instance, the JCL API for HPC. Finally, JCL integrates with MQTT [53] brokers to publish sensing data on them, but also to consume data from them.

The evaluations demonstrated that JCL scales when doing the IoT-HPC services, which implies that it can manage IoT, HPC and IoT-HPC `UAs` with few components. Comparative sensing evaluations demonstrated that JCL is a competitive IoT alternative in terms of battery, memory consumption and CPU. Its super-peers introduced small overheads, interconnecting peers of distinct networks, including those with invalid IP addresses. We identify that customized JCL deployments become an alternative when Java-Android and vice-versa code conversion is necessary. Usually, this conversion process generates a high overhead in general `UAs`. Based on these results JCL can be considered a feasible middleware alternative to implement integrated IoT-HPC `UAs`. A short example for monitoring moving objects, like pets, phones, children and people with special needs, exemplify JCL facilities for HPC and IoT development.

Previous JCL [3] version described how to put three decades of HPC requirements in a single API using task-oriented together with DSM programming models. In that work, it was evaluated the JCL capability to execute a task, i.e., to invoke any Java class method, with different numbers and types of arguments. Task scheduling strategy impacts, the map distributed data structure API access costs and the throughput for instantiating Java objects as global variables over a cluster were also evaluated. In this new work, we extend the evaluations by adding the IoT ones. Precisely, the sensing services and the presence of `JCL-SuperPeer` components were evaluated. Comparative analysis against Zanzito [30] was made to understand JCL strengths and drawbacks. Additionally, we evaluated JCL Android-Java and Java-Android code conversion costs, presenting alternatives when automatic code conversions times are unacceptable.

In summary, the main contributions of this work are:

- A single API to program different device categories, specifically HPC, high-end (PCs and laptops), low-end (smartphones and tablets), sink node (Raspberry Pi, Beaglebone and Galileo), and sensor node devices (Arduino);
- Support for different programming models, specifically event-based, distributed shared memory and task-oriented models;

[2] The interface Future in Java - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

- Support for Java and Android applications interoperability;
- Integration with MQTT technology;
- Security, context awareness and actions services introduced through JCL API;
- The first middleware in the literature integrating IoT and HPC technologies.

The remain of the work is organized as follows: Section 2 presents the main related work and discusses the improvements and drawbacks of these solutions to attend several IoT and HPC requirements; Section 3 details JCL architecture and how it implements IoT and HPC fundamental requirements, including some experimental evaluations; Section 4 presents general middleware evaluations; and finally in Section 5 the work is concluded and future work is listed.

2 | RELATED WORK

We describe several middleware solutions and evaluate them according to fundamental requirements. Table 1 summarizes the requirements from IoT [8, 62, 67], HPC [25, 20, 77, 3] and the ones they have in common. The sensing analytics services provided by Cloud of Things (CoT) solutions are also part of the requirements. For a better understanding, the solutions are separated into IoT, CoT and HPC middlewares.

TABLE 1 Essential IoT/HPC middleware requirements [15]

Requirements	HPC	IoT
Security and Privacy (SaP)	✓	✓
Context Awareness (CA)	–	✓
Different Programming Models (DPM)	✓	✓
Distributed Storage (DS)	✓	✓
Heterogeneity (H)	✓	✓
Simple Deploy (SD)	✓	✓
Fault Tolerance (FT)	✓	✓
General-purpose Computing (GPC)	✓	–
Low Refactoring (LR)	✓	–
Task Scheduler (TS)	✓	–
Super-peer Support (SP)	✓	–
Collaborative Development (CD)	✓	✓
Performance and Scalability (PaS)	✓	–
Sensing Analytics (SA)	–	✓

2.1 | IoT Middlewares

In terms of security and privacy, SMEPP [14] is one of the most promising solutions. SMEPP offers three different security models, including encryption and user authentication. In addition to SMEPP, there are other middleware systems with security and privacy services: X-GSN [13], Sofia2 [75] and Cayenne [50] have authentication and access control mechanisms. FamiWare [28] and VIRTUS [9] allow data encryption. Hydra [22] uses XACML (eXtensible Access Control Mark-up Language) [52] to provide policies for access control.

The context awareness support is part of many solutions and C-MOSDEN [59] is the most promising one. C-MOSDEN offers six types of activity contexts as well as localization context. Other middlewares, such as X-GSN, Cayenne and SMEPP, have context awareness services that trigger notification actions like sending an e-mail or SMS. Cayenne also allows events to trigger actuators. FamiWare uses ContextUML [72] and Hydra uses the Drools [68] platform to provide context awareness services. Sofia2 allows the creation of rules that can execute scripts in the Groovy programming language. The scripts can only perform specific actions, such as sending an e-mail, invoking a URL or managing an ontology instance in Sofia2.

Some of the evaluated IoT middlewares provide easy deployment in different ways. VIRTUS and SIXTH [54] use the Open Services Gateway Initiative (OSGi) [56] to partition the application into modules that can be installed or removed dynamically while X-GSN and C-MOSDEN have similar strategies using XML files and are the IoT alternatives with the easiest deployment. SIXTH and SMEPP provide fault-tolerance services where anomalies are not propagated to the other services, keeping the correct functioning of the application; however, it was not clearly specified how this requirement is achieved.

Heterogeneity is considered by all IoT middlewares, but at different levels. Java portability is part of all alternatives; in this way, X-GSN, using a semantic approach with the Semantic Sensor Network [18] ontology, can be considered the most promising solution. In terms of multiple language support, SMEPP is implemented in the Java and nesC programming languages. C-MOSDEN and Cayenne allow sensing data retrieval using RESTful services. Cayenne can be managed using popular browser applications or using iOS and Android applications. It can be executed on different Arduino and Raspberry models. VIRTUS and FamiWare can be executed on many operating systems, such as TinyOS, VxWorks, Symbian, Windows and Linux. Hydra was evaluated in more than 40 different devices and it adopts Web services protocols for this purpose. SIXTH also supports many communication protocols. Sofia2 uses protocols such as JSON and a RESTful API. It also has an API for many languages, such as Java, Android, C and Python, among others. Regarding sensing analytics, Cayenne offers a drag-and-drop dashboard that allows the user to filter data or generate charts. Sofia2 is integrated with Hadoop and has a module that allows the user to apply machine learning algorithms over the sensing data.

Sofia2's integration with Hadoop also allows distributed storage and uses MongoDB as a real-time database. According to the authors, the use of MongoDB provides scalability to the solution. The previously explained solutions implement several requirements, but there are many other alternatives that address specific requirements. In general, they are modeled as Wireless Sensor Networks (WSNs): i) Mate [44] only details the sensing services; ii) COSMOS [42] implements security and context awareness; iii) Garnet [85] provides a basic security service; iv) DSWare [45] implements sensing data storage according to data semantics and v) Impala [46] focuses on the user application development, attempting to avoid common mistakes.

There are also solutions focused on specific IoT protocols, like the MQTT. The Zanzito [30] solution is one promising MQTT publisher for Android that has a very simple deployment process, allowing publishing sensing data via MQTT brokers in few steps. It supports safe communication through SSL/TLS connections, the creation of alarms through text messages and it has a module to remotely manage other Zanzito devices.

The main limitations of IoT solutions in integrating IoT-HPC are as follows: i) few approaches deal with processing and storage services, where processing are normally sensing analytics and data aggregation services and ii) the contexts are normally restricted to few options and there are few notification mechanisms to users or UAs. General-purpose tasks should be notified when certain criteria are met, thereby enabling their start. In summary, contexts generally developed from any sensor with few simple steps are not an option for IoT middlewares.

These capabilities are presented in the proposed middleware. JCL's HPC features allow the execution of any registered class or compiled module across the cluster. This capability can also be used to execute any algorithm over the stored objects and data structures, such as machine learning or data mining techniques. Contexts can also trigger the execution of any registered class or compiled module. The same contexts can trigger sensing and actuating services in one JCL API call.

2.2 | CoT Middlewares

In terms of security and privacy, OpenIoT [76], Kaa [40], Xively [47] and Mango [38] provide authentication, authorization and encryption services, being the CoT middlewares with more advanced techniques for data safety. SensorCloud [48] and CloudPlugs [16] only provide encryption services. Kaa and SensorCloud provide context awareness services that allow notifications via custom alarms, such as sending e-mails or SMSs. ThingSpeak [84] allows actuating services, started in remote devices and when contexts are triggered. CloudPlugs has geo-location-based triggers that allow both mobile things management and e-mail/SMS delivery.

OpenIoT uses X-GSN as one of its modules; therefore, it adopts the same mechanism for simple deployment, being the most promising alternative in this requirement. Mango has a USB utility that can be used to configure various settings. The user edit the configuration files and plug them into a device.

In terms of heterogeneity, OpenIoT also considers the SSN ontology. Kaa can be executed in many boards, such as ESP8266 and QNX Neutrino. Kaa also has APIs for different programming languages, such as Objective-C and C++, besides supporting a vast amount of communication protocols. It also implements a RESTful API. SensorCloud has a RESTful API to allow sending data to its infrastructure. Xively, Nimbits [51], ThingSpeak and Mango support different communication protocols and data formats. CloudPlugs has various libraries for different platforms and programming languages. All CoT middlewares offer different ways for achieving heterogeneity, but Kaa is by far the most advanced alternative and it is hardware-agnostic according to its authors.

In terms of collaborative development, all evaluated CoT middlewares support sharing the sensing data, allowing different users to concurrently access the same historical sensing data source. Kaa is fault tolerant with no single point of failure and it also has geographical redundancy. Kaa enables distributed sensing data storage using NoSQL technologies. All others are able to store sensing data in the cloud, but they did not mention how the sensing data are partitioned and distributed in the cloud.

Kaa, SensorCloud and Mango mentioned that they provide scalable IoT services and, for this purpose, they integrate with different technologies, such as Cassandra [80] and Hadoop [74]. The data analytics and storage services have good performance according to the authors, but they do not clearly specify how scalability and performance are achieved.

All evaluated CoT solutions provide sensing analytics services. Kaa is the most advanced alternative and it uses Spark [82] for large-scale sensing data processing, including machine learning services. ThingSpeak provides sensing analytics with MATLAB [83], allowing different chart options, data conversions and many math analysis methods for pattern discovery. SensorCloud has a MathEngine module that allows users to deploy Python and Octave applications to analyze data. Xively integrates with Amazon Kinesis [4] and Nimbits uses Google Analytics Management [34]. OpenIoT, CloudPlugs and Mango have iterative dashboards, but it was not clearly specified which tools are used to analyze

sensing data.

CoT solutions attempt to satisfy various requirements by introducing several integrations, making them the most robust. However, none of them enable a full integration between IoT and HPC since requirements such as general-purpose computing and task scheduling are not detailed. Collaborative development is limited to sharing sensing data access, e.g., it is not possible to share variables, class methods or data structures to simulate a DSM address space. Moreover, different programming models are not provided to the user in an integrated manner. In general, processing and storage services do not operate concomitantly with sensing, actuating and context awareness services. Kaa is one of the most robust alternative; however, its idea of distributed processing and storage is based on Hadoop only and thus code refactoring is mandatory, as well as other limitations [87].

The proposed middleware share all compiled module, classes, sensing data and global variables to all components of the cluster in a transparent way using a DSM abstraction, improving the collaborative development. JCL also provides the task-oriented and event-based programming models in an integrated way. Besides that, different algorithms are used to schedule task processing and data storage, including sensing data storage.

2.3 | HPC Middlewares

Middlewares such as JCL (previous one) [3], JPPF [17], RAFDA [86], Jessica [90], Infinispan [69], Hazelcast [37], Gridgain [36] and Apache Ignite [81] have low or minimal code refactoring to provide distribution issues. Safety is achieved by Hazelcast, JPPF, Oracle Coherence [55], RAFDA, PJ [41], ProActive [6] and ICE [88] via encrypted communication. Infinispan and Gridgain have authentication and auditory services, being the most advanced middlewares in terms of this requirement.

In terms of heterogeneity, previous JCL and PJ can be executed on devices with JDK support. Apache Ignite, Gridgain, JPPF, Hazelcast, Oracle Coherence and ICE have support for many programming languages. Infinispan supports numerous communication protocols. JPPF and previous JCL are able to run over small devices, but only JPPF includes the Android platform.

Infinispan, Hazelcast and Oracle Coherence have storage fault tolerance. ICE, Apache Ignite, Open MPI [35], P2P-MPI [29], Gridgain and ProActive have processing fault tolerance. JPPF implements both and it is the most advanced alternative for this requirement. None of the HPC alternatives address solutions for Byzantine faults.

Middleware systems such as previous JCL, JPPF and Gridgain implement different scheduling techniques for tasks, but other systems, such as Java RMI [64] and MPI [27], delegate scheduling issues to users. All of the evaluated HPC solutions support general-purpose computing using one programming model, normally DSM, message-passing or task-oriented alternatives. Previous JCL, JPPF, Oracle Coherence, RAFDA, PJ, Apache Ignite and Jessica have easy deployment processes while JPPF and Apache Ignite are the most promising alternatives since they have service discovery facilities.

Distributed implementations of Java Collections Framework (JCF) data structures are offered by previous JCL, Hazelcast, Infinispan, Apache Ignite and Gridgain. Oracle Coherence has a Partition Backing Map, which is a Map-like structure, but it is incompatible with JCF code. Previous JCL, Infinispan, Hazelcast, Oracle Coherence, RAFDA, Apache Ignite and Gridgain provide a multi-developer environment where applications can access methods and user-typed objects from each other without explicit references. Previous JCL, RAFDA and Gridgain support super-peers; so, they can be considered the most promising middlewares in terms of the collaborative development requirement.

Most of the evaluated middlewares offer different mechanisms and integrations with different technologies to provide performance and scalability; however, few approaches detail how they guarantee performance. PJ implements several high-level programming abstractions, such as ParallelRegion (code to be executed in parallel), ParallelTeam

TABLE 2 Comparison of middleware solutions

Solution	Type	SaP	CA	DPM	DS	H	SA	SD	FT	GPC	LR	TS	SP	CD	PaS	Active	Free
JCL	IoT-HPC	✓✓	✓✓	✓✓✓	✓✓✓	✓	–	✓✓	–	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	Yes	Yes
X-GSN [13]	IoT	✓✓	✓	–	–	✓✓	–	✓✓	–	–	–	–	–	–	–	Yes	Yes
C-MOSDEN [59]	IoT	–	✓✓✓	–	–	✓	–	✓✓	–	–	–	–	–	–	–	Yes	Yes
FamiWare [28]	IoT	✓	✓	–	–	✓	–	–	✓✓	–	–	–	–	–	–	–	–
SMEPP [14]	IoT	✓✓✓	✓	–	–	✓	–	–	–	–	–	–	–	–	–	No	–
SIXTH [54]	IoT	–	–	–	–	✓	–	✓	✓	–	–	–	–	–	–	–	–
Hydra (LinkSmart) [22]	IoT	✓✓	✓✓	–	–	✓✓	–	–	–	–	–	–	–	–	–	Yes	Yes
VIRTUS [9]	IoT	✓	–	–	–	✓✓	–	✓✓	–	–	–	–	–	–	–	–	–
Sofia2 [75]	IoT	✓✓	✓✓	–	✓✓✓	✓✓	✓✓	–	–	–	–	–	–	–	✓✓✓	Yes	Yes
Cayenne [50]	IoT	✓✓	✓✓	–	–	✓✓	✓✓	–	–	–	–	–	–	–	–	Yes	Yes
OpenIoT [76]	CoT	✓✓	✓	–	–	✓✓	✓✓	✓✓	–	–	–	–	–	✓	–	Yes	Yes
Kaa [40]	CoT	✓✓	✓	–	✓✓✓	✓✓✓	✓✓✓	–	✓	–	–	–	–	✓	✓✓✓	Yes	Yes
SensorCloud [48]	CoT	✓	✓	–	–	✓✓	✓✓	–	–	–	–	–	–	✓	✓✓✓	Yes	No
Xively [47]	CoT	✓✓	–	–	–	✓✓	✓✓	–	–	–	–	–	–	✓	–	Yes	No
CloudPlugs [16]	CoT	✓	✓	–	–	✓✓	✓✓	✓	–	-	–	–	–	✓	–	Yes	No
Nimbits [51]	CoT	–	–	–	–	✓	✓✓	–	–	-	–	–	–	✓	–	Yes	Yes
ThingSpeak [84]	CoT	–	✓	–	–	✓	✓✓	–	–	-	–	–	–	✓	–	Yes	No
Mango [38]	CoT	✓	–	–	–	✓	✓✓	✓	–	-	–	–	–	✓	✓✓✓	Yes	No
Previous JCL [3]	HPC	–	–	–	✓✓✓	✓	–	✓✓	–	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	Yes	Yes
Infinispan [69]	HPC	✓✓	–	–	✓✓✓	✓✓	–	–	✓✓	✓✓✓	✓✓	–	–	✓✓	✓✓✓	Yes	Yes
JPPF [17]	HPC	✓	–	–	–	✓✓	–	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	–	✓✓✓	Yes	Yes
Hazelcast [37]	HPC	✓	–	–	✓✓✓	✓✓	–	–	✓✓	✓✓✓	✓✓	–	–	✓✓	✓✓✓	Yes	Yes
Oracle Coherence [55]	HPC	✓	–	–	✓✓✓	✓✓	–	✓✓	✓✓	✓✓✓	–	–	–	✓✓	✓✓✓	Yes	No
RAFDA [86]	HPC	✓	–	–	–	✓✓	–	✓✓	–	✓✓✓	✓✓✓	–	✓✓✓	✓✓✓	–	–	–
PJ [41]	HPC	✓	–	–	–	✓✓	–	✓✓	–	✓✓✓	–	–	–	–	✓✓✓	Yes	Yes
ProActive [6]	HPC	✓	–	–	–	✓✓	–	–	✓✓	✓✓✓	–	–	–	–	✓✓✓	Yes	Yes
Apache Ignite [81]	HPC	–	–	–	✓✓✓	✓✓	–	✓✓✓	✓✓	✓✓✓	✓✓	–	–	✓✓	✓✓✓	Yes	Yes
Gridgain [36]	HPC	✓✓	–	–	✓✓✓	✓✓	–	–	✓✓	✓✓✓	✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	Yes	No
ICE [88]	HPC	✓	–	–	–	✓✓	–	–	✓✓	✓✓✓	–	–	–	–	✓✓✓	Yes	Yes
Jessica [90]	HPC	–	–	–	–	–	–	✓✓	–	✓✓✓	✓✓✓	–	–	–	✓✓✓	No	Yes
Open MPI [35]	HPC	–	–	–	–	✓	–	–	✓✓	✓✓✓	–	–	–	–	✓✓✓	Yes	Yes
P2P-MPI [29]	HPC	–	–	–	–	✓	–	–	✓✓	✓✓✓	–	–	–	–	✓✓✓	No	Yes
MPJava [66]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	✓✓✓	–	Yes
MPJ Express [7]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	✓✓✓	No	Yes
F-MPJ [78]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	✓✓✓	Yes	Yes
Java RMI [64]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	–	Yes	Yes
FlexRMI [79]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	–	–	–
KaRMI [63]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	–	–	Yes
RMIX [43]	HPC	–	–	–	–	✓	–	–	–	✓✓✓	–	–	–	–	–	–	–

(group of threads that execute a ParallelRegion) and ParallelForLoop (work parallelization among threads). Previous JCL, Hazelcast and Oracle Coherence allow the execution of a task in a specific device of the cluster or a task to handle multiple method invocations with identical or different arguments.

HPC alternatives normally implement various requirements, but they still do not integrate with IoT. None of the HPC middlewares provide context awareness, sensing or actuating services. Various solutions are not designed to run over small platforms and few alternatives support super-peers. Existing HPC task schedulers do not consider the mobility, speed, direction and battery of some devices. CoT middleware alternatives are far from providing the services offered by the alternatives described in this section, which reaffirms the existing gap between IoT and HPC. The HPC capabilities of the proposed middleware were inherit by the previous version of JCL, which implements all of them except fault-tolerance.

2.4 | Perspectives

Table 2 presents a comparison of the IoT, CoT and HPC middlewares using the same set of requirements. Solutions that have improved versions changing their names such as GSN and X-GSN, are omitted. Table 2 has two new requirements that were not previously explained as they are simple: i) if the solution is active and ii) if it is free of charge. Each requirement can be fulfilled in three levels (✓, ✓✓ and ✓✓✓), where ✓ indicates basic implementations, ✓✓ indicates fundamental ones and ✓✓✓ indicates advanced designs. The '–' wildcard indicates that no information was found about the ability of the middleware in attending the requirement.

Some IoT applications with HPC needs are common in businesses related to smart buildings, factories and logistics; therefore, their demands are explained in some studies [23, 11], but the existing IoT/CoT/HPC middleware solutions did

not combine sensing, context awareness, actuating, processing and storage. In this way, it is currently impossible to develop a unique code demanding all services. HPC simulators, particularly spatial-temporal ones, WSN ones, game ones and robotic ones, should provide context, sensing and actuating services in their models. Some of them will require real sensing and actuating services operating with virtual ones. As a main code normally exists in HPC approaches, the event-based programming model is not suitable, even though it is common in IoT and CoT. Therefore, redesigns should occur in the integrated solutions, probably with support of more than one programming model.

Kaa can be considered the most promising alternative, but its IoT nature omits several details about its HPC services. Therefore we understand that it is far from being classified as an IoT-HPC solution. HPC alternatives capable of running over small platforms, such as JCL and JPPF, are well designed for HPC, so they can be extended to support IoT, although programming efforts must be considered. The gap between the two fundamental technologies is substantial and `UA`s exist, but no middleware solution attenuates the gap significantly. In the next section, we detail an alternative solution that extends the JCL HPC version to produce the first IoT-HPC middleware solution for Java and Android communities.

3 | JCL MIDDLEWARE PROPOSAL AND EVALUATION

The JCL was proposed to cover different middleware capabilities, illustrated in Figure 1 and presented as follows:

- General middleware capabilities, i.e., the expected or basic services present in all middleware:
  - **Simple deployment** without rebooting the system or devices used.
  - **Heterogeneity** allowing the interoperability of applications with different software and hardware specifications.
  - **Code refactoring** without implementing several interfaces to guarantee general user tasks.
  - **Scheduling** optimized to integrate HPC and IoT applications.
- JCL service capabilities represented though the JCL API:
  - **Basic services** allowing the user to register or unregister services, to start asynchronous run and to synchronized and get the processing results.
  - **DSM services** allowing the user to access and manipulate distributed variables, for instance, global, synchronized (lock and unlock) and hashmap variables.
  - **Sensing and actuating services** allowing the user to configure the sensor devices, to get the sensed and stored data and to interact with the environment through acting commands.
  - **Context awareness services** allowing the user to add and remove context rules and decisions.
  - **Security services** allowing the user to add and remove security elements as cryptography or authentication.
- Besides the commons services present in JCL, other specific capabilities are supported:
  - **Super-peer support** performed to make the interface among networks and to partition the cluster into logical entities.
  - **Collaborative development** allowing different `UA`s to share common JCL abstractions, such as sensor, sensing data, global variables, maps and any Java or Android object methods without explicit references.

The JCL middleware is composed by the following components: i. `JCL-User` component exposes the API to `UA`s and also enables JCL integration with MQTT brokers. ii. `JCL-Host` component used to handle the services numbered in Figure 1. It also helps the integration with MQTT brokers, publishing sensing data on them. The `JCL-Host` helps the scheduling of tasks and data. iii. `JCL-Server` component is responsible for managing `UA`s with their dependencies. There is only one `JCL-Server` per deploy. It also notifies devices with control data about the cluster of devices, therefore it

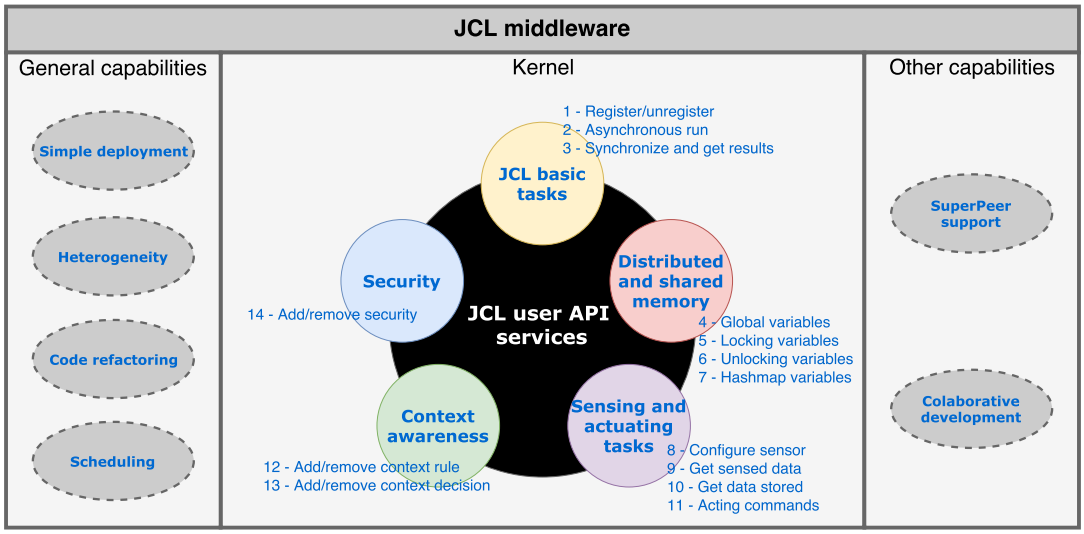

FIGURE 1 JCL Services

must be reachable by the remaining JCL components. iv. `JCL-Super-Peer` component redirects all data and commands to `JCL-Host` components under its control. It also creates tunnels with a `JCL-Server` component to enable clusters with invalid IP addresses to exchange data and commands. In our applications, `JCL-User` communicates with all other JCL components. The `JCL-User` component runs in Java and Android devices. `JCL-Host` runs in Java, Android and Arduino devices. `JCL-Server` and `JCL-Super-Peer` components run only in Java devices. The 14 services illustrated in Figure 1 represent the core improvements proposed in JCL. All these services are provided by `JCL-User` component and portable for all HPC, IoT and IoT-HPC applications.

In the next subsections we explain how JCL addresses solutions for most of the requirements presented in Table 2. We also present each component evaluation, thus we discuss the requirements, like super-peer support, performance and scalability, required to experimental evaluations. In our evaluation, the cluster of devices (IoT-HPC environment) used is composed of:

- Three microcontroller devices: Intel Galileo Gen 2, processor Intel® Quark™ 400MHz, 256MB RAM, Yocto Poky OS equipped with a 8GB SD-card; Raspberry Pi 2 Model B, processor BCM2837 Arm7 Quad Core 900MHz, 1GB of RAM, Raspbian Jessie OS equipped with a 8GB SD-card; and Beaglebone Black Rev B, processor Sitara™ AM3358 1GHz, 512MB of RAM, Debian Jessie OS, equipped with a 8GB SD-Card, are used to evaluate: the impact of JCL execution in different applications scenarios; the impact of getting and/or store the sensing data concurrently; and the JCL performance and scalability.
- Three smartphones: Lenovo Vibe K5 Plus, processor Snapdragon™ 616 (Quad core 1.5 GHz + Quad-core 1.0 GHz), 2GB of RAM, Android 6; Moto X, processor Snapdragon™ 801 (Quad core 2.5GHz), 2GB of RAM, Android 6; and Moto Z Play, Snapdragon™ 625 (Octa core 2GHz), 3GB RAM, Android 7, are used to evaluate: the impact of JCL execution in different applications scenarios; the impact of getting and/or store the sensing data concurrently. Specifically, the Lenovo device was used in the comparison of JCL and Zanzito solutions and in the JCL performance and scalability evaluation.
- Arduino Mega, micro-controller ATmega1280, clock 16MHz, 128KB of Flash memory, 8KB SRAM equipped with

Ethernet Shield W5100, is used to evaluate: the impact of getting and/or store the sensing data concurrently; and the JCL performance and scalability.

- One notebook Dell Inspiron 3421, processor Intel Core™ i3, 4GB of RAM, OS Ubuntu 16.04, is used to evaluate: the impact of JCL execution in different applications scenarios; and the impact of store the sensing data in a remote high-end device.

3.1 | General middleware capabilities

3.1.1 | Simple deployment

Deployment is a time-consuming process in most middleware systems. In some cases, its necessarily to reboot the system to deploy a new application. To reduce the deployment time, JCL adopts a simple deployment process based on both Java's reflection capabilities and the adoption of discovery services.

The JCL simple deployment process is illustrated in Figures 2 – 4. Only one `JCL-Server` exists for each JCL deployment and it should be deployed first, because it registers and manages the remaining components (Figure 2-A). Furthermore, at least one `JCL-Host` component must be deployed after the `JCL-Server` to guarantee that other JCL components will be registered correctly; this component is deployed in the same network as the `JCL-Server`. JCL supports one or many `JCL-Hosts` per cluster, as shown in Figure 2-B.

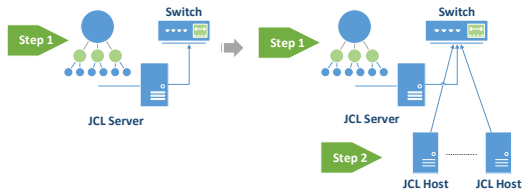

FIGURE 2 JCL multi-computer deployment view

Steps 3 and 4 of Figure 3 are optional, i.e., they are required only when interconnecting different data networks or when creating logical clusters according to specific needs, such as a group of `JCL-Hosts` to support machine learning services or collect sensing data from a smart building's garden. `JCL-Super-Peer` component deployment takes place in a network gateway (Figure 3) or in the same network as the `JCL-Server` to create logical groups of `JCL-Hosts`. Several `JCL-Hosts` can be deployed after a `JCL-Super-Peer` deployment, as shown in Figure 3. Several `JCL-Super-Peers` are feasible in a JCL multi-cluster or grid environment; in addition, nested `JCL-Super-Peers` can be deployed to produce a hierarchical network topology. The hierarchical topology in Figure 3 shows a `JCL-Super-Peer` inside a network managed by another `JCL-Super-Peer`. This network topology can be useful in many scenarios; for instance, a house cluster may contain a garden cluster, and the garden cluster may contain a swimming pool cluster. For this situation, three `JCL-Super-Peers` could be interconnected to form a hierarchical or tree topology.

In Figure 4, several `JCL-User` components are deployed on different types of machines (desktops, smartphones, tablets and laptops). We assume each machine is running a different Java or Android `UA`.

Following the deployment steps, multiple `UAs` can run over the same JCL cluster, sharing JCL abstractions (registered modules, maps, sensors, actuators, contexts, actions and global variables) without cluster reboots. When it is necessary to update a previously registered module, JCL requires only a new registration API call to perform all new registrations

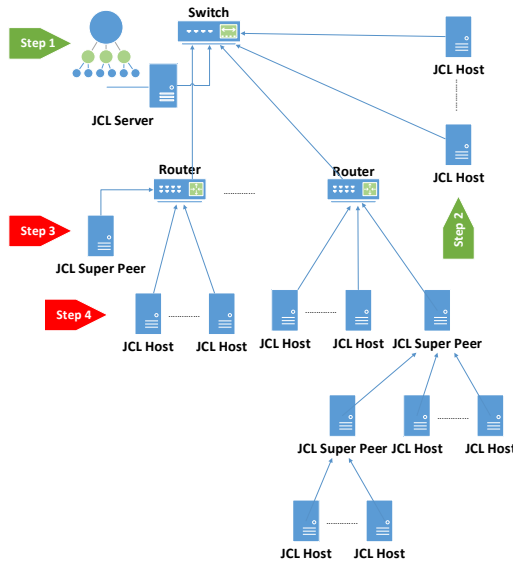

FIGURE 3 JCL multi-computer deployment view

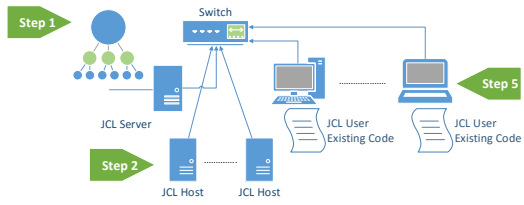

FIGURE 4 JCL multi-computer deployment view

in the cluster; thus, it is not necessary to stop the middleware's execution even in live update scenarios. To avoid having to register modules in the entire cluster each time, a selective registration approach exists in which only `JCL-Hosts` that will execute a `UA` must register it before its first execution.

JCLs discovery services implementation provides another useful advantage because a `JCL-Host` can become a `JCL-Server` if it is deployed first, i.e., it tries to find a `JCL-Server` and if it fails it becomes both a `JCL-Server` and a `JCL-Host`. If a `JCL-Server` is deployed later, it assumes control of the cluster, removing that responsibility from the `JCL-Host` that was previously working as a server. Furthermore, JCL components find each other in a network, which avoids having to perform manual configuration of each component, a time-consuming activity that is impractical in dynamic cluster deployments.

3.1.2 | Heterogeneity

`JCL-Host` is implemented for different platforms such as Linux compatible boards, Android and Arduino. This way, JCL runs over three huge community platforms: Java, Android and Arduino. The `JCL-Host` for Linux boards uses MRAA

library [39] to access the pins of the board to guarantee portability and interoperability since MRAA can be executed on many boards and it enables transparent sensor or actuator accesses regardless their communication particularities. Besides using TCP communication to provide all IoT-HPC services and UDP for discovery and other internal control services, JCL is integrated with MQTT protocol, enabling external entities to subscribe to sensing data provided by `JCL-Hosts` or a new scenario where `JCL-Hosts` subscribe to sensing data from external entities.

There are `JCL-User` components for Java and Android, enabling the development of both types of IoT-HPC `UAs` and their integration in a single cluster transparently. In summary, the `JCL-Host` component can convert `.class` files into `.dex` files and vice-versa using Android SDK DX Tool [33] and reverse engineering [57], respectively.

We evaluated the cost of this conversion in both scenarios: i) when a `UA` registered an Android compiled module in a Java `JCL-Host`; and ii) when a `UA` registered a Java compiled module in an Android `JCL-Host`. To perform this evaluation we registered three different `UAs` with different sizes to observe the impact of the application size on the conversion. The `UAs` used in this experiment were a Big Sorting application with 10KB size, a Dijkstra solver application with 500KB size and an e-mail sender application with 1MB size.

Three devices were used when `UA` is Java (Lenovo, Moto X and Moto Z phones, precisely). Figure 5 illustrates the runtime values to convert `.class` files into `.dex` files. In general, the runtime deteriorates two times when Dijkstra is registered and near ten times when Mail is registered, but the Mail application is twice the size of Dijkstra, so the conversion overheads is not proportional to the application size. The main justification is the complexity of the code, so a small complex code with recursive calls, complex types, several dependencies and so forth takes more time to be converted than a bigger code with simple methods and only primitive types. The Moto Z phone increased its runtime only three times when the same applications are registered, what means that code conversion is feasible only in devices with more computational resources. The standard deviation values were less than 10% in the worst cases, so the results can be considered conclusive.

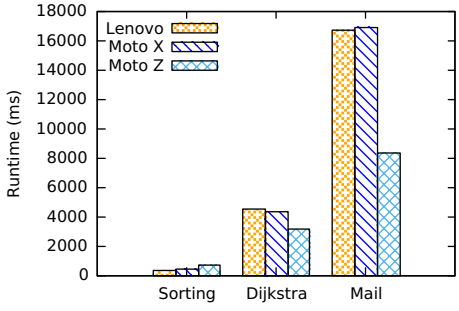

FIGURE 5 Runtime to register an application, requiring code conversions - Java to Android

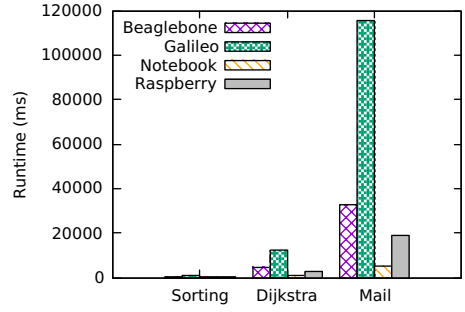

FIGURE 6 Runtime to register an application, requiring code conversions - Android to Java

When `UA` is deployed in Android (Figure 6) four devices with Java support were used: Beaglebone, Galileo, Notebook (Dell Inspiron) and Raspberry Pi. The Galileo did not handle the register of the Mail application satisfactorily, requiring nearly 120 seconds (two minutes) to register it in a single `JCL-Host`. In general, code conversion introduced overhead, being not satisfactory in some scenarios. However, JCL offers an alternative strategy when these unacceptable results occur: Users can start `JCL-Host` components with their business logic dependencies, meaning that JCL can do conversions and registrations once and during its starting phase, so no further registrations are necessary when running a `UA`. Customized JCL deployments of `JCL-Hosts` are very useful when `UAs` size are large, when the codes are complex or

when devices have limited computational resources. The higher standard deviation was 12.8% in these experiments, reinforcing the quality of the results.

3.1.3 | Code Refactoring

Normally, existing middleware solutions force their users to implement several interfaces to guarantee distributed storage or asynchronous distributed tasks. In JCL, no such interface needs to be implemented because JCL adopts Java reflection to avoid code refactorings of existing and well tested methods, variables, components or algorithms.

To explain this feature, we use the ubiquitous "Hello World" application. Figure 7 depicts a class with a sequential method named "print" that represents part of the business logic of a UA. In this example, the method simply prints the sentence "Hello World!", but the concept works the same for any other demand.

```
1 public class HelloWorld {
2
3     public void print() {
4         // Prints ''Hello World!'' in the console.
5         System.out.println("Hello World!");
6     }
7 }
```

FIGURE 7 Business logic - Hello World

It is important to highlight that JCL does not automatically partition the user business logic into several distributed tasks; therefore, the method "print" is never automatically partitioned by JCL, only allocated and executed by it in a JCL deployment. Figure 8 illustrates how JCL achieves distribution for existing sequential code. At line 4, the user obtains an instance of the JCL, and at line 5, the class HelloWorld is registered; subsequently, it is visible to the entire JCL cluster. At line 6, JCL starts one task per JCL-Host in the cluster to enable execution of the "print" method of the registered class. The JCL "executeAll" method requires the class nickname ("Hello"), the method to be executed ("print"), and the arguments to the method (or null if no arguments are required).

```
1  public class JCLHelloWorld {
2
3      public static void printAll() {
4       JCL_facade jcl = JCL_FacadeImpl.getInstance();
5       jcl.register(HelloWorld.class, "Hello");
6       List<Future> tickets = jcl.executeAll("Hello","print",null);
7        // do any other task, including new JCL calls
8       List<JCL_result> result = jcl.getAllResultBlocking(tickets);
9      }
10 }
```

FIGURE 8 Distribution logic - Hello World

Another way to work with JCL is to execute distributed tasks remotely. For example, we can use JCL to start other JCL tasks, as shown in Figure 9. Basically, the main application sends ten parallel tasks that represent JCL distributed tasks, as shown in Figure 8. In Figure 9, at line 5, the UA requests a JCL instance, and line 7 registers a Jar file containing the "JCLHelloWorld" (Figure 8) and "HelloWorld" (Figure 7) classes. Finally, at line 10, the JCL instance launches ten tasks by calling its "execute" API method.

```
1  public class MainClass{
2
3    public static void main(String[] args) {
4
5     JCL_facade jcl JCL_FacadeImpl.getInstance();
6     File[] complexApplJars = {new File("./JCLHelloWorld.jar")};
7     jcl.register(complexApplJars, "ComplexHello");
8     List<Future> tickets = new ArrayList<Future>(10);
9
10    for(int i=0;i<10;i++) tickets.add(jcl.execute("ComplexHello","printAll",null));
11
12    // do any other task, including new JCL calls
13
14    for(Future t:tickets){
15      Object result = t.get();
16       //do any computation with the obtained result
17    }
18   }
19 }
```

FIGURE 9 Distribution logic - Complex Hello World

The methods "executeAll" and "execute" are asynchronous; therefore, the UA can execute other code while waiting for the results. When these methods are called, the JCL kernel assigns a thread, named "worker," to handle the task's execution in a specific `JCL-Host`. The `JCL-Host` initially starts as many "worker" threads as the number of cores available; any remaining tasks are added to a "worker" queue to wait for execution. This strategy can generate deadlocks because a task running can be waiting for a task that is not running (i.e. in the queue). To avoid this problem the `JCL-Host` starts a new task whenever all the running tasks have been in the wait state for a given time. This strategy causes more thread yields and, consequently, more context switches, but it avoids deadlocks because the JVM always guarantees that a CPU is available for all the started threads. This strategy also reduces the number of tasks in the "worker" queue, reducing the possibility of moving tasks from overloaded `JCL-Hosts` to less-loaded ones. This aspect is discussed in detail in the Scheduling Section.

3.1.4 | Scheduling

JCL adopts different strategies for scheduling processing and storage services in a cluster. More precisely, the scheduling of tasks, global variables, `JCL-HashMap` and their <*key*, *value*> pairs occur in two ways.

To schedule a task to invoke a method or a group of methods, JCL adopts a two-phase distributed solution. In the

first phase, the JCL-User component quickly dispatches a task to a JCL-Host using a circular list of JCL-Hosts. The list of JCL-Hosts is obtained from JCL-Server, which is responsible for notifying the cluster members when changes occur. After selecting a JCL-Host, the JCL-User determines the number of tasks per JCL-Host using its information about the number of available cores provided by the JCL-Server. The JCL-User can group the tasks into chunks before submitting them if UAs explicitly call the "jcl.executeAll" API service or if the users configure a property file for that purpose.

The number of chunked tasks to invoke methods remotely and asynchronously is always proportional to the number of cores available in each JCL-Host; thus, it is designed to work with different nodes in a cluster. To deal with UAs where the number of method invocation calls is not proportional to the chunk size, JCL implements a watchdog. A watchdog is a thread that wakes up every 100 milliseconds. At each run it flushes the chunk regardless of the number of processing calls on it.

In the second scheduling phase, the JCL-Hosts components collaborate with each other to better balance the JCL cluster workload. Each JCL-Host, after executing its last task (i.e., when its last "worker" thread finishes execution), tries to obtain and execute a new task from other threads in the JCL cluster. Each time, it obtains only one task from a "worker" queue to avoid new redistributions in the cluster. This collaborative behavior mitigates problems caused by the circular list scheduler, implemented in JCL-User; therefore, even non-deterministic heuristics can be scheduled efficiently in JCL, requiring few tasks replacements and dramatically reducing the runtimes. Previous experimental results [3] demonstrated that JCL scales well in different scenarios, precisely with different number and type of arguments in each task.

When using a circular list scheduling technique, it is possible for a scenario to occur in which one JCL-Host receives most of the CPU-bound tasks. In JCL, the second phase redistributes these tasks with all the other JCL-Host "worker" threads. A JCL-Host that handles a task from another JCL-Host must notify the JCL-User component to update its control data, since it contains the task metadata for each task, including the JCL-Host that handles it. However, to avoid architectural bottlenecks, the JCL-Server component is not notified after JCL-Host scheduler decisions.

This load balancing technique is classified in [58] as a neighbor-based approach because it is a dynamic load balancing technique in which nodes transfer tasks among their neighbors. Consequently, after a number of iterations, the whole system is balanced without having to introduce a global coordinator. As mentioned earlier, if deadlocks are detected in a JCL-Host, using the state of a task in wait mode for a long time, the number of tasks in the "worker" queue will be reduced because the JCL-Host starts new queued tasks to try to eliminate the existing deadlock; consequently, task replacement will also be reduced.

To schedule global variables, maps, map <*key*, *value*> pairs and sensing data stored as <*key*, *value*> pairs of maps, the JCL-User component calculates a function F to determine the JCL-Host in which they will be stored. Equation 1 is responsible for performing a fair distribution, where $hash(vn)$ is the hashcode of the global variable name or the map name or the key of a <*key*, *value*> pair, nh is the number of JCL-Hosts and F is the remainder of the division corresponding to the node's position. JCL adopts the default Java hash code for strings and primitive types, but user-typed objects require a hashcode implementation.

$$F = Remainder(\frac{|hash(vn)|}{nh}) \tag{1}$$

Previous JCL work [3] demonstrated that JCL achieved fair distribution in many scenarios, including when aleatory

names are used. To achieve distributed and fair sensing data storage, the `JCL-Host` component uses a `JCL-HashMap` data structure to store each sensor sensing data entry. The sensing data timestamps guarantee the uniqueness of entries in a map and there is a single distributed map per sensor in JCL. Forward in this paper, we detail how JCL provides distributed storage services, including sensing ones.

One drawback introduced by d is that JCL must check $(2 * d) + 1$ nodes to search for a stored object, i.e., if d is equal to 2, JCL must check five nodes (two before and two after the `JCL-Host` identified by function F in the logical ring); therefore, JCL performs parallel checks to reduce this overhead. Experiments demonstrated that the extra communications introduced by the parallel checks are compensated for when compared with only two sequential checks, which is possible when check of the first node did not find the desired object.

The F function also introduces a problem for the scheduler when a new `JCL-Host` enters or exits the cluster. In this scenario, the location of the previously stored objects changes. To avoid storage replacements, which can become a time consuming activity, the `JCL-Server` and the `JCL-User` components maintain all previous cluster sizes after the first object instantiation in JCL; thus, F can be applied to each cluster size, increasing the number of parallel checks in the network and reducing their benefits, but avoiding having to replace huge objects very often. Normally this compensatory strategy is sufficient. However, users can opt to replace all objects after all cluster changes by modifying a property file.

3.1.5 | General-purpose computing

JCL was originally designed as a HPC middleware; therefore, as any other HPC middleware, it supports the execution of general-purpose tasks. It also supports the storage of any Java or Android Object and has has mechanisms to convert Java classes to Android and vice-versa. Previous experimental evaluations [3] demonstrated that JCL scales well while performing task processing and object storage as global variables or as map <*key*, *value*> pairs. The Code Refactoring Section demonstrates that JCL introduces general-purpose computing with low code refactoring, an uncommon issue in HCP middleware literature.

3.2 | JCL services capabilities

3.2.1 | Different Programming Models

It is possible to develop applications over three programming models in JCL: The first one is the task-oriented programming model. Each call to a JCL API service starts a JCL task. The HPC task-oriented programming model is represented by **JCL basic tasks (services 1, 2 and 3)** in Figure 1. They represent the three API methods: i. the `register()` method to register a compiled module in a JCL cluster (Ex. JAR, class, dex and compiled module extensions); ii. the `execute()` method to schedule and run a task that invokes a class method with any number and type of arguments. The `executeAll()` and `executeAllCores()` variants start one task per `JCL-Host` or per JCL cluster core, respectively; and iii. the `get()` method to synchronize the caller and get the task result or an error. The `get(timeout)` is an alternative to synchronize for a specific period, avoiding indefinite wait. There is the `getAll()` variant to wait for a group of tasks.

The second JCL programming model is the Distributed Shared Memory (DSM). One of the main limitations of the task-oriented programming model is that the tasks do not share a common address space over a cluster. The DSM programming model is an alternative, being represented by the **Distributed and shared memory (services 4, 5, 6 and 7)** in Figure 1. In JCL, there are two options for distributed storage, as previously mentioned:

Global variables By using global variables, which means any Java object can be stored in a `JCL-Host` component in a cluster. This storage option is present only in JCL.

HashMap By using `JCL-HashMap`, which means that `UA` can instantiate a distributed data structure and, for instance, put a pair <*key*, *value*> on it. JCL supports all Java Map interface method implementations; thus, code refactoring is reduced since users continue to use the Map interface, present in Java and Android since their beginning, but internally it is a distributed JCL implementation.

The third programming model is the event-based model. This is achieved through the integration with MQTT brokers, where an `UA` subscribes itself to receive sensing data from a sensor. When external events, called contexts, happens the `UA` can start different types of tasks, such as sensing or actuating tasks, but also any general-purpose task. This can be done programmatically, but also by simply configuring a text file, indicating the conditions to trigger the tasks and which tasks must be started.

JCL also introduces what we call active sensing. It is a synchronous way to obtain sensing data from a sensor without intervals. It is represented by the `getSensingDataNow()` API method. We evaluated this method in a cluster with six different devices. Eighteen sensors were used in the experiment, so each device handled three simple sensors in Figure 10. First, sixteen concurrent `UAs` entered in a loop, calling `getSensingDataNow()` from all sensors of the cluster repeatedly. Two Dell Inspiron simulated sixteen `UAs` because each device had a quad-core processor with hyper-threading technology.

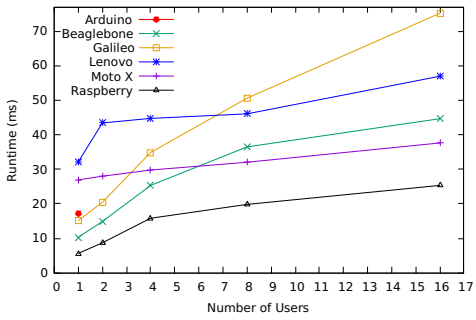

FIGURE 10 1 to 16 concurrent `UA` `getSensingDataNow()` calls

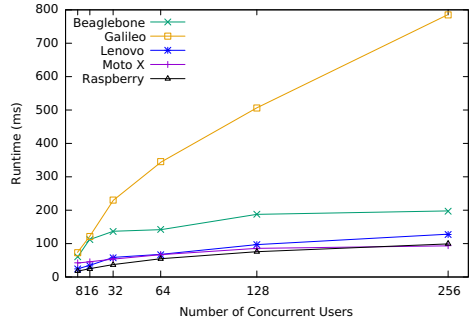

FIGURE 11 1 to 250 concurrent `UA` `getSensingDataNow()` calls

Arduino has a single entry in the graphic because it did not support concurrent `getSensingDataNow()` calls, so this experiment was useful to understand that sensor node device category, like Arduinos, cannot be concurrently controlled, so the best option is their utilization as passive sensing devices only, storing sensing data in distributed `JCL-HashMap` or in MQTT brokers.

The Raspberry Pi is by far the best device in the six alternatives, varying its runtime five times to handle sixteen concurrent `UAs` performing eighteen sequential `getSensingDataNow()` calls each. The worst result was observed in Galileo, which consumed 75 milliseconds per `getSensingDataNow()` call at the same concurrency level, but even the worst result maintained the runtime difference within a factor of five times to process sixteen concurrent calls. The Android Moto X device had the most stable behavior, varying only 10 milliseconds from the worst to best results. In general, the standard deviation of these concurrency experiments were low, reaching a maximum of 10%.

Figure 11 illustrates an extreme scenario, where 1-250 concurrent `UAs` performed eighteen `getSensingDataNow()`

calls sequentially. These requests were submitted from a single device of type Dell Inspiron. The idea was to observe JCL behavior in the presence of high concurrency levels. Four of five devices supported such an extreme scenario and three devices spent less than 100 milliseconds to handle each of these method calls. The runtime increased, on average, ten times, but the concurrency increased 250 times, which seemed very promising. The `JCL-Host` deployed in the Galileo device was the only one that degraded more as the concurrency increased. Since all devices uses the same executable `JCL-Host` we can argue that it is not a problem caused by JCL.

In summary, the `JCL-Host` multi-thread component can handle concurrent active sensing calls to several sensors. When the devices have several sensors, the threads used to handle them compete with the threads used to handle API calls; thus, the runtimes are expected to increase, but, as this experiment confirmed, JCL is also designed for these scenarios.

3.2.2 | Distributed Storage

JCL has a distributed implementation of the Java Map interface (**service 7** in Figure 1), allowing users to adopt a data structure that is familiar to the Java/Android communities, requiring minimal refactoring of existing code. In general, users just replace a Map sub-type object (Ex. Tree Map, Hash Map, etc.) with a `JCL-HashMap`, so the storage, which before was done locally, is now done in a distributive manner over a cluster of multi-processor devices.

The sensing data produced by `JCL-Hosts` are stored in `JCL-HashMaps`, so for each sensor of a cluster there is a distributed hash map data structure. The keys of each `JCL-HashMap` are the timestamps of sensing data, so uniqueness is preserved. The `UA` never sees a sensor sensing data as a map, but always via API methods. Internally, JCL designs two alternatives for sensing data storage: i. Small devices, like Arduino, always ask the `JCL-Server` or `JCL-Super-Peer` to perform distributed map insertions; and ii. Higher device categories insert their sensing data directly in `JCL-HashMaps` stored over `JCL-Hosts`. These sensed data are available by JCL-User through **Sensing and actuating tasks** (**services 8, 9, 10, and 11** in Figure 1.

We evaluated the cost to store sensing data on `JCL-HashMap` and also the cost of simply publishing the sensing data using MQTT technology. The service that publishes sensing data to `UA`s was evaluated in three scenarios: i) when the number of sensors increased; ii) when the sensing data interval was reduced; and iii) when the number of devices increased.

Internally, a `JCL-Host`, deployed seven times in five different devices, stored sensing data in a MQTT Mosquitto [21] broker or in a `JCL-HashMap`, which implies the utilization of any `JCL-Host` component in the cluster for the <*key, value*> storage, as explained before. A Dell Inspiron device was used as the only `JCL-Host` capable of storing data in the cluster and as the MQTT broker to receive the sensing data. The entries in graphic illustrated by Figures 12 and 13 represent JCL using `JCL-HashMap` or MQTT broker alternatives, respectively. Basically, entries with suffix names equal *MQTT* are deployments using the Mosquitto broker.

Four out of the seven deployments illustrated in Figure 12 had stable runtimes to publish sensing data from simple sensors in a MQTT broker or in a `JCL-HashMap`, varying the runtime from 5-17 milliseconds. The higher runtime variations occurred in Lenovo phone and the main reason is because it is the only device with WiFi connection, which is more unstable than wired communications. In general, Mosquitto broker is almost 10-30% faster than `JCL-HashMap` to handle simple sensors with high intervals. The MQTT protocol proved its efficiency against TCP alternatives, indicating that we must reconsider JCL's solution to transmit sensing data.

When the interval was reduced to 20 milliseconds, the runtimes of all tested devices continued to be stable, as Figure 13 illustrates. Each device handled three simple sensors, as previously explained, and MQTT Mosquitto was faster than `JCL-HashMap`, specially in the Lenovo phone, thus the MQTT technology is better designed for wireless

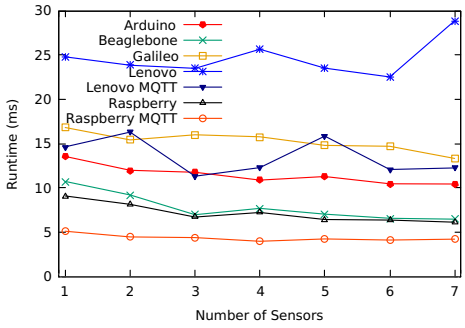

FIGURE 12 Runtime to store sensing data when the number of sensors varied

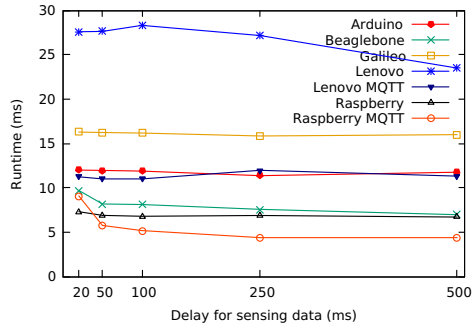

FIGURE 13 Runtime to store sensing data when the interval time varied

communications than JCL. An important issue is that in JCL there is the impact of a distributed sensing storage, not performed by Mosquitto, so overheads are introduced, but sometimes they are compensated by the distributed sensing storage benefits. In summary, we understand that JCL must consider MQTT alternatives to transmit sensing data, but the distributed sensing storage improvements should be continued.

When the number of devices increased the runtimes continued stable. In Figure 14, each new device had three simple sensors publishing sensing data according to a interval of 500 milliseconds. A single Dell Inspiron was used to store all sensing data, so a single `JCL-Host` or a Mosquitto instance was deployed. The Mosquitto was 30% to 100% faster than JCL to publish sensing data concurrently and the Android mobile phones had the worse results, explained by both wireless TCP communications overhead and distributed sensing storage, regardless of the fact that the cluster had a single `JCL-Host` for that. Again, the results demonstrated that JCL must introduce lightweight wireless IoT protocols to publish sensing data in `JCL-HashMaps`, especially when simple sensors are used; however, we can consider these results also very promising, since they were stable, being justified in part because each sensor has a `JCL-HashMap` to store its sensing data, attenuating concurrency with other sensors and devices.

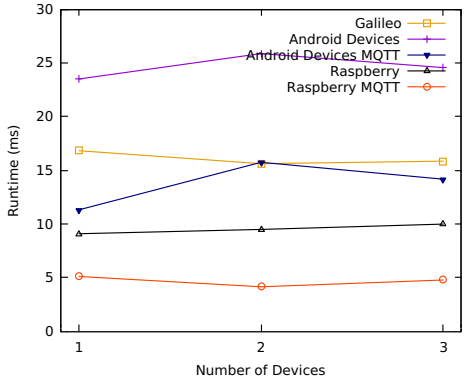

FIGURE 14 Runtime to store sensing data when the number of devices increased

3.2.3 | Context Awareness

IoT context awareness services are represented by **services 12 and 13** in Figure 1. Three methods are necessary to provide context awareness: i. the `registerContext()` method to synchronously register a context in a device, so it is necessary to have the device, the sensor, an expression and a nickname for the context. The expression in JCL is defined per sensing value; e.g., a GPS sensor produces two double values after sensing, one for latitude and one for longitude, so users can easily define expressions like $s_0 >$ `lat`; $s_1 <$ `lgt`, where s_0 and s_1 represent the two GPS sensing data and 'lat' and 'lgt' represent users limits for an area. The `registerContext()` method guarantees that the sensing data are being stored during the period the context continues to be true; ii. the `addContexAction()` method to asynchronously add an action to a context, where an action means a sensing task, an actuating task or any HPC general-purpose task. To add an action, users must inform the context previously defined, a sensor to be accessed after a context is reached and the commands to operate such a sensor. There is a second type of action, where a class, its method and arguments must be informed, enabling asynchronous task executions when contexts are reached. A `Future` object is returned when actions are configured, enabling local computations, JCL calls or any other business logic in `UA` before waiting for a context to occur. When a context is reached an action runs and only when a context is reached again it runs a second time and so on; and iii. the `get()` method to synchronize the `UA` to get the result of an asynchronous action. Note that, the `UA` waits until a context is reached, so it represents a JCL alternative to develop a publisher and a subscriber in the same Java class with few portable API calls. The `Future` object has a `get(long timeout)` method variant, so `UA` can wait a predefined time for a context, perform other commands/computations and repeat these steps many times while the context is unreached.

More complex context awareness applications can be developed in JCL, e.g., the `UA` can start many tasks, where in each task the following API calls can be made: `registerContext()`, `addContexAction()` and `get()` to synchronize the caller to wait until a context is reached or to wait for a predefined time. This way, this new `UA` can add both contexts and actions to sensors concurrently, enabling IoT subscribers to register themselves to several contexts of several sensors, being notified sequentially and according to an order, using nested `get` calls in `UA` for that, or concurrently and asynchronously, using a single `get` call inside each started task. These development options for subscribers or ordinary HPC Java/Android applications are present only in JCL.

3.2.4 | Security

JCL introduces security in API; therefore, the communications done to provide a service in JCL API occur safely in `UA` when it calls the `setEncryption()` method before any API call. The method is used to synchronously apply/suppress security policies of JCL, **representing service 14** of JCL API illustrated in Figure 1.

Internally, JCL components adopt the Advanced Encryption Standard (AES) [19] with a 128-bit key in Cipher Block Chaining (CBC) operation mode. JCL also adopts HMAC-SHA256 to guarantee data integrity in communications. The HMAC [10] is a hash based Message Authentication Code (MAC). Each message transmitted is partitioned into 16-byte blocks. The last block can have empty bytes; therefore, JCL uses PKCS#7 [70] to assert all blocks with exactly the same number of bytes. In general, JCL encrypted messages are 48 bytes longer than insecure ones.

JCL is the only middleware solution that allows enabling/disabling data encryption through API, i.e., during the execution of a certain block of code. IoT is huge in many aspects [60], so security at the API level can be very useful for Intranet HPC clusters together with sensing things in the Internet, therefore imposing security restrictions. JCL can also apply security polices when a `JCL-Host` starts, so it works also without explicit API calls, similar to several solutions in the literature [9, 54, 1].

3.3 | Other JCL capabilities

3.3.1 | Super-peer support

In order to make the interface among networks and to add the capacity to partition the cluster into logical entities, JCL adopts the concept of super-peer. The `JCL-Super-Peer` component was designed with two internal components. The first one behaves as a `JCL-Server` component (referred to as `JCL-Super-Peer-Server`) for a given network, and the second behaves as a `JCL-Host` component (referred to as `JCL-SuperPeer-Host`) for the network where the `JCL-Server`, other `JCL-Hosts` and `JCL-Super-Peers` are deployed. Figure 15 illustrates an example of a cluster partitioned into logical networks in a smart-house appliance. Although all devices are on the same network, we partitioned the cluster using two `JCL-Super-Peers` components, thus creating two logical partitions. In this way, it is possible to send control commands and data to all `JCL-Hosts` of a certain logical group. We evaluated the overhead caused by this component in two scenarios: i) when the number of `JCL-Host` components managed by a `JCL-Super-Peer` increased; and ii) when the number of `JCL-Super-Peer` components increased.

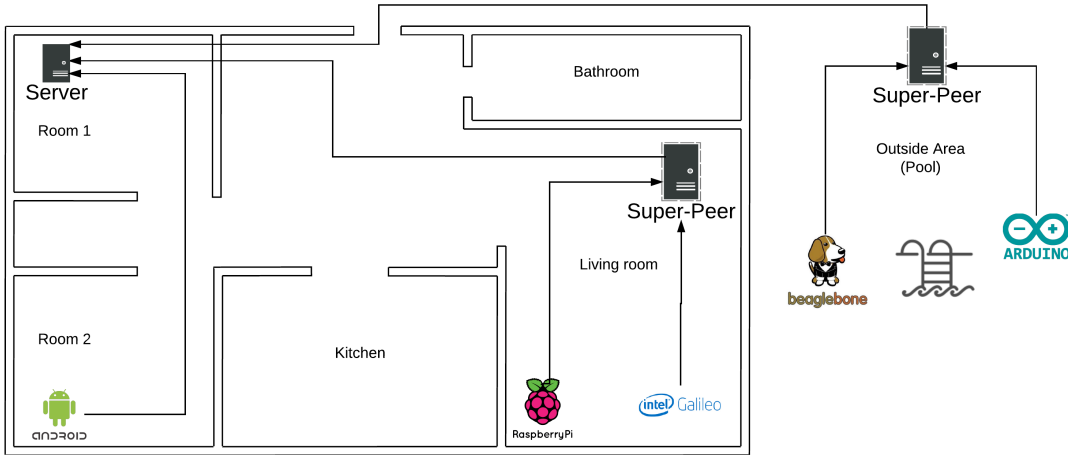

FIGURE 15 Logical network partition using `JCL-SuperPeer` components

Both scenarios used a cluster with six `JCL-Host` components deployed in six devices and eighteen sensors, each device having three simple sensors. Initially, no `JCL-Super-Peer` was deployed and the `UA` started eighteen `getSensingDataNow()` calls repeatedly, one for each sensor. The `UA` application was deployed in the Dell Inspiron device and the results of this cluster configuration were used as a baseline to other cluster configurations.

Figure 16 illustrates a single `JCL-Super-Peer` overhead when the number of `JCL-Hosts` under its control increased. The first value of Figure 16 means that a cluster with a single `JCL-Super-Peer` managing two `JCL-Hosts` is nearly 3% slower than the baseline cluster configuration without `JCL-Super-Peer`. When the `JCL-Super-Peer` had four and six `JCL-Hosts` under its control, the results were 9 and 13% slower when compared to the baseline, respectively, but the number of `JCL-Hosts` increased 100% or even 150%, so the results were promising. The standard deviation values of `JCL-Super-Peer` experiments were about 7%, so very conclusive.

The same cluster was partitioned into three other configurations, where in the first configuration it is considered one `JCL-Super-Peer`, in the second two `JCL-Super-Peers` and in the third three `JCL-Super-Peers`. The same number of `JCL-Hosts` was preserved, so each `JCL-Super-Peer` had two `JCL-Hosts` in each round. Figure 17 illustrates a small

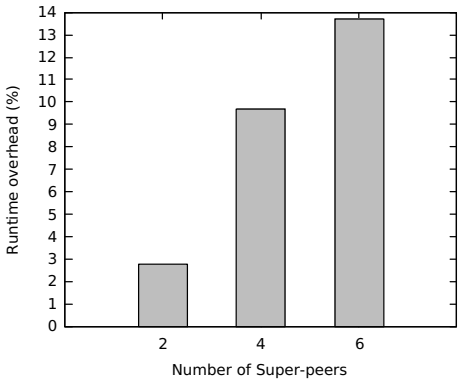

FIGURE 16 JCL-SuperPeer managing different number of JCL-Hosts

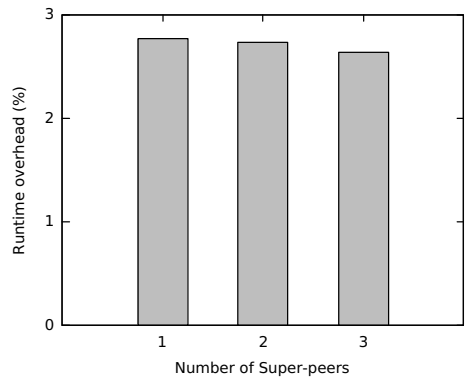

FIGURE 17 Overhead when the number of JCL-SuperPeers increased

overhead of almost 3% when compared with a cluster without JCL-Super-Peer, and two fundamental reasons for that are: i) JCL-Super-Peer does not open the messages while forwarding them, avoiding decoding overhead; ii) as the number of JCL-Super-Peer increased, the communications are partitioned among them, attenuating the concurrency level at each JCL-Super-Peer, thus reducing the runtime overhead as the number of JCL-Super-Peers increased.

3.3.2 | Collaborative Development

JCL applications can share registered UAs, instantiated global variables and maps, but also sensors and actuators without explicit references, so an UA *A1* in a machine can access an object instantiated by another application *A2* using only its nickname. The same capability is performed for IoT abstractions such as contexts, actions, sensors, actuators or sensing data. By introducing this requirement, UAs around the world can share their algorithms, data structures, IoT abstractions and computational power of multiple clusters, forming grid environments. The JCL-Super-Peer component plays an important role since it enables the interconnection of distinct networks.

```
1 Map<Integer, Integer> JCLMap = JCL_FacadeImpl.getHashMap("Test");
2 JCLMap.put(0, 1);
3 JCLMap.put(1, 10);
```

FIGURE 18 UA one in machine one

To exemplify the collaborative behavior of JCL, consider one UA starting a JCL-HashMap named "Test" at line one of Figure 18 and storing two <*key*, *value*> pairs in lines two and three, respectively. UA two can recover the JCL-HashMap named "Test" at line one and print the values of key "1" and "2" at lines two and three of Figure 19. It can also put other values, as line four illustrates. The UA can also lock an entry of an existing map and update its value, as lines five, six and seven of Figure 19 demonstrate. Executing registered methods as tasks, instantiating global variables, accessing sensors and actuators, reusing contexts and actions, and retrieving sensing data over JCL clusters follow the same collaborative idea.

```
1 Map<Integer, Integer> JCLMap = JCL_FacadeImpl.getHashMap("Test");
2 System.out.println(JCLMap.get(0));
3 System.out.println(JCLMap.get(1));
4 JCLMap.put(2, 100);
5 int value = JCLMap.getLock(0);
6 value++;
7 JCLMap.putUnlock(0, value);
```

FIGURE 19 UA two in machine two

4 | GENERAL MIDDLEWARE EVALUATIONS

4.1 | Comparative analysis

To complement the previously presented results, we tested `JCL-Host` against Zanzito. As explained in Section 2, Zanzito is a MQTT publisher for Android. We ran the experiments on the same Lenovo smartphone and we used only the smartphone since Zanzito runs over Android devices.

We repeated some experiments to compare CPU, memory and battery usages. In these experiments, seven simple sensors were configured and the sensing interval was fixed at one second because it is the minimum rate allowed by Zanzito. JCL used less CPU in all scenarios (Figure 20), thus we can consider that JCL is a feasible Android alternative in terms of CPU usage. Furthermore, the results were also consistent, since the standard deviation was very uniform and low. In terms of memory consumption, `JCL-Host` consumed less memory than Zanzito, precisely 10-20% less memory (Figure 21).

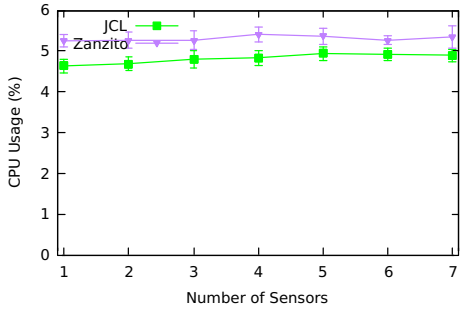

FIGURE 20 `JCL-Host` CPU usage versus Zanzito when sensors increased

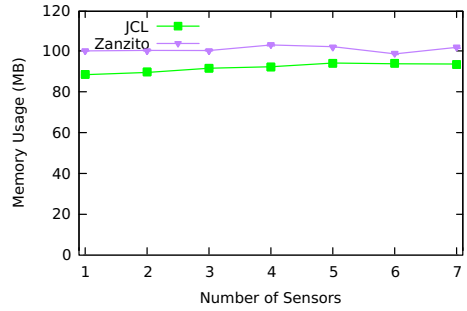

FIGURE 21 `JCL-Host` memory consumption versus Zanzito when sensors increased

In Table 3, we compared JCL and Zanzito in terms of CPU usage and memory consumption when a camera sensor is configured with a fixed interval. As mentioned before, JCL compresses the photos, but Zanzito does not, being unable to publish photos with one second interval. This way, we disabled the compression in JCL, so both solutions generated photos with a very close size, precisely 3MB each photo. Five seconds interval for sensing was sufficient for both alternatives to publish photos without errors and the results demonstrated that JCL consumed less CPU and memory than Zanzito in these scenarios. The average standard deviation in this experiment was 13%, so the results can be

considered conclusive.

TABLE 3 Battery and memory consumption comparison between JCL and Zanzito with a camera sensor

Solution \ Description	CPU (%)	Memory (MB) (%)
JCL	10.013	94.563
Zanzito	11.360	111.201

We repeated the battery experiments with seven simple sensors publishing data at one second rate to establish a comparison between JCL and Zanzito. We can observe that JCL consumed less battery than Zanzito in all experiments (Figure 22). In the experiment with seven sensors JCL consumed almost 40% less battery than Zanzito, a promising result to reinforce that JCL is competitive.

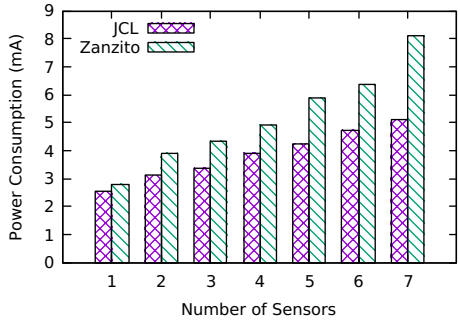

FIGURE 22 `JCL-Host` battery usage versus Zanzito when sensors increased

We measured the battery consumption when a camera was configured to publish photos at five second rate in both JCL and Zanzito. In JCL, the battery consumption reached almost 13mA while in Zanzito it was close to 17mA. The average standard deviation of this experiment was 4.12%. In general, JCL is a feasible solution to deal with special camera sensors regarding battery consumption.

4.2 | Performance and Scalability

JCL uses different strategies to ensure performance and scalability. It adopts the Protocol Buffer [32] to compress the messages. As [31, 24] highlighted, Protocol Buffer can achieve a higher data compression and has a faster processing when compared to protocols such as XML and JSON. `JCL-HashMap` also implements pre-fetching techniques to improve its performance and it allows the execution of a task in a specific device of the cluster or a task to handle multiple method invocations, the last possible option using the "executeAll" or "executeAllCores" API alternatives.

Additionally, when loops occur and JCL API is called many times, JCL groups successive "execute" calls and submits them in chunks to `JCL-Hosts` to improve performance. However, there are the "executeAll" and "executeAllCores" methods, as mentioned before, allowing multiple executions of an object with identical or different arguments in many devices. The "executeAll" and "executeAllCores" methods are by far faster than several "execute" calls due to less data transmission and, consequently, less communication overhead.

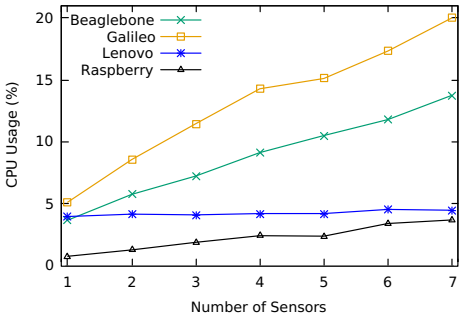

FIGURE 23 `JCL-Host` CPU usage to publish sensing data periodically from several sensors

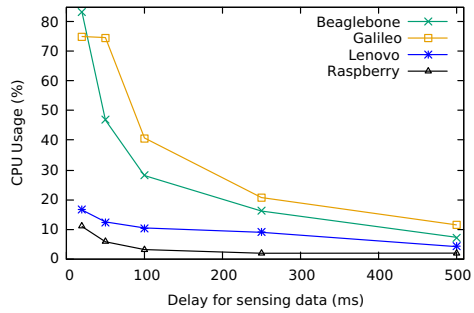

FIGURE 24 `JCL-Host` CPU usage to publish sensing data according to different intervals

To evaluate the scalability of JCL, we test it regarding CPU usage, memory consumption and battery usage. In theses experiments, a single `JCL-Host` component was evaluated in two scenarios: i) when the number of sensors was increased; and ii) when the sensing data interval was reduced.

Figure 23 illustrates the CPU usage of four devices. Each device was tested with seven simple sensors. The interval for publishing sensing data was fixed at 500 milliseconds. The only processing task of each `JCL-Host` in this experiment is to publish sensing data. The `JCL-Host` component enables multi-core computing, so it has a JCL multi-thread version inside, as mentioned before. In general, JCL maintains the CPU usage at less than 20% while handling seven simple sensors, being its best results with the Raspberry Pi and Lenovo phone, where CPU usage maintained at less than 5%. The concurrent sensing services could be handled by a pool of threads, by default configured to start as many threads as the number cores in the device. The evaluations increased the number of sensors in seven times, but the CPU consumption increased only two to four times, so JCL compensates the adoption of multiple sensing services together. We could not collect the CPU usage of an Arduino device and there is no thread-pool inside it because of hardware limitations; however it published sensing data from seven simple sensors very easily. The `JCL-Host` for Arduino is a C++ sequential code without protocol buffer and other performance improvements. The standard deviation of the CPU usage experiments varied from 0.09% to less than 5%, demonstrating that the results are consistent.

The second scenario evaluated `JCL-Host` with three simple sensors, varying the interval time to publish sensing data from them (Figure 24). The right most values, where interval is set to 500 milliseconds, are identical to those of the previous experiments, so now JCL increased the number MRAA library calls or native calls to basically acquire sensing data from sensors. Standard deviation for this experiment remained low, varying between 0.1% and 6.6%.

Above 100 milliseconds of interval, all devices reduced their CPU usage drastically. Near 20 milliseconds of interval, devices like Beaglebone and Galileo consumed almost 80% of CPU usage, but when the interval is above 100 milliseconds the CPU usage dropped to almost 40%, while in devices like Android Lenovo and Raspberry the CPU usage varied more linearly, being the extreme scenario with only 20% of CPU usage. This experiment demonstrated that short periods for sensing degraded more the CPU usage than concurrent sensing data acquisition calls because threads are executed more often. Furthermore, some sink nodes, like Galileo and Beaglebone, cannot manage many sensors and cannot deal with short sensing intervals, so they are useful only for sensing and actuating services, but others, like Android phones or Raspberry Pi single-board computers, can do also processing and storage services while performing sensing and actuating ones.

The memory consumption experiments are illustrated by Figure 25 in KB and by Figure 26 in MB. In Figure 25 the

sensor node device category is evaluated and in Figure 26 the low-end and sink node device categories are evaluated. It is important to reinforce that no sensing data were stored locally in the devices. `JCL-Host` for Arduino (Figure 25) had a low and linear memory consumption increase as the number of sensors increased, requiring only 2.5KB of memory in a device with 8KB SRAM memory capacity, so more than seven sensors can be easily handled by Arduinos. This occurs because memory is dynamically allocated for new sensors and because it is a sequential code without threads, since Arduino cannot deal with them. Also, in Figure 26, we had a stable memory consumption for the seven simple sensors in more powerful device categories, being justified by the similarity of the started threads in `JCL-Host`, since they performed the same logical steps, requiring the same dependencies. Standard deviation variations were close to 0% in these experiments, so they can be considered conclusive.

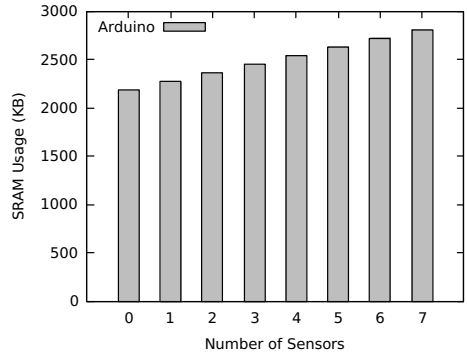

FIGURE 25 Arduino memory usage to publish sensing data periodically from several sensors

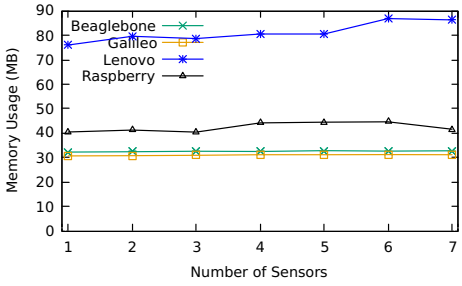

FIGURE 26 `JCL-Host` memory usage to publish sensing data periodically from several sensors

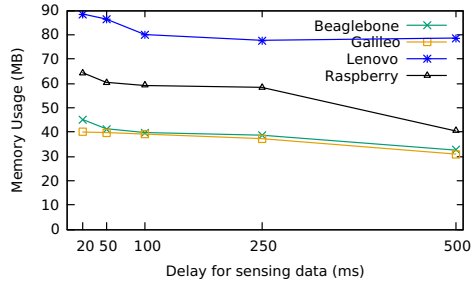

FIGURE 27 `JCL-Host` memory consumption to publish sensing data according to different intervals

In terms of memory consumption, we also tested the interval impact, by fixing the number of sensors in three simple ones and varying the interval time from 20 to 500 milliseconds, as illustrated in Figure 27. In contrast with CPU usage results, we can observe that the interval variation for sensing data did not cause much impact in memory usage. This occurred because no extra memory allocations were necessary to deal with smaller intervals. In summary, Arduino maintained in 2.5KB the memory usage with three sensors, and the other higher device categories maintained the memory usage between 40MB and 100MB. Standard deviation was also close to 0% in this experiment.

We made experiments to observe how special sensors impact the CPU usage and memory consumption. We

evaluated the special sensor camera with the interval configured in 500 milliseconds to take each photo with the best resolution available in the device, precisely 4160x3120 in Lenovo phone and 2592x1944 in the Raspberry using a RaspiCam. JCL compresses the images, preserving their resolutions, to generate less data to be transmitted, but also to be able to deal with low interval rates. As a baseline, we first configured three simple sensors publishing sensing data at 500ms interval, similarly as in Figures 23 and 26. However, the fourth configured sensor was the camera instead of a simple sensor. When the camera was configured in a Raspberry Pi the CPU usage surpassed 12%, while memory consumption almost duplicated, consuming close to 85MB. The Lenovo phone had the smaller overheads when the camera was introduced, precisely CPU usage reached 9% and the memory consumption was close to 90MB. The experiments demonstrated that CPU usage and memory consumption increased significantly when a single camera was introduced, mainly on small devices as Table 4 presents. This is explained because processing large amounts of data, such as photos, requires the allocation of more resources, thus causing a more remarkable impact in smaller devices. The average standard deviation on this experiment was 8.18%, so the results were consistent.

TABLE 4 `JCL-Host` battery and memory consumption with a camera sensor configured

Device \ Description	CPU (%)	Memory (MB)
Lenovo	9.031	89.882
Raspberry	12.318	84.013

TABLE 5 `JCL-Host` battery consumption to publish sensing data periodically from 7 sensors

# of Sensors \ Description	Power consumption (mA)	Standard Deviation (%)
1	2.540	0.105
2	3.396	0.145
3	3.984	0.139
4	4.553	0.215
5	6.193	0.532
6	6.671	0.601
7	6.833	0.613

The only device category that supports mobility in our cluster, e.g., with wireless communications and battery autonomy, is Android; therefore, we evaluated the battery usage of JCL over the same Lenovo phone model. The same seven simple sensors per device, as well as the sensing intervals, were considered in this experiment. We uninstalled all possible applications of the Android devices to perform the battery experiments as an alternative to get realistic milliampere values.

Table 5 illustrates a linear behavior of battery consumption in the presence of simple sensors, i.e., the number of sensors increased seven times while battery consumption increased near three times. The interval reduction, in contrast, affected battery usage substantially, as Table 6 illustrates, but it was expected, since the CPU usage results demonstrated that interval reduction degraded more than concurrent sensing services executions. Precisely, the battery

consumption increased five times, but the interval was reduces twenty five times, thus even in those extreme scenarios JCL proved to be a feasible IoT middleware alternative.

TABLE 6 `JCL-Host` battery consumption to publish sensing data according to different intervals

Description / Interval (ms)	Power consumption (mA)	Standard Deviation (%)
20	15.62	1.614
50	12.9	0.357
100	10.338	0.758
250	6.472	0.284
500	3.984	0.139

4.3 | Developing an IoT-HPC application in JCL

To evaluate JCL in terms of simplicity and usefulness, a JCL IoT-HPC case study with a single JCL Java class and less than 25 lines of code was developed. Figure 28 illustrates how the application works. It is responsible for observing a moving object, e.g., a pet, bike, person or even a phone, inside a previously defined location and when the object is outside of the location, a context is reached, and thus actions are started.

In the case study, a general-purpose task, developed from an existing code in the Internet[3], to send an email to a Gmail mail server starts. Next, a task to turn-on a lighting alarm also starts. Figure 29 presents this IoT-HPC application, which can be downloaded at the JCL website[4]. Precisely, at lines 5-6 the code obtains two devices from a JCL cluster and at lines 7-8 the devices are configured programmatically. Two sensor references, a led and a GPS, are retrieved at lines 9-10 and a context is configured at line 12. The action to turn the led on is configured at line 13 and a general-purpose email task is configured to run at lines 14-17. The code `JCLAppl` synchronizes until the GPS context is reached at line 18 of Figure 29. This kind of IoT-HPC development is very familiar for Java multi-thread community, but also for Android ones, thus we understand JCL is very simple for deployment and for the development of portable code.

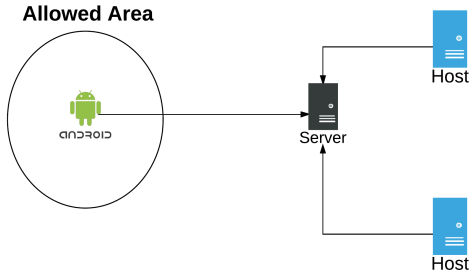

FIGURE 28 The Moving Object Observer application

[3] `https://www.tutorialspoint.com/java/java_sending_email.htm`
[4] http://www.javacaela.org

```
1  public class JCLAppl {
2    public static void main(String[] args) {
3       int numRecords = 1000; int delay = 2000; Object on[] = {1};
4       JCL_IoTfacade jcl = JCL_IoTFacadeImpl.getInstance();
5       Device raspberry = jcl.getDeviceByName("raspberry").get(0);
6       Device androidGPS = jcl.getDeviceByName("android").get(0);
7       jcl.setSensorMetadata(androidGPS, "TYPE_GPS", Constants.IoT.TYPE_GPS,
            numRecords, delay, Constants.IoT.INPUT, Constants.IoT.GENERIC);
8       jcl.setSensorMetadata(raspberry, "TYPE_LED", 7, numRecords, delay,
            Constants.IoT.OUTPUT, Constants.IoT.GENERIC);
9       Sensor led = jcl.getSensorByName(raspberry, "TYPE_LED").get(0);
10      Sensor gps = jcl.getSensorByName(androidGPS, "TYPE_GPS").get(0);
11      String contextNickname = "gpsContext";
12      jcl.registerContext(androidGPS, gps, new JCL_Expression("S0<-20.396107;
            S1<51.0468"), contextNickname);
13      jcl.addContextAction(contextNickname, raspberry, led, on);
14      File[] dependencies = {new File("lib/emailSender.jar"), new
            File("lib/javax.mail-1.5.0.jar")};
15      jcl.PacuHPC.register(dependencies, "EmailSender");
16      Object[] args = {"Object is outside the location", "A photo of the location is
            attached."};
17      Future result = jcl.addContextAction(contextNickname, false, "EmailSender",
            "sendEmail", args);
18      result.get();
19         ...
20    }
21  }
```

FIGURE 29 Moving Object Observer IoT Application Code

5 | CONCLUSION

This work presents a Java/Android IoT-HPC middleware solution, called JCL, which implements an alternative solution to attenuate the gap between IoT and HPC technologies. It gathers in a single API several HPC and IoT requirements, such as device control, security, and context awareness. In terms of heterogeneity, JCL enables Java and Android applications to run over Java, Android and Arduino devices. The `JCL-Super-Peer` component also attenuates heterogeneity, and in terms of interoperability, JCL integrates with MQTT brokers.

In terms of device control, all IoT related works support only the event-based programming model using standard IoT brokers, like MQTT ones. This way, they are not designed for HPC requirements where, for instance, DSM and task-oriented programming models are more suitable to control other device categories. In terms of security, JCL introduces a unique security level for its components communications, activated programmatically using its API, so a method call or a sensing data acquisition, for instance, can occur safely. A context can be inserted or removed programmatically in the API, using the task-oriented programming model, so very familiar to Java thread developers.

Experiments demonstrated that JCL scaled well when the number of sensors and devices increased. The same

promising behavior was observed when the interval time for sensing was reduced. Several UAs simulated various scenarios where concurrency was highlighted and JCL supported well all of them. The JCL-Super-Peer overhead was minimal, but Java-Android conversion costs were sometimes high, so customized JCL-Host deployments might be mandatory. Comparative experiments reinforced that JCL is a feasible Java middleware alternative for an integrated IoT-HPC, but the TCP protocol must be changed to MQTT to transmit small amount of sensing data, since MQTT is more efficient.

As future work we intent to include support to other communication protocols, such as Bluetooth and LPWAN. Fault tolerance in storage, processing and sensing must be inserted. A dashboard to observe a JCL cluster health is fundamental. New API services to enable, for instance, the construction of pipelines running sequentially or in parallel over a cluster can be useful when developing subscribers. Better scheduling strategies, binders to other programming languages, and a JCL-Host for both PIC microcontrollers and IoS platform are part of JCL's future plans. Support to multiple GPUs in JCL will enrich its HPC API services. Useful sensing analytical services, like OLAPing and Mining, should be introduced in JCL. Graph processing and storage services are also interesting areas to extend JCL. A new set of HPC experiments considering Android and single-board computers, like Raspberry, must be conducted to evaluate how these small device categories scale while performing services of JCL HPC API, such as `execute`, `executeAll` and `instantiateGlobalVariable`.

ACKNOWLEDGEMENTS

We thank UFOP and IFMG for the infrastructure. A. L. L. Aquino gratefully acknowledges his research support from FAPEAL, FAPESP, and CNPq.

REFERENCES

[1] Aberer K, Hauswirth M, Salehi A. A Middleware for Fast and Flexible Sensor Network Deployment. In: International Conference on Very Large Data Bases; 2006. p. 1199–1202.

[2] Al-Jaroodi J, Mohamed N. Middleware is STILL Everywhere!!! Concurr Comput : Pract Exper 2012 Nov;24(16):1919–1926.

[3] Almeida ALB, Silva SED, Jr ACN, de Castro Lima J. JCL: A High Performance Computing Java Middleware. In: International Conference on Enterprise Information Systems; 2016. p. 379–390.

[4] Amazon Web Services, Amazon Kinesis; 2017. Available: https://aws.amazon.com/kinesis [Accessed July 12, 2017].

[5] Androutsellis-Theotokis S, Spinellis D. A survey of peer-to-peer content distribution technologies. ACM computing surveys 2004;36(4):335–371.

[6] Baduel L, Baude F, Caromel D. Object-oriented SPMD. In: IEEE International Symposium on Cluster Computing and the Grid; 2005. p. 824–831.

[7] Baker M, Carpenter B, Shafi A. MPJ Express: towards thread safe Java HPC. In: International Conference on Cluster Computing; 2006. p. 1–10.

[8] Bandyopadhyay S, Sengupta M, Maiti S, Dutta S. A survey of middleware for internet of things. In: Recent Trends in Wireless and Mobile Networks; 2011.p. 288–296.

[9] Bazzani M, Conzon D, Scalera A, Spirito MA, Trainito CI. Enabling the IoT paradigm in e-health solutions through the VIRTUS middleware. In: IEEE International Conference on Trust, Security and Privacy in Computing and Communications; 2012. p. 1954–1959.

[10] Bellare M, Canetti R, Krawczyk H. HMAC: Keyed-hashing for message authentication. Internet Request for Comment 1997;p. 1–11.

[11] Botta A, De Donato W, Persico V, Pescapé A. Integration of cloud computing and internet of things: a survey. Future Generation Computer Systems 2016;56:684–700.

[12] Brynjolfsson EM. Big Data: The Management Revolution. Harvard Business Review 2012;90(10):60–66.

[13] Calbimonte JP, Sarni S, Eberle J, Aberer K. XGSN: An open-source semantic sensing middleware for the web of things. In: International Workshop on Semantic Sensor Networks; 2014. p. 51–66.

[14] Caro RJ, Garrido D, Plaza P, Roman R, Sanz N, Serrano JL. SMEPP: A Secure Middleware for Embedded P2P. In: ICT Mobile and Wireless Communications Summit; 2009. p. 1–8.

[15] Cimino LdS, d Resende JEE, Silva LHM, Rocha SQS, d O Correia M, Monteiro GS, et al. IoT and HPC Integration: Revision and Perspectives. In: Brazilian Symposium on Computing Systems Engineering; 2017. p. 132–139.

[16] CloudPlugs Inc , CloudPlugs; 2017. Available: https://cloudplugs.com [Accessed July 15, 2017].

[17] Cohen L, Java Parallel Processing Framework; 2017. Available: http://www.jppf.org [Accessed July 10, 2017].

[18] Compton M, Barnaghi P, Bermudez L, GarcíA-Castro R, Corcho O, Cox S, et al. The SSN ontology of the W3C semantic sensor network incubator group. Web Semantics: Science, Services and Agents on the World Wide Web 2012;17:25–32.

[19] Daemen J, Rijmen V. The Design of Rijndael. Berlin, Germany: Springer; 2002.

[20] Diaz J, Munoz-Caro C, Nino A. A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on parallel and distributed systems 2012;23(8):1369–1386.

[21] Eclipse, Mosquitto Broker; 2017. Available: https://mosquitto.org/ [Accessed June 11, 2017].

[22] Eisenhauer M, Rosengren P, Antolin P. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In: IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks; 2009. p. 1–3.

[23] El Baz D. IoT and the need for High Performance Computing. In: International Conference on Identification, Information and Knowledge in the Internet of Things; 2014. p. 1–6.

[24] Emeakaroha VC, Healy P, Fatema K, Morrison JP. Analysis of data interchange formats for interoperable and efficient data communication in clouds. In: International Conference on Utility and Cloud Computing; 2013. p. 393–398.

[25] Engelmann C, Ong H, Scott S. Middleware in modern high performance computing system architectures. In: International Conference on Computational Science; 2007. p. 784–791.

[26] Eugster PT, Felber PA, Guerraoui R, Kermarrec AM. The many faces of publish/subscribe. ACM computing surveys 2003;35(2):114–131.

[27] Forum MP. MPI: A Message-Passing Interface Standard. University of Tennessee; 1994.

[28] Gámez N, Fuentes L. FamiWare: a family of event-based middleware for ambient intelligence. Personal and Ubiquitous Computing 2011;15(4):329–339.

[29] Genaud S, Rattanapoka C. P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids. Journal of Grid Computing 2007;5(1):27–42.

[30] Gianluca Barbaro, Zanzito; 2017. Available: http://www.barbaro.it/cms/index.php/it/android/zanzito [Accessed June 21, 2017].

[31] Gligorić N, Dejanović I, Krčo S. Performance evaluation of compact binary XML representation for constrained devices. In: International Conference on Distributed Computing in Sensor Systems and Workshops; 2011. p. 1–5.

[32] Google, Protocol Buffers; 2015. Available: https://developers.google.com/protocol-buffers [Accessed June 20, 2017].

[33] Google, Android SDK DX Tools; 2017. Available: https://developer.android.com/studio/command-line/index.html [Accessed June 27, 2017].

[34] Google, Google Analytics; 2017. Available: https://analytics.google.com/kinesis [Accessed July 12, 2017].

[35] Graham RL, Shipman GM, Barrett BW, Castain RH, Bosilca G, Lumsdaine A. Open MPI: A High-Performance, Heterogeneous MPI. In: International Conference on Cluster Computing; 2006. p. 1–9.

[36] GridGain Systems, Inc, GridGain; 2017. Available: http://www.gridgain.com [Accessed July 07, 2017].

[37] Hazelcast Inc , Hazelcast; 2017. Available: https://hazelcast.com/ [Accessed July 10, 2017].

[38] Infinite Automation Systems, Mango; 2017. Available: http://infiniteautomation.com [Accessed May 10, 2017].

[39] Intel IoT DevKit, Libmraa; 2017. `http://iotdk.intel.com/docs/master/mraa/`.

[40] KaaIoT Technologies, Kaa IoT Development Platform; 2017. Available: https://www.kaaproject.org [Accessed July 12, 2017].

[41] Kaminsky A. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In: IEEE International Parallel and Distributed Processing Symposium; 2007. p. 1–8.

[42] Kim M, Lee JW, Lee YJ, Ryou JC. Cosmos: A middleware for integrated data processing over heterogeneous sensor networks. Electronics and Telecommunications Research Institute 2008;30(5):696–706.

[43] Kurzyniec D, Wrzosek T, Sunderam V, Slominski A. RMIX: A multiprotocol RMI framework for Java. In: Parallel and Distributed Processing Symposium; 2003. p. 6.

[44] Levis P, Culler D. Maté: A tiny virtual machine for sensor networks. ACM Sigplan Notices 2002;37(10):85–95.

[45] Li S, Son SH, Stankovic JA. Event detection services using data service middleware in distributed sensor networks. In: International Conference on Information Processing in Sensor Networks; 2003. p. 351–368.

[46] Liu T, Martonosi M. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. ACM SIGPLAN Notices 2003;38(10):107–118.

[47] LogMeIn, Xively IoT Platform; 2017. Available: https://www.xively.com [Accessed July 12, 2017].

[48] LORD MicroStrain, SensorCloud; 2017. Available: http://www.sensorcloud.com [Accessed July 15, 2017].

[49] Lua EK, Crowcroft J, Pias M, Sharma R, Lim S. A survey and comparison of peer-to-peer overlay network schemes. Communications Surveys & Tutorials, IEEE 2005;7(2):72–93.

[50] myDevices, The Cayenne Project; 2017. Available: https://mydevices.com [Accessed June 25, 2017].

[51] Nimbits Inc , Nimbits Platform; 2017. Available: https://nimbits.com [Accessed July 17, 2017].

[52] Oasis, eXtensible Access Control Markup Language; 2017. Available: https://www.oasis-open.org/committees/xacml [Accessed July 02, 2017].

[53] OASIS Standard, MQTT;. Available: http://mqtt.org/ [Accessed June 24, 2017].

[54] O'Hare GM, Muldoon C, O'Grady MJ, Collier RW, Murdoch O, Carr D. Sensor web interaction. International Journal on Artificial Intelligence Tools 2012;21(02):1240006.

[55] Oracle Corporation, Oracle Coherence; 2017. Available: http://coherence.java.net/ [Accessed July 12, 2017].

[56] OSGi Alliance, OSGi; 2017. Available: https://www.osgi.org/ [Accessed July 11, 2017].

[57] Pan B, Dex2Jar; 2017. Available: https://github.com/pxb1988/dex2jar [Accessed June 17, 2017].

[58] Patel DK, Tripathy D, Tripathy C. Survey of load balancing techniques for Grid. Journal of Network and Computer Applications 2016;65:103–119.

[59] Perera C, Talagala DS, Liu CH, Estrella JC. Energy-Efficient Location and Activity-Aware On-Demand Mobile Distributed Sensing Platform for Sensing as a Service in IoT Clouds. IEEE Transactions on Computational Social Systems 2015;2(4):171–181.

[60] Perera C, Jayaraman PP, Zaslavsky A, Georgakopoulos D, Christen P. MOSDEN: An internet of things middleware for resource constrained mobile devices. In: Hawaii International Conference on System Sciences; 2014. p. 1053–1062.

[61] Perera C, Liu CH, Jayawardena S, Chen M. A Survey on Internet of Things From Industrial Market Perspective. IEEE Access 2014;2:1660–1679.

[62] Perera C, Zaslavsky A, Christen P, Georgakopoulos D. Context aware computing for the internet of things: A survey. Communications Surveys & Tutorials 2014;16(1):414–454.

[63] Philippsen M, Haumacher B, Nester C. More efficient serialization and RMI for Java. Concurrency: Practice and Experience 2000;12(7):495–518.

[64] Pitt E, McNiff K. Java.Rmi: The Remote Method Invocation Guide. 1 ed. Harlow, England: Addison-Wesley; 2001.

[65] Protic J, Tomasevic M, Milutinovic V. Distributed shared memory: Concepts and systems. IEEE Parallel & Distributed Technology: Systems & Applications 1996;4(2):63–71.

[66] Pugh W, Spacco J. Mpjava: High-performance message passing in java using java.nio. In: International Workshop on Languages and Compilers for Parallel Computing; 2003. p. 323–339.

[67] Razzaque MA, Milojevic-Jevric M, Palade A, Clarke S. Middleware for internet of things: a survey. IEEE Internet of Things Journal 2016;3(1):70–95.

[68] RedHat, Drools; 2017. Available: http://www.drools.org [Accessed July 12, 2017].

[69] RedHat, Infinispan; 2017. Available: http://infinispan.org/ [Accessed July 13, 2017].

[70] RSA Laboratories East, PKCS #7: Cryptographic Message Syntax; 1998. Available: https://tools.ietf.org/html/rfc2315 [Accessed June 28, 2017].

[71] Shahrivari S, Sharifi M. Task-oriented programming: A suitable programming model for multicore and distributed systems. In: International Symposium on Parallel and Distributed Computing; 2011. p. 139–144.

[72] Sheng QZ, Benatallah B. ContextUML: a UML-based modeling language for model-driven development of context-aware Web services. In: International Conference on Mobile Business; 2005. p. 206–212.

[73] Sheng X, Tang J, Xiao X, Xue G. Sensing as a service: Challenges, solutions and future directions. IEEE Sensors journal 2013;13(10):3733–3741.

[74] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In: Symposium on Mass storage systems and technologies; 2010. p. 1–10.

[75] Smart & Cloud Architectures and Platforms, The SOFIA2 Project; 2017. Available: http://sofia2.com [Accessed July 05, 2017].

[76] Soldatos J, Kefalakis N, Hauswirth Mea. OpenIoT: Open Source Internet-of-Things in the Cloud. In: Interoperability and Open-Source Solutions for the Internet of Things - International Workshop; 2014. p. 13–25.

[77] Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R. Java in the High Performance Computing arena: Research, practice and experience. Science of Computer Programming 2013;78(5):425–444.

[78] Taboada GL, Touriño J, Doallo R. F-MPJ: Scalable Java message-passing communications on parallel systems. The Journal of Supercomputing 2012;60(1):117–140.

[79] Taveira WF, de Oliveira Valente MT, da Silva Bigonha MA, da Silva Bigonha R. Asynchronous remote method invocation in java. Journal of Universal Computer Science 2003;9(8):761–775.

[80] The Apache Software Foundation, Apache Cassandra; 2016. Available: http://cassandra.apache.org [Accessed July 13, 2017].

[81] The Apache Software Foundation, Apache Ignite; 2017. Available: https://ignite.apache.org [Accessed July 11, 2017].

[82] The Apache Software Foundation, Apache Spark; 2017. Available: http://spark.apache.org [Accessed July 07, 2017].

[83] The Mathworks Inc , MATLAB; 2017. Available: https://mathworks.com/matlab [Accessed July 11, 2017].

[84] The MathWorks Inc , ThingSpeak; 2017. Available: https://thingspeak.com [Accessed July 16, 2017].

[85] Ville LS, Dickman P. Garnet: a middleware architecture for distributing data streams originating in wireless sensor networks. In: International Conference on Distributed Computing Systems Workshops; 2003. p. 235–240.

[86] Walker S, Dearle A, Norcross S, Kirby G, McCarthy A. RAFDA: A policy-aware middleware supporting the flexible separation of application logic from distribution. University of St Andrews; 2006.

[87] Wang K, Liu N, Sadooghi I, Yang X, Zhou X, Li T, et al. Overcoming hadoop scaling limitations through distributed task execution. In: International Conference on Cluster Computing; 2015. p. 236–245.

[88] ZeroC Inc , Ice - Comprehensive RPC Framework; 2017. Available: https://zeroc.com/products/ice [Accessed July 03, 2017].

[89] Zhang Q, Cheng L, Boutaba R. Cloud computing: State-of-the-art and research challenges. Journal of internet services and applications 2010;1(1):7–18.

[90] Zhu W, Wang CL, Lau FCM. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. In: International Conference on Cluster Computing; 2002. p. 381–388.