A Service-Oriented Middleware Enabling Decentralised Deployment in Mobile Multihop Networks

Luc Hogie

HAL Id: hal-03886521
https://inria.hal.science/hal-03886521

Submitted on 6 Dec 2022

A Service-Oriented Middleware Enabling Decentralised Deployment in Mobile Multihop Networks

Luc Hogie[1,2,3][0000−0001−8044−5672]

[1] CNRS (Centre National de la Recherche Scientifique)
[2] I3S laboratory Université Côte d'Azur
[3] Inria Sophia Antipolis, France
luc.hogie@cnrs.fr

Abstract. The number of computing devices, mostly smartphones is tremendous. The potential for distributed computing on them is no less huge. But developing applications for such networks is challenging especially as most middleware solutions for distributed computing are tailored to managed grids and clusters, so they lacks the elasticity needed to deal with the difficult conditions brought by multi-hops, mobility, heterogeneity, untrustability, etc. To solve this, several middleware were released, but none of them feature workable deployment solutions. This paper presents the deployment service of the IDAWI middleware, which implements a fully decentralized and automatised deployment strategy into a Open Source middleware tailored to enabling distributed computing in difficult networking conditions like in the IoT/fog/edge.

Keywords: Decentralised systems, overlay networks, middleware, deployment, IoT, Java

Today smartphones are the most common computing devices. Billions of them have been sold all over the world. They all feature network interfaces to communicate through 4G/5G, Wi-Fi and Bluetooth. Broadband 4G/5G networks and Wi-Fi access points enable them to have direct connections to the Internet, while the Wi-Fi ad hoc mode and Bluetooth enable them to have direct P2P connections to *each other*, and to talk to other devices of the IoT. The ubiquity, capabilities, and communication abilities of smartphones make their population a computing platform of a paramount importance. Unfortunately, existing middleware to distributed computing can hardly be used in this context. Indeed, many such middleware dedicated to general distributed computing, like ProActive [8], MPI, JXTA, JMS (Jakarta Messaging API), ActiveMQ [13], 0MQ, JGroups, RMI, and Akka have been proposed through the years. Because they are most often tailored to grid/cloud/cluster environments, they do not accommodate strong heterogeneity, node mobility, etc. that is found in networks involving mobile devices, such as the IoT, the fog, etc. More flexible models and elastic tools were proposed by Researchers, like JavaCà&Là (JCL) [23], GoPrime [7], ParallelTheater [21], ActorEdge [2], and EmbJXTAChord [3], but in spite

of their numerous good features, they often are Research tools designed to solve particular scientific problems, which make them hardly usable out-of-the-box in projects involving practical distributed computations. Adapting one of them would be a cumbersome work that has no guarantee of success as source codes, when they are fully available, are always very hard to embrace. Anyway, there is no consensus about which middleware would be the right candidate to start that work from.

Further, we found out that no tool was geared to experimentation, which has specific requirements. In particular, no single tool support the *trial and errors* working process that is usually employed to design and tune an (distributed) algorithm. This consists in doing countless small adjustments in the source code and running many new executions to see their impact. To do that, the middleware needs to be able to deploy and bootstrap distributed applications very quickly, and it needs to provide an easy way to deal at runtime with remote executions.

This article introduces a deployment strategy for Java application, which operates even in the difficult networking conditions of the mobile multihop networks of smartphones, and the IoT. It relies on the IDAWI middleware [18], [17], whose the aspects crucial to deployment are first presented here.

Our initial motivation originates from applied Research projects conducted at I3S/Inria, whose goals were to provide effective solutions to practical distributed problems related to graph algorithms [25], networking [10], decentralised protocols [16] for MANETs and more recently the IoT, etc.

1   The Idawi middleware

IDAWI is a middleware for distributed computing. It gathers (and improves when possible) the good features of existing tools in a comprehensive fresh one, and it goes beyond by proposing effective solutions to problems not tackled by existing tools. In a few words, on top of its *fully decentralised architecture*, it proposes a SOA-like application model. Its design is mixed: it uses elements of object/message/queue/component/service-based models wherever they proved appropriate. It proposes elastic *collective* and *asynchronous* communication and computation models, augmented with facilities for synchronous calls. IDAWI is Open Source, and can be found at:

<div align="center">

`https://i3s.univ-cotedazur.fr/~hogie/idawi/`

</div>

We describe in the following the elements of IDAWI that are essential to the development of our deployment solution. A good understanding of them is required to embrace the potential of the deployment strategy.

1.1   An overlay network of components

IDAWI defines a component model. Components represent business entities. They self-organize as a multihop overlay network. In this overlay, two given components are neighbours if they have direct interactions. Any two components can

be neighbours unless the underlying network infrastructure prevents it. This may happen in the presence of NATs/firewalls, or because of inherent constraints of wireless technologies such as their limited range, hidden nodes, etc. Two non-neighbours must then rely on intermediary nodes, which then behave as *routers*.

In the usual use case, there will be only one component per device. The role of this component is then to represent the device that hosts it. But in order to facilitate experimentation on large systems, components can deploy other components in their JVM or in another JVM in the same device. Emulation can then be achieved by having more components than devices (then some device host multiple components).

The default routing protocol defines a destination address as a triplet (C, e, d) where C is descriptor matcher (if C is not defined, the address is considered to be a broadcast address); e is the expiration date of the message, and d is the maximum number of hops allowed to travel. This routing protocol is intrinsically multicast/broadcast, which suits the very nature of dynamic multihop networks. *unicast* comes naturally when one single target component is specified in the set of recipient names. In order to address node mobility and scarce connectivity, messages are not dropped after they are forwarded. Instead they are stored into the routing service's internal tables until they expire. Each time a new neighbour component pops up, stored messages are reconsidered for re-emission.

1.2   An application model in the style of SOA

IDAWI proposes a *structuring model* of distributed applications, meaning that applications must conform to a certain organisation defined by the Object-Oriented model (OO). This ensures *consistency* of application source codes, severely reduces the risk of design errors (as most design work is in the middleware), and enables the development of high-level functionalities such as "deployment" which cannot be implemented if applications do not follow a standard pattern.

Component expose their functionality via *service*s. A *service* is an object within a host component. It holds data and implements functionality about the specific concern it is about. Also a service has queues which enable them to receive messages. Services are the standard way to incorporate functionality in an IDAWI system. An application is defined by a set of services. System-level functionality is also brought by specific builtin services like the *routing* service internally maintains routing tables and its public API enables other services (hence applications) to obtain topological information like distances between components, paths to reach them, etc.

In turn, services expose functionality via *operations*. An operation is a piece of code that can be triggered remotely from any other one in the system. Just like services, operations are identified by their class. Technically, an operation is described by an inner class of its service class. Its ID (class) then holds the ID of its host service. Operations constitute the *only way* to execute code in an IDAWI system. When the code of an operation is started, it is fed by an input queue of messages. This operation can start/feed new operations, as well as it can send messages to others (already running) operations.

The deployment functionality described in this article comes in the form of a specific service accessible to all other components/services in the component system. It has a particular *deploy* operation that can be triggered from anywhere in the network. As explained in Section 2, the deployment service is fully decentralised, meaning that a component does not need any exogenous element to deploy another component. This enables a component to pro-actively deploy new components where it is asked to do so, or it can be asked by another component to perform a deployment. In the latter case, the component requesting the deployment does not need to provide any information other than where the new component has to be deployed to.

1.3   A *many-to-many* message-based communication model

At the lowest layer, (running) operations communicate by *explicitly* sending/receiving messages of a bounded size. Sending a message is always an *asynchronous* (non-blocking) operation. It provides no guarantee of reception. A message has a probabilistically unique random 64-bit numerical ID. It carries a content (which can be anything), the target service/queue IDs, the route it took so far, and optional routing-specific information. When a message reaches its destination service, it is delivered into its target message queue. Queue are then fetched by operations, in a *synchronous* fashion.

A message queue is a thread-safe container of messages exposing the following primitives: $size()$ gets the number of messages currently in the queue; $get(timeout)$ retrieves and removes the first message in the queue, waiting until the timeout expires if the queue was empty; and $add(timeout)$ adds a message in the queue, waiting until the timeout expires if the queue was full. Using finite timeout ensures that no dead-locks will occur in the system.

IDAWI comes with a default routing protocol which suits the very nature of mobile multihop networks: it defines a destination address as a triplet (C, e, d) where C is set of component names (if C is not defined, the address is considered to be a broadcast address); e is the expiration date of the message, and d is the maximum number of hops allowed to travel. This routing protocol is intrinsically *multicast/broadcast*. *Unicast* comes naturally when one single target component is specified in the set of recipient names. In order to address node mobility and scarce connectivity, messages are not dropped after they are forwarded. Instead they are stored into the routing service's internal tables until they expire. Each time a new neighbour component pops up, stored messages are reconsidered for re-emission.

1.4   A collective computation model

IDAWI defines an innovative computing model, based atop the communication model described hereinbefore, from which it benefits the *collective* approach. It defines a special message called the *exec message*, whose the reception triggers the execution of a particular operation. This enables an operation to be executed in parallel on multiple components.

When it is executed, an operation is provided with a queue that it will use to receive input data. An operation is then able to receive multiple messages at runtime, constituting *unbounded input*. An operation can produce output (intermediary results, final result, warnings, exceptions, progress information, etc) at any time by sending messages. In most cases, output will be sent to the sender of the exec message. To receive output messages, the caller creates a new local queue, called the *return queue* that aims at storing messages from the running operation. This message queue can play the role of a *future*. Once again, other running operations may obtain the address of the return queue, and send *directly* messages to it. A running operation may execute another one. This enables *composition of services* and *workflows*.

From a programmatic point of view, The *exec*() primitive makes it easy the remote execution of operations. It takes as input the address of the operation to execute, an optional address of the return queue, as well optional initial input data. Just like sending a message (which it does behind the scene), calling *exec*() is asynchronous, but synchronicity can be achieved by invoking synchronous primitives on the return queue. Calling *exec*() then immediately returns a proxy to the remotely running operation. This proxy features the address of the (remote) input queue of the running operation, as well a reference to the aforementioned return queue.

If the operation address describes multiple components, the operation will be executed on all of them. In this case the caller may receive output messages from these multiple executions. To deal with that, the *collect* algorithm can be used to demultiplex messages according to where they come from.

This model allows the parallel execution of the *deploymentoperation* on multiple nodes: multiple nodes get deployed at the same time at different locations in the network.

2 Deployment and bootstrapping

In many distributed systems, the deployment process involves human work to provide deployment configuration files (which specify the location of binaries, their destination, the mean to deploy them, and how to bootstrap/stop remote elements), and to execute command-line packagers, remote shells, deployment scripts, etc. Generic deployment systems like DeployWare [14][20] target the deployment of any software, along with their complex dependencies, with support to multiple languages. But in fact the number of constraints/requirements/assumptions can be much more that that, which makes deployment hardly automatizable. This problem has motivated several research initiatives like [11][6][9][19]. In practise distributed systems generally come with their own *ad hoc* deployment sub-system, like in ProActive [1], Jade [5][20], OSGi [24], and Frogi [12], an OSGI-based solution to the deployment of Fractal [4] components. Modern solutions often resort to solutions from the Docker ecosystem. These enable deployment in centralized IoT and fog network configurations. But deployment in such scarse networks, as suggested by recent works on deploying applications

across unreliable networks foster the use of a decentralized technology like Bittorrent as a support to deployment [22]. Even closer to IDAWI's considerations, deployment in infrastructure-less mobile heterogeneous networks have been investigated in [15].

Deployment is a generic word that encompasses a wide variety of use cases. There can be deployment of executable code or data, deployment of virtual machines in a grid, deployment of docker images in the cloud, etc. IDAWI deploys Java runtimes: it *clones* the execution environnement (JVM and bytecode) to remote hosts, and it bootstraps a new component in a new JVM on each of them. It is able to do that to any POSIX node providing an access via SSH. In other words, deployment in IDAWI lies in the ability of any component to deploy other components on remote nodes (but also in the same JVM or in a new JVM on the same local node). Deployment in IDAWI can be seen as a zero-configuration process. More precisely, a component does not need any exogenous element to perform a deployment: it has all the functionality to do it. Unlike most tools, IDAWI does not require any pre-installed library to be installed on target nodes, nor it requires any daemon to run. It only requires that an access to SSH is possible and that a few POSIX commands are available (`mkdir`, `rsync`).

Just after deployment of a JVM, a child component is started in it. Bidirectional communication between child and parent is achieved through the SSH connection that was used for the deployment. When this SSH connection breaks, the child component may keep running on its own, or may stop its execution. This is an application-specific policy.

The deployment functionality comes as a specific service available in every components. This service exposes operations that allow any component to request any other component to deploy new ones.

2.1   The problem of shared file systems

In a context of *trial and errors* workflow, deployment times are crucial for it has a direct impact on the developer quality of Life. Cloning a JVM to a single remote node is a significantly long process. It takes several seconds when best conditions are met (deploying to desktop computers on the LAN). Then performing a sequential deployment on multiple nodes is prohibitive, as deployment durations sum up. Thus, in order to fasten the deployment process, IDAWI resorts to parallelism: it deploys to multiple nodes simultaneously. In practise, deploying to n nodes has roughly the same duration as deploying to a single node. But such parallel deployment is problematic when target nodes share their filesystem. In particular, parallel uncontrolled writes to a file system often produce data inconsistencies. When the computing hardware infrastructure is managed and documented (which is the case usually in grids ands clusters), users know if nodes share or not the file system, and the deployment process can be configured on the basis of this information. But IDAWI is not restricted to these networks. In the target computing environment it was designed for, it must detect shared file systems.
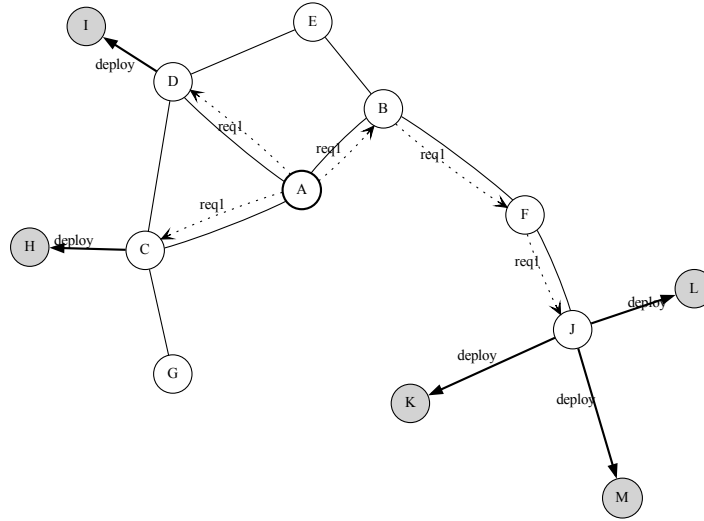
Fig. 1. Component A sends a *same* deployment request to res. nodes C, D and J, asking them to deploy resp. 1, 1 and 3 children components

To do this, IDAWI first executes a distributed algorithm that computes a partition of a set of computers by grouping those which share a same file-system. This algorithm, initiated by a node a uses SSH to create on every node (in parallel) a marker into their file system. This marker takes the shape of a directory, as the *mkdir* system call is the only atomic one (which implies that concurrent calls to mkdir are sequentialized). This mark directory is named against the DNS name of the node that executes mkdir). Mark directories are all created in a parent directory whose the name is the same and known by all. This parent directory is initially empty. This way, at the end of the marking process, every node in N will has in its file-system the marks of other nodes it shares the file-system with. This on-disk information is then retrieved in parallel by node a which computes a partition of the nodes according to file-system shares.

One single node is picked up at random from every set, resulting in a subset of the initial set of nodes. This new set of nodes exhibits the following properties:

- no two nodes share a file-system
- there exists a representative node for every file-system

Deploying code to every node in this set will make it available to all target nodes, avoiding any problem of concurrent access. The process is described in Algorithm 1.

Data: A set N of nodes
Result: a partition of the set of nodes, grouping nodes with a common file
           system
foreach *node $n \in N$* **do** in parallel
    | mark n;
end
let S be a set of set of nodes
foreach *node $n \in N$* **do** in parallel
    | retrieves markers from n into set M;
    | lock S;
    | **if** $M \notin S$ **then**
    | | $S \leftarrow S \cup M$;
    | **end**
    | unlock S;
end
foreach *node $n \in N$* **do** in parallel
    | remove all markers from n;
end

Algorithm 1: `sharedFSGroups()`: Determination of shared filesystems

2.2   Duplication of the binaries

Most of the time, deployment tools require the user to provide a configuration specifying the location of binaries and JVM specifications. IDAWI takes benefit of the ability of Java programs to discover them at runtime. Regardless of how the program was started (via the command line on a server, from an IDE, etc), a Java program knows its classpath, which lists the class directories and `jar` files constituting it. Thanks to system properties, it also knows the exact version of the JVM that is executing it. An IDAWI component then has a exhaustive view of its binaries, which its deployment service clone when its creates new remote nodes. As a consequence, children nodes end having the exact same binaries as their parent. This eliminates interoperability problems related to incompatible versions.

IDAWI is geared towards experimentation. This implies that IDAWI's users will use it in a particular way: most often in a *trials and errors* mode. This way of working involves many subsequent runs, each one with a slight variation of the source code or of its dependencies. It must be quick in order to have a minimal impact on the user experience.

Many deployment systems consider an image as the deployment unit. Such image can weight up to gigabytes of data and take long times to transfer. IDAWI takes a fine-grained approach. It takes benefit of the Java platform. The deployment unit in IDAWI is the class/resource file. In order to minimize the amount of data transferred, IDAWI resorts to synchronization instead of doing a complete copy: only what is not already there gets transferred. The process of deployment of bytecode and resources is then *incremental*. To do that, IDAWI performs the following steps:

1. in parallel:
   - retrieves a set R of s $(name, size, timestamp)$ for every remote file
   - builds such a set L for local files
2. transfers missing or out-of-date files in $L - R$ (out-of-date files will be over-written)
3. deletes file no longer in use in $R - L$

By doing this, the whole set of files constituting the executable code is transferred only once. This applies to both the byte code and the Java runtime. Subsequent transfers will consider only the differences from previous ones. As these differences are most of the time insignificant, both the amortised time and space complexity is constant.

Thanks to the determination of shared file systems described in section 2.1, IDAWI is able to transfer code to multiple computers *in parallel*. Transferring to one single node or to multiple ones is not much different in time.

2.3 Bootstrapping remote components

When the executable code has been deployed and the right JVM has been installed, the remote component is ready to be bootstrapped. To do that, IDAWI uses SSH to execute the previously-verified JVM on the child node. This JVM is bootstrapped with a specific *main* class that uses the SSH input/output to:

1. read information about the child component to be created in the new JVM
2. send an acknowledgement to its parent node to indicate that the child was successfully created and initialised

On the other side, the parent component:

1. sends deployment information for the component to be created in the new JVM
2. waits for an acknowledgement from its child node that was successfully created and initialised

As long as the SSH connection is open, the two components can use it to communicate with each other.

The entire deployment process is described in Algorithm 2.

3   Conclusion

The number and variety of computing devices is growing, in particular billions of smartphones have been sold all over the world in the past years, making the potential for distributed computing huge. Unfortunately most middleware for distributed computing is tailored to grids and clusters, so it lack the elasticity needed to accomodate the difficult conditions brought by the dynamics and heterogeneity of mobile networks of smartphones, MANETs, the fog, the IoT. For

Data: A set N of target nodes
$S \leftarrow sharedFSGroups(N)$ **foreach** *set of node* $s \in S$ **do** in parallel
    picks a random component c in s;
    synchronizes local binaries to c;
end
foreach *node* $n \in N$ **do** in parallel
    start a JVM on n;
    bootstrap a new component on n;
end

<div align="center">

Algorithm 2: `deploy()`: Full deployment process

</div>

these purposes, several software solutions were released, but none of them feature workable deployment solutions. This paper presents the deployment service of the IDAWI middleware, which implements a fully decentralised and automatised deployment strategy into an Open Source middleware tailored to enabling distributed computing in mobile heterogeneous networks.

References

1. Aloisio, G., Cafaro, M., Epicoco, I.: A grid software process. In: Cunha, J.C., Rana, O.F. (eds.) Grid Computing: Software Environments and Tools, pp. 75–98. Springer (2006). https://doi.org/10.1007/1-84628-339-6"4, https://doi.org/10.1007/1-84628-339-6_4

2. Aske, A., Zhao, X.: An actor-based framework for edge computing. In: Anjum, A., Sill, A., Fox, G.C., Chen, Y. (eds.) Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017. pp. 199–200. ACM (2017). https://doi.org/10.1145/3147213.3149214, https://doi.org/10.1145/3147213.3149214

3. Battaglia, F., Bello, L.L.: A novel jxta-based architecture for implementing heterogenous networks of things. Comput. Commun. **116**, 35–62 (2018). https://doi.org/10.1016/j.comcom.2017.11.002, https://doi.org/10.1016/j.comcom.2017.11.002

4. Blair, G.S., Coupaye, T., Stefani, J.: Component-based architecture: the fractal initiative. Ann. des Télécommunications **64**(1-2), 1–4 (2009). https://doi.org/10.1007/s12243-009-0086-1, https://doi.org/10.1007/s12243-009-0086-1

5. Bouchenak, S., Palma, N.D., Hagimont, D., Taton, C.: Autonomic management of clustered applications. In: Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain. IEEE Computer Society (2006). https://doi.org/10.1109/CLUSTR.2006.311842, https://doi.org/10.1109/CLUSTR.2006.311842

6. Cañete, A., Amor, M., Fuentes, L.: Supporting iot applications deployment on edge-based infrastructures using multi-layer feature models. J. Syst. Softw. **183**, 111086 (2022). https://doi.org/10.1016/j.jss.2021.111086, https://doi.org/10.1016/j.jss.2021.111086

7. Caporuscio, M., Grassi, V., Marzolla, M., Mirandola, R.: Goprime: A fully decentralized middleware for utility-aware service assembly. IEEE Trans. Soft-

ware Eng. **42**(2), 136–152 (2016). https://doi.org/10.1109/TSE.2015.2476797, https://doi.org/10.1109/TSE.2015.2476797

8. Caromel, D., di Costanzo, A., Mathieu, C.: Peer-to-peer for computational grids: mixing clusters and desktop machines. Parallel Comput. **33**(4-5), 275–288 (2007). https://doi.org/10.1016/j.parco.2007.02.011, https://doi.org/10.1016/j.parco.2007.02.011

9. Chen, Y., Sun, Y., Feng, T., Li, S.: A collaborative service deployment and application assignment method for regional edge computing enabled iot. IEEE Access **8**, 112659–112673 (2020). https://doi.org/10.1109/ACCESS.2020.3002813, https://doi.org/10.1109/ACCESS.2020.3002813

10. Coudert, D., Hogie, L., Lancin, A., Papadimitriou, D., Pérennes, S., Tahiri, I.: Feasibility study on distributed simulations of BGP. CoRR **abs/1304.4750** (2013), http://arxiv.org/abs/1304.4750

11. Dautov, R., Song, H., Ferry, N.: Towards a sustainable iot with last-mile software deployment. In: IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021. pp. 1–6. IEEE (2021). https://doi.org/10.1109/ISCC53001.2021.9631250, https://doi.org/10.1109/ISCC53001.2021.9631250

12. Desertot, M., Cervantes, H., Donsez, D.: Frogi: Fractal components deployment over osgi. In: Löwe, W., Südholt, M. (eds.) Software Composition - 5th International Symposium, SC@ETAPS 2006, Vienna, Austria, March 25-26, 2006, Revised Papers. Lecture Notes in Computer Science, vol. 4089, pp. 275–290. Springer (2006). https://doi.org/10.1007/11821946‟18, https://doi.org/10.1007/11821946_18

13. Estrada, N., Astudillo, H.: Comparing scalability of message queue system: Zeromq vs rabbitmq. In: 2015 Latin American Computing Conference, CLEI 2015, Arequipa, Peru, October 19-23, 2015. pp. 1–6. IEEE (2015). https://doi.org/10.1109/CLEI.2015.7360036, https://doi.org/10.1109/CLEI.2015.7360036

14. Flissi, A., Dubus, J., Dolet, N., Merle, P.: Deploying on the grid with deployware. In: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), 19-22 May 2008, Lyon, France. pp. 177–184. IEEE Computer Society (2008). https://doi.org/10.1109/CCGRID.2008.59, https://doi.org/10.1109/CCGRID.2008.59

15. Guidec, F.: Déploiement et support à l'exécution de services communicants dans les environnements d'informatique ambiante. Habilitation à diriger des recherches, Université de Bretagne Sud ; Université Européenne de Bretagne (Jun 2008), https://tel.archives-ouvertes.fr/tel-00340426

16. Hogie, L.: Mobile Ad Hoc Networks: Modelling, Simulation and Broadcast-based Applications. (Réseaux Mobile Ad hoc : modélisation, simulation et applications de diffusion). Ph.D. thesis, University of Luxembourg (2007), https://tel.archives-ouvertes.fr/tel-01589632

17. Hogie, L.: IDAWI: a decentralised middleware for achieving the full potential of the iot, the fog, and other difficult computing environments. In: Proceedings of MiddleWedge 2022 ACM International workshop on middleware for the Edge. Collocated with ACM/IFIP/USENIX Middleware 2022, Québec, Canada. ACM (2022), to be published

18. Hogie, L.: Idawi: a middleware for distributed applications in the IOT, the fog and other multihop dynamic networks. Research report, CNRS - Centre National de la Recherche Scientifique ; Université Côte d'azur ; Inria (Feb 2022), https://hal.archives-ouvertes.fr/hal-03562184

19. Kayal, P.: Kubernetes: Towards deployment of distributed iot applications in fog computing. In: Amaral, J.N., Koziolek, A., Trubiani, C., Iosup, A. (eds.) Companion of the 2020 ACM/SPEC International Conference on Performance Engineering, ICPE 2020, Edmonton, AB, Canada, April 20-24, 2020. pp. 32–33. ACM (2020). https://doi.org/10.1145/3375555.3383585, https://doi.org/10.1145/3375555.3383585

20. Lacour, S., Pérez, C., Priol, T.: Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids. Research Report PI 1757 (2005), https://hal.inria.fr/inria-00000645

21. Nigro, L.: Parallel theatre: An actor framework in java for high performance computing. Simul. Model. Pract. Theory **106**, 102189 (2021). https://doi.org/10.1016/j.simpat.2020.102189, https://doi.org/10.1016/j.simpat.2020.102189

22. Shiau, S.J.H., Huang, Y., Tsai, Y., Sun, C., Yen, C., Huang, C.: A bit-torrent mechanism-based solution for massive system deployment. IEEE Access **9**, 21043–21058 (2021). https://doi.org/10.1109/ACCESS.2021.3052525, https://doi.org/10.1109/ACCESS.2021.3052525

23. de Souza Cimino, L., de Resende, J.E.E., Silva, L.H.M., Rocha, S.Q.S., de Oliveira Correia, M., Monteiro, G.S., de Souza Fernandes, G.N., da Silva Moreira, R., de Silva, J.G., Santos, M.I.B., de Aquino, A.L.L., Almeida, A.L.B., de Castro Lima, J.: A middleware solution for integrating and exploring iot and HPC capabilities. Softw. Pract. Exp. **49**(4), 584–616 (2019). https://doi.org/10.1002/spe.2630, https://doi.org/10.1002/spe.2630

24. The OSGi Alliance: OSGi service platform core specification, release 4.1. http://www.osgi.org/Specifications (2007)

25. Trolliet, T., Cohen, N., Giroire, F., Hogie, L., Pérennes, S.: Interest clustering coefficient: a new metric for directed networks like twitter. J. Complex Networks **10**(1) (2021). https://doi.org/10.1093/comnet/cnab030, https://doi.org/10.1093/comnet/cnab030