Towards Better Middleware for Distributed Computing

Luc Hogie

Towards Better Middleware for Distributed Computing

Abstract

New middleware for distributed system is frequently released by computer science laboratories. Each new proposal comes with specific models of strategies to solve particular problems related to distributed computations. In our lab, pratical studies on a variety of networks configurations (i.e. in clusters, edge, IoT, etc) led us think about the design of a middleware that would be practicable in the worst conditions: high dynamics/mobility, heterogeneity, low security, reliability, etc. Because less is more, such middleware could then be used in any application setting. This paper describes the inventive collection of intricate models for application architecture, communication, computation, dynamics management, etc which can be found in the Idawi Open Source Java implementation.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords: Digital twin, Decentralised systems, overlay networks, framework, edge, fog, IoT, Java, distributed computing, idawi

ACM Reference Format:
. 2024. Towards Better Middleware for Distributed Computing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Distributed/parallel computing is difficult by nature. It dramatically augments the initial complexity of algorithms with issues related to communications and concurrency. Also, because distributed computing involves more hardware elements (computers, networks, etc), it leads to even higher risk of failures. Worse, concurrency problems are often counter-intuitive, and the distributed nature of the computing environment makes them harder to detect, understand and solve. Because of this, the availability of middleware/frameworks for distributed computing is of paramount importance.

In this context this article introduces the Idawi middleware (see technical report [11]), which is built on the basis of ideas not found in existing tools, in the objective of improving the adequacy of distributed middleware to recent applications/networks settings. 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 [15], networking [6], decentralised protocols [9] for MANETs and more recently the IoT, etc. From the very beginning, the design of Idawi has been driven the requirements of these projects, as we quickly faced problems existing tools did not solve. *How make trial-and-errors working mode practicable? How to trigger new deployments from previously deployed nodes so as to deploy a* tree *of nodes? How to deal with NATs/firewall? How to process streams/workflow of data in a distributed way? How to monitor a remote execution? How to interact with it in a synchronous way?* We discovered that these problems were most of the time correlated, as they all involve the core application/communication/computations models. We reckon they must be considered them *as a whole* be solved by an unified set of solutions.

To this purpose Idawi does a synthesis of the tools mentioned hereinbefore, by gathering (and improving when possible) their good features in a comprehensive fresh one, and it goes beyond by proposing effective solutions to problems not tackled by existing tools. Among the numerous middleware solutions we looked at, ProActive has been a primary source of inspiration. Since it was developed in our lab, meeting its authors was easy, so we could extensively discuss the design choices they had made. Doing this really helped us make thoughtful decisions when it came to Idawi. As a result, Idawi is in many aspects different from other tools, and it comes with unique features. More precisely, on top of a *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 doing synchronous calls. It comes with automatised incremental deployment/bootstrapping of components/applications through SSH,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

operating even in the presence of firewalls/NATs, with a REST-based web interface, and with a multi-thread computation engine. Finally it supports *emulation*. Many of these features can be found in some grid middleware and in modern orchestration tools, but these do not offer the elasticity intrinsically brought by decentralization.

IDAWI is Open Source, and can be found at https://github.com/lhogie/idawi.

2 Existing tools

Existing middleware for distributed computing (like ProActive [5], MPI, JXTA, JMS (Jakarta Messaging API), ActiveMQ [8], JGroups [2], RMI, and Akka) is most often tailored to grids, clouds or clusters. With the advent of Mobile Ad hoc Networks (MANETs), the IoT (Internet of Things), Delay Tolerant Networks (DTNs), Edge computing, VANETs, etc, middleware now needs to take into serious consideration the development of applications for networks involving mobile devices. Indeed existing middleware was not designed to operate on these networks, and it poorly accommodates their intrinsic properties. To solve this problem, more flexible models and elastic tools have been proposed by Researchers, like JavaCà&Là (JCL) [7], GoPrime [4], ParallelTheater [14], ActorEdge [1], and EmbJXTAChord [3]. But in spite of their numerous good features, in particular their wise use of *decentralization* as a solution to dynamics/mobility, these Research tools are often 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 their source codes, when they are fully available, are always very hard to embrace. To increase difficulty, there is no consensus on the tool that is the most appropriate to start this work from.

3 Adequacy to experimentation

Proposition: *middleware for distributed computing should facilitates experimentation.*

Interestingly, we found out that no tool was designed for experimentation, which brings its lot of specific requirements. In particular, Researchers and engineers usually work in a *trial and errors* mode from their favorite Integrated Development Environment (IDE). At the design/tuning stage, they do countless small adjustments in their source code and run many new executions to see their impact. In order to facilitate this daunting work, the middleware needs to be able to deploy and bootstrap distributed applications very quickly, and it needs to be well integrated to the IDE so that remote code can be easily dealt with at runtime. In this regard, having transparent and on-the-fly reporting of messages/errors is crucial, as working with distributed logs is not the right way to go, even if it is still a very common practise today.

4 A structuring application model

Proposition: *middleware for distributed computing should impose applications a common architectural pattern.*

The application model of IDAWI relies on the Service Component Architecture (SCA) style of programming. SCA defines that functionality is brought by services/endpoints in components. *component*s are first-class citizens in IDAWI. They represent business entities, and in most situations (where emulation is not used) every device hosts a component that represents it in the component-system. Components self-organize as an 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 limited range, hidden nodes, etc. Two non-neighbours must rely on intermediary nodes which then behave as *routers*. Routing policies are defined by optional routing services.

In IDAWI, components are *atomic*, meaning that a single component cannot be split on several nodes. Distribution is brought by specific services dedicated to network messaging: *transport* services (OSI layer 2) and *routing* services (OSI layer 4).

Such a *structuring model* forces distributed applications to conform to a certain organization defined by a specific OO model, which ensures *consistency* of source codes, significantly reduces the risk of design errors (for most design work is into the framework), and it enables the addition of high-level functionalities such as *deployment* and *service discovery*.

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

Proposition: *middleware for distributed computing should take maximum profit of compilers to strengthen the code* When most middleware uses strings to identify things, Idawi takes advantage of Java meta-class system by using top-level classes as identifiers whenever possible. Since classnames are statically checked by the compiler (string content is not), doing this eliminates the risk of using undefined IDs. Then any service is identified (within its host component) by its implementation class.

Proposition: *middleware for distributed computing should be flexible on thread usage* Unlike ProActive active objects which have their own control thread, Idawi uses a common a pool of threads, for the sake of performance and scalability. This approach also used by actor systems (like Akka). Doing this, components have no threads associated to them. Executions requests are submitted to the entire pool, whose the first available thread will satisfy. This implies that multiple executions of endpoints for a single service may execute in parallel. In this situation where there may be concurrent access to the service data, it is desirable that this data employs *lock-free* data structures, so as to reduce the use of locks. Idawi does not provide implementation of such data structures as the standard Java library already features a good variety of them, and there exist third-party libraries dedicated to that.

5 A message-based communication model

Proposition: *middleware for distributed computing should make communication explicit and as much flexible as possible.*

At the lowest layer of Idawi, (running) endpoints communicate by *explicitly* sending/receiving messages of a bounded size. Sending a message is always an *asynchronous* (non-blocking) endpoint. 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 endpoints, in a *synchronous* fashion.

A message queue is a thread-safe container of messages exposing the following blocking 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 remains empty; and $add(timeout)$ adds a message in the queue, waiting until timeout expires if the queue was full. Using finite timeout ensures that no dead-locks will occur in the system.

6 A *many-to-many* collective computation model

Proposition: *middleware for distributed computing should consider group computation as the default paradigm.*

Most existing solutions rely on the Remote Procedure Call (RPC) model, in spite of its limitations to a client/server endpoint, its inability to provide progress information, temporary results, nor to process streams of data, etc. To overcome these problems Idawi defines an innovative computing model, based atop the communication model from which it benefits the *collective* approach.

It defines a special message called the *exec message*, which carries the name of the endpoint to be executed. Within the recipient address, the name of the target queue is chosen randomly by the sender of the exec message. On reception, the target service creates this queue on-the-fly and puts the exec message into it. This queue will be used as the input queue for this new execution of the endpoint.

Example applications of *collective computing* include fault-tolerant distributed computing, resource discovery, opportunistic computing, etc, querying of distributed databases, etc.

The exec message may carry input data, and further messages may come later, carrying extra input data. This enables a running endpoint to receive *unbounded input* at runtime. In most cases the input queue will be fed by the caller of the endpoint. However, any other component may obtain in, and use it to feed the endpoint. Then, just like any queue, the input queue can be fed by *multiple sources*.

An endpoint 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 endpoint. This message queue can play the role of a *future*. Once again, other running endpoints may obtain the address of the return queue, and send *directly* messages to it. This enables *composition of services* and *workflows*.

From a programmatic point of view, The $exec()$ primitive makes it easy the remote execution of endpoints. It takes as input the address of the endpoint 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 (by using the *collect* algorithm for example), as described hereinafter.

6.1 Flexible computation facilities

Proposition: *middleware for distributed computing should provide a flexible API for computation, supporting synchronous and asynchronous calls, and group-computation by default*

So as to enables the design of implementation of a large variety of communication/computation scheme, a middleware should provide a highly flexible dedicated API. The *collect* algorithm introduces an higher-level approach by

proposing an asynchronous/reactive/event-driven API on top of synchronous API of queues.

Its design is inspired by the iterator API in the HPPC library [**?**], and by the stream APIs in the Java standard library. More precisely, in order to enable an efficient way to iterate over a container of primitive values, HPPC iterators do not return "boxed" objects, instead they return a single same object, called *cursor* which carries (as one of its attribute) the primitive value of the current element in the iteration process. Following the elegant style of functional programming, the stream API of Java proposes a way to iterate over the elements in a stream by each time invoking a user code written in the form of a lambda.

The *collect* algorithm employs these two ideas to what it does: enabling the iteration over message lists. Every time a new message is discovered in a queue, the algorithm calls a business code provided by the developer. Instead of receiving a reference to the current message, this code receives as input a reference to the collector itself (which stands as its "cursor"), whose current state provides the (modifiable) list of messages collected so far as well as technical information on how the algorithm performs (timings). The collector also exposes its parameters, which can be *altered* on-the-fly. This enables a deep control of the runtime of the *collect* process.

The parameters of the *collect* algorithm are the following:

- *endDate* indicates the deadline at which the algorithm will stop waiting for new messages
- *timeout* the longest tolerated duration when waiting for the next message
- *stop* a boolean value for stopping the *collect* process before *endDate*
- *blackList* a set of components whose messages are simply ignored
- *deliverProgress* set if progress messages should be delivered
- *deliverError* set if error messages should be delivered
- *deliverEOT* set if EOT messages should be delivered

In order to illustrate the use of the *collect*() algorithm, let us consider a few examples. Let *q* be a queue of messages. The following call to the *collect* algorithm prints every single messages until the queue expires.

```
1  q.collect(c -> System.out.println("new message: " + c.
       messages.last()));
```

This second example only obtain the three first messages:

```
2  q.collect(c -> {
3            System.out.println("new message: " + c.messages.
                 last());
4            c.stop = c.messages.size() == 3;
5          });
```

Third, let us consider a toy example in which each reception of a message alters the *timeout* parameter by imposing that the next message needs to arrive at least as fast as the current one.

```
6  q.collect(c -> {
7            if (c.messages.size() > 1) {
8              var newMsg = c.messages.last();
9              var previousMsg = c.messages.last(1);
10             c.timeout = newMsg.receptionDate - previousMsg
                   .receptionDate;
11           }
12         });
```

This last example obtains data frames from many source components and reconstructs the *merge* data stream out of them, by paying attention to not duplicate frames. A frame is assumed to have an ID and to carry data. This collect process stops when 1000 frames have been received.

```
13         var receivedFrames = new HashSet<Long>();
14
15         q.collect(initialDuration, initialTimeout, c -> {
16               var f = (Frame) c.messages.last().content;
17
18               if (!receivedFrames.contains(f.id)) {
19                 receivedFrames.add(f.id);
20                 pipe.write(frame.data);
21               }
22
23               c.stop = receivedFrames.size() == 1000;
24           });
25 }
```

The *collect* algorithm is the essence of IDAWI. It uses many concepts the framework relies on, it gives full access to the platform functionality via a coherent interface. The rest of this article will explain the techniques which have been employed to translate the behavior of the *collect* algorithm to the world of Web services.

7 A Web interface to the *collect* algorithm

Proposition: *middleware for distributed computing should provide extensive Web monitoring and HTTP-based control systems.*

For the sake of interoperability with other tools and of the controllability by an human operator, services should be exposed to the Web via services conforming to Web standards, using standard Web technologies. This is done by a specific built-in service whose it is the specific concern.

Web services endpoints in other systems usually work in a RPC fashion: their input data are carried by the URL that triggered them, similarly to a function/method that accepts a list of parameters, and they respond (a result or an error) in the form of a JSON document. IDAWI departs from this idea. The aim of its Web interface is to reflect the endpoint of its native services, and in particular to translate the "stream of messages"-oriented behavior of the *collect*() algorithm to the Web. Thus, it enables to:

- process streams of data
- trigger any endpoint on any component
- feed it on-the-fly
- obtain responses as they get delivered
- terminate it

Each service is identified by a specific URL path. The path specifies the target components, as well as the ID of the endpoint which is to be invoked on these components. Once the service is running, it can be fed on-the-fly via POST by a stream of data. Responses then are obtained by using an ad hoc protocol called *Idawi Web Protocol* (IWebP) over Server Side Events (SSE). Because there may be multiple responses coming from multiple components, or because components may not reply as expected, the question of *service termination* arises. Idawi solves it by specifying the termination condition in the URL of the Web service.

8 Distributed digital twin

Proposition: *in a distributed system, the concept of digital twin is highly adequate as a paradigm for representing remote nodes and interacting with them*

In a distributed system, entities (components, objects, functions, etc) live in different address spaces. For this reason they cannot refer to each other via (process-scoped) pointers/references: a distributed referencing system is needed.

In many systems, remote entities are represented by numerical or textual identifiers. In this approach there is no way for the compiler to assist the developer and verify its source code, as the numerical of textual IDs representing remote things have no semantics a compiler can manipulate: they are just values. This approach has the following pros and cons.

On the one hand, a significant advantage of this approach is that it makes distribution explicit: remote calls can be clearly seen in the source code of applications. The developer can then have a clear idea of when his application will use the network, allowing him to minimize as much as possible communications, which are a common bottleneck in distributed applications.

On the other hand, the natural—elegant—constructs of the programming language cannot be used to refer to remote things. This tends to obfuscate the source code to make the code prone to bugs. A naming system based on types is always desirable as it offers much more security and consistency of the code.

Object-oriented programming proposes an elegant solution to the problem of referring to remote entities, by associating them to specific types, and forcing their local counterpart to be of this same type: both the remote entity and its local *stubs* follow the same specification (they extend the same class). The code for stubs, which is generated automatically by a preprocessor, implements bidirectional communication with the remote object. The business implementation must be provided by the developer. This approach has interesting properties. First, it takes full advantage of OO designs (relying on abstraction, polymorphism, inheritance, and encapsulation), ensuring enhanced readability and maintainability, extensibility, code reuse, and reduced development time. Second, it provides distribution transparency. A stub can then be used as a placeholder for the remote object it refers to. Consequently, algorithms manipulate stubs just like they would do with business objects. Transparency seems highly profitable at first glance, but it actually exhibit two major drawbacks:

But by hiding distribution, it hides its inherent cost. When invoking a method on a stub, there is no indication whatsoever that the invocation will not only entail a jumps in memory like this happens for any method invocation, but also and bi-directional synchronous network communication. From a more technical perspective, while invoking a local implementation take just a few nanoseconds, adding a network communication on the fastest local networks today is more of the order of a millisecond (in the order of 10^3 slower). This performance problem is not visible in the source code. Transparency then tends to produce too high-level and largely inefficient code. Also, because the functional interface is the same for stubs and business objects, the programmer is tempted to think they behave the same. But there exists a major difference in terms of behavior. As a matter of fact, when invoking the business object *locally* (via its native reference), mutable parameters are passed to methods as references, and immutable ones are passed by value. When invoking business methods via a stub, (de)serialization enters the scene, and parameters get all passed by value, be they mutable or immutable. This also applies to return values. Unfortunately, this difference of behavior is not reflected by the source code. The developer must be aware of the nature (stub or business object) of the instance it manipulates, so as to treat parameters and return values differently. An intuitive solution to this problem is to dynamically/transparently create stubs to all mutable objects, and to transport these stubs instead of the values of objects. Remote algorithms would then deal with stubs of objects living on the same node/process from which the execution request originates. Unfortunately doing this worsens the impact of transparency, increases the amount of communication, thereby dramatically reducing the global performance of applications.

Idawi Digital Twining System (IDTS) lies on the idea that *digital twin* (DT) are adequate to the manipulation of distributed elements. A DT is a model of a physical object. It evolves alongside its physical counterpart, and it offers the ability to obtain information on it, and to simulate it.

The main idea in IDTS is to represent, as well as refer to, remote so-called "physical" objects as DTs. IDTS takes advantage of the very nature of the physical object it considers: software objects. These are of the same nature as DTs: they both are made of code. IDTS defines that a digital twin has the same code as the real object. They only differ in the data they carry, and the role they play. When queried, an object declared as the digital twin of another object, can do several things:

- it can act like a stub, forwarding the query to the real object (optionally caching the result locally);
- or if it has the necessary data and its host node has enough resources available, it can execute the business code (optionally reporting its action to the real object, and caching the result locally);
- or it can simulate the behavior of the real object (optionally reporting its action to the real object, and caching the result locally);
- or it can return a previously cached value.

In addition to its natural conformance to the OO approach, as DT can serve as a placeholder for the real object (it can actually even become the real object if it obtains the necessary data), IDTS has a number of desirable impacts on distributed systems, not found in existing designs:

- It makes them simpler (both in design and source code) by making local and remote objects of the very same nature;
- It enhance their flexibility by natively providing query delegation, result caching, and simulation;
- By enabling each node to simulate the distributed system (not just the network), it can help applications to improve the management of resources, hence their performance.

8.1 An use case: message broadcasting

In order to illustrate the advantages of being able to simulate locally in this context, please consider the following use case: node needs to broadcast a message into the network. To this purpose it needs to evaluating the best diffusion strategy, which is a multi-objective problem aiming at (among others):

- minimizing network congestion;
- minimizing energy consumption;
- maximizing delivery rate.

In order to broadcast a message, every node has a set of so-called *broadcast services*. Each of them comes with its own set of parameters. If we assume that only one broadcasting algorithm is going to be used, mathematical analysis can compute a Pareto front, which will permit the calibration of the values of each parameter. But if we relax that unrealistic assumption, maths become way too complex, and simulation becomes the only way to go. In a DT-based system, the node will be able to simulate the broadcast process on its *memory-local network of DTs*. In this simulation, the full diversity of behaviors found in the physical system can be considered for further decisions on what to do..

Just like link-state routing protocols do, physical nodes disseminate topology information across the network, but not only. Indeed "routing" is not our sole concern. A framework for distributed computing must offer services for resource/services discovery, among others. Then physical nodes must also send structural and system information, like the services they offer, their computational/storage capabilities, their current load, etc. This information can then be used to update individual physical nodes to update their local digital twins of remote nodes so as their local view of their network environment remains as accurate as possible. Accurate views can then be cleverly used by applications to distribute at best.

9 On decentralization

Proposition: *middleware for distributed computing should impose no form of centralization to atop applications, yet enabling the design/implementation of centralized application when desired by the designers of the applications.*

In the contexts of pervasive/edge/mobile computing, VANETs, the IoT, etc, recent tools consider decentralization. Indeed in these environments, no assumption can be made on dynamics, especially on mobility. Decentralization is required in situations in which no infrastructure is available, when devices cannot be trusted, etc. Then devices must self-organize with each other, communicate with other devices in their radio range, or with the the neighbors they trust, in a decentralised fashion to provide the minimal functionality required to distributed computations.

On one hand, a middleware should not prevent applications built on top of it to resort to centralization. Further, for the sake of completion, it should provide helpers to achieve centralization. On the other hand, it should not impose application any form of centralization.

9.1 Node identification

Proposition: , *from a general point of view, nodes'ID should be generated randomly, and that the responsibility of providing "friendly names", should be delegated to atop applications, which would drive their management according to their own needs.*

If decentralization has many good properties, it also brings new issues. In particular arises the question of how to identify the component when no central trusted directory is available. IDAWI assumes that components are identified by a random value they generate on their own, and whose length ensures acceptable statistical unicity.

9.2 Encryption

that encryption should be considered anywhere it proves useful in a distributed system, so as to enhance not only local node security but also the global behavior of the system.

A unique feature in IDAWI is to use this random identifier as a public key for decrypting messages outgoing messages. Identifiers are then generated locally, in a decentralized manner, by the key-pair generators of asymmetric ciphers. Another unique feature is that before fowarding a message, nodes add their own layer of encryption (using their own private key) to the message. This augments encrypted communication (à-la SSH) by route validation: being able to

decrypt a message guarantees that it took the same route as the one it advertizes.

10 Reference implementation: the Idawi framework

Idawi [12][10][13] is a framework for decentralized computing on large heterogeneous dynamic networks. It aims at providing applications built atop of it with appropriate communication and computation models, as well crucial services like messaging, routing, service discovery, provisioning, deployment, etc. The initial motivation for the design and implementation of a new framework was to meet the technical requirements of the scientific experimentation of the COATI/I3S/Inria/CNRS Research group (2010-2020) in the contexts of its Research projects on graph algorithms [15], networking [6], decentralized protocols [9], etc. More specifically we needed software services for the deployment of workers, their monitoring and the control of their jobs in progress.

In 2021, our production code turned into a Research platform when we worked at making it effective at large scale (i.e. when a large number of computational nodes are involved). In this context, the difficulties lied to node failures, churn, mobility and heterogeneity arose naturally. In particular we considered heterogeneity anywhere it can be encountered: on the hardware of nodes (RAM, CPUs, disks, etc), and on the communications mediums (wired, wireless). The latter aspect impacts deeply the communication strategies for using wireless technologies inevitably leads to the presence of directed links into the communication graph.

The very nature of the network we consider (i.e. large, dynamic and heterogeneous) relaxes most of the assumptions made by other frameworks. This led us to rethink the design of most of our initial code. In this process, as much as possible we look at the state of the Art of existing tools, and try to reuse concepts, ideas, and codes.

All of the concepts presented hereinbefore are implemented in Idawi.

11 Conclusion

The design of frameworks for distributed computing has been changing a lot during the past 3 decades. OO modeling, SOA, and microservices architectures have brought new concepts and possibilities which proved highly beneficial to the development of distributed applications. that, in addition to these paradigms, the use of digital twins to represent node and simulate the network, the use of random identifiers as keys for encryption/decryption, flexible stream-oriented execution models can play a major role in the design of future middleware for distributed applications.

In order to demonstrate the implementability and effectiveness of theses approaches from a programmatic point of view, we implemented them all at the core of the Open Source Idawi framework.

Acknowledgments

I am grateful to all of those with whom I have had the pleasure to work since the beginning of this project, and especially to Master's students Valentin Mascaro and Sami Joudet.

References

[1] Austin Aske and Xinghui Zhao. 2017. An Actor-Based Framework for Edge Computing. In *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, Ashiq Anjum, Alan Sill, Geoffrey C. Fox, and Yong Chen (Eds.). ACM, 199–200. https://doi.org/10.1145/3147213.3149214

[2] Bela Ban. 1997. *Adding group communication to Java in a non-intrusive way using the ensemble toolkit.* Technical Report. Citeseer.

[3] Filippo Battaglia and Lucia Lo Bello. 2018. A novel JXTA-based architecture for implementing heterogenous Networks of Things. *Comput. Commun.* 116 (2018), 35–62. https://doi.org/10.1016/j.comcom.2017.11.002

[4] Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. 2016. GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly. *IEEE Trans. Software Eng.* 42, 2 (2016), 136–152. https://doi.org/10.1109/TSE.2015.2476797

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

[6] David Coudert, Luc Hogie, Aurélien Lancin, Dimitri Papadimitriou, Stéphane Pérennes, and Issam Tahiri. 2013. Feasibility study on distributed simulations of BGP. *CoRR* abs/1304.4750 (2013). arXiv:1304.4750 http://arxiv.org/abs/1304.4750

[7] Leonardo de Souza Cimino, José Estevão Eugênio de Resende, Lucas Henrique Moreira Silva, Samuel Queiroz Souza Rocha, Matheus de Oliveira Correia, Guilherme Souza Monteiro, Gabriel Nata de Souza Fernandes, Renan da Silva Moreira, Junior Guilherme de Silva, Matheus Inácio Batista Santos, André Luiz Lins de Aquino, André Luís Barroso Almeida, and Joubert de Castro Lima. 2019. A middleware solution for integrating and exploring IoT and HPC capabilities. *Softw. Pract. Exp.* 49, 4 (2019), 584–616. https://doi.org/10.1002/spe.2630

[8] Nicolas Estrada and Hernán Astudillo. 2015. Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. In *2015 Latin American Computing Conference, CLEI 2015, Arequipa, Peru, October 19-23, 2015.* IEEE, 1–6. https://doi.org/10.1109/CLEI.2015.7360036

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

[10] Luc Hogie. 2022. 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.

[11] Luc Hogie. 2022. *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. https://hal.archives-ouvertes.fr/hal-03562184

[12] Luc Hogie. 2022. A Service-Oriented middleware Enabling Decentralised Deployment in Mobile Multihop Networks. In *Proceedings of FMCIoT 2022 International Workshop on Architectures for Future Mobile Computing and Internet of Things. Collocated with 20th International*

Conference on Service-Oriented Computing (ICSOC 2022), Sevilla, Spain. LNCS. To be published.

[13] Luc Hogie. 2023. A Decentralized Web Service Infrastructure for the Interoperability of Applications in Multihop Dynamic Networks. In *CIoT 2023 - 6th Conference on Cloud and Internet of Things*. DNAC, IEEE, Lisbon, Portugal, 211–218. https://doi.org/10.1109/CIoT57267.2023.10084876

[14] Libero Nigro. 2021. Parallel Theatre: An actor framework in Java for high performance computing. *Simul. Model. Pract. Theory* 106 (2021), 102189. https://doi.org/10.1016/j.simpat.2020.102189

[15] Thibaud Trolliet, Nathann Cohen, Frédéric Giroire, Luc Hogie, and Stéphane Pérennes. 2021. Interest clustering coefficient: a new metric for directed networks like Twitter. *J. Complex Networks* 10, 1 (2021). https://doi.org/10.1093/comnet/cnab030