Idawi: a middleware for distributed applications in the IOT, the fog and other multihop dynamic networks

Luc Hogie

IDAWI: a middleware for distributing applications in the IOT, the fog and other multihop dynamic networks

Luc Hogie

luc.hogie@cnrs.fr
I3S Computer Science laboratory, Université Côte d'Azur/CNRS/Inria, France

February 8, 2022

Contents

Abstract

As technology improves, new kind of devices are given the ability to interconnect to others, thereby shaping today's networks. Sensors, smartphones, tablets, laptops, desktops, servers, be they organized or not, altogether form the Internet of Things (IoT), the cloud, and the glue between them. All these devices are heterogeneous in terms of hardware (CPUs, memory type and size, network interfaces, etc), software (Unix, Windows, Android, etc) and behavior (mobility, reliability). However they all have in common the ability to perform computations. But in this context of extreme heterogeneity, distributed computing has become even harder, and the tools we use for clusters, grids and clouds cannot be used. A few middleware solutions to distributed computing this is difficult environment have been proposed recently, but these tools address only a limited number of issues, and they can hardly be coupled together to a single usable solution. As a consequence, Researchers most often opt for developing their own solution, or resorting to simulation. At I3S we also did that. Through several Research projects which implied many aspects of cluster computing and IoT, we built a codebase of successful solutions to many problems we faced along the years. Revamped and generalized into the most consistent, efficient and cleanest possible middleware, we now expose it to the Research communities as Idawi: an Open Source Java message/queue/service/object-oriented decentralized overlay network of components designed to meet the conceptual and practical needs of Researchers working on distributed algorithms.

1 Introduction

This paper describes IDAWI, a new middleware designed to assist Researchers and R&D Engineers in the experimentation of distributed algorithms. IDAWI is developed at the COMRED Research Group of the I3S Computer Science laboratory of Université Côte d'Azur (France), and Inria Sophia Antipolis. Its source code can be found at: `https://github.com/lhogie/idawi`. It is released under Apache V2 license.

The design of IDAWI is driven by the needs of R&D projects we have been getting involved into, and it benefits from the experience we acquired working on them. Indeed, in addition to studying distributed systems from a theoretical point of view, for many years we have been using, evaluating, designing, and tuning algorithms for High Performance Computing (HPC), the Internet of Things (IOT), Mobile Ad hoc Networks (MANETs), fog computing, peer-to-peer (P2P) computing [4], etc. In this context, we evaluated many existing frameworks/libraries available to Researchers. Most of the time we found out that these solutions hardly match what experimentation usually need in terms of models and features. As a consequence, during this projects, we most often ended up writing a significant amount of code/time to adapt existing solutions or to design/implement fresh ones. This led us to write and maintain thousands of line of code along the years. The components of the BigGrahs CNRS Research platform are ones of them: Grph [31], BigGraph [26], and JMaxGraph

[32, 41, 40]. Others include their satellite tools [24, 28, 20, 23, 29, 27, 21]. IDAWI stands as a synthesis of these past developments, and our experience in the field of distributed computing. As such it gathers the concepts and solutions which proved effective in our past studies, and that we believe will be useful to future Research not only of our local Research groups but of a broader Scientific community. The following of this introduction details the requirements IDAWI proposes solutions to.

A general network model

Research in distributed system consider a wide variety of network settings. An experimentation tool in this field then should not be restricted to a specific one. In particular the IOT, fog computing and MANETs operates on a network in which nodes are heteroegenous, mobile, unreliable, etc. These nodes must be considered to have a view of the network that is restricted to their surroundings, and that they need to self-organized in a decentralized way. Decentralized algorithms are heavily studied today. Indeed all these fields consider networks featuring mobility, extreme heterogeneity, and high number of nodes. These networks do not accommodate centralized software management systems. Unfortunately most existing distributed middleware have very limited support for decentralization, if any. IDAWI proposes a multi-hop overlay communication network in which each node acts as a router: nodes collectively participate to the delivery of messages without relying on any centralized service or super-node.

Communication in such an networking environment can hardly be achieved using existing communication primitives which usually implement point-to-point communication. In the networking conditions we assume, nodes must be considered unreliable and likely to vanish in the middle of a transaction. Also, the very nature of the applications running on top of them is peer-to-peer (P2P), which implies that the global behavior of the application is the result of the collective behavior of each of its part. In this context, the communication model must be natively collective. This means that a (dynamic) set of recipient nodes should be addressable just like a single one, and this should be as much transparent as possible to the application.

Also, as these network involve mobility, which entails intermittent connections, delays, timeouts, communication must take time into account. We are not talking about a real-time system here, but communication primitive must enable the application to deal easily with time.

A flexible computational model

Most object-oriented distributed systems rely on the RPC paradigm and they foster transparency as much as they can. But as distributed calls take the shape of local ones, they are targetted to one specific recipient. Indeed the OO model for execution does not suit well group-communication. Just like the communication model (which it relies on), the computation model should be natively collective, so as to support distributed queries, redundancy, etc.

First, the RPC model defines that values are returned when the remote executed has completed. This prevents the obtention of information at runtime just like intermediary results, progress informations, etc. In the context of distributed computations, this proves to be a problem on the caller side as computations are likely to take time, but no information on their status can be obtained.

Second, we reckon transparency should not be favoured. As remote calls are inherently costly as they involve network communication, they have to be avoided as much as possible. Thus it is desirable that they are clearly identifiable within the source code. We believe that disguising a remote call into as local one, like many system like to do, tend to the implementation of low performance code.

Finally, most middlware define event-based computations, which suit most applications. However algorithms requiring interactions can hardly be implemented using that model: they are more naturally expressed using an imperative programming model. A computation engine should support both event-based and imperative programming models.

Network agnosticism

Network technologies are numerous, besides the ubiquitous Ethernet for wired networks dans Wi-Fi for wireless ones, Infiniband is widely adopted in clusters, Bluetooth is common in IOTs, etc. These technologies not only differ at the physical layer: their APIs do not allow to do the same things. For example, TCP, regardless of the PHY layers it operates, provides data connections through what it calls *socket*s. On the contrary, Bluetooth, UDP and Infiniband do no offer such connected mode: data has to be sent as chunks. Also they do not always garantee of reception, and that chunks be will be received in the same order they were sent. This heterogeneity cannot be accounted for at the level of applications. Application must be provided with an abstract communication model which exposes basic universal primitives for network communication. This paradigm is well-adapted as most middleware now have agnoticism of the network layer.

The ability to work in a trials and errors mode from within the user's favorite IDE

In link with the theoretical aspects of architecture and computational complexities, the process of designing distributed software (and software in general) deeply involves implementation, executions and observation of factual behavior. In practice the implementation/tuning work is done within an Integrated Development Environment (IDE). This work consists of countless small adjustments of the source code and verification by new executions. When the software under work is a distributed system, executions requires the prior deployment of the new binaries onto target nodes. Then, at running time, it is highly desirable than remote code can be easily monitored as if it were running locally.

In particular, having a transparent and on-the-fly reporting of errors is crucial. Then, upon completion, remote resources must be releases without requiring any manual operation from the developer. IDAWI features a SSH-based deployment service to enable the seamless deployment, execution, communication and termination of distributed computations even in the presence of firewalls and NATs.

Ability to process large datasets

Experimentation works start using toy data set so as to evaluate the behavior of the system on specific configurations. When the behavior is validated, or trusted enough, experimentation most often consists of processing large data sets. Multi-thread parallel is a powerful tool to solve large problems, but it is also known to be inherently difficult. Many modern solutions incorporate a lock-free computation engine which maximizes the utilization of the CPU cores through the use of a thread pool.

Unique features:

- it is decentralized
- it has a collective message/queue-oriented communication model
- it has a collective computation model
- it has automatized deployment/bootstrapping of components through SSH
- it provides interoperability through a REST web interface
- it has the ability to do emulation

2 Architecture of the system

IDAWI is a middleware for dynamic decentralised distributed systems. It provides an architectural model and implementations to assist Engineers and Researchers at building their own system. Systems based on IDAWI do not need to be either dynamic or decentralised. They actually don't even need to be distributed at all. But IDAWI does support distribution, decentralisation, and dynamics if needed. One of the main design objectives of IDAWI is to be useful in the context of Research and experimentation, from which it was born.

To explain what IDAWI is, let us first start by stating what it is not. IDAWI is not a library. Indeed libraries do not impose that user's application have any particular design characteristic. They merely expose objects and/or primitives to them so as enable them to extend their functionality. On the contrary, IDAWI is an object-oriented middleware that structures underlying applications. These must conform a certain class organisation. If this constraint may seem frustrating at first sight, it actually strongly guides programmers towards a polished application model, by providing a clean framework into which that have

to plug their business functionalities. IDAWI is not the reference implementation of a theoretical system with proofs of correctness or optimality. IDAWI models and implementation are driven by the practical use cases we identified and its relies on our experience in designing and programming software system.

Many software rely a particular design paradigm, and push it as far as possible. This is especially true with object-oriented (OO) designers which often try to describe every single concept of their business by a class, everything in their application ending being an object. Indeed, the OO approach has many intrinsic advantages of the, and it often leads to beautifully designed applications, but designers are always tempted to push it too far. There indeed are situations in which other paradigms apply smoother. Following this statement, the design of IDAWI does not favour a specific paradigm. Instead it borrows good properties of several design approaches and applies them to its own purposes when they are appropriate. Overall, it is organised in the cleanest possible object-oriented model, so as to maximize robustness, extensibility and understandability. Wherever it makes sense, IDAWI concepts are describes by classes. It relies on the concept of components exposing services, following the SOA style of programming. The communication model is message/queue-oriented: entities communicate with one another by sending/receiving explicit messages targeted to queues. On top of this native communication scheme, IDAWI exposes a REST interface which enables interoperability with other tools. This makes IDAWI conform to the Web of Things (WoT) ideas that any device in the network is observable through a web browser.

Let us detail all this in the coming sections.

2.1 Components/services/operations

First-grade classes in IDAWI are *components*. Conceptually, components represent entities of the user application domain. Technically, a component is an object that does nothing but embedding a set of services that it exposes to its peer components. A component without service cannot do anything. It is not even able to communicate with its peers: functionality, including networking, is brought by services.

A *service* implements functionality about the specific concern it is about. Services are the way to incorporate functionality in an IDAWI system. Builtin system-level functionality (networking, routing, deployment, etc) comes as services. Users applications will come as services as well. A service has a unique identifier in its host component. A service exposes functionality through its set of *operations*.

An *operation* is a piece of code that can be triggered remotely from any component. An operation has a unique identifier in its host service. An execution of an operation happens in one single thread. There is no guarantee that two subsequent executions will use the same thread. The execution of an operation is fed by an input queue of messages. This queue provides the input data. It may contain input parameters sent by the initiator when it requested the execution, but it can also provides data at runtime. Unlike the input parameters is always

comes from the initiator of the execution, runtime input data may have been provided by any component in the system. To the purpose of producing output, an operation is free to send any message to any component at runtime. The execution of an operation can hence be seen as a many-to-many interaction.

Among the builtin system services, the *service management* service enables components to discover the services available on their peers. They can also remotely start and stop services.

2.1.1 Referring to components

In IDAWI, components have no reference to their peers. They refer to them as component *descriptor*. A component descriptor is an object holding information about a component. The content of such a descriptor is unspecified. It may store no more than the name of the peer, but it may also contains detailled information about the target component. As the whole system is dynamic, components change and knowledge about them outdates. To overcome this, a specific service (optionally) running on components periodically creates and broadcasts fresh descriptors. Components then automatically receive frest information about their peer, enabling them to maintain an (as much as possible) accurate view of the system. The set of descriptors a component holds constitutes the knowledge is has of the system. Because of the significant time needed to disseminate a descriptor, the larger the system is, the less accurate is the individual components' knowledge of it. The use of this knowledge is not restricted to the services within the component: there exists a specific service which exposes this knowledge to other components, which can then known what other components know about the system.

2.1.2 Referring to services and operations

Object-oriented middleware generally rely on stubs to refer to remote objects. A stub is a lightweight local (proxy) object that stands for its remote (business) object. Both share the same public interface (set of public methods). Stubs then make remote method invocation transparent to the callers, which deal with stubs the same way they would do with full-fledged business objects. Because of this, referring to a remote method benefits from the support of the compiler, which guarantees that (method) names are accurate. This static way of doing also takes advantage of the powerful refactoring features of modern IDEs. On the contrary, middleware solutions based on queues (like IDAWI is generally use strings as identifiers for target queues. Using strings (or any other literal) prevents the compiler to do any verification of the validity of identifiers. IDAWI purposely does not do transparent method invocation, but it still benefits from the compiler/IDE, as it relies on the type system of Java by identifying things by their top-level class. More precisely, within a component (resp. service), services (resp. operations) are identified by their class. Further, an operation class is required to be a static member of its service class, so as to automatically link an operation ID to the ID of its enclosing service.

2.1.3 Location of components

In an IDAWI system, components may live the same JVM, or in different JVMs, on the same host computer or not. In most settings, there will be a 1-1 component-host relationship and all interaction will involved the network, but the ability to instantiate components in the same computer (in the same JVM or not) has two significant benefits. First it enables IDAWI to operate like an *emulator*. More precisely, an IDAWI physically distributed system can be virtually extended by creating new components in the same node of already existing components. This is especially relevant when evaluating the scalability of a distributed application. Second, it makes it possible *testing*, which would be impracticable otherwise. To this purpose, if two components in the same JVM will be preferably use shared-memory to communication, it is possible to force them to use the networking layers instead. This enables the local testing of networking code, which is normally used only when components are in different JVMs.

2.2 Communication

An IDAWI network of components is structured as a graph. In this graph, two given components are *neighbour*s if they have direct interactions. Any two components can be neighbours unless the underlying network infrastructure prevents direct interaction. This may happen in the presence of NATs/firewalls. Also, wireless communication introduces additional constraints like limited range, hidden nodes, limited number of neighbours (Bluetooth), etc.

Any two neighbours components can communicate with each other using a specific protocol that is independant to the underlying network infrastructure. The communication model of IDAWI abstracts existing transport layers by exposing a generic communication model, and implementations for TCP, UDP, file-based communication and inter-process piping. The latter enables a component in a JVM to interact with another component in another (JVM) process, through SSH or not. This provides other services with agnosticism of the network stacks.

In this graph, two non-neighbours components can communicate through *intermediary* components that will relay messages. The relaying policies are defined by routing services. Routing services compute relays out of the local knowledge they have of the system. As the network is dynamic, this knowledge may have a certain degree of inaccuracy, which leads to a difficulty to find the shortest routes. This provides other services with agnosticism of the network topology.

The network of components forms an *overlay network* atop the existing infrastructure.

2.2.1 Message-passing

IDAWI relies on the message-passing paradigm. Components interact with each other by sending/receiving messages of a bounded size. A message has a prob-

abilistically unique random 64-bit numerical ID. It carries:

- a content, which can be anything
- target service/queue IDs
- the route it took so far
- routing information, which consists of:
  - the component destination address
  - some technical data, if any

There is no connection involved in communications. Consequently when a component stops getting reachable, no error occurs in the code of other components. They simply stop receiving messages from the unreachable node and messages sent to it never reach it.

2.2.2 Message queues

When a message arrives at (a) destination, it is delivered into a message queue. A message queue is a thread-safe container of messages. It features the following primitives:

size() gets the number of messages currently in the queue

get(timeout) gets and removes the first message in the queue, waiting until timeout expires if the queue was empty.

add(timeout) adds a message in the queue, waiting until timeout expires if the queue was full.

Sending a message to a queue is achieved using the *send*() primitive. This is an asynchronous (non-blocking) operation. It provides no guarantee of reception.

2.2.3 Advanced message management

So as to provide a convient API for the management of messages, message queues also propose a (Java)streams-oriented API. The $forEach(code)$ primitive invokes an user-specified code each time a new message is received. Let q be a queue of messages.

```
1 q.forEach(msg -> {
2     System.out.println("new␣message:␣" + msg);
3     return Enough.no;
4   });
```

The user code must return whether it has received enough messages so far, or further ones are required.

$forEach(result, error, progress, eot)$ adds demultiplexing of messages according to their kind. Four kind of message are supported:

error the message carries an error

progress the message carries progress information, which is either a progress ratio (in $[0, 1]$) or a progress message indicated what is happening

eot the message carries an marker of end of tranmission from a given component

result the message carries a result of some computation

For each of this message kinds, the user must provide a specific handler.

Through the *collect*(*msg*) primitive, message queues also make it possible to collect incoming messages to a message list, until a certain condition. The following example collects messages into a list until an EOT is received. Further message will then be ignored.

```
1  q.collect(msg -> {
2   System.out.println("new␣message:␣" + msg.content);
3   return msg.isEOT();
4  });
```

A message list features a number of primitives for browsing, filtering, and classifying the message in contains.

IDAWI proposes an API for the manipulation of data streams on top of IDAWI message-based native communication scheme. The $Streams.split(i, id, s, l)$ primitive converts a data stream to a sequence of message.

i is an input stream

id specifies a description for the data carried. This description will be used as an ID by the target operation for demultiplexing

s the expected number of bytes in the stream

l a lambda that defines what to do with any new message generated out of the stream

For example, let f by a file:

```
1  var in = new FileInputStream(f);
2
3  // split the file data and put in messages
4  Streams.split(in, f.getName(), f.length(), msg -> {
5   // sends the message
6  })
```

On the server-side, the *join*() method creates a binary input stream out of incoming messages, assuming they transport byte arrays. This stream facility natively supports multi-source (many-to-one) streaming.

2.2.4 Routing

When the component receives a new message (it is actually the networking service which receives it), it passes it to the routing service for that message. The corresponding routing protocol computes a set of relays to which the message is forwarded.

IDAWI comes with a default routing protocol. This protocol is decentralised. It defines that all components behave as routers, forwarding the messages coming from their neighbours to other neighbours that will do the same. This way, non-neighbours components can communicate just like if they were actually neighbours.

This protocol is multicast defines message recipient as a compound queue address. This makes it multicast/broadcast by nature. It defines that a component address is a triplet (C, e, d) where:

C is the set of component names. If C is not defined, the address is considered to be a broadcast address.

e the expiration date of the message

d is the maximum number of hops allowed to travel

This way to address components suits the very nature of dynamic multihop networks. It enables to address all, one or multiple components at at maximum distance and/or reacheable until a given timeout. For example, "all my neighbor components", "all the components closer than 10 hops", "all components reacheable within the next 2 seconds", "this component", "these two components". Unicast comes naturally when one single target component is specified in the set of recipient names.

After a message has been forwarded by a component, it is not dropped. Instead, until it expires, it is stored and considered for re-emission each time a new neighbour component pops up. This enables IDAWI to deal with node mobility and scarce connectivity found for example in delay tolerant networks (DTNs).

3 Computation model

The RPC model is used by most systems, in spite of its inherent limitations. It defines that a list of parameters is passed to a procedure when it starts, and that the return value is transferred back the caller when the procedure has completed. This model limits the expressiveness of remote code to what functions/methods offer. In particular this model prevents procedures to get and to provide information at runtime. However this enables remote codes to provide progress ratios, temporary results, to use streams (which is required when dealing with large data), etc.

The computation model of IDAWI tackles all this problems at once, by doing a major departure from this RPC-oriented model. It defines that an running

operation has an unbounded input queue and it is able to send as many messages as needed. Also, just like IDAWI communication model which is relies on, the computation model is inherently *collective*. In other words, *group computation* benefits from native support. Killer application of collective computing include fault-tolerant distributed computing, distributed resource discovery, querying of distributed databases etc.

3.1 Input/output

This computation model is built on top of it communication model. It defines that upon the reception of an *exec* message, a component creates a queue in the target service, and puts the message in it. Then the operation is scheduled for execution. When it starts, the operation will be able to consume the message, as well as further ones, if any.The number of incoming message is not limited.

In most cast the input queue will be fed by the caller of the operation. But this address can also be passed to other components which will then become able to feed the running operation. Then a running operation accepts input from multiple sources.

An operation does not return data in the sense a function does. To make its output available, an operation can send messages at any time during its execution, using the *send*() primitive. Result messages are usually sent back to the caller of the operation (whose the address can be found in the trigger message), but just like any other message, they can be sent to any other component/service/operation in the system. This feature is useful for example in the situation where a component A triggers the transmission of large data from a component B *directly* to a component C. This situation is common in home systems when a user of a phone within the LAN wants to download a movie directly into the internet box, in order to watch it on the TV. Downloading first on the device then uploading to the box is a waste of time and resources as the movie data transfers anyway through the box to reach the phone.

This enables running operations to invoke other operations in other components (composition of services) as well as to send results (if any) to any other service, thereby forming workflows (operations feeding other operations, in a graph). But results are actually optional and operations may just alter the local state of the service they implement.

Once the operation has completed, and "end of transmission" (EOT) message is automatically sent to the initiator of the operation, and the input queue is deleted.

3.2 Invocation of operations

To order to make it easy the execution of remote operations, the *exec*() primitive is provided to services. It takes as input:

r the address of the operations to execute

p the initial input data

q an optional queue for collecting results

As a consequence o The *exec*() primitive creates and sends an exec message. It is asynchronous, immediately returning a proxy to the remotely running operation. This proxy features the address of the input queue of the running operation, which can be used (by any component) to send it input data.

Here is a short example of how to call the same remote operation simultaneously on two components.

```
1 // addresses two components
2 var componentsAddress = new To(Set.of("c1", "c2"));
3 var operationAddress = componentsAddress.s(aService.
    anOperation);
4 RunningOperation proxy = exec(operationAddress, true,
    null);
```

3.3 Results

On the execution of an operation, the caller optionally creates a new local input queue (as defined by the parameter of *exec*()) that will store optional output the data transferred back by the running operation. This message queue can be seen as an extension of a *future*. It is where results, if any, will be obtained.

exec() is asynchronous (just like any message emission is), but synchronicity can be achieved by invoking synchronous (blocking) primitives of message queues.

Errors (thrown exceptions) are transferred just like any other results. Component then receive the exception in their result message queue. They can treat it in different ways. A common way is to fire the exception locally, just like if it the error had happened locally. Please note that IDAWI makes it non-ambiguous the obtention of an exception as a regular result or as an error.

3.4 Creating an operation

Operations can added/removed at runtime. The primitive $registerOperation(name, code)$ adds a new operation in a service. The user code of the operation is here expressed as a lambda expression.

```
1   registerOperation("stringLength", new q -> {
2    var execMsg = q.get_non_blocking();
3    var s = (String) execMsg.content;
4    reply(execMsg, s.length());
5   });
```

Its counterpart *unregisterOperation* removes an operation.

```
1   Operation o = lookup("stringLength");
2   unregisterOperation(o);
```

In order to benefit from the compiler assistance to resolve operation names, an operation can be statically declared as an inner class.

```
1  public class grep extends BasicOperation {
2   @Override
3   public void exec(MessageQueue q) throws Throwable {
4    var execMsg = q.get_non_blocking();
5    var s = (String) execMsg.content;
6    reply(execMsg, s.length());
7   }
8  }
```

Registering such an operation is done via:

```
1   registerOperation(new grep());
```

On top of this native model, IDAWI proposes method-oriented flavour declaration of operations. Operation declared this way are called *typed* operations. A typed operation describes an operation as a Java method. The signature of this method represents both the type of the input data of the operation as well as its return type.

```
1  public class stringLength extends TypedOperation {
2   public int f(String s) {
3    return s.length();
4   }
5  }
```

The name of the method does not matter. In this mode, the operation has no way to obtain the exec message that triggered its execution. Instead it gets the list of parameters that were embedded in the exec message, and whose the types are statically specified by the signature of the method. Similarly, its return value is specified by using the *return* Java keyword, just like in any method. If this way of declaring seems more natural to most, it loses many benefits of IDAWI computational model. In particular, data cannot be passed at runtime, and there cannot be more than 1 result. This result cannot be sent before the operation has completed.

3.5 Scheduling

In order to enable take profit of multi-core computer architectures which are not ubiquitous, distributed computing libraries and framework embed a parallel computation engine. Many approaches exist. For example active objects have their own control thread: when the number of entities grows, the number of thread increases correspondingly, thereby dramatically worsening the overhead of the thread-scheduler and quickly reaching the system limit. Actor system *à-la* Akka overcome this inability to scale up by using a fix number of pre-allocated threads. IDAWI uses this approach.

3.5.1 Pool of threads

In IDAWI, components have no threads associated to them. Operations exposed by services in all components in a JVM are executed by a shared pool of threads. There exists only one pool of threads per JVM and its size if determined by the number of physical cores on the hardware node. This architecture is profitable to scalability because, the number of threads remains constant, regardless of the number of components (in the JVM). When using multiple JVMs in parallel in the same node, each JVM must be explicitly set up to use only a fraction of the threads, depending on the application requirements.

3.5.2 The special case of preemptive executions

IDAWI has not notion of priority. Operations are executed in the same order they were submitted, according to a FIFO strategy. There however is an exception. A *preemptive* execution does not involve the pool of thread. Instead the operation is executed immediately and synchronously (in the current thread). This feature is essential to system-level services which should not wait. This is for example the case of the `kill` service, whose the functionality is to force the target component to stop immediately. This feature has the disadvantage of being harmful to the system as too many preemptive executions will unavoidably introduce a bottleneck to the scheduling of non-preemptive executions, thereby decreasing the overall performance of a node.

3.6 Web/REST interface

The Rest service exposes the IDAWI distributed system through HTTP. There can be an HTTP service in any component, each of them exposing the entire system. This makes the Web access to IDAWI is then *redundant*.

3.6.1 Query URL

The HTTP server accepts URLs following this pattern:

```
http://ip:port/c1,c2,...,cN/s/o/p1/p2/.../pN
```

Where:

ip:port is the IP address and TCP port number where the Web server can be reached

c1,c2,...,cN is a list of target component names

s is the name of a service within component *c*

o is the name of a typed operation within service *s*

p1, p2, ..., pN are string representations of parameters for operation *o*

The REST web service represented by an URL is a dynamic representative of IDAWI native services. There is nothing specific to write in an operation to make it usable via REST. The only constraint is that it need to be a typed operation. Its parameters are automatically converted from the URL strings to the actual types specified in the operation implementation method, and the returned value of this method is serialised to as to be sent back to the Web client.

Additionally, the behavior of the Web server can be controlled using HTTP parameters to the URL:

```
?o1=v1,o2=v2,...,oN=vN
```

3.6.2 Response format

The default output format of REST web services is the JSON text obtained out of the GSON serialisation process. But several other formats are available, via the *format* option:

- Apache Jackson's JSON
- Apache Jackson's XML
- raw Java or FST serialisation binaries, for Java applications willing to go Web anyway
- toString() raw output
- printStackTrace() raw output, useful to understand errors, at development time

Regardless of its syntax (JSON, XML, etc), the result always follows the same structure. It consists of 3 sections:

results contains all the results obtained,

warnings contains all the warning issued,

errors contains all the errors that have happened. The presence of an error does not necessarily mean that results are not available. Remote code are able to send error but still pursuing the computing to the end.

The Web server supports the collective computation model of IDAWI. The overlay network is used to route the request to the target components, and to retrieve results back. This design enables any component to be a Web *proxy* to other components. These other components do not need to run the REST service as they will be contacted by the proxy via the overlay. Thanks to this use of the overlay network, a Web requests can reach components even if their host is not reachable via HTTP. When multiple components are queried in parallel, their results appears as a list in the result JSON text.

3.6.3 GET/POST

The web server supports GET and POST requests, indifferently. In the case of POST requests, the binary data sent by the client is mapped to the last parameter of the type operation that is then expected to be of type `byte[]`.

3.7 Deployment

As deployment is a unique feature of IDAWI, it deserves to be extensively described in its own Section 4.

3.8 Other builtin services

This section details services that are builtin the IDAWI package.

3.8.1 Error log

As explained in Section 3.3, operation (remote) errors can be handled programmatically when operations run. In addition to this, when the error log service is loaded, errors are logged on the components where they happened. Considered altogether, the error log services on all components constitutes a distributed database that can be queried as a whole, thanks to the collective computation model.

3.8.2 Exit

The exit service is a very minimal service that kills every (reachable) components in the system. As the system is decentralised, the components are organised as a graph. The exit service visits each component by running a distributed depth-first search algorithm.

3.8.3 Ping

The service does like the POSIX ping command: it sends a message to a given addresse and waits for one or several replies.

3.8.4 System monitoring

This service enables components to retrieve information about load/memory of other components.

3.8.5 Publish/subscribe

This service maintains a number of topics on a component. Topics contains publications submitted by components. Each submission to a topic in a component is broadcasted to all addresses subscribing to this topic.

Publications are stored on the local file system.

3.8.6 Map/Reduce

The MapReduce service is an implementation of MapReduce for IDAWI. It proposes strategies (random, round robin, all-to-all) for mapping tasks to components, and a computation engine with support to fault tolerance.

As it is based on the IDAWI messaging model, it makes it possible to get progress information about tasks at runtime.

3.8.7 Gossiping

The gossiping service provides to other services a way to periodically disseminate information to specific destinations. Destinations can be either specific components, or broadcast addresses.

4 Deployment and bootstrapping

In many distributed systems, the deployment process involves human work to provide deployment configuration files (specify the location of binaries, their destination, how 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 [17, 33] target the deployment of any software, along with their complex dependencies, with support to multiple languages. But in fact deployment is hardly automatizable because it is closely dependent to the architecture of the systems, of the applications, as well as to the technologies they involve. For this reason, most distributed libraries come with *ad hoc* deployment procedures, like in ProActive [1], Jade [10, 33], and OSGi [39]. OSGi is worth detailing here at it exposes its deployable elements (called *bundle*) to user applications, which can manipulate it at runtime. Also OSGi enabled the development of Frogi [15], a solution to the deployment of Fractal [9] components as OSGi services.

All of the aforementioned solutions consider managed clusters (or grids) as their deployment target. Recent works on deploying applications across unreliable networks suggest the use of Bit-torrent as a support to deployment [37]. Considering even more difficult networking conditions, deployment in mobile networks have been investigated in [18]

Deployment/bootstrap in IDAWI address several difficulties:

- computing resources are heterogeneous in terms of hardware, operating systems;
- they may belong to heterogeneous institutions/companies;
- they may share a common file-system;
- computing resources are unreliable;
- computing resources participate in the process by themselves deploying/-bootstrapping to other resources otherwise unreachable.

Also, as IDAWI is geared towards experimentation, developers use it in a particular way: they will use it in a trials and errors mode, which involves frequent subsequent runs, each with a new version of the source code or different dependencies. This particular usage makes by-hand deployment inadequate: deployment must be automatized as much as possible (ideally it is zero-configuration), and it must be quick so as to have a minimal impact on the user experience.

4.1 Requirements

Deployment in IDAWI does the following assumptions:

1. all the executable code (bytecode, scripts, native libraries, etc) of the user application can be found in the classpath;

2. computational resources are accessible via SSH.

These assumptions are a reasonable bet as 1) is a good practise to packaging Java programs, and 2) relies on ubiquitous tools.

All the deployment systems we found impose that their core libraries have been pre-installed on the target nodes. Sometimes they even requires that specific services (daemons) run.

4.2 The problem of shared file systems

In order to fasten the deployment process, IDAWI makes use of parallelism: a component is able to deploy multiple other components simultaneously. In managed grids or clusters, the architecture is documented and users know whether nodes share or not a file system, which enables the operator to configure its deployment process.

As IDAWI is expected to execute on any network conditions, it cannot assume to know if two given node share or not a file-system. This may lead conflicting access to shared file systems.

To solve this, IDAWI first executes a distributed algorithm for the classification of computers according to their file-system. This algorithm computes set of computers sharing a same file-system.

It works like this: let a be an IDAWI node which need to know which nodes in set N their their filesystem. a marks every node b in N with the ID of b, in parallel; the mark is implemented using the *mkdir* system call which is atomic. At the end of the marking process, every nodes in N will have in their file-system the ID of other nodes it shares the file-system with. This information is, then fetched by a (in parallel), enables a to build sets of node sharing their file-systems.

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

- no two computers share a file-system

- there exists a representative node for every file-system

Copying executable code to every computer in this set will ensure that the executable code will be available to all target computers, avoiding any problem of concurrent access.

4.3 Deployment of the binaries

4.3.1 Identifying binaries

Unlike other solutions which most often require the user to provide a configuration specifying the location of binaries and JVM specifications, IDAWI takes benefit of the ability of Java program to discover them at runtime. This information can be obtained from the system properties any Java program has access to. Regardless of how the program was started (via the command line on a server, from an IDE, etc), a Java program knows its classpath as well as the home directory of the JVM it runs into. As any IDAWI component is able to locate all the binaries for the application it belongs to, it gets in capacity to clone its runtime. Clones having the exact same binaries, interoperability problems relation to versions are suppressed.

4.3.2 Transfer of the JVM

Unlike other solutions, IDAWI does not consider pre-installation of a Java Runtime Environment (JRE) on target computer. Just like it imposes that all computers have the very same version of the bytecode (which it taken care of by the synchronization process), it imposes that the JVMs also are the exact same. In other words it considers the JVM to be part of the IDAWI, and that a specific version is required. Because the source code of IDAWI complies to a specific version of the JVM (just like any Java code does), the responsibility of determining the correct version of the JVMs is left to the source itself.

Hence, a parent component checks that the correct JVM is present on the remote computer. If not, the parent forces the target computer to download it from the official website and install it.

This process is executed in parallel on every target computer.

4.3.3 Transfer of the Java bytecode

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 benefits of the Java platform to which it is specific. The deployment unit in IDAWI is the file. In order to minimize the amount of data transferred, thereby fastening the deployment process, IDAWI resort to synchronization instead of doing a complete copy: only what is not already there gets transferred. The process of deployment bytecode and resource is then *incremental*. To do that, IDAWI relies on the *rsync* POSIX utility, which performs the following steps:

1. in parallel,
    - retrieves a set R of triplets $(name, size, timestamp)$ 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 overwritten)
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. Subsequent transfer will consider only the differences from previous ones. As these differences are most of the time insignificant, both the amortized time and space complexity is constant.

Thanks to the determination of shared file systems described in section 4.2, 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.

4.4 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 start a new JVM with an appropriate *main* class that is part of IDAWI's binaries. Using the SSH input/output streams, 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 is was successfully created and initialized

As long as the SSH connection is open, the two components can use it to send/receive messages.

4.5 Killing remote components

When the SSH connection linking a parent component to its child gets closed, if the the child was set to be *autonomous*, it remains alive. Otherwise it terminates. This process is transitive. In turn, if the child component was a parent of non-autonomous children components, those will terminate.

4.6 Deployment as a service

In IDAWI, the functionality of deployment is exposed as a service, making it a decentralized application whose the operations are available to any component/service in the system. In other words, deployment from a component can be triggered remotely by any other component.

5 Related works

5.1 Some elements of History

It is primitive form, back in the early days of programming, distributed computing was implemented using application-specific *request-response* network protocols. The early 80s saw the arrival of Remote Procedure Calls (RPCs), and their popular implementation from Sun Microsystems. RPCs use the calling conventions of the C language which defines functions accepting parameters and returning values. RPCs serializes parameters and returns values using the External Data Representation (XDR) protocol in order to transfer them through the network, from/to computers possibly using incompatible encoding/endianness. This paradigm is still used today in most systems, in spite of the major drawback that it does not permit any from of interaction with a running procedure. More precisely, a remotely running procedure cannot return any information back to its caller until it completes, making it difficult to monitor it, to grab progress information, or to get intermediary results if any, etc. Also RPC cannot get input as they run, which prevents any form of remote execution control. But many computing projects today relate to big data, which implies that they involve time-consuming computations, thereby making the ability to interact crucial to applications. In the 90s, with the increasing popularity of the object-oriented (OO) paradigm, RPCs libraries evolved to middleware solutions such as CORBA and Sun's RMI. At the same time, the message-based appraach got revigorated by the initial release of the Message Passing Interface (MPI) (1992). MPI immediately became a *de facto* standard and still is the communication library which is the most frequently found in HPC projects today. Rapidly, the distributed OO approach got augmented with the high-level architectural abstractions of components and agents. This permitted the development of specifications like Fractal [9], and assorted tools like Julia [**julia**], ProActive [1] (distributed implementations of ProActive), Jade [**jade**], etc. In 2006, when Map/Reduce came along with its reference implementation Apache Hadoop, things changed significantly. A large part of distributed computing communities now stared at the suddenly resurrected Bulk Synchronous Parallel (BSP) paradigm. Designing a distributed application as a map/reduce process became the norm. Map/Reduce had tremendous success, and it opened the path to a new generation of tools like Apache Spark, and the vertex-centric computing frameworks Google Pregel and Apache Giraph. Recent approaches to distributed system foster the use of lock-free parallel multi-thread computing in actor systems *à la* Akka [19] and Zio. Also, for the sake of interoperability, many of these solutions have adopted Service-Oriented Architectures (SOA), exposing WSDL/SOAP or, more recently, JSON/RestFul services. Also, the current state of Research in distributed computing can be seen through projects like [**javacaetla**] and EmbJXTAChord [7] which sum up the good features found in all past developments in comprehensive tools. IDAWI can be seen as one of these.

All of the aforementioned tools are very different from each other, as they

where developed by communities working in distinct areas, following different targets, etc. Each of them excels at what it is made for, but we have not found one of them that is really effective we use as a research tool. Indeed most often they are geared to production rather than experimentation, meaning that they do not (need to) provide facilities to monitor, debug, understand the internals of distributed algorithms. Also, their application fields generally are restricted to specific use cases, data models, and computing infrastructures. In particular, most of these works are designed to work in grids or managed clusters, which are specifically tailored to distributed computations: they feature shared file-systems, resource managers (like Slurm, Torque, OAR, Globus Toolkit, gLite, and UNICORE), and they usually benefit from a dedicated engineer for maintenance and assistance to users. But in practise, R&D in distributed computing goes way further: mobile Ad hoc networks (MANETs), the Internet of Things (IOT), and fog computing, just to name a few, consider a much more hostile computing environment made of untrustable heterogeneous mobile devices. They reinvigorates the idea that any device can be used to perform computations, like the Sarah [18] and SoNi [22] projects did in the 2000s when mobile phones and PDAs invaded cities, forming spontaneous MANETs. This idea of an ubiquitous computing implies multiscale [2] decentralized multi-hop networks in which heterogeneity, mobility, untrustability, unpredictability and unreliability are the rule. These networks are populated with any kind of communicating devices, like mobile phones, smart devices, workstations, gateways, servers, clouds, etc, which interact by resorting to a variety of wired (Ethernet, Infiniband, etc) and wireless (Wi-Fi, Bluetooth, Wi-Fi direct, etc) technologies. In this environment the availability of an IP stack cannot be always accounted for. Also it imposes software (middleware, libraries, user-applications, etc) to implement a computing model featuring a high degree of elasticity, which grid software does not (need to) have. For three decades, Researchers in the fields of MANETs/WSNs/DTNs have been designing protocols and simulators [25] in order to enable computations in such difficult networking conditions, but few truly distributed software were proposed until the recent [**javacaetla**] and Idawi. Extensive information can be found in topic-specific surveys [6][34].

5.2 Position against existing tools

This section gives an overview of the distributed libraries and frameworks suited to the construction and experimentation of distributed systems for Science. For each of them it provides a short description and it details their advantages and limitations which are of interest in our context, and that are taken in consideration in the design of Idawi. In this article we deliberately omitted lab tools which were designed for a particular proof-of-concept but never reach sufficient maturity for a public distribution/usage.

5.2.1 Simulators

Dealing with a real distributed systems is notoriously difficult, especially when mobility and heterogeneity are involved. Simulators are not the real thing but they are much easier to handle, they are tailored to assist experimentation, and they enable faster prototyping/evaluation of distributed algorithms, which makes them especially handy at design-time. Consequently, many studies in distributed systems resort to simulation, at least in a first stage. Conversely simulators are primarily used by academics, to the purposes of Research and teaching. *ns* is the de facto standard when it comes to the simulation of networked systems. It consists of a C++ core wrapped by Python modules. This architecture is very common in modern systems as it offers performance close to the speed of compiled code, and at the same time the flexibility of Python scripting language.
→*But* IDAWI *takes another angle: it is written entirely in Java, as this provides better portability while keeping comparable performance. Also the flexibility of Java has improved a lot of the recent advances of the language, in particular thanks to the introduction of lambdas/functional programming and the addition of numerous facility methods to the standard API.*
Executions of *ns* generate timestamped traces, which can then be statistically or graphically analyzed using tools like R, GNUPlot, Matplotlib, etc. *ns* comes with a great wealth of communication protocols, and its behavior is generally trusted, despite a significant degree of inaccuracy of its physical layer had been pointed out [8]. Unfortunately *ns* is a complex tool with a steep learning curve. Also, as it was designed to simulate wired networks, its support for mobility and distributed computations is scarce. This specific concern is tackled by a specific class of simulators like SimGrid [13] which focuses on the simulation of large MPI computations, and GSSim [5] which is geared towards experimentation related to resource management. These simulators all consider the grid at the sole computing environment, which make them hardly usable for the investigations of algorithms targeted to more difficult computing environment like the IOT, fog computing, etc. For this reason, a new class of simulator emerged, with tools like NetSim [42], MadHoc [30] and JBotSim [14]. These tools propose models for mobility, for intermittent networking, as well as APIs at a higher level of abstraction, closer to the concerns of applications developers.

5.2.2 MPI

The *de facto* standard to parallel/distributed computing as soon as it was initially released in 1992. MPI is still today the most popular library. Its written in C but has bindings to many other languages. Its API is geared towards communication: it consists of primitives at a low level of abstraction, which makes it highly flexible, but does not help structuring the design of distributed systems in any way. IDAWI takes the best of both worlds: it provides a SOA-style framework to the construction of distributed application but keeps, within this framework, communication schemes at a low-level of abstraction, so as to

provide a high degree of flexibility. MPI features a large set of primitives for both point-to-point and group (collective, gather and reduce operations) communication and synchronization. It supports synchronous and asynchronous messaging.
→*Communication in* IDAWI *is collective an asynchronous by nature, thereby enabling point-to-point communication as a particular use case. Synchronicity is achieved in* IDAWI *by the use of blocking queues.*
It has network agnosticism of the transport layer, and it comes with drivers for TCP, UDP and Cisco usNIC. MPI-3 introduces explicit shared-memory programming.
→*Shared memory is not available in* IDAWI*: communication is achieved via the use of messages so as to eliminate the risk of concurrent access to a shared data.*
Operating system processes are first-class citizens within a MPI computation: they are what MPI considers as the communicating entities. As a consequence, no two entities can live in the same process. This limitation hinders the execution of large distributed systems, as the number of processes in a running OS is limited.
→*Scalability is a major concern of* IDAWI*, which makes it possible to have numerous entities in a same JVM (OS process).*
Another major drawback of MPI is its support for structured data. It is written in C but structures (in the sense of the C language) are not directly usable within MPI: they must be wrapped in to MPI-specific data types. This is an obstacle to the design of complex distributed systems as interchanged data cannot be structured.
→IDAWI *messages can carry any (serializable) object*
.

5.2.3 JXTA

Developed by Sun Microsystems until 2010, JXTA was a object-oriented framework for P2P networking. It defines an overlay network of communicating *peer*s. JXTA has a hierarchical organizations which defines that peer belong to groups, and groups can be nested. Just like MPI (and many others), JXTA has network agnosticism. Peers are able to exchange messages regardless of the underlying network topology. On top of its agnostic network layer and its P2P architecture, JXTA features a strategy enabling peers to interact even if they are separated by firewalls and NATs.
→*There is no such notion of group in* IDAWI*. Peers organisation is completely flat. Note that a structure can be defined at application-level.*
Two peers can communicate using virtual channels, named pipes for asynchronous unicast connections or JXTA-sockets for synchronous unicast connections. Pipes can be reliable or unreliable, whereas sockets are always reliable. In addition, JXTA provides also *propagate pipes*, which ensure multicast communications within a group.

5.2.4 P2P-MPI

The Inria middleware P2P-MPI [36] project introduces a message-passing P2P computing framework providing a communication primitives inspired by those of MPI, hence its evocative name. P2P-MPI was developed at Strasbourg University in France. It is now discontinued but it had a number of good features which make it worth mentioning in this paper. Firist it is built on the idea of a decentralized model, though not completely flat as it relies on the presence of super-nodes.
→*There are no super-peer in* IDAWI*: all peers are seen equal in terms of responsibility.*
Second, it proposes an unique deployment system which automatically migrates code and data to remote hosts prior to executing code on them. Third, P2P-MPI does dynamic discovery of computational resources in the network, which allows it to allocate execution requests to resources. The allocation strategy can be configured to perform replication. P2P-MPI has a Swing-based graphical visualization tool to navigate through the platform.
→*A web-based navigation interface for* IDAWI *is under construction. It uses the REST interface o fthe plateform.*
P2P-MPI is written in Java but it does not have an object-oriented design: P2P-MPI is an implementation of the MPJ specification, which defines only primitives. No objects needed.
→*Far beyond communication,* IDAWI *proposes a complete framework to guide the design of distributed applications, and this framework is object-oriented.*
Until its last release, P2P-MPI relied on JXTA. JXTA got then discarded because of its too high complexity.

5.2.5 JMS

The Jakarta Messaging API (formerly known as Java Messaging API) is a specification for message-oriented communication that originates from Sun Microsystems. Just like IDAWI, it has real-time consideration by considering the expiration of messages. It supports both one-to-one and many-to-many (publish/subscribe) communication. Notable implementations of JMS include RabbitMQ and ActiveMQ [16]. It is to be mentioned that the latter can be set to work in a P2P setting. Similarly to IDAWI JMS defines composite destinations as a way to enable multicast; and it features a web server which exposes a REST interface.

5.2.6 0MQ

ZeroMQ is not to be confused with RabbitMQ and ActiveMQ which both relate to JMS, It is not a distributed application framework, it is a distinct *communication* library. Like JMS, 0MQ also both enables data-transfer in one-to-one and many-to-many modes. A strong feature of 0MQ it the interoperability of components written in different programming languages. 0MQ has a fully flat P2P topology: there is no centralization whatsoever. In top of its communication

layer, it supports RPCs. This RPC system has a feature of interest: at runtime, when the remote function raises an exception, that exception is forwarded back the the caller, which raises it on the client side. This mechanisms makes error management on the client transparent.
→*This features is implemented in* IDAWI*, in addition to a specific error management service enabling distributed application to adapt their behavior when errors occur.*
It efficiently exploits TCP connections by enabling multiple concurrent RPC calls over one single TCP connection.
→*The TCP driver of* IDAWI *also does this.*
0MQ also has support for redundancy of executions.
→*Redundancy in* IDAWI *is a inherent consequence of its collective communication model.*
It has been frequently reported that ZeroMQ has much higher overhead than MPI, which makes it a poor candidate to HPC. It is written in C but has bindings to many languages, including Java.

5.2.7 JGroups

JGroups is a toolkit for reliable messaging. To this target, it has a unique high-level mechanism which re-transmit lost messages if necessary, and which automatically discard crashed nodes. Like many other toolkits, it supports one-to-one and one-to-many communication, and provides network agnosticism over TCP and UDP. JGroups represents message recipients as literals (strings), thereby not exploiting the compiler.
→*It is to be noted that in many tools, remote elements are represented a string value storing their name. This impedes the compiler to check the validity of the name as the expression of this name does not relies on anything known by the compiler. In* IDAWI*, remote services are identified by their class. It becomes then impossible to mistype the name of a service as this would make the compiler generate an error. In other words, in* IDAWI *service identification relies on the type system.*
It does fragmentation of large messages.

5.2.8 RMI

RMI (Remote Method Invocation) is the Java builtin mechanism for calling remote code. It is mentionned here for it relies on the notion of stubs, which are the local counterpart of remote object to which they act as a bridge to. A stub has the same interface as its implementation object, thereby bringing transparency of distribution. But in reality transparency cannot be fully achieved for (at least) the following reason: a call of a local method returns a reference to a local object (unless this object is a primitive or immutable type in such case is it copied) but calling a remote one always returns (via serialization) a copy. Because the call in syntactically identical is either case, the caller must know if its calling a local or remote method, which contradicts the principle of

transparency.
→*This ambiguity related to the way objects are returned by reference of by value, depending on where they actually are, is one of the reasons why* IDAWI *opts for explicit distribution rather than transparency.*

5.2.9 ProActive

ProActive [12] is a parallel/distributed library which was formerly developed in our lab at Côte D'Azur University. It is now maintained and extended by Activeon. It is a industrial-strength Java middleware designed around the concept of an active object. The principle of active object is to decouple the invocation of methods from their execution, which are then executed its different threads, as each active object in the system runs in its own thread. A major issue of having one thread per active object is the impossibility to scale, as the number of threads is limited.
→*For the sake of scalability,* IDAWI *uses a fixed number of threads, which are shared by entities.*
The goal is to enable parallelism by delegating method executions to a scheduler, and distribution by making the location of active-objects transparent to others. In ProActive all method calls are asynchronous. When invoked, the client is immediately provided with a future object, which is of the same class of the return type of the method. This future object, whose the creation if implicitly and the class can be dynamically generated using the JavaAssist technology, enables the client to handle the future just like if it were a definitive result, but also to use future-specific methods like waitByNecessity(). However when the method returns a primitive or an instance or a final class (or more generally a class that cannot be inherited from, which ProActive calls reifiable[1]) no future can be produced, preventing asynchronism.
→*Similarly, in* IDAWI *communication is always asynchronous, however synchronicity can be achieved via the use of message queues (which have a similar role than futures in this context.*
ProActive provides a P2P network model atop TCP which provide resilience upon failure. Unfortunately it does not accommodate NATs and firewalls.

5.2.10 Akka

A common approach to parallel computing is the use of shared data whose the access is ruled by locks. But using locks hinders performance and reasoning upon shared access is error-prone. The actor model is proposed as a solution to this problem. Actors do not share date and they interact with one another by sending messages. Message are processed in a sequence. Upon the reception of a message, and actor can modify its internal state, and send messages to other actors. As the actor model involves no concurrent access to data, to

[1] not to be confused with Oracle's definition of a reifiable type which is a type whose type information is available at runtime

enables lock-free programming. The truth is that it is commonly omitted that the access to the message boxes is a concurrent operation as mailboxes can be fed by message-delivery layer and fetched by the owner actor at the same time. Akka is a actor framework. Unlike ProActive and Monix, Akka has by default bidirectional communications. Its extension akka-stream is about defining workflow of processing data. But streams have single output, unlike Idawi.

In Akka, every single message targeted to a given actor trigger its execution, meaning that an execution cannot be fed by more that 1 message. The computing scheme of Akka is hence purely event-based.
→*On the contrary Idawi's operation are triggered by messages specifically tagged has "trigger" messages. Other messages are delivered to the input queue of the running operation. If this allows event-based computing à-la Akka, it also enables imperative programming of operations, which is desirable in many situations.*

5.2.11 ParallelTheater

ParallelTheater [35] defines the notion of *theater* which acts as a container for actors. A JVM can contain only one theater at a time. In ParallelTheater, actors belonging to a same theater are invoked sequentially: no parallelism occurs within a theater. Like in JMS and IDAWI, a real-time flavor is brought by messages having a timestamp as well as a *deadline*.

For communication, ParallelTheater relies on the RPC paradigm. Client theaters invoke methods on others by using serialization. A method is candidate to invocation is annotated. Just like in [38] and unlike IDAWI, methods are identified by a string value: the compiler cannot verify the existence of a method. A similar weakness of static checking is also found when parameters are passed to the method. Indeed parameters are passed as varargs, which is flexible and easy to implement, but prevents any assistance from the compiler as there is no specification of parameter types. This is an important source of mistake when writing the code. Like Akka, ParallelTheater and IDAWI has lock-free parallel computations.

5.2.12 JavaCà&Là (JCL)

JCL [38] is a fresh HPC middleware with IOT flavors, in the same vein à IDAWI. It gathers concepts and technologies in both IOT and HPC. Its boasted objective is to federate in a single tool a set of relevant technologies that can be found in different tools in both worlds. It exposes 3 programming model: event-based, distributed shared memory (it provides an implementation of a distributed map) and task-oriented. The programming model of IDAWI mixes event-based and imperative programming. On top of this, a map/reduce service provides the ability to reason in terms of tasks.

5.2.13 ActorEdge

ActorEdge [3] relies on the actor model. Like ProActive, each actor has its own control thread. For the sake of interoperability, ActorEdge uses JSON formatted messages over TCP/IP connections.
→IDAWI *opted for binary-encoded messages generated by serialization of objects, which provide higher throughput, but hinders interoperability with non-Java software. For this reason, every entity in* IDAWI *has a REST interface accepting a number of text-based format like JSON and XML.*
ActorEdge is written in Objective-C.

5.2.14 GoPrime

GoPrime [11] is a fully decentralised middleware for the adaptive self-assembly of distributed services. GoPrime proposes a model for QoS which encodes the quality of various characteristics of services (such as reliability, accuracy, speed, etc) as numeric vectors, thereby allowing the clients to estimate the adequacy of services to their own specific requirements. It uses REST for communication Just like IDAWI, GoPrime core features a number of *manager*s responsible for providing system-level functionality such as gossiping, service deployment, etc.

5.2.15 EmbJXTAChord

EmbJXTAChord [7] aims at combining the good features of existing tools into a new one. It is decentralized and enable transparent routing. It does not support UDP, which we believe is a lack, but does support Bluetooth It tackles integration issues more than computing. Service architecture based on a RESTful-API Support for HTTP tunneling and NAT traversal

References

[1] Giovanni Aloisio, Massimo Cafaro, and Italo Epicoco. "A Grid Software Process". In: *Grid Computing: Software Environments and Tools*. Ed. by José C. Cunha and Omer F. Rana. Springer, 2006, pp. 75–98. DOI: 10.1007/1-84628-339-6_4. URL: https://doi.org/10.1007/1-84628-339-6%5C_4.

[2] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. "Automatic deployment of distributed software systems: Definitions and state of the art". In: *J. Syst. Softw.* 103 (2015), pp. 198–218. DOI: 10.1016/j.jss.2015.01.040. URL: https://doi.org/10.1016/j.jss.2015.01.040.

[3] Austin Aske and Xinghui Zhao. "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*. Ed. by Ashiq Anjum et al. ACM, 2017, pp. 199–200. DOI: 10.1145/3147213.3149214. URL: https://doi.org/10.1145/3147213.3149214.

[4] Mohammad Sadegh Aslanpour et al. “Serverless Edge Computing: Vision and Challenges”. In: *ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021*. Ed. by Nigel Stanger et al. ACM, 2021, 10:1–10:10. DOI: 10.1145/3437378.3444367. URL: https://doi.org/10.1145/3437378.3444367.

[5] Slawomir Bak et al. “GSSIM - A tool for distributed computing experiments”. In: *Sci. Program.* 19.4 (2011), pp. 231–251. DOI: 10.3233/SPR-2011-0332. URL: https://doi.org/10.3233/SPR-2011-0332.

[6] Sharu Bansal and Dilip Kumar. “IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication”. In: *Int. J. Wirel. Inf. Networks* 27.3 (2020), pp. 340–364. DOI: 10.1007/s10776-020-00483-7. URL: https://doi.org/10.1007/s10776-020-00483-7.

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

[8] Alessandro Bazzi and Gianni Pasolini. “On the accuracy of physical layer modelling within wireless network simulators”. In: *Simul. Model. Pract. Theory* 22 (2012), pp. 47–60. DOI: 10.1016/j.simpat.2011.11.004. URL: https://doi.org/10.1016/j.simpat.2011.11.004.

[9] Gordon S. Blair, Thierry Coupaye, and Jean-Bernard Stefani. “Component-based architecture: the Fractal initiative”. In: *Ann. des Télécommunications* 64.1-2 (2009), pp. 1–4. DOI: 10.1007/s12243-009-0086-1. URL: https://doi.org/10.1007/s12243-009-0086-1.

[10] Sara Bouchenak et al. “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. DOI: 10.1109/CLUSTR.2006.311842. URL: https://doi.org/10.1109/CLUSTR.2006.311842.

[11] Mauro Caporuscio et al. “GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly”. In: *IEEE Trans. Software Eng.* 42.2 (2016), pp. 136–152. DOI: 10.1109/TSE.2015.2476797. URL: https://doi.org/10.1109/TSE.2015.2476797.

[12] Denis Caromel, Alexandre di Costanzo, and Clément Mathieu. “Peer-to-peer for computational grids: mixing clusters and desktop machines”. In: *Parallel Comput.* 33.4-5 (2007), pp. 275–288. DOI: 10.1016/j.parco.2007.02.011. URL: https://doi.org/10.1016/j.parco.2007.02.011.

[13] Henri Casanova et al. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: http://hal.inria.fr/hal-01017319.

[14] Arnaud Casteigts. "JBotSim: a tool for fast prototyping of distributed algorithms in dynamic networks". In: *Proceedings of the 8th International Conference on Simulation Tools and Techniques, Athens, Greece, August 24-26, 2015*. Ed. by Georgios Theodoropoulos. ICST/ACM, 2015, pp. 290–292. DOI: 10.4108/eai.24-8-2015.2261067. URL: https://doi.org/10.4108/eai.24-8-2015.2261067.

[15] Mikael Desertot, Humberto Cervantes, and Didier Donsez. "FROGi: Fractal Components Deployment over OSGi". In: *Software Composition - 5th International Symposium, SC@ETAPS 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*. Ed. by Welf Löwe and Mario Südholt. Vol. 4089. Lecture Notes in Computer Science. Springer, 2006, pp. 275–290. DOI: 10.1007/11821946_18. URL: https://doi.org/10.1007/11821946%5C_18.

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

[17] Areski Flissi et al. "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*. IEEE Computer Society, 2008, pp. 177–184. DOI: 10.1109/CCGRID.2008.59. URL: https://doi.org/10.1109/CCGRID.2008.59.

[18] Frédéric Guidec. *Déploiement et support à l'exécution de services communicants dans les environnements d'informatique ambiante*. 2008. URL: https://tel.archives-ouvertes.fr/tel-00340426.

[19] Philipp Haller and Martin Odersky. "Actors That Unify Threads and Events". In: *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*. Ed. by Amy L. Murphy and Jan Vitek. Vol. 4467. Lecture Notes in Computer Science. Springer, 2007, pp. 171–190. DOI: 10.1007/978-3-540-72794-1_10. URL: https://doi.org/10.1007/978-3-540-72794-1%5C_10.

[20] Luc Hogie. *A SDK enabling the generation of progress/feedback information for algorithms*. https://github.com/lhogie/long_process. 2018-2019.

[21] Luc Hogie. *JOAR: a library for manipulating processes running on OAR clusters from Java*. https://github.com/lhogie/joar. 2018-2019.

[22] Luc Hogie. "Mobile Ad Hoc Networks: Modelling, Simulation and Broadcast-based Applications. (Réseaux Mobile Ad hoc : modélisation, simulation et applications de diffusion)". PhD thesis. University of Luxembourg, 2007. URL: https://tel.archives-ouvertes.fr/tel-01589632.

[23] Luc Hogie. *raw-csv: a library for CPU/memory-efficient parsing of CSV data*. https://github.com/lhogie/raw-csv. 2018-2019.

[24] Luc Hogie. *Tooools: a toolkit for scientific software developments in Java*. https://github.com/lhogie/tooools. 2001-2021.

[25] Luc Hogie, Pascal Bouvry, and Frédéric Guinand. "An Overview of MANETs Simulation". In: *Electron. Notes Theor. Comput. Sci.* 150.1 (2006), pp. 81–101. DOI: 10.1016/j.entcs.2005.12.025. URL: https://doi.org/10.1016/j.entcs.2005.12.025.

[26] Luc Hogie and Nicolas Chleq. *BigGrph - a distributed graph library*. https://github.com/lhogie/biggrph/. 2014-2017.

[27] Luc Hogie and Nicolas Chleq. *Jacaboo: a library for the deployment of Java distributed applications*. https://github.com/lhogie/jacaboo. 2014-2018.

[28] Luc Hogie and Nicolas Chleq. *Live-Distributed Java Objects: a middleware for the manipulation of objects spanning on multiple computers*. https://github.com/lhogie/ldjo. 2014-2018.

[29] Luc Hogie and Nicolas Chleq. *Octojus: a middleware for parallel RPC*. https://github.com/lhogie/octojus. 2014-2018.

[30] Luc Hogie et al. "A Context-Aware Broadcast Protocol for Mobile Wireless Networks". In: *Modelling, Computation and Optimization in Information Systems and Management Sciences, Second International Conference, MCO 2008, Metz, France - Luxembourg, September 8-10, 2008. Proceedings*. Ed. by Le Thi Hoai An, Pascal Bouvry, and Pham Dinh Tao. Vol. 14. Communications in Computer and Information Science. Springer, 2008, pp. 507–519. DOI: 10.1007/978-3-540-87477-5_54. URL: https://doi.org/10.1007/978-3-540-87477-5%5C_54.

[31] Luc Hogie et al. *Grph - a high-performance graph library for mixed graphs*. https://github.com/lhogie/grph/. 2008-2014.

[32] *JMaxGraph - a graph library for the computation of huge graphs*. https://github.com/lhogie/jmaxgraph/. 2018-2019.

[33] Sébastien Lacour, Christian Pérez, and Thierry Priol. *Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids*. Research Report PI 1757. 2005, p. 21. URL: https://hal.inria.fr/inria-00000645.

[34] Anne Hee Hiong Ngu et al. "IoT Middleware: A Survey on Issues and Enabling Technologies". In: *IEEE Internet Things J.* 4.1 (2017), pp. 1–20. DOI: 10.1109/JIOT.2016.2615180. URL: https://doi.org/10.1109/JIOT.2016.2615180.

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

[36] Choopan Rattanapoka. "P2P-MPI : A fault-tolerant Message Passing Interface Implementation for Grids". PhD thesis. Louis Pasteur University, Strasbourg, Alsace, France, 2008. URL: https://tel.archives-ouvertes.fr/tel-00724132.

[37] Steven J. H. Shiau et al. "A BitTorrent Mechanism-Based Solution for Massive System Deployment". In: *IEEE Access* 9 (2021), pp. 21043–21058. DOI: 10.1109/ACCESS.2021.3052525. URL: https://doi.org/10.1109/ACCESS.2021.3052525.

[38] Leonardo de Souza Cimino et al. "A middleware solution for integrating and exploring IoT and HPC capabilities". In: *Softw. Pract. Exp.* 49.4 (2019), pp. 584–616. DOI: 10.1002/spe.2630. URL: https://doi.org/10.1002/spe.2630.

[39] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1*. http://www.osgi.org/Specifications. 2007.

[40] Thibaud Trolliet et al. "Interest Clustering Coefficient: A New Metric for Directed Networks Like Twitter". In: *Complex Networks & Their Applications IX - Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications, COMPLEX NETWORKS 2020, 1-3 December 2020, Madrid, Spain*. Ed. by Rosa M. Benito et al. Vol. 944. Studies in Computational Intelligence. Springer, 2020, pp. 597–609. DOI: 10.1007/978-3-030-65351-4_48. URL: https://doi.org/10.1007/978-3-030-65351-4%5C_48.

[41] Thibaud Trolliet et al. "Interest Clustering Coefficient: a New Metric for Directed Networks like Twitter". In: *CoRR* abs/2008.00517 (2020). arXiv: 2008.00517. URL: https://arxiv.org/abs/2008.00517.

[42] Akanda Wahid-Ul-Ashraf, Marcin Budka, and Katarzyna Musial. "NetSim - The framework for complex network generator". In: *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference KES-2018, Belgrade, Serbia, 3-5 September 2018*. Ed. by Robert J. Howlett et al. Vol. 126. Procedia Computer Science. Elsevier, 2018, pp. 547–556. DOI: 10.1016/j.procs.2018.07.289. URL: https://doi.org/10.1016/j.procs.2018.07.289.