A Decentralized Web Service Infrastructure for the Interoperability of Applications in Multihop Dynamic Networks

Luc Hogie

A Decentralized Web Service Infrastructure for the Interoperability of Applications in Multihop Dynamic Networks

Luc Hogie

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

Abstract—The advent of mobile computing has changed the shape of distributed computing. Mobile devices now have enough computational capabilities to take part of distributed computations. Thanks to their wireless networking interfaces, they can have direct interactions with their peers, self-organize in dynamic multi-hop networks, like the Internet of Things (IoT), the Edge, Mobile Ad hoc Networks (MANETs), etc. Such networks exhibit a number of inherent characteristics (mobility, unpredictability, scarce connections, etc) which make it difficult the use of their resources by (distributed) applications. Because existing middleware for distributed computing was most often designed to operate on clouds/grids/clusters, it can hardly be used in these networks. IDAWI belongs to a new class of middleware solutions which rely on decentralization to federate the resources of dynamic multi-hop networks, to the purpose of facilitating the design and implementation of distributed applications in networks of mobile devices. IDAWI goes further by proposing a collective stream-oriented computing model that matches the very nature of such networks. This permitted us to design, implement and describe in this article the architecture of a specific decentralized Web backend exposing as a whole the resources of a dynamic multi-hop network as a set of Web services. In order to profit from the stream-oriented and collective natures of IDAWI 's computation model, these Web services make a clever use of several standard Web technologies they rely on. All the concepts presented in this paper can be found in an Open Source reference implementation.

Index Terms—Decentralised systems, overlay networks, middleware, edge, fog, IoT, Java, distributed computing, idawi

Today, thanks to the advances of micro-electronics, devices of very diverse natures are capable to perform computations. The computational landscape, previously consisting of only computers in grids or managed clusters, has been augmented by newcomers. Indeed, mobile devices, such as laptops/tablets/smartphones, which were previously restricted to personal computing applications, now have computational capabilities that enable them to take part of computation-intensive distributed applications. In particular, considering the computational power, laptops are now on par with mid-range fixed workstations, and tablets and smartphones have no less than half of their power. In this context, smartphones are particularly worth considering. Indeed, billions of them have been sold in years, making them the most common computing
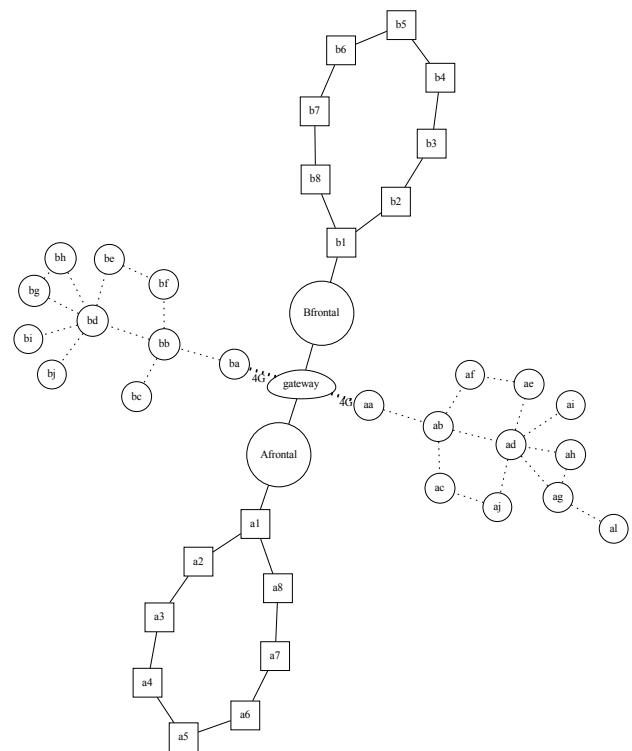
Fig. 1. An example of a network consisting of mobile nodes communicating with each other through wireless links (dotted lines). They 4G broadband connection enables them to connect to a gateway to the Internet, giving them access to frontals of clouds/grids/clusters whose worker nodes are protected by a firewall.

devices on the planet today.

All these computation-able devices are very heterogeneous in terms of capacity, speed, memory, mobility, as well as in their ability to interact with other devices. Those in cloud/clusters usually communicate with each other via both Infiniband and IP-over-Ethernet. While Infiniband provides computers in

clusters with a high-throughput low-latency communication media, IP connects them to the Internet (and distant clusters), via optical fiber connections. Mobile devices all use broadband 4/5G or Wi-Fi to connect to routers, and they can themselves behave as routers by sharing their connection via Wi-Fi. Also, they all use Bluetooth to communicate with the devices of the IoT. Finally, the less known the Wi-Fi ad hoc mode and Bluetooth enables them to have direct P2P connections to *each other*. This paper considers such networks formed out of the diversity of devices and networking technologies, that serve as the hardware infrastructure of IoT application, Edge computing, etc.

Distributed computing is inherently complex. The advent of mobile devices made it even more complex, for it introduced new constraints and new possibilities. To deal with them, we need to change the way we envisage distributed applications today, by rethinking the middleware that support them. Indeed, existing middleware for distributed computing (like ProActive [1], MPI, JXTA, JMS (Jakarta Messaging API), ActiveMQ [2], JGroups [3], RMI, and Akka) is most often tailored to grids, clouds or clusters. Middleware now needs to take into serious consideration the development of applications for networks involving mobile devices, like Mobile Ad hoc Networks (MANETs), the IoT (Internet of Things), Delay Tolerant Networks (DTNs), the Edge, etc. 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) [4], GoPrime [5], ParallelTheater [6], ActorEdge [7], and EmbJXTAChord [8]. 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.

Because of this, in order to meet the requirements of our lab applied Research projects in the fields of graph algorithms [9], networking [10], decentralized protocols [11] for MANETs and more recently the IoT, etc, we had initiated the design and development of a new middleware called IDAWI [12][13]. In IDAWI, we have designed, implemented and arranged together many features like a structuring component-oriented application model in the style of SOA, and automatic decentralized deployment system [14], etc, as well as a flexible stream-oriented collective computational model suited to the requirements of IoT applications, and which is automatically accessible to other applications via its Web interface. This feature empowers IDAWI with deep ability to interoperate with other tools, which is what this article is about.

This article tackles the problem of providing an unified framework exposing the resources of mobile multi-hop networks (like the IoT) to applications. To solve it, we propose an innovative decentralized Web service infrastructure, relying on the IDAWI framework, to which it exploits many base features such as its decentralized nature, SOA (Service Oriented Architecture) design, and collective computing model. This Web-service infrastructure exposes to exogenous applications the resources of the underlying dynamic multi-hop network as a *unified* set of Web services, which can be used regardless of where they are called from in the network: the system provides agnosticism to location. Unlike usual Web services which conform to the RPC (Remote Procedure Call) model, the system we propose makes a clever use of existing Web standard technologies to achieve a stream-oriented mode of operation, conforming the execution model of the IDAWI framework.

All ideas and algorithms presented here have been implemented in the Open Source IDAWI middleware.

Section I describes the concepts within IDAWI which are required to comprehend the Web service infrastructure that is then presented in Section II.

I. THE IDAWI MIDDLEWARE

IDAWI is an innovative framework for distributed computing in mobile multi-hop networks, like the IoT, the Edge, MANETs, etc. It synergizes powerful concepts found in existing tools, and many ideas we introduced all along its design and development. As a result, its Open Source Java reference implementation exhibits a number of unique features which constitute effective solutions to problems not solved by existing middleware, like automatic decentralized deployment, operation through firewalls/NATs, and stream-based Web interfacing, hence this article.

The source code of IDAWI is available at

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

In a few words, the polished object-oriented design of IDAWI proposes a SOA-like application model atop a fully decentralized component system. It proposes flexible *collective* message-oriented communication and computation models through an API supporting both reactive (asynchronous) and imperative (synchronous) programming. In order to avoid confusions, instead of introducing a specific taxonomy for our elements of design, we foster the reuse of the concepts familiar to Web-services communities, when possible.

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

This structuring model is described in the coming sections.

A. An overlay network of components

IDAWI defines a component model. Components represent business entities. They form a multi-hop overlay network

providing agnosticism of the underlying network topology. In this overlay, two given components are neighbors if they *do* have direct interactions. Any two components can be neighbors unless the underlying network infrastructure prevents it. This may happen in the presence of NATs/firewalls, or because of inherent constraints of wireless technologies like a limited range, hidden nodes, etc. Two non-neighbors must then rely on intermediary nodes to communication, which then behave as *routers*. This overlay network of components is then multi-hop by nature. It can be seen as a *dynamic mixed multi-graph*.

In the usual use case, there will be only one component per device. The role of this component is then to represent its host device: it acts as an *agent*. But, in order to enable the simulation of large systems, components can deploy other components in their Java Virtual Machine (JVM) or in any other JVM in the same device. Emulation can then be achieved by deploying multiple components to specific devices whose role is to simulate parts of a large system.

B. An application model in the style of SOA

Components expose their functionality via *service*s. A service holds data and implements functionality about the specific concern it is about. Services must communicate with other services in other components by sending/receiving messages. To this purpose, services have message *queues*. Services can communicate with other services in the same component using shared memory.

Services are the standard way to incorporate functionality in an IDAWI system. In this regard, an application consists of a set of services. System-level functionality is also brought by specific built-in services like *transport* and *routing* services, which enable components to send messages to their neighbors, and to forward messages not targeted to them. Other built-in system services endow components with abilities to deploy other components, to shut down the entire system, to provision/discover services, to execute map/reduce computations, etc.

Services expose their functionality via *operations*. Operations constitute the *only way* to execute code in an IDAWI system. An operation is a routine that can be triggered remotely from any other one in the system. Just like a method, it has an ID. This ID is a literal, whose validity is hence guaranteed by the compiler. While a method is fed by a list of parameters before its starts, an operation takes as input a queue of messages. This queue provides a running operation with new input (messages) on-the-fly. Similarly, while a method returns a value after it has completed, an operation does not: instead it (can) send output messages while it executes.

This example shows the standard form for declaring an operation:

```
public class MyOperation extends Operation {

    @Override
    public void exec(MessageQueue input) {
        replyTo(m, "some output");
    }
}
```

A specific kind of operation, called *typed operation* makes more natural the expression of operations which conform to the RPC model. The parameter list of these operations provides a specification for their input. This makes it possible automatic conversion of input data. For example, in our use-case, parameters coming as a list of strings from an URL can be automatically converted to instances of the types expected by the operation. This is an example of declaring a typed operation:

```
public class MyOperation extends TypedOperation {
    public int sum(int a, double b) {
        return a + b;
    }
}
```

Operations execute in parallel, in a JVM-local pool of threads. The number of threads in this pool can be set by the application. The default is "as many threads as the CPU cores". This rule of thumb provides excellent scalability (in the number of simultaneously running operations).

C. A many-to-many *message-based communication model*

At the lowest layer, (running) operations communicate by *explicitly* sending/receiving messages of a bounded size. Sending a message is always *asynchronous* (non-blocking). 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 route it took so far, the ID of the target queue, and routing-specific information like the recipients components, parameters, etc. When a message arrives at destination, it is delivered into a message queue. Queue are then fetched by running operations, in a *synchronous* fashion.

A message queue is a thread-safe container of messages exposing the following primitives: $size()$ gets the number of messages currently in the queue; $take(timeout)$ retrieves and removes the first message in the queue, waiting until the timeout expires if the queue was empty; and $add(timeout)$ tries to add a message to the queue, possibly waiting (until the timeout expires) for an available slot. The use of finite timeouts ensures that no deadlocks will occur in the system.

IDAWI comes with a default routing protocol which suits the very nature of mobile multi-hop networks: it defines a destination address as a triplet (C, e, d) where C is a set of component names (if C is not defined, the address is considered to be a broadcast address); e is the expiration date of the message, and d is the maximum number of hops allowed to travel. This routing protocol is intrinsically *multicast/broadcast*. *Unicast* comes naturally when one single target component is specified in the set of recipient names. This routing scheme permits components to send messages to *any/any set of/all* components up to a given distance, and reachable before a given deadline. In order to address node mobility and scarce connectivity, messages are not dropped after they are forwarded. Instead, they are stored until they expire, and they reconsidered for re-emission each time a new neighbor connects.

D. A collective computation model

IDAWI defines an innovative computing model, based atop the communication model described herein before, from which it inherits the *collective* nature. It defines a special message called the *exec* message, whose reception triggers the execution of a particular operation. Just like any message, the *exec* message can be targeted to multiple components. This enables redundancy and parallelism.

When it is executed, an operation is provided with a message queue that it uses to receive input data. This enables operations to receive multiple messages at runtime. As there is no theoretical limit in the number of messages a queue can accept, operations can have *unbounded input*. An operation can produce output (intermediary results, final result, warnings, exceptions, progress information, etc) at any time by sending messages. Just like their input, operations output is unbounded. In most cases, output will be sent to the sender of the exec message, whose return address is carried by the exec message. But output messages can be sent to any queue in the distributed component-system. To receive output messages, the caller creates a new local queue, called the *return queue* that aims at storing messages from the running operation(s). This message queue can play the role of a *future*. Here again, other running operations may obtain the address of the return queue, and send *directly* messages to it. A running operation may execute another one. This enables *composition of services* and *workflows*.

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

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

E. The collect algorithm

As described in Section I-C, obtaining messages can be done using the synchronous (blocking) $take(timeout)$ and $size()$ primitives of queues, which enable iterating over queues' content. The *collect* algorithm introduces an higher-level approach by proposing an asynchronous/reactive/event-driven API.

Its design is inspired by the iterator API in the HPPC library [15], 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.

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

This second example only obtain the three first messages:

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

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.

```
q.collect(c -> {
        if (c.messages.size() > 1) {
            var newMsg = c.messages.last();
            var previousMsg = c.messages.last(1);
            c.timeout = newMsg.receptionDate - previousMsg.
                receptionDate;
        }
    });
```

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.

```
26        var receivedFrames = new HashSet<Long>();
27
28        q.collect(initialDuration, initialTimeout, c -> {
29            var f = (Frame) c.messages.last().content;
30
31            if (!receivedFrames.contains(f.id)) {
32              receivedFrames.add(f.id);
33              pipe.write(frame.data);
34            }
35
36            c.stop = receivedFrames.size() == 1000;
37        });
38 }
```

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.

II. A WEB INTERFACE TO THE *collect* ALGORITHM

In IDAWI, services are automatically 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 operation 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 operation 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 operation 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.

A. An IDAWI *service*

The Web functionality in IDAWI is provided by a native service, available by default in each component. When a component activates this service, it becomes a *gateway*: the component will be able to receive HTTP requests from Web
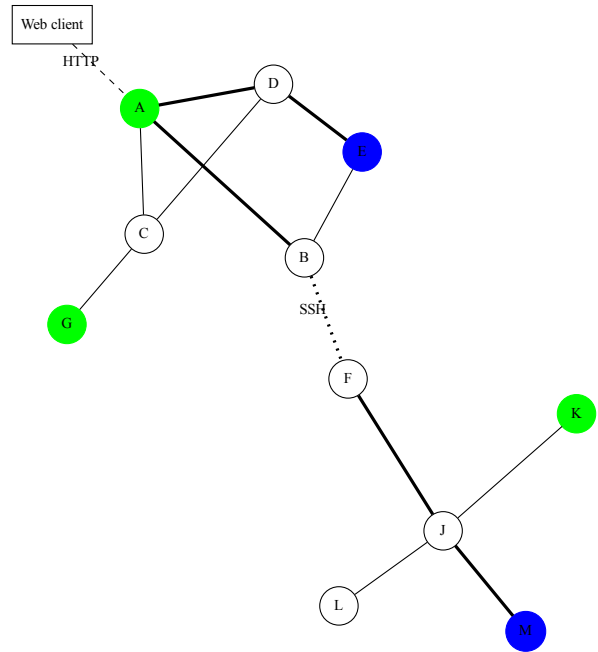

Fig. 2. Gateway components A, K and G run the Web service. An HTTP client (such as a browser) connected to gateway component A will be able, through A, to execute operations on any component in the system. Here, E and M are targeted. Note that M can be reached (via the overlay network) even though it is protected by a firewall. The client can then query M via HTTP just like if it were directly connected to it. The bold edges indicate the path of the operation execution request initiated by A.

clients and translate them to IDAWI's world. To do that this service executes an HTTP server on the device which hosts the component. Executing the Web service entails the execution of an operation into the component system: every operation in the system can be invoked from the Web. The most crucial notion here is that the request is received by the HTTP service of a specific component, but the component that will do the computation can be any component in the system, according to what the routing section of the URL specifies. A gateway component is in reality just an entry point to the entire distributed system. It plays the role of a bidirectional translator from/to the Web. This concept is illustrated by Fig. 3.

Except if it is described as one that will do the computation, the gateway component does not play any other role than translating and relaying input/output data. Whichever gateway is used, the result reported to the Web client will be the same. The only difference will be the way the input/output is routed within the component system.

There is no fundamental need that all components expose the Web service/behave as a gateway. Only one is enough if this meets the need of the software ecosystem the user application lives in. This said, more gateways are present, more flexibility and redundancy will be exhibited.
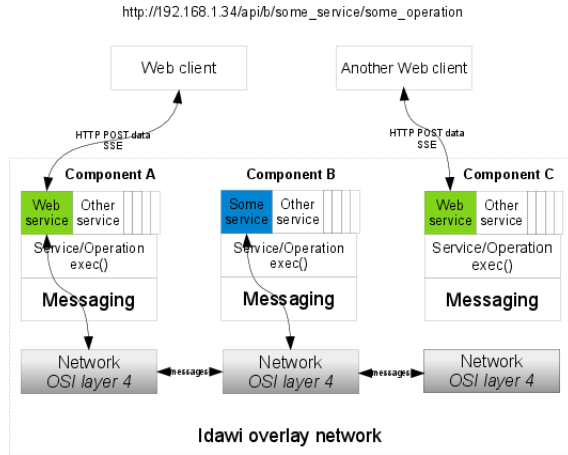
Fig. 3. Components A, B and C form an overlay. A and C run the Web service so they are gateways, translating requests from the Web clients to component B, as specified by the second element of the path of the URL.

B. Passing data

The HTTP standard defines two ways to pass data to Web services: either encoded in the URL, or provided as POST data. The latter allows running operations to be fed at runtime with new input.

1) Structure of the URL: The minimal URL for executing an operation that has no parameter, on one single component, follows this pattern:

$$\text{http://host\_name/api/r/s/o}$$

Where:

- r is the routing information (targets, parameters, etc)
- s the ID of the recipient service on this component
- o the ID of the recipient operation on this service

In the following, the prefix http://host_name/api/ will be purposely omitted in all URLs.

The second path element of the URL describes the input and parameters from the routing protocol that will be used to deliver the message to components. The syntax of this path element depends on the routing algorithm used by the gateway component. Different routing protocols would use a different language to describe the delivery process. When using blind broadcasting, the path element must be left empty as blind broadcast has no parameters nor specific destination. For example, when using the default routing protocol which comes as a built-in service in IDAWI, the following URL requests the execution of the o operation on service s of both components c_1 and c_2.

$$\text{c1,c2/s}$$

Currently, targeting multiple components can only be achieved by mentioning their names in the URL. Specifying no name at all implies the parallel execution of the operation on all reachable components (broadcast). The maximum distance from the source, as well as the deadline for message lifetime are not yet supported by the parser. We are currently working

on defining a better Domain Specific Language (DSL) for the expression of routing rules within a single URL path. In parallel, we are studying the relevance of using base64 URLs or of triggering an operation by sending POST data. In the latter situation, the initial URL-based HTTP call would do nothing but waiting for execution information from POST).

The third path element provides the name of the target service. Here the fully qualified name of the service class is expected. The fourth path element gives the name of target operation is that service. The simple name of the operation inner class is the service class is expected.

Subsequent path elements constitute the parameters of the service as a list of strings. They apply when the target operation is typed (see Section I-B. In the case of a non-typed operation, the exec message carries no initial input. Input is then transferred to the running service via new messages, out of the POST data the gateway receives from the Web client. A call to a typed service then follows this pattern:

$$\text{c/s/o/p1/p2/.../pN}$$

There is a type mismatch between URL path elements, which are strings, and the parameters of a typed operation which are instances of particular classes. To solve this problem, the Web service benefits from the flexibility of the IDAWI middleware: just before executing, operations attempt to convert the parameters they are given to instances of the types they expect (as defined by their signature). In the specific case of the Web system, parameters always are strings, so recipient operations need to be able to convert from strings to objects. All primitive types and many common classes are supported by default, and application can provide specific converters from string to the specific types their operations need.

Finally, the query section of the URL is used to pass information to the Web server—not to the running service. In particular, it is used to set the *initial* values for the:

- duration for the whole collect process
- the timeout for receiving the next message

For example:

$$\text{c/s/o/\textbackslash?duration=10\textbackslash?timeout=1}$$

2) Using POST data: IDAWI's operations have native support for streams, as they accept an unlimited number of input messages. In order to reflect this feature into the Web API, the web service accepts data via POST. POST data arrives in the form of a binary input stream. As the content of IDAWI messages must be bounded in size, this stream of binary is cut into chunks of a fixed size. The Web service then embeds each of these chunks into a message that is sent to the running operation.

C. Obtaining values

HTTP responses do not have to specify the size of the body they carry. More technically, the HTTP standard does not impose the server to specify a `Content-Length`. In theory, this feature enables the client to receive a stream of data, until the connection gets closed, or the client, which has

agreed upon an application-specific protocol, receives End Of Transmission (EOT). Unfortunately, most Web clients have been designed to receive a single bounded response from Web services, and available APIs are also designe that way. It turns out that the most client-side Web APIs make it impossible to deal with multiple responses from the service. This is surprising as their essentially reactive (event-driven) nature is particularly well suited to dealing with multiple responses. Moreover, they propose no way to deal with the HTTP response in a synchronous fashion.

This technical restriction has been a serious problem as IDAWI operations are likely to multiple output messages. An obvious solution to this problem is to use Websockets. But these have a number of disadvantages. First they require their specific support by the Web server, but as far as our knowledge, there is no Web server for Java that is both lightweight and Websocket-compliant, so we could hardly have it working within IDAWI. Second Websockets requires the Web client to open a second connection to the Web server. In our case where the networking infrastructure is likely to have firewalls and NATs, opening an additional Web connection could be impossible.

Another solution is to use a less known Web API called Server Side Events (SSE). SSE enables servers to send events back to clients, which are then notified in a reactive way. The MIME type associated to such events is `text/event-stream`, which indicates that events are made of text. More specifically, an SSE event is a sequence of lines. In order to enable the transfer of binary data back to the client, we define a IDAWI SSE Event (ISSE) as an SSE event consisting of 2 elements:

- a JSON header specifying:
  - the semantics (a description)
  - the syntax of the body
  - a boolean indicating if the body is transcoded in base64
- a body, encoding the content of the message in a given syntax

An example event containing a result from the running operation.

```
1  data: {semantics: "result", syntax: "Jackson JSON"}
2  data: {value: "42", nb_iterations: "100"}
```

An example event indicating that the running operation has almost completed.

```
1  data: {semantics : "progress ratio", syntax : "text"}
2  data: 0.9
```

This last example illustrates the ability to provide binary data in a space-efficient format. A binary image is sent back to

```
1  data: {semantics: "result", syntax: "JPEG", base64="true"}
2  data: GDTGEIERISEVDSVZGIERGIERIGER110CFKSEF995KJSDFKER
3  data: MMSD65ZDFZEDFGGDFKZEIZE675Z49ZFDSAZJMMSD65ZDFZED
4  data: 110CFKSEF995KJSDFKER9ZFDSAZJMMSD65ZDFZEDFGGDFKZE
```

1) Output format: The exec messages generated by the gateway component (on request from the Web client) carry the address of the return queue on this gateway component. The enables output messages of the running operation to be transferred back to the gateway. Upon the reception of such output message, the gateway component will transcode it to a text or binary output format and pass it to the Web client via an ISSE. In order to provide the Web client with maximum information, it is the message itself that will be passed, not just its content.

IDAWI has an object-oriented design. In this regard, a message is an object. But Java objects have no existence in the Web. In order to convert messages to something that can be manipulated by the Web client, IDAWI relies on serialization. Wikipedia gives a very accurate definition of it: "serialization [...] is the process of translating a data structure or object state into a format that can be stored [...] or transmitted [...] and reconstructed later (possibly in a different computer environment)". IDAWI has built-in support to serialize its internal messages to the following text formats: Google JSON (GSON), Jackson JSON, JSONEx JSON, Jackson XML, TOML, YAML. It also supports the following binary formats: native Java serialization format (SER) and FST. The output format can be set by the client by specifying the name of the serializer in the query string of the URL. Unfortunately, the binary formats can hardly be used out of the Java world and none of the serialization tools we tried enable a seamless automatic serialization of *any* Java object. To solve this problem, we designed and implemented a specific Java→JSON serializer called Jaseto [16]. Jaseto focuses on a specific—yet classic—application of serializers: exposing object models to Web applications written in JavaScript, using JSON encoding. To this particular purpose, Jaseto features programmatic customization API (many other tools annotation-based descriptive approaches), which enables highly flexible and deeper customization abilities; and it supports cyclic aggregation, making it possible to serialize *graphs*. Jaseto does not consider deserialization as it does not aim a generating Java objects out of JSON texts.

2) Stopping a running Web service: In the RPC model, a procedure is known to have completed once it has returned a value. This assumption cannot be used in IDAWI as:

- an execution request may trigger the same service on multiple components, in parallel
- any single execution on one component may return multiple values, thus the gateway may receive multiple messages from multiple components
- it is possible that some (maybe all) output messages never reach the gateway component

Because of this, the gateway has no way to determine if a request it had triggered has completed or not. To solve this problem, the termination condition of the collect algorithm can be specified in the URL. It can hence be defined by the Web client. Unfortunately, the textual expression of a termination condition into a URL is severely constrained, and defining a programmatic control of when the *collect* algorithm

should stop collection, IDAWI proposes a number of predefined *termination conditions*, such as:

- wait an EOT message from all the component mentioned in the routing section
- stop when the first result is received
- stop when the first message is received
- stop when the first error is received
- never stop

In any case, the collect algorithm will run until its initial deadline has been reached or the timeout on message reception has not been satisfied.

III. CONCLUSION

Dynamic multi-hop networks, like Edge networks and the IoT, exhibit a number of inherent characteristics (like mobility, unpredictability, scarce connections, etc) which make it difficult the use of the resources by distributed applications. We presented in this article an advanced Web service infrastructure to solves this problem. It relies on the architecture of the IDAWI middleware whose key ideas is to use decentralization to federate resources, to rely on a component-oriented design in the style of SOA, and to run collective "stream of messages"-oriented asynchronous communication and computation model. We managed to translate these features to the Web world by using/extending appropriate Web standard technologies. The ideas presented in this article have all been validated by a reference implementation which has been incorporated into the Open Source code of IDAWI.

Current and future works include the implementation of a graphical demonstrator for the Web service functionality, the definition of a better DSL (Domain Specific Language) for deeper expressiveness of routing rules, the reuse of IWebP on top of POST to provide running operations with new structured input on-the-fly (currently only streams of byte are supported), as well as the deployment of IDAWI-powered applications onto recent Android devices.

ACKNOWLEDGMENT

REFERENCES

[1] D. Caromel, A. di Costanzo, and C. Mathieu, "Peer-to-peer for computational grids: mixing clusters and desktop machines," *Parallel Comput.*, vol. 33, no. 4-5, pp. 275–288, 2007. [Online]. Available: https://doi.org/10.1016/j.parco.2007.02.011

[2] N. Estrada and H. 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. [Online]. Available: https://doi.org/10.1109/CLEI.2015.7360036

[3] B. Ban, "Adding group communication to java in a non-intrusive way using the ensemble toolkit," Citeseer, Tech. Rep., 1997.

[4] L. de Souza Cimino, J. E. E. de Resende, L. H. M. Silva, S. Q. S. Rocha, M. de Oliveira Correia, G. S. Monteiro, G. N. de Souza Fernandes, R. da Silva Moreira, J. G. de Silva, M. I. B. Santos, A. L. L. de Aquino, A. L. B. Almeida, and J. de Castro Lima, "A middleware solution for integrating and exploring iot and HPC capabilities," *Softw. Pract. Exp.*, vol. 49, no. 4, pp. 584–616, 2019. [Online]. Available: https://doi.org/10.1002/spe.2630

[5] M. Caporuscio, V. Grassi, M. Marzolla, and R. Mirandola, "Goprime: A fully decentralized middleware for utility-aware service assembly," *IEEE Trans. Software Eng.*, vol. 42, no. 2, pp. 136–152, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2015.2476797

[6] L. Nigro, "Parallel theatre: An actor framework in java for high performance computing," *Simul. Model. Pract. Theory*, vol. 106, p. 102189, 2021. [Online]. Available: https://doi.org/10.1016/j.simpat.2020.102189

[7] A. Aske and X. 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*, A. Anjum, A. Sill, G. C. Fox, and Y. Chen, Eds. ACM, 2017, pp. 199–200. [Online]. Available: https://doi.org/10.1145/3147213.3149214

[8] F. Battaglia and L. L. Bello, "A novel jxta-based architecture for implementing heterogenous networks of things," *Comput. Commun.*, vol. 116, pp. 35–62, 2018. [Online]. Available: https://doi.org/10.1016/j.comcom.2017.11.002

[9] T. Trolliet, N. Cohen, F. Giroire, L. Hogie, and S. Pérennes, "Interest clustering coefficient: a new metric for directed networks like twitter," *J. Complex Networks*, vol. 10, no. 1, 2021. [Online]. Available: https://doi.org/10.1093/comnet/cnab030

[10] D. Coudert, L. Hogie, A. Lancin, D. Papadimitriou, S. Pérennes, and I. Tahiri, "Feasibility study on distributed simulations of BGP," *CoRR*, vol. abs/1304.4750, 2013. [Online]. Available: http://arxiv.org/abs/1304.4750

[11] L. Hogie, "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, 2007. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01589632

[12] ——, "IDAWI: a decentralised middleware for achieving the full potential of the iot, the fog, and other difficult computing environments," in *Proceedings of MiddleWedge 2022 ACM International workshop on middleware for the Edge. Collocated with ACM/IFIP/USENIX Middleware 2022, Québec, Canada*. ACM, 2022.

[13] ——, "Idawi: a middleware for distributed applications in the IOT, the fog and other multihop dynamic networks," CNRS - Centre National de la Recherche Scientifique ; Université Côte d'azur ; Inria, Research Report, Feb. 2022. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03562184

[14] ——, "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, 2022, to be published.

[15] "Hppc: High performance primitive collections for java," https://labs.carrotsearch.com/hppc.html.

[16] "The jaseto json serializer for java objects," https://github.com/lhogie/jaseto.