Lazy Services: A Service Oriented Architecture based on Incremental Computations and Commitments

Joskel Ngoufo Tagueu, Eric Badouel, Adrián Puerto Aubel, Maurice Tchoupé Tchendji

HAL Id: hal-03353118
https://inria.hal.science/hal-03353118v2

Preprint submitted on 24 Sep 2021

Lazy Services: A Service Oriented Architecture based on Incremental Computations and Commitments [*]

Joskel Ngoufo Tagueu[1], Éric Badouel[2], Adrián Puerto Aubel[3], and Maurice Tchoupé Tchendji[1]

[1]University of Dschang, Cameroon
[2]Inria Rennes Bretagne-Atlantique, IRISA, University of Rennes, France
[3]University of Groningen, The Netherland

Abstract

A service oriented architecture (SOA) aims to structure complex distributed systems in terms of re-usable components, called services. To guarantee a good service interoperability these services must be weakly coupled and their description must be separated from their implementations. The interface of a service provides information on how it can be invoked: the logical location where it can be invoked, the supported communication protocol and the types of its input (parameters) and output (result). Traditionally, a service can only be invoked when its parameters are fully defined and, symmetrically, these services only return their results after they have been totally processed. In this paper, we promote a more liberal view of services by allowing them to consume their data lazily (i.e., as they need it) and produce their results incrementally (i.e., as they are produced). We develop this notion as 'lazy services' by building up from the model of guarded attributed grammars that was recently introduced in the context of distributed collaborative systems. We abstract from this model and limit somewhat its expressiveness so that it can comply more broadly to SOA principles. We introduce an improvement on subscription management to optimize the distributed implementation of lazy services.

Keywords: *Service-oriented computing, Guarded Attribute Grammars, lazy and parallel processing, user-centric systems, micro-services, IoT Workflows, peer-to-peer computing, distributed systems, language-oriented programming.*

[*]This research was conducted in the frame of Lirima Fuchsia Associate Team (`https://lirima.inria.fr`

1 Towards a notion of lazy service

A service oriented architecture (SOA) aims to structure complex distributed systems in terms of re-usable components, called services. Services can be atomic or composed of several other sub-services. They operate independently and autonomously, and communicate via open protocols specified by their interfaces. In this way they are ideal candidates for implementing distributed systems where the components have a certain autonomy and where coordination is asynchronous and decentralized. They have been widely adopted in business-to-business interactions [6] and today constitute the basis of web applications through the notion of web service [15].

To guarantee a good service interoperability, these services must be weakly coupled and their description must be separated from their implementations. The interface of a service provides information on how it can be invoked: the logical location where it can be invoked, the supported communication protocol and the types of its input (parameters) and output (result). A service is thus presented as a black box that provides a precise functionality. The latter is specified by the shape of its input parameters and its output result, regardless of the way the result is produced. It thus corresponds to the mathematical notion of function. Service-oriented systems rely on two formalisms. The first one is used to specify the interface of a service (for example WSDL for web services). The second (like BPEL) expresses how these services can be used to design complex applications.

In order to define a new service through the orchestration or the choregraphy of more basic services, one should be able to derive the interface of the combined service from the interfaces of its components. In order to facilitate this type of modularity while keeping a separation between interface specification and service composition, the traditional approach to SOA restricts to a fairly simple interaction scheme: a service can only be invoked when its parameters are fully defined, and symmetrically, these services only return their results after they have been completely processed. Nonetheless, the SOA principle of asynchronous communication prescribes a message passing mechanism (no shared memory) while allowing a client process to continue its execution without waiting for the answer of the invoked service as long as it does not need it. This principle thus calls for a more flexible view of service composition, that would allow services to consume their parameter data lazily (i.e., as they need it) and produce their results incrementally (i.e., as they are produced).

In this paper, we present a solution for this more general service interaction scheme that improves concurrency. The exchanged data are lists of attribute/value pairs where values can be specific (i.e. determined), but also 'future' values (i.e. subscriptions to values that remote services have committed to produce). These commitments have an impact on the course of future actions (even if the corresponding values are not yet known). For instance, if one asks a report on a file by an expert, the answer can take the form of two attributes: the first one is a Boolean indicating whether the expert accepts to produce the report, and the second one is a link to the report to be produced. The commitment of the expert allows the process to progress (even if the report is not yet produced) and the other services that are dedicated to future treatments, using that report, must also be subscribed to it.

We develop this notion of lazy services, building up from the model of guarded attributed grammars introduced in the context of distributed collaborative systems [7]. We abstract from this model and limit somewhat its expressiveness so that it can comply more broadly to SOA principles, and we develop a distributed implementation of lazy services. In addition, we outline some practical situations where we believe that lazy services can provide a significant improvement.

2 Guarded Attribute Grammars

In this section we present a simplified version of Guarded Attribute Grammars (GAG) adapted to the scope of this work. The reader is not expected to be familiar with the original GAG model since we provide a complete description of its adaptation to the present purpose. Our goal is to define a notion of lazy service compatible with the requirements of a service oriented architecture. We proceed by associating each service with an interface that specifies how this service can be used regardless of its implementation. The interface thus allow for a loose-coupling between the implementation of a service and its use in larger applications. This conforms to the image of a service as a black box and of its interface as instructions for using it, that specify both the assumptions on how the caller should invoke the service and the guarantees that the service offers in return. Then, we introduce the so-called *productions* which are a means to combine several services. One can statically check the correctness of a production. This consist in verifying that each service is invoked according to its interface and that no cyclic dependencies between values (produced by a service and used by others) can be created during execution. The latter property allows us to associate an interface to a production thus providing a new (more complex) service.

2.1 Running example

We will use as running example a simple book ordering process between two bookstores A and B, corresponding to an adaptation of the ordering process presented in [16]. The process starts when a bookstore A places an order to a larger bookstore B according to its needs. Once the order is received, bookstore B checks for the available books and their quantities in its warehouse and makes an offer to A. If Bookstore A is satisfied with the offer, it must validate it and pay the corresponding fees. Upon receipt of payment, the books are delivered to A and the process terminates. The BPMN representation of the process nominal case is showed in Figure 1.

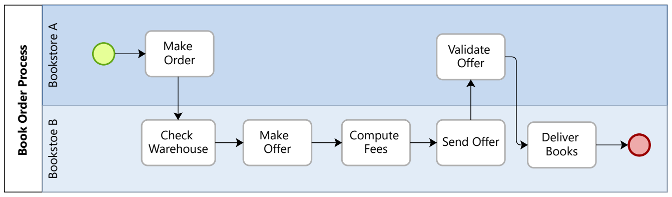


Figure 1: Book order process between two bookstores

2.2 The interface of a lazy service

We can invoke a service by providing it with input parameters. It takes the form of a list of attribute/value pairs which we call the input form. For instance, the input form for admission in a hospital would contain the name of the person, her address, social security number, attending physician, the results of certain medical examinations etc. It may be the case that not all of this information can be determined at the time of service call, if another service is responsible for providing it. Indeed, in order to do so, this other service could in turn require information from the service we are just invoking, in such a way that none of the two services can be fully executed before the other. In this case, the missing information should be provided as soon as it is available. Therefore, it is not necessary for attributes in the form to be associated to actual values. What we mean by 'actual value' can be a basic data: a symbol, a string of characters, a numerical value, a Boolean value; but also any value obtained from these by constituting lists, associative lists, arrays, hash tables etc. Thus, actual values can be complex hierarchical data. The attributes in the input form are called *inherited attributes*, since their values are produced by the environment, and the attribute in the output form are called *synthesized attributes* as their values are produced by the service during its execution. The remaining ingredient needed for defining the interface of a service is a dependency relation between inherited and synthesized attribute:

Definition 2.1 (Interface)**.** *The interface of a service is given by a disjoint sets of (names of) inherited attributes* **Inh** *and of synthesized attributes* **Syn** *and a dependency relation* $D \subseteq \mathbf{Inh} \times \mathbf{Syn}$.

The dependency relation gives only *potential* dependencies because the value of a synthesized attribute may depend on different sets of inherited attributes depending on the way the service is rendered. A (realization of a) service conforms to an interface when the value of a synthesized attribute is determined whenever all the values of the inherited attribute it (potentially) depends on have actual values. In particular, when all inherited values have an actual value, then the values of all synthesized attributes are determined. Thus, conformance to an interface entails that a service *commits* itself to produce the values of all its synthesized attributes, and will do so as soon as the required information is available in the input form. The rationale for introducing the dependency relation in the interface of a service is to stipulate how services can be safely combined without introducing cyclic dependencies between attributes.

2.3 Incremental Computations

A *realization* (or *implementation*) of an interface is a computation of the values of the synthesized attributes in terms of the values of inherited attributes, in accordance with the dependency relation. We call it an *incremental computation.* The purpose of this section is to define these incremental computations.

The basic building block for incremental computations is the notion of production. A production is a rewriting rule that allows for expanding a service call as a composition of calls to sub-services.

Definition 2.2 (Service Call). *A service call is an expression of the form $s(x_1, \ldots, x_n)\langle y_1, \ldots y_m\rangle$ where s is the name of the service, the x_i's are variables associated bijectively with its inherited attributes, and the y_i's are variables associated bijectively with its synthesized attributes. The variables y_j's are pairwise distinct, they are said to be* defined *by the service call. The variables x_i's are said to be* used *by the service call.*

Definition 2.3 (Configuration). *A configuration of the system is given by a set of variables, and an assignment of values to some of these, together with a set of service calls that use these variables.*

As it is usual in rewriting systems, we also consider *formal* service calls whose variables do not appear in the configuration: they are just formal names for placeholders. When a rewriting rule is applied, these formal variables are replaced by actual variables: some already exist in the current configuration where others are created and added to the configuration at the time of rewriting.

Definition 2.4 (Composite service). *A composite service C is a set of (formal) service calls $s(x_1, \ldots, x_n)\langle y_1, \ldots y_m\rangle$. All variables appearing in a given service call are pairwise distinct; although a same variable can occur in several service calls. We say that a variable is* defined *(respectively* used*) in the composite service C if it is defined (resp. used) by some service call in C. We assume that each defined variable is defined by a unique service call, but it may be used by several other service calls. The variables of a composite service can then be classified into three categories:*

- *The Input variables, $\mathbf{In}(C)$, are those that are used but not defined.*
- *The Output variables, $\mathbf{Out}(C)$, are those that are defined but not used.*
- *The Local variables, $\mathbf{Loc}(C)$, are the remaining cases, namely the variables that are both used and defined.*

Definition 2.5 (Acyclicity). *If each service s comes with an interface, and thus a dependency relation $D_s \subseteq \mathbf{Inh}(s) \times \mathbf{Syn}(s)$, then one gets an instance of this dependency relation for each call of service s by replacing each attribute by the* *corresponding variable of the call of s in D_s. The composite service is said to be* well-formed *if D_C, the transitive closure of the union of these dependency relations, is acyclic; i.e., these equations do not induce cyclic dependencies between attributes. Then a well-formed composite service C is an incremental computation with inherited attributes $\mathbf{In}(C)$, and synthesized attributes $\mathbf{Out}(C)$, which conforms to the dependency relation $D_C \cap (\mathbf{In}(C) \times \mathbf{Out}(C))$.*

Any procedure f in the host language, for which we assume a call-by-value evaluation strategy, can be transformed into an incremental computation, denoted $(\mathbf{lift}\ f)$. It behaves as follows: it first waits for the value of each of its arguments to be known, then applies procedure f with its arguments substituted by these values, and finally associates the values returned by f to the corresponding output variables, which refines the current configuration.

For service calls defined by calling procedures of the host language, we shall use notation $(y_1, \ldots y_m) = f(x_1, \ldots, x_n)$ as a shorthand for

$$(\mathbf{lift}\ f)(x_1, \ldots, x_n)\langle y_1, \ldots y_m\rangle.$$

The set of equations $(y_1, \ldots y_m) = f(x_1, \ldots, x_n)$ so obtained constitute the *semantic rules* of the composite service. Thus, a semantic rule can in particular include user-interactions, requests to a local database, and calls to distant services (usual services not defined by the GAG specification and using a traditional call-by values evaluation strategy). Hence, the definition of a lazy service can involve calls to usual services.

If we consider the book order example presented previously, the definition of the composite service handling orders in bookstore B may look like this

$$\begin{array}{lcl} B(\text{order})\langle\text{deliveryInfo}\rangle & = & \{ \\ & & \quad \text{Validate}(\text{offer}, \text{paymentFees})\langle\text{validation}\rangle; \\ & & \quad \text{Deliver}(\text{offer}, \text{validation})\langle deliveryInfo\rangle; \\ & & \quad \text{check} = checkWareHouse(\text{order}); \\ & & \quad \text{offer} = \mathbf{user}(\text{check}); \\ & & \quad \text{paymentFees} = \text{computeFees}(\text{offer}); \\ & & \qquad\qquad \} \end{array}$$

where *Validate* and *Deliver* are two (sub-)services that should be defined elsewhere in the GAG specification, and the three last equations constitute the semantic rules. Here, each semantic equation returns a unique result, but we may imagine that several results are returned if the host language allows it. In this example, the (local) variables *check* and *paymentFees* are computed by ordinary functions of the host language. The offer sent to bookshop A is provided by the user. We may also imagine that *checkWareHouse* or *computeFees* refer to ordinary services rather than local functions. Finally, the synthesized result *deliveryInfo* will be available as soon as the (sub-)service *Deliver* terminates.

Each computation in a composite service runs in parallel (each one in a particular thread) and the scheduling of the computations is only constrained by the dependency relations between attributes. This allows for maximal parallelism since it avoids any arbitrary sequencing of computations. This is why the acyclicity condition is required to ensure that the overall computation terminates.

The defined and used variables of the semantic rules are respectively the variables defined and used by some of its equations. This allows, as in Def. (2.4), for defining the input, output and local variables of the semantic rules. See Fig. (2) to observe the following.

Remark 2.6. *The input variables of the semantic rules are*

1. *the input variables of the composite service together with*
2. *the defined variables of the service calls.*

The output variables of the semantic rules are

1. *the output variables of the composite service together with*
2. *the used variables of the service calls.*

The local variables of the semantic rules are used to store preliminary results that may be used in several places for subsequent computations (and thus avoiding redundant computations). They also contribute to reinforce the incremental character of a composite service.

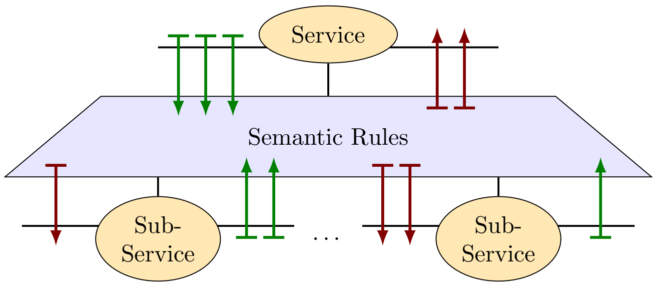


Figure 2: Composition of lazy services

If we abstract from the attributes and the semantics rules to focus on the decomposition of tasks (the service), we end up with a rewriting system. For instance the example above is reduced to the rewriting rule $B \rightarrow \text{Validate} \quad \text{Deliver}$ expressing the fact that the composite task B decomposes into its two sub-tasks *Validate* and *Deliver*. More generally, we expect that a GAG specification gives rise in this way to an abstract context-free grammar whose syntactic symbols are service names. This means that the decomposition is hierarchical (a subtask itself can be a composite task), non-deterministic (several productions may exists with the same symbol in its left-hand side, i.e. different decompositions may exist to solve a given task) and possibly recursive (a task may indirectly invoke itself as a sub-task).

Definition 2.7 (Basic Computation)**.** *A basic (incremental) computation is a composite service that contains only semantic rules, i.e. liftings of procedures of the host language.*

Definition 2.8 (Guarded Attribute Grammar)**.** *A guarded attribute grammar is given by a set S of services, each of which is equipped with a set of inherited attributes and synthesized attributes and a defining equation*

$$s(x_1,\ldots,x_n)\langle y_1,\ldots y_m\rangle \longrightarrow G \rhd \left(\sum_{i=1}^{k} RHS_i\right)$$

whose left-hand side is a (formal) service call and the right-hand side is a set of so-called guarded productions $G \rhd \left(\sum_{i=1}^{k} RHS_i\right)$ *where:*

- *The* guard $G = G(\overline{\mathbf{x}})\langle p\rangle$ *is a basic incremental computation with inherited attributes $\overline{\mathbf{x}} \subseteq \{x_1,\ldots x_n\}$, a subset of the set of inherited attributes of service s, and with one synthesized attribute p whose value in $\{1,\ldots,k\}$ should indicate which production is triggered.*

- *Each RHS_i is a composite service whose inherited attributes is a subset of the inherited attributes of s and its output attributes coincide with the set of synthesized attributes of s.*

If each service is associated with a dependency relation then, the attribute grammar is said to be well-formed *whenever each right-hand side of guarded productions is well-formed and their dependency relation is included in the dependency relation of the service they define.*

Note that from the set of defining equations of a guarded attribute grammar, one can inductively detect whether it is well-formed (free of cyclic dependency) and return the least dependency relations to assign with each service, so that the GAG is well-formed with respect to this assignment. Thus when specifying a GAG one can omit to specify the interfaces of the services and let the system check that the specification is well-formed and infer the less constraining interfaces.

The fact that the synthesized attributes of each production which appears in the definition of a service coincide with the set of its synthesized attributes, guarantees that the service indeed commits to the computation of a value for each of these attributes, regardless of the choice of the production. Nonetheless, a given production needs not rely on the values of all the inherited attributes of the service, which is why only set inclusions between sets of inherited attributes are required.

The evaluation of the guard in the current configuration determines which production is triggered. Triggering a production amounts to replacing the corresponding service call with the composite service in the right-hand side of the selected production. Most often, the guard is decomposed in several threads, in the form $G = C \circ \sum_{i=1}^{k} G_i$, where a guard G_i is associated with each production. It produces a Boolean value p_i from the inherited attributes of service s

indicating if and when the corresponding production is enabled. Then, C can be regarded as a 'choice function', namely another basic incremental computation having $p_1, \cdots, p_k$ as inherited attributes, and p as synthesized attribute, and whose role is to select which production to trigger among those currently enabled. In user-centred systems, the choice will be proposed to the user in charge of solving the given task, so that the system will be guided by the different decisions of the users. In other cases, additional queries may be made (e.g. to local databases) in order to remove the indeterminacy. Mixed solutions can also be used. In any case it is important to note that the guard is a non-deterministic process: even though it must eventually select exactly one production, the resulting choice is not based exclusively on the values of the inherited attributes.

The monotonicity of incremental computations facilitates a distributed version of a GAG specification. The operational semantic for the distributed execution of a GAG specification is detailed in Section 4. Before delving into these technical aspects, we present in the next section some targeted application domains.

3 Domain Specific Applications

This section presents some informal discussion on the possible applications of the presented model. Having introduced the formalism in the previous section, and before engaging in further technical details, we wish to provide the reader with some intuition on how existing solutions may benefit from the features of GAG. To this aim, we consider a few fields where we believe our contribution may provide an advantage, and we underline it with some informal examples.

3.1 Micro- and Web-Services

Service-oriented, or more recently micro-service architectures have grown in popularity among firms, due to the flexibility they provide in managing large projects. On one hand, they allow for an easier integration of outsourced functionalities.

As a matter of fact, the publication of services through Internet has further opened a portal of collaboration between companies. Companies are able to offer their services on the Internet by just specifying their interfaces (often called API). A common example is the secure payment services used by e-commerce companies and offered by specialized payment companies. Such services, provided over the Internet, are commonly referred to as web services. Cloud computing is also turning to towards this model, offering computation capabilities in the form of web services.

On the other hand, this flexibility provides advantages when managing large projects internally. Tasks may be assigned to different working teams in such a way that they leverage each other's work by the means of services. One team may make use of a service whose implementation was assigned to a different team, focusing only on its functionality. Indeed, a service is composed of sub-services, but the way these are put together does not rely on their implementation, only on their interface. The implementation of this composite service then only relies on the input and output parameters of the corresponding sub-services. Such an implementation can furthermore be provided by a service assigned to yet another team, so that this modular approach structures the overall workflow of the project at stake.

In this way, each actor is provided with a service tool-box, and may use it to solve the tasks they were assigned, without caring about the actual implementation of each tool. Only service interfaces are relevant at that stage, and ultimately, the task resolution can be reduced to the appropriate combination of input and output data of services.

Service-oriented architectures are nowadays firmly settled in the industry, underlining the success of this approach. This trend leads to an always finer granularity in the provided services, where more complex services are composed at will by the users. Cloud computing is further fueling this momentum, by providing, computational capabilities in the form of web services. Indeed, rather than remotely running the programs of the users, it is in the advantage of the provider to offer elementary functionalities in the form of web services, that can be used as instructions with which a user may write source code in some scripting language, as composition of services. The race is in this sense for the versatility of the provided services, that increases their flexibility in the hands of the users. This has led to the advent of the so called cloud functions, as Amazon's Lambda Service, Microsoft's Azure Functions, or yet Google and IBM's.

The model we propose offers a qualitative enhancement in the service oriented approach. Indeed, whereas services behave like black boxes, that communicate only upon call and termination, **lazy services are permeable to data flow**. While the user composing services may still use the latter relying solely on their interface, **these will produce each output value as it is available, and while the underlying process is still in execution.** Symmetrically, they can be invoked even if all the required data is not available yet.

It is to be noted, that any service may be lifted to an incremental computation. In this sense, our model comes as a complement to existing web-services and APIs. An atomic lifted service does not present the features of our model, but the composite services built from it do.

Example 3.1. *Consider a filming crew. The director is responsible for providing a service "shooting" whose output data is the footage required for editing. Another service "post-production" takes this footage as input and produces the final theater-ready film. It is apparent that some editing may be done before the shooting is finished, as the story-board indicates which footage is required in which scene. The services "shooting" and "post-production" may be decomposed into sub-services according to each scene. By lifting each such sub-service, both composite services become lazy, and the edition of each scene may start as soon as the corresponding footage is available. In this way, the shooting crew and the* *post-production technicians may work concurrently instead of following a sequential workflow. Since each scene is composed of possibly several sequence shots, we may further refine the "shooting" and "post-production" services according to these. By lifting sequence shots instead of whole scenes, smaller chunks of footage are sent for edition as soon as they are available, thus increasing the flexibility of the whole workflow, and further exploiting the concurrency between the two teams.*

Lazy services are hence consonant with the trend to provide finer and finer grained services. Indeed, a finer decomposition of a service into sub-services, allows for lifting services which provide more elementary functionalities, and thus for taking better advantage of the lazy approach.

Web services have become common practice, as shown by the development of standards to describe service interfaces [31]. Nevertheless, the tools available today for their composition (BPEL[20], Petri net [13], etc.) assume that the services are rendered in a limited execution time (usually a few seconds). Although this is efficient for automatic services (provided by computers), it prevents the publication of services that integrate manual user tasks and therefore require longer execution time. Lazy services appear beneficial in this case, as they allow to combine automatic services with manual tasks while deriving the resulting interface. They are therefore good candidates for the integration of more complete workflows over the internet, involving human and software actors.

3.2 IoT Workflows

The growing availability and diversity of portable devices, both for the general public and the industry, have risen an interest in the research community for their integration in larger systems. Indeed, these devices are equipped with a variety of sensors (cameras, GPS, thermometers, ...) that provide valuable information on the state of the real world. Symmetrically, some devices may interfere with the world, either by communicating with the users, or by means of robotic solutions. The field dealing with the integration of these devices into larger systems, so as to take better advantage of their capabilities has been tagged Internet of Things, IoT for short.

We represent devices as components, that may have a very limited computing capacity, but can provide data, and execute tasks in the form of services. The ordeal of exploiting these data and commanding or scheduling actuator behaviour to the best of their capacity then reduces to an efficient coordination of services. For instance, the use of sensors for quality control in assembly lines has become a widespread practice in the industry. Solutions involving visual inspection have been greatly enabled by the use of neural networks for image analysis. These tasks are extremely costly in terms of calculations, and the capture devices lack the computation capacity to perform them. Solutions to such problems thus rely on the coordination of sensors with the devices able to process the data, and further with the intended consequences of such an analysis: detected defects need be communicated so that they may be taken care of.

The coordination of the prescribed interactions has been tackled thanks to workflows, in their different implementations ([12]), but problems arise, particular to this setting. The unreliability of these devices may occur in a variety of forms. First, sensors have a quality range which greatly depends on their price, incurring into the quality of the provided data. Indeed, specific problems may require the acquisition of a large number of poor quality devices rather than a few costly ones. Cheap sensors may be extremely noisy, and the data they provide proportionally unreliable. On the other hand, such device networks may be spread over large geographical areas, which introduces a dependency on the quality of the connection linking them together. In this case, a particular device may often fail to communicate its output, so a coordination featuring strong data dependency would risk to stall the whole system due to the failure of a single irrelevant component. This points at a necessity for loose device integration: systems in which at most a few devices are essentially required, and where noisy data may be discarded with a limited effect on the system behaviour.

Workflows constructed with lazy services have a high tolerance to faulty, or unavailable data. To provide an example, we focus on the case of agriculture, since it can greatly benefit from sensor networks in using resources more efficiently, and limiting its environmental impact.

Example 3.2. *Consider a large cropping field in a fairly arid region. Water is in this case an extremely valuable resource, and irrigating vast areas could be even too costly to be an option. A grid of sensors measuring rainfall or soil moisture may be implanted over the area, providing information on the locations where water is most needed. This data may be communicated to the workers responsible for irrigation, or to automatic actuators, that may activate a selected set of water pumps. A system coordinating these devices could be structured hierarchically as follows. Sensors may be clustered according to their geographical location, and each cluster subordinated to a controller, responsible for the closest water pump. These controllers are then in turn subordinated to the coordinator of the general water supply, that may guide the resource to the wells in the areas in most need. The supply coordinator may invoke a service "estimate water requirement" on each of the local controllers, either on a regular basis, or upon availability of the resource. Each controller may then invoke a service "measure moisture" on each sensor. This latter service shall provide the actual measured moisture together with the particular position of the sensor. Note that the former service is a composition of the many instances of the latter, combined with some calculations. With a lazy implementation, the failure of a sensor to perform the actual measure can be mitigated by the fact that it has provided its position, and the controller may compensate this lack of information statistically. Furthermore, statistical analysis at the controller level can be used to enhance estimation of the errors in the data provided by each sensor, thus mitigating the eventual noise they present. We could consider that the controllers do not have enough computation capacities to perform these tasks, in which case these can* *be assigned to the general coordinator, thus limiting the controller functionality to sensor data aggregation.*

Such solutions can certainly be done without recurring to laziness or even a service approach, but lazy services permit to structure the system with a far more straightforward design. Furthermore, lazy services provide additional flexibility. For instance, when the moisture measured by a sensor is below some critical threshold, the controller may directly establish a high need for water in the area, and return the estimation before all the sensors have provided the corresponding data. With a lazy service implementation, this can be done without handling additional exceptions, as these events can be integrated within a particular service. The pump activation is not decided locally, since the supply coordinator must validate the resource attribution. Thus, local controllers transfer their estimates to the supply coordinator as service outcomes, so that the latter may efficiently distribute the water geographically. At this level again, data provided by the different controllers may be aggregated through a lazy composite service, so that decisions may be taken without waiting for all controllers to have transferred their estimates. In this way a straightforward service architecture may be deployed so that given decomposition patterns present no risk of stalling the system when some of its components fail, and with a more agile ability to react to critical situations.

The presented approach transfers well to, for example, plague control, permitting a wiser use of chemicals, costly both economically and in their ecological impact.

The robustness, and tolerance to failure of workflows built with lazy services enables their use in the design of safety critical systems, where shorter reaction times may prevent catastrophic events. It is the data permeability of lazy services that makes them suitable for coordinating systems of devices interacting in the real world, such as (automated) road traffic, air space management, or human teams working in crisis situations (military, sanitary, ...).

3.3 Distributed Computation

The processes underlying lazy services may run concurrently where their standard counterparts would stall while waiting for some unnecessary data. This issue can, in general, be overcome by providing a different architecture that would allow for a more efficient distribution of the calculation loads. While these efficiency considerations still impose that atomic services be fine enough to pertain to a single location, composing them as lazy services provides more flexibility in the design of solutions than the conventional approach.

In general, composite services may be distributed according to their decomposition patterns, in a way restricted by the data dependencies. Lazy services partially lift this restriction, as inter-dependent services may still run concurrently. Indeed, **the incremental computations allow for data dependencies to breach through service interfaces, stalling the underlying**

processes only when strictly required.

Example 3.3. *Consider a paradigmatic computation-intensive task, as is often found in image processing. Suppose that a particular object needs to be found and tracked along a sequence of frames. For the sake of simplicity, we may imagine that a service meant to find the object statically is instantiated on each frame, and that it is composed of sub-services that locally search for the object on different regions of each frame. The object we are tracking is supposed to have a roughly continuous motion, and when found on a frame, it is likely to be found in nearby areas of the next and previous frames. This information can be used to avoid unnecessary calculations, by killing too distant local searches, or giving a higher priority to the nearby ones. This however, creates a data dependency between the different frame instances of the global service. Indeed, in this setting, each frame instance of the service would need for the object to be found on the previous frame before being invoked. As a matter of fact, this view would discard this solution to the problem, in favor of one that would better exploit concurrency. However, with a lazy service approach, all frame-wise instance of the service may be invoked concurrently, and their execution distributed. Upon invocation of each instance, the input parameter corresponding to the position of the object in the neighbouring frames is left as intentional. When one instance succeeds in finding the object, it may communicate the outcome position to the instances of the neighbouring frames, that may use this information to optimise their own search.*

Thus, lazy services may offer solutions that would be unpractical in a standard framework. Indeed, in this example, solutions could be optimised in other ways, but our approach permits a natural service based way to tackle the problem while taking effective advantage of concurrency.

It is noteworthy that lately, researchers are exploiting fine grain cloud computing to solve such computation-intensive tasks by launching parallel swarms of cloud functions (see [10] for a short review). Our model represents a possible solution for efficiently structuring such projects while leaving the data dependency restrictions at the level of the cloud functions.

3.4 Human Centric Systems

Guarded attribute grammars were originally introduced for distributed collaborative systems, and user-centred systems remain the primary intended area of application of our model. These systems range from rigid workflow (task-flow systems) to more dynamic and flexible systems, such as corporate social networks. The former are adapted to well-defined processes, that however generally depend on specific contexts and are not expected to evolve over time. The latter on the contrary, may rely less strictly on a formal process so as to provide the users with a higher degree of freedom.

As an example of a rigid workflow, we may consider crowd-sourcing. There, some repetitive tasks requiring human intelligence are entrusted to external actors. **Lazy services can be used to coordinate the participation of**

external actors in such situations. Qualified external actors (the crowd) could receive requests of the service that the company wants to outsource, together with a defined set of productions meant to solve the corresponding task. The crowd must implement the services by selecting the appropriate productions among those made available to them, and these interact directly with the company's services (database, automated programs, other lazy services, ...). The outsourcing of such services allows for optimising the costs, while the availability of productions regulate the way the crowd participates in the process, their activation being guarded by contextual information.

Example 3.4. *In an insurance company, for example, the validation of insurance or reimbursement claims can be outsourced following this principle. An independent insurer could be given claim validation tasks from nearby customers. The decision would be formalised by a particular choice of productions for task resolutions. The guards of these productions would prevent discriminatory or unfounded decisions, while the insurer could still take her final decision based on her experience and possible interactions with the customer. Indeed, by accommodating user interactions through the selection of productions, screening of user activity is made possible, providing the system with a better control over human interventions.*

Regarding more flexible and less formal systems, we may consider corporate social networks. These are more likely to harness the creativity of participants enabling the emergence of collective intelligence towards the resolution of a task, but they are too often not sufficiently structured. This lack of structure represents a major obstacle for the cooperative production of a solution to the problem at stake. In order to fully exploit the potential of such systems, rules must be added to regulate the exchanges between the participants, in a way that will not coerce their creativity. **Guarded attribute grammars** are well suited for analysing how tasks may be decomposed into more elementary sub-tasks, and as such, they can provide a normative framework for collective problem solving. The semantic rules are written in the host language in which the system is designed, so that the normative model can interact with the utilities offered by the corporate social network (share documents, agendas, discussion threads). Note that the possibility to select appropriate production rules, or even create new ones, **provides enough flexibility for the users to freely exploit their creativity**. Furthermore, the expertise of each user is reflected in her choices, or the rules she creates, and this knowledge may be recorded for further use. The social network allows for sharing most useful information (e.g. documents or spreadsheets) but lacks control over its role in the process of task resolution. This knowledge is left to the human participants. Our data-aware normative approach allows for structuring the informal processes (or workflows) performed by the users **as they collaborate for solving their respective tasks**. This enhances the system in that it records the interaction patterns of the various relevant expertise. In this way, the system may collect, not only the individual knowledge of each user, **but also gather a trace of the emergent collective intelligence**. This structure, together with the information provided by the synthesized attributes may also be used to offer a shared mental representation to the users. Such a representation can be particularly effective in preventing detrimental human behaviour, such as dilution of responsibility (by knowing who has committed do doing what) or cognitive bias (like the illusion of unanimity, the incomplete evaluation of alternatives, the pressure of divergent opinions, self-censorship, feeling of invulnerability). Altogether this solution improves the quality of the collaboration.

For a thorough example of human-centring application using the formalism of Guarded Attribute Grammars, we refer to [19], which presents an application to epidemiological surveillance.

4 Operational semantics of lazy services

The operational semantics of lazy services are defined by a distributed system, consisting of elementary computational units called components. More precisely, each component of the system proposes to solve certain tasks, the services it offers, for which it may require to call external services offered by other components. To this end, it is associated with a local GAG, as defined in Section (2, with the difference that external services do not have a defining equation. As in the case of the variables of a composite service, the services of a local grammar can be classified into three categories:

- The provided services: those defined but not used.
- The required services: those used but not defined.
- The local tasks: those both defined and used.

The local tasks are not visible from outside the component and thus their names are lexically bound to the scope of the component. Several components can offer the same service and use different defining equations for it, i.e. different components may render the same service in different ways.

4.1 Configuration

Each component maintains its own configuration as in Def. (2.3), a data structure storing a list of pending tasks (the current service calls), a partial valuation for the involved variables, but also, for this distributed version, a set of subscriptions to the variables defined locally but used in another component. It is modeled by the four following sets:

1. A set of variables $\mathbf{Var}(\Gamma)$.
2. A set $\mathbf{Eq}(\Gamma)$ of equations of two types:
   (i) A service call $s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle$ where y_i, x_j are variables and s is either a provided service or a local task.

This form models the services being executed within the component. The presence of such an equation means that the current component must produce the data $y_1, ..., y_m$ from the inputs $x_1, ..., x_n$ through service s.

(ii) A semantic rule $(y_1, ..., y_m) = f(x_1, ..., x_n)$ where y_i, x_j are variables and f a procedure of the host language.

This form means that the current component must execute procedure f to compute the data $y_1, ..., y_m$ from the data $x_1, ..., x_n$. Unlike for services, the procedures used in semantic rules are not lazy. They correspond to procedures of the host programming language used by the component to implement its services.

We deduce from $\mathbf{Eq}(\Gamma)$ the inputs and outputs of the component. We shall say that a variable is used (resp. defined) if it used (resp. defined) by one of its instances of services or semantic equations. The inputs, noted $\mathbf{In}(\Gamma)$, are the variables used but not defined: $\mathbf{In}(\Gamma) = Used(\Gamma) \setminus Def(\Gamma)$; whereas the outputs are those which are defined but not used: $\mathbf{Out}(\Gamma) = Def(\Gamma) \setminus Used(\Gamma)$. The defined and used variables correspond to local variables used internally.

3. A valuation $\sigma(\Gamma)$ which is a partial substitution of values to variables. Each equation of $\sigma(\Gamma)$ takes the form $x = v$ meaning that the computation of the variable x is completed and has resulted in the value v. The valuation is updated whenever a semantic rule finishes its computation or when the present component receives a notification providing the value from another component.

4. The last set $\mathbf{Sub}(\Gamma)$ is the set of subscriptions on the variables that the component must compute and publish. Each element of $\mathbf{Sub}(\Gamma)$ is of the form (x, c') such that $x \in \mathbf{Out}(\Gamma)$ and $c' \neq c$ (c being the present component).

We associate some additional constraints ensuring the validity of the execution:

$(\mathbf{C_1})$**: Non-double definition**. A variable cannot be defined more than once in Γ.

$(\mathbf{C_2})$**: Acyclic dependency**. The set $D(\Gamma)$ of dependencies between variables of Γ is acyclic. This set is given as the transitive closure of the union of the dependency relation associated with each equation: $D(\Gamma) = (\bigcup_{e\in\Gamma} D(e))^*$, where

- $D(e) = \{x_1, ..., x_n\} \times \{y_1, ..., y_m\}$ if e is a semantic rule $(y_1, ..., y_m) = f(x_1, ..., x_n)$,
- $D(e) = D_s[x_i/in_i, y_j/out_j]$ if e is a service call $s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle$ and the dependency relation of service s is $D_s \subseteq \{in_1, ..., in_n\} \times \{out_1, ..., out_m\}$.

$(\mathbf{C_3})$**: Input/output consistency.** Each input variable x of a component c must appear as an output variable of a single other component c' in the system. Consequently, c' must have a subscription of c on the variable x.

4.2 State of a Component

The initial computational state of a component is obtained from its initial configuration by replacing each instance of a provided service by its definition. The rationale is that all the guards are active in the current state and are evaluated concurrently. When a production is triggered, we should activate (expand) the corresponding right-hand side of the production. The state of a component should then expose a structured set of equations that allows such a dynamic evolution.

Definition 4.1. *The state of a component consists of a set of variables, a valuation (partial substitution of values to these variables), a set of subscriptions and a set of* instances of service. *An instance of a service is a pair, denoted as $G \triangleright \left(\sum_{i=1}^{k} RHS_i\right)$, made of a instanciated guard G (a set of semantic rules) and a list RHS_i of equations (service calls and semantic rules). The analogs of conditions (C_1), (C_2), and (C_3) are also required.*

The expansion of the set $\mathbf{Eq}(\Gamma)$ of equations in a configuration Γ is the state $\overline{\Gamma}$ obtained by keeping all semantic rules unchanged and expanding each of the instance of service in $\mathbf{Eq}(\Gamma)$ as defined below. The variables, valuation, and subscriptions of $\overline{\Gamma}$ are those of Γ but they are updated during the expansion of its equations as it is described below.

Recall that the definition of a service in a guarded attribute grammar is given by a rewriting rule:

$$s(in_1, ..., in_n)\langle out_1, ..., out_m\rangle \longrightarrow G \triangleright \left(\sum_{i=1}^{k} RHS_i\right)$$

where:

- Guard G is given by a set of semantic rules that meets the constraints C_1 and C_2. Their inputs form a subset of the inherited attributes of service s ($\mathbf{In}(G) \subseteq \{in_1, ..., in_n\}$) and $\mathbf{Out}(G) = \{p\}$.
- Each RHS_i is given by a set of equations (service calls and semantic rules) which verify C_1 and C_2. This set represents a potential implementation of service s. Thereby, $\mathbf{In}(RHS_i) \subseteq \{in_1, ..., in_m\}$ and $\mathbf{Out}(RHS_i) = \{out_1, ..., out_n\}$.

All the variables that appear in such a rule are formal variables that play the role of placeholders. Expanding a service call $s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle$ in a configuration Γ is done as follows:

- If s is a defined service (provided service or local task) with a definition as above, then replace $s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle$ with the corresponding right-hand side where the formal parameters $in_1, ..., in_n$ and $out_1, ..., out_m$ are replaced by the actual variables $x_1, ..., x_n$ and $y_1, ..., y_m$. To avoid name clashes, the other variables of the expression are renamed with new names.

These variables are added to the set of variables of Γ (each with an undefined value). The expansion of the service call $s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle$ is then the instance of service $G' \rhd \left(\sum_{i=1}^{k} RHS'_i\right)$ with $G' = G\left[x_i/in_i; y_j/out_j\right]$ and $RHS'_i = RHS_i\left[x_i/in_i; y_j/out_j\right]$.

- If s is a required service then (1) a service call request is sent to a component c' providing s; (2) subscriptions (x_i, c') are created for the x_i not defined in $\sigma(\Gamma)$; and (3) the values of the defined x_i are embedded in the request. A service call query thus takes the form $\langle s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle, \sigma_s\rangle$ where $\sigma_s = \{x_j = v_j | x_j = v_j \in \sigma(\Gamma) \wedge \; x_j \in \{x_1, ..., x_n\}\;\}$.

4.3 System dynamics

The dynamics of the system relies on the notion of component state introduced above and the operations that make it evolve (internal computation, service call, notification, etc). We later show that the semantics so defined lends itself equally well to a centralized architecture (central orchestration unit) and a decentralized architecture (no central unit).

Note that, expanding the initial configuration of a component into its initial state can generate events (calls to services provided by other components) and thus already contributes to the system dynamics.

The execution dynamics is given by a set of operations that govern the evolution of the set of component states and preserve the conditions of their validation (C_1 to C_3).

Internal computation and notification

The atomic level of computation is an internal computation. For any semantic equation $(y_1, ..., y_m) = f(x_1, ..., x_n)$ the procedure f is called when, and as soon as, each variable x_i has a value v_i in the current state: $(x_i = v_i) \in \sigma(\Gamma)$. Then for each y_j, an equation $y_j = v'_j$ is added in $\sigma(\Gamma)$ where v'_j is the value returned by f for variable y_j, i.e. $(v'_1, \ldots, v'_m) = f(v_1, \ldots, v_n)$. This can potentially result in notifications if there are subscriptions in $\mathbf{Sub}(\Gamma)$ for some of the y_j's. In such a case, equation $y_j = v'_j$ is sent to the corresponding subscribers.

Triggering a production

If $G \rhd \left(\sum_{i=1}^{k} RHS_i\right)$ is an instance of service in the current state and $v(p) = i$, then the corresponding guarded production is triggered. Triggering this production amounts to replacing this instance of service with the expansion of RHS_i. Recall that this expansion generates a service call for each requested service in RHS_i.

Message processing

A component can receive two types of messages corresponding to notifications and service calls which are generated respectively after internal computations and activation of productions.

In the case of reception of a service call $\langle s(x_1, ..., x_n)\langle y_1, ..., y_m\rangle, \sigma_s\rangle$ coming from a c', the component receiving the call executes the expansion of this service call in its current state; updates the set of variable values: $\sigma(\Gamma) = \sigma(\Gamma) \cup \sigma_s$; and creates subscriptions (y_j, c') (in $\mathbf{Sub}(\Gamma)$) to notify the caller c' whenever an output value becomes available.

When a component c receives a notification $\langle x = v\rangle$, it updates its configuration with the received value : $\sigma(\Gamma) = \sigma(\Gamma) \cup \{x = v\}$. In some cases, it may happen that the component c being notified had already stored subscriptions to the notified value. This corresponds to the situations where the component has used an output variable of a remote service as input of another remote service. In such a case, it becomes a transiting node and must notify all the component requiring the variable value as soon as it gets notified.

5 Implementation and Deployment

In this section we provide some guidelines for implementing and deploying lazy services in a distributed environment with respect to the previously defined operational semantics. We propose, as a proof of concept, the development and deployment of the running example in a distributed environment. The guidelines given in this section are general, and can be refined or adjusted according to the targeted application.

5.1 Deployment architecture

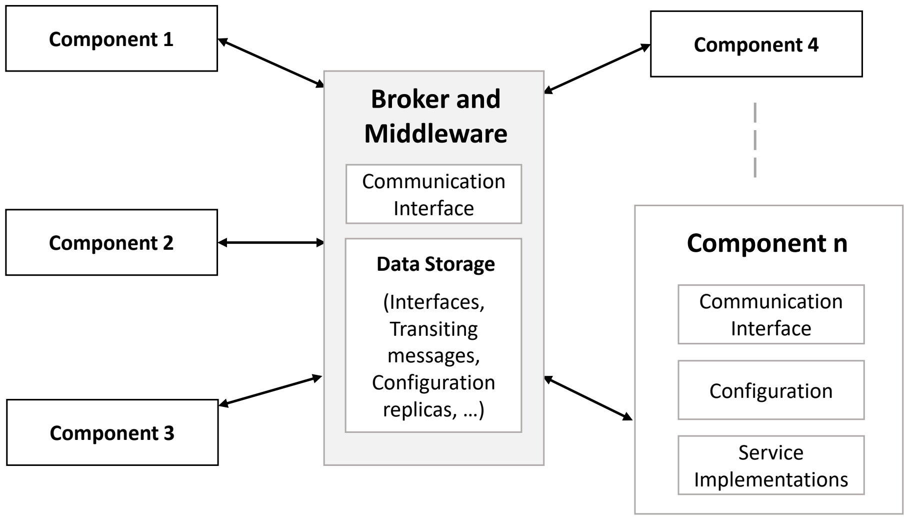


Figure 3: Centralized deployment architecture for lazy service

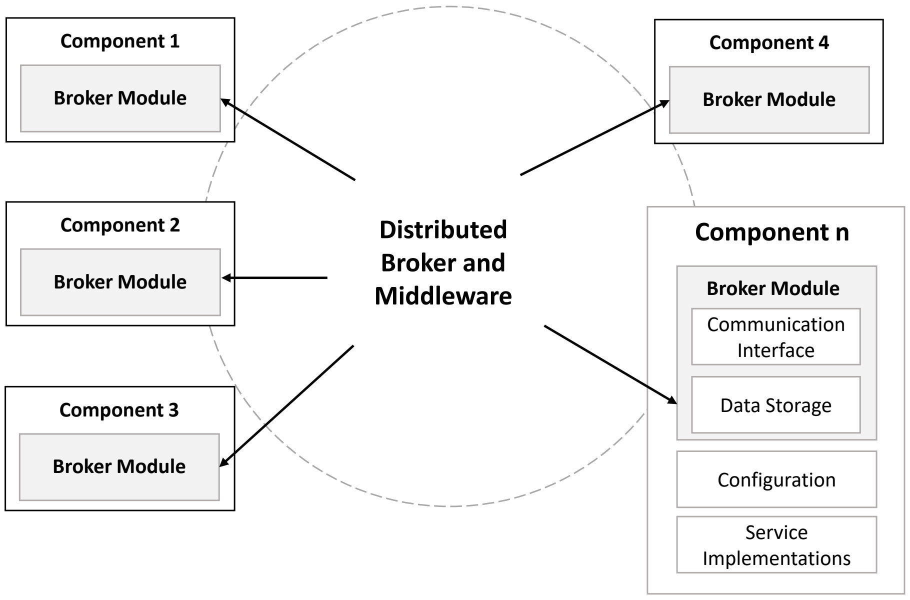


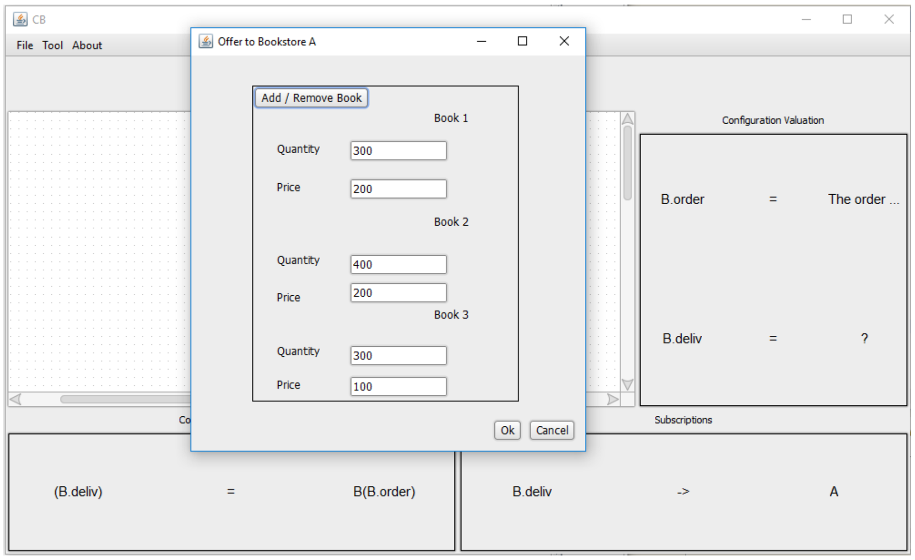
Figure 4: Distributed deployment architecture for lazy service

The easy way to deploy lazy services is to use a central broker where each component publishes the services it implements and searches for those it needs. The broker can also hold middleware functions handling the connections and disconnections of components, to ensure the routing of messages. In the case of crowd-soucing systems where the components maintained by users can have failures, the broker can also store the components' sessions (i.e configurations), to ensure the robustness of the system. The resulting architecture is then a centralized architecture where interaction is provided by the broker in the middle (see Figure 3). In a more elaborate way, we can optimize the deployment by distributing the broker within the components, similar to what is done in distributed publish/subscribe systems ([28, 17, 9, 27]). Each component will then hold a part of the broker and share the responsibility of the reliability of the system (routing of messages, secure replication of data, etc.). This second architecture (see Figure 4) also matches peer-to-peer systems, where each participant plays an equal role towards other participants, and which are free of central server bottlenecks.

Thus, the application domain can also guide the used architecture, and the shared data. For an enterprise application, a centralized architecture seems appropriate, while for a collaboration between different companies, a distributed architecture where each component ensures the routing of its messages and the reliability of its data, is more appropriate. Mixed architectures are also to be considered. For example, an enterprise application based on lazy services and communicating with other applications of the same type could have a centralized internal broker securing its data and interfacing with other brokers.

5.2 An application prototype based on lazy services

We have developed a demonstrator based on an instantiation of the architecture in Figure 4. It is the implementation of the book order example introduced in section 2.1. Its interface has four principal panels (see Figure 6). The first one (top left) shows the graphic visualization of the configuration, and the three others (bottom left, top right and bottom left) show respectively the different parts (**Eq**(Γ), $\sigma(\Gamma)$ and **Sub**(Γ)) of the configurations described in section 4. Our example has two components C_A and C_B. The first component C_A implements the two services offered by the bookstore A: a service to initiate the transaction and a service to validate or reject the offers made by B. The component C_B implements the composite service used by B to handle A's orders. This service uses the validation service offered by A to confirm $B's$ offers, and an internal service to deliver the books upon confirmation. In case of offer rejection, the same internal service returns a null delivery detail value. Figures



Book 1
Quantity 300
Price 200
Book 2
Quantity 400
Price 200
Book 3
Quantity 300
Price 100

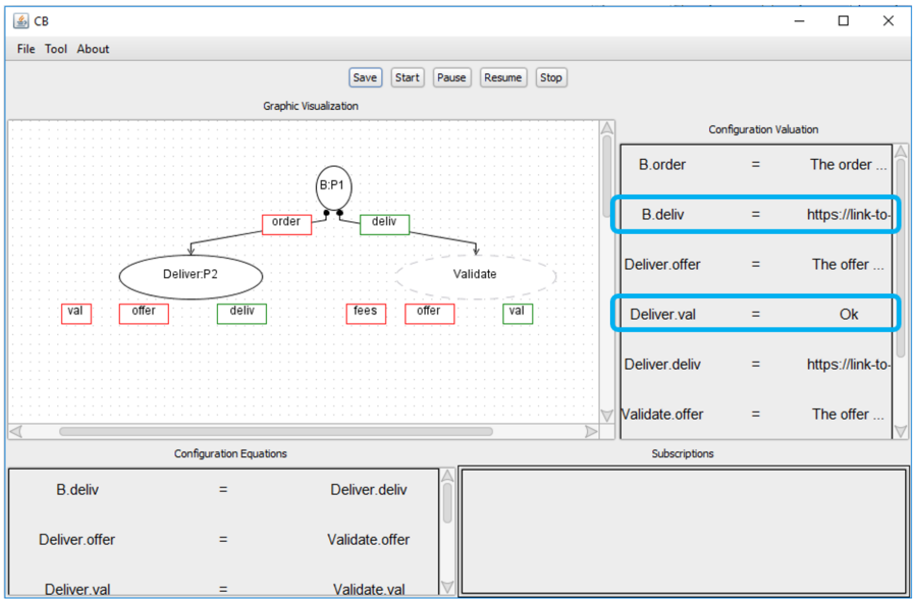
B.order = The order ...
B.deliv = ?

(B.deliv) = B(B.order)

B.deliv -> A

Figure 5: User interface allowing B to make its offers

5 and 6 display screenshots of the application. The first figure presents the user interface that allows B to make its offers. This interface is activated by the local function **user** of the bookstore B. We can also observe in the figure (in the *subscription* panel), the subscription of A to the delivery detail that B has to produce. The second figure corresponds to the final interface of the bookstore B after delivery. We can observe in the *configuration valuation* panel that the offer has been validated and that the delivery detail is now available.



B.order	=	The order ...
B.deliv	=	https://link-to-
Deliver.offer	=	The offer ...
Deliver.val	=	Ok
Deliver.deliv	=	https://link-to-
Validate.offer	=	The offer ...

B.deliv	=	Deliver.deliv
Deliver.offer	=	Validate.offer
Deliver.val	=	Validate.val

Figure 6: Final execution interface of the bookstore B after the delivery

The application is implemented using Java as host language, and a DSL based on XML and the jaxb library[1] to specify the GAG of each component. The communication between components is ensured by means of the distributed SON middleware[2]. The source code of the prototype is available at [1].

6 Related works and discussions

In this section, we discuss the relation of the present work with other approaches, in line with the lazy service concept.

6.1 Lazy data computing: data with embedded calls

A popular lazy data computing technique consists in embedding functions into data. The data is then composed of two types of information; some explicit and available called *extensional information* and others available only when needed, upon embedded function call, called *intensional information*. This computational principle has been used by Microsoft Office, to provide information to

[1]The Java Architecture for XML Binding (JAXB) provides an API and tools that automate the mapping between XML documents and Java objects[22]

[2]SON is a generic lightweight P2P middleware that assists application developers by providing an automatic code generation which handles several requirements (e.g., communication mechanisms, message queue management, broadcasting messages, etc.)[14].

users on demand, by means of “smart tags” [2, 18]; and by datalogs [5, 11] to deduce new (intensional) information from pre-existing one.

The active XML documents (AXML) approach [3, 4, 24] then proposed to replace embedded functions by embedded web services to integrate the use of intensional information in distributed computing over the internet. Our approach differs from this one, in that data which are not available are here considered as *commitments*, that must be delivered by the system as soon as they are produced. The main advantage of this is that, it is henceforth possible to exchange intensional data on the basis of the commitments of their producers: lazy and/or parallel processing are therefore enhanced. In fact, one can now perform an operation and/or invoke a service before all the necessary data are available if a significant part already is. Thus, we are not only interested in lazy evaluation of data, but also in lazy and parallel evaluation of services so as to optimize the overall computing speed.

6.2 Service composition

The composition of services is a technique regularly used in the distributed execution of business processes. It is generally handled via one of two approaches: orchestration or choreography. Service orchestration relies on a central service dedicated to orchestration and coordination; the orchestration model being given by a dedicated language such as BPMN [21], BPEL [20] or Petri nets [13, 8]. The orchestration technique generally suffers from scaling drawbacks due to the centralization of coordination. Service choreography tends to solve the problem by defining specification languages (WSCI [29], WS-CDL [30]) that consider services as autonomous entities collaborating without intervention of a central server. They can also be used to ensure the collaboration of several orchestration models [30].

Since no specific orchestration or collaboration language is defined in the lazy services approach, it can be used to efficiently execute distributed business processes, improving lazy and parallel processing, from the orchestration point of view, as well as from the choreography one.

6.3 Peer-to-peer computing

Peer-to-peer computing is progressively gaining popularity for the development of internet applications [14, 25]. In a peer-to-peer application, each participant plays an equal role towards the other participants. Therefore, no server is required to centralize the processing and thus scaling constraints are leveraged.

Lazy services are well suited for peer-to-peer computing, as they operate independently and do not require a central server for their execution. Moreover, since communication between lazy services is asynchronous and data are delivered as soon as possible, they can be used to better address connectivity problems generally encountered in peer-to-peer network.

6.4 Declarative models

Declarative models are a promising means to exploit concurrency and avoid over-specifications when designing systems. Nowadays, declarative computing models [23, 26] allow the user to specify only the mandatory constraints to be met when executing a process and let the system decide on the other contextual requirements to be fulfilled during the execution. These contextual requirements may depend on the execution platform or on the availability of resources. A representative case study was provided by the Condec approach [23] that proposes to execute business processes in the form of task constraints without specifying the execution flow, which is determined at runtime by the user. Lazy services enrich these approaches by providing a service-based model for building declarative systems where the only mandatory constraint to be specified before execution is the data dependency. The remaining constraints are determined by the execution dynamics.

7 Conclusion

In this paper, we have developed a notion of lazy service starting from the model of guarded attribute grammars previously introduced in the context of collaborative systems. Lazy services provide flexibility to SOA systems, allowing services to start with only some of their inputs and to return outputs at the earliest possible time, as they are produced by sub-services and local functions. Lazy service behaviour relies on productions of guarded attribute grammars. Each production of the grammar defines how a composite service relies on sub-services and local functions of a host programming language to produce its outputs. It is worth emphasising that a GAG specification is always relative to a host language, but any arbitrary language can host a GAG specification. GAG formalism should be seen as a means of extending the host language. The approach followed is therefore clearly language-oriented: our intent is not to develop a fixed application but to offer a means to extend the host language. We have shown how so defined lazy service may bring more flexibility in user-centered systems or show more resilience in systems subject to failures or connectivity problems. Time execution is also improved, as scheduling constraints are reduced down to data dependency.

We introduced a lightweight syntax (section 2.3) to specify lazy services and used it to implement a prototype through xml. This lightweight syntax can be refined to target concrete applications. One may want to offer higher-level notations more fitted for describing a given problem, and hence provide extensions of the core syntax. For instance, if the host language allows for macros, new syntactic patterns can be defined. Any expression using these macros can then be expanded into a specification of the core model of GAG.

Extending the core model thus constitutes the primary intended follow-up of this work. More specifically, it will consist in providing frameworks, libraries and Domain specific languages (DSLs), built on top of the core model, for each of the domain specific applications discussed above. DSLs built on top of the base model will then provide an additional level of abstraction, easing the user into the specification of lazy services, while exploiting the features of the base model.

References

[1] Lazy service with son middleware. `https://github.com/Service-BP-Dev-Team/lazy-service-with-son-middleware`.

[2] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: peer-to-peer data and web services integration. In *Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong*, pages 1087–1090. Morgan Kaufmann, 2002.

[3] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active xml. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–45, New York, NY, USA, 2004. ACM.

[4] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Active XML. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] Erika Asnina and Gundars Alksnis. Survey on information monitoring and control in cross-enterprise collaborative business processes. In Kurt Sandkuhl and Ulf Seigerroth, editors, *Proceedings of the 7th International Workshop on Information Logistics and Knowledge Supply*, volume 1246 of *CEUR Workshop Proceedings*, pages 1–12. CEUR-WS.org, 2014.

[7] Eric Badouel, Loïc Hélouët, Georges-Edouard Kouamou, Christophe Morvan, and Nsaibirni Robert Fondze, Jr. Active workspaces: Distributed collaborative systems based on guarded attribute grammars. *SIGAPP Appl. Comput. Rev.*, 15(3):6–34, October 2015.

[8] Sofiane Chemaa, Mouna Bouarioua, and Allaoua Chaoui. A high-level Petri net based model for web services composition and verification. *IJCAT*, 51(4):306–323, 2015.

[9] N. Doraswamy, M. Abrams, and A. Mathur. Chitra: Visual analysis of parallel and distributed programs in the time, event, and frequency domains. *IEEE Transactions on Parallel & Distributed Systems*, 18(06):672–685, nov 1992.

[10] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference, Renton, WA, USA*, pages 475–488, 2019.

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[12] S. Haller and C. Magerkurth. The real-time enterprise: Iot-enabled business processes. In IAB IETF, editor, *Workshop on Interconnecting Smart Objects with the Internet*, 2011.

[13] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for web service composition. In Klaus-Dieter Schewe and Xiaofang Zhou, editors, *Database Technologies 2003, Adelaide, South Australia*, volume 17 of *CRPIT*, pages 191–200. Australian Computer Society, 2003.

[14] Ayoub Ait Lahcen and Didier Parigot. A lightweight middleware for developing P2P applications with component and service-based principles. In *15th IEEE International Conference on Computational Science and Engineering, CSE 2012, Paphos, Cyprus, December 5-7, 2012*, pages 9–16, 2012.

[15] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.*, 48(3):33:1–33:41, 2016.

[16] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.*, 48(3):33:1–33:41, 2016.

[17] G. Li, P. Zhao, and S. Gao. Marshmallow: A content-based publish-subscribe system over structured p2p networks. In *2013 Ninth International Conference on Computational Intelligence and Security*, pages 290–294, Los Alamitos, CA, USA, dec 2011. IEEE Computer Society.

[18] Microsoft. Smart tags overview. `https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/ms178786(v=vs.100)`, 2013.

[19] Robert F. Jr Nsaibirni, Eric Badouel, Gaëtan Texier, and Georges-Edouard Kouamou. Active-Workspaces: A Dynamic Collaborative Business Process Model for Disease Surveillance Systems. In *Worldcomp'16, Las Vegas, United States*, July 2016.

[20] OASIS. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`, April 2007.

[21] OMG. BPMN specification: Business process model and notation.

[22] Oracle. Java architecture for xml binding. `https://javaee.github.io/jaxb-v2/`.

[23] Maja Pesic and Wil M. P. van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *BPM 2006 Vienna, Austria*, volume 4103 of *LNCS*, pages 169–180. Springer, 2006.

[24] Binh Viet Phan and Eric Pardede. Active XML (AXML) research: Survey on the representation, system architecture, data exchange mechanism and query evaluation. *J. Netw. Comput. Appl.*, 37:348–364, 2014.

[25] Jan Sacha, Bartosz Biskupski, Dominik Dahlem, Raymond Cunningham, René Meier, Jim Dowling, and Mads Haahr. Decentralising a service-oriented architecture. *Peer Peer Netw. Appl.*, 3(4):323–350, 2010.

[26] Yutian Sun, Wei Xu, and Jianwen Su. Declarative choreographies for artifacts. In Chengfei Liu, Heiko Ludwig, Farouk Toumani, and Qi Yu, editors, *ICSOC 2012, Shanghai, China*, volume 7636 of *LNCS*, pages 420–434. Springer, 2012.

[27] Peter Triantafillou and Ioannis Aekaterinidis. Content-based publish-subscribe over structured p2p networks. In *26th International Conference on Software Engineering - W18L Workshop "International Workshop on Distributed Event-based Systems"*, 2004.

[28] Peter Triantafillou and Ioannis Aekaterinidis. Peer-to-peer publish-subscribe systems. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2069–2075. Springer US, 2009.

[29] W3C. Web service choreography interface (wsci) 1.0. `https://www.w3.org/TR/wsci/`, aug 2002.

[30] W3C. Web services choreography description language version 1.0. `https://www.w3.org/TR/ws-cdl-10/`, nov 2005.

[31] W3C. Web services description language (wsdl) version 2.0. `https://www.w3.org/TR/wsdl/`, june 2007.