2018 6th International Conference in Software Engineering Research and Innovation

SagaMAS: a software framework for distributed transactions in the microservice architecture

Xavier Limón*, Alejandro Guerra-Hernández†, Angel J. Sánchez-García‡, Juan Carlos Peréz Arriaga§
*‡§*Facultad de Estadística e Informática,* †*Centro de Investigación en Inteligencia Artificial*
Universidad Veracruzana
Xalapa, México
*hlimon@uv.mx, †aguerra@uv.mx, †angesanchez@uv.mx, ‡juaperez@uv.mx

***Abstract*—This paper introduces SagaMAS: a Multi-Agent based framework on development, dealing with distributed transactions in the microservices architecture. Microservices are an architectural style where the distributed system is decomposed in a series of highly cohesive and independent services. Each microservice can have different implementations and data persistence technologies, resulting in heterogeneous distributed systems. Given its heterogeneity and distributed nature, an open challenge in this architecture is the proper management of distributed transactions that expand through several microservices. Multi-Agent Systems are by definition distributed systems suited for complex coordination tasks, such as this. The proposed framework can be seen as a decoupled autonomous layer that coordinates the distributed transactions of the system, relieving the microservice developer from such tasks, and simplifying microservice interactions. Unlike existing approaches, our proposal is conceived and usable at an abstraction level appropriate to express reliability and robustness issues in terms of agent coordination.**

***Keywords*—Microservices, Software Architectures, Distributed Transactions, Multi-Agent Systems**

I. Introduction

Microservices are a software architecture inspired by service-oriented computing that in recent years has enjoyed great popularity [6]. It has arisen as a response to fast technological changes, extensibility and scalability concerns, and the necessity of shorter delivery cycles. The general idea of this architecture is to decompose the system into independent services, usually distributed, that do the least possible amount of work, i.e., highly cohesive and small services that eventually collaborate with each other. Being independent, each microservice has its own life cycle, which allows them to be developed by different work groups at different times and using heterogeneous technologies [17]. All the aforementioned characteristics facilitate the integration and experimentation of new technologies, as well as the extensibility and scalability of the system itself.

There are different ways to deal with heterogeneity of components, being RESTful [11] the most prominent example. The combination of RESTful web services and microservices is very common [17] as it provides ease of distribution, deployment flexibility, fault tolerance and speed of delivery. SagaMAS considers this combination as the way to develop and consume microservices.

Considering its independent, open and heterogeneous nature, micro-services tend to follow a design pattern known as database per service [24], which establishes that each microservice has its own database, which may include various SQL or non-SQL persistence technologies.

Given the potential diversity of data persistence technologies in microservices, an open problem is the management of distributed transactions involving several microservices. A transaction is basically a set of operations that must be executed in an atomic manner, ensuring data integrity if something goes wrong. The necessary coordination to perform each step of the transaction, and potentially clean it in case of error, while at the same time trying to maintain good performance and scalability of the system, is a non-trivial problem that requires technologies and design patterns.

Multi-Agent systems (MAS), facilitate the realization of complex coordinated tasks in distributed systems, since they are, by definition, distributed systems that have the appropriate level of abstraction to facilitate communication between components, as the required flexibility, modularity, robustness, autonomy and scalability [14].

Given the mentioned characteristics, it is natural to think about the possibility of merging MAS with the microservices architecture. This merging, although currently not very common, can offer interesting possibilities such as the one presented in this paper, where an approach based on MAS is proposed to solve the problem of distributed transactions in the architecture of microservices. The main objectives of the proposed approach are the following:

1) Scalability. Suitable for large scale systems, with various computational nodes and a large number of microservices.
2) Reliability. It is possible to adopt strategies to maintain data integrity even in fault situations.
3) Heterogeneity. Open to any type of implementation technology and data persistence in microservices.
4) Appropriate abstraction level. The proposed model, its workflow, its concepts and associated strategies can be understood in a high abstraction level thanks to the use of agents. This aspect facilitates the use and configuration by the microservice developers.

The rest of this work is organized as follows. Section II reviews the background on distributed transactions in mi-

978-1-5386-6577-0/18/$31.00 ©2018 IEEE
DOI 10.1109/CONISOFT.2018.00015

croservice architectures. In section III SagaMAS is presented, describing the general model to represent transactions, a configuration format to describe the flow of a transaction and finally a set of strategies for handling errors. Section IV deals with the formalization of SagaMAS through the Prometheus software development methodology, for which the system specification phase is introduced. In section V design an implementation choices for SagaMAS are addressed. Finally, in section VI several conclusions of the work are presented together with the future work to be done.

II. Background

Distributed transactions are not a new issue in software engineering, they are an inherent problem of distributed systems and there are several efforts in the literature to address them, such as the Microsoft Distributed Transaction Coordinator (MSDTC) of the .NET platform [13], and various implementations in database managers such as Oracle and MySQL which usually use a protocol for transactions known as two-phase commit protocol [15].

However, such traditional solutions tend to suffer from the following problems in the context of microservices [17]:

1) Homogeneity. For the mechanism to work, each part of the system must be technologically compatible, whether in persistence or implementation technology.
2) Low scalability. Many solutions work in a synchronous fashion, that is, the components can stop their work or block tables in the database waiting for the response of other components.

Another possibility is that microservices themselves coordinate the transactions. This approach, although direct and heterogeneous, is difficult to implement adequately and reliably, especially in regard to error handling.

Following the above problems, especially scalability, a persistence model known as eventual consistency [1] has been adopted. This model does not require blocking and enables concurrent work on incomplete transactions and partially updated data. The model states that if there are no future updates, the database eventually becomes consistent. Eventual consistency allows an asynchronous model, which does not require coordinators in the transaction, nor wait for another component to finish its work, each component can work independently.

Eventual consistency carries certain risks. In order to be implemented properly, the developer must take into account concurrent situations that could generate inconsistencies, complementing them with domain code [17]. For example, in an application to make purchases, instead of directly subtracting money from the client's account, first reserve the necessary credit to make a purchase and if the transaction goes well, the money is subtracted from the account as a final step.

There is a need to have concrete ways to deal with distributed transactions in the context of microservices. In the next section, we present a series of related design patterns that attempt to solve the stated problem.

A. Related design patterns

Possibly the most important and known design pattern for handling distributed transactions with eventual consistency is the Saga pattern [8]. A saga is a sequence of local transactions where each transaction updates data from a single microservice. The first transaction is initiated by an external request, then each subsequent step is triggered after having completed the previous step. If something goes wrong at any step, a series of compensation actions that try to clean the transaction are executed. This pattern can be implemented in two general ways:

- Orchestrated. There is a central transaction coordinator, who orchestrates each step of the transaction and coordinates compensation operations in case of failure. Each time a step is completed, the microservice notifies the coordinator that it has finished. This orchestration allows the microservices to remain decoupled. The orchestration can be synchronous or asynchronous.
- Choreographed. Based on asynchronous events. Each microservice produces an event each time it completes one step of the transaction or an error occurs, other microservices listen to the events and continue with the process accordingly.

The traditional orchestrated approach has the disadvantage of having a central point of failure, and leads to the centralization of traffic. Moreover, the transaction coordinator can be a very complex module to create and maintain. Another disadvantage of this approach is that, in order to be reliable, it requires atomicity between the completion of a local operation and the notification to the coordinator, in the absence of such atomicity it can be the case that the local operation finished but due to some error the coordinator could not be notified.

In order to be reliable, the choreographed approaches also required atomicity between the completion of the local operation and the publication of the associated event. One possible approach to solve this problem is a design pattern known as Event Sourcing [24]. Event Sourcing establishes the persistence of domain entities, as a sequence of state changing events. Each time the state of a domain entity changes, a new event is added to its event queue. If you want to recover the current state of the domain entity, simply run the events in order. Since storing an event is a single operation, this approach is inherently atomic.

In Event Sourcing, applications persist events in an event store, which has the necessary mechanisms to add and obtain the events of an entity. When coupled with the Saga pattern, the event store also serves as an intermediary for choreographed transaction coordination, allowing microservices to subscribe to events that other microservices can store.

When using Saga and Event Sourcing together, the problem of not having directly available the current state of an entity to make queries arises. To solve this problem another design pattern known as CQRS [25](Command Query Responsibility Segregation) is used. In this pattern the application is divided into two parts: the command side and the query side. The

command side handles creation, update and deletion operations, issuing pertinent events; essentially this is the Event Sourcing side. The queries side handles queries, which are executed against one or more materialized views of the entity; for example, a materialized view can be obtained through executing the events of the entity's queue in the event store. To create views more efficiently, you can save periodic images of the entities in a database, in this way the reconstruction of the entity is also simplified by not having to execute all the historical events to reconstruct the current state. Eventuate [9] is an example of framework that follows Event sourcing and CQRS for microservices.

The choreographed approach is a natural way to implement the Saga pattern, which promotes low coupling between microservices, however, as more microservices participate in the transaction, it becomes more complex to understand the flow of it and determine which services listen to which events. It is also necessary to avoid cyclic dependencies that can create infinite loops of events. In addition to this, to be reliable and usable, it needs to be accompanied by other design patterns, e.g., Event Sourcing and CQRS, which can complicate implementation and bring other problems with it. For example, when using Event Sourcing, the event store represents a central point of failure. Likewise, in spite of the reliability improvement that Event Sourcing establishes, a way or recommendation to handle cases in which an event is generated but the receiving service is inaccessible has still not been defined; if cases like this are not treated, the transaction is incomplete and with inconsistent data in the event store.

Many of the mentioned problems of the choreographed approach and their associated patterns are due to the fact that the transaction itself does not exist as a high level concept, the transaction is hidden at a low level in the form of events, which complicates its atomic treatment. In the proposed approach, attempts are made to solve the problems of existing approaches, while at the same time simplifying the process by maintaining a high-level view of transactions, exploiting the high level of abstraction that the systems Multi-Agent allow.

III. SagaMAS general model

In this section we present a general model to implement distributed transactions for microservices architecture, this approach has the following characteristics:

- Based on agents. MAS concepts such as communication based on speech acts [16] and coordination are exploited.
- Based on eventual consistency. Given that it is the data persistence model that allows greater scalability and heterogeneity.
- The proposed approach can be seen as a possible implementation of the Saga pattern, thus the name SagaMAS.
- Based on a semi-orchestrated asynchronous model. It is simi-orchestrated because there is no central command unit but each agent asks another to perform the next step; It is asynchronous since agents are able to communicate or ask for something without waiting for an answer, likewise agents can exhibit reactive behavior when an event happens.
- It defines a transaction model that allows to describe workflows as well as transaction settings.
- It proposes error handling strategies for detection, tolerance and error recovery, these strategies exploit the high level of abstraction of MAS.

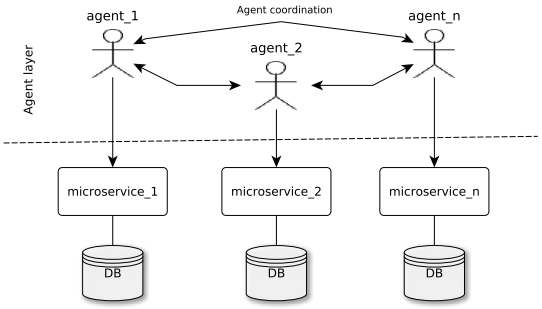

Fig. 1. General model. Each microservice has a database BD associated and an agent. Agents, in the independent agent layer, coordinate among themselves to perform transactions between microservices.

The general idea of SagaMAS is that each microservice has a particular agent associated, that agent may or may not be on the same server. Each transaction is born in a microservice, which has the means to communicate, directly or indirectly, the start of the transaction to its associated agent. The concrete form of communication depends on the implementation, for example, if both the microservice and the agent are on the same server, communication can be established simply by means of a pipe, or specific agent technologies such as CArtAgO [22] allowing the agent to react to signals emitted by the microservice. Once the start of the transaction is reported to the agent, the transaction is handled by an independent Multi-Agent layer, which is also in charge of handling errors regarding transactions and the agent layer itself. The figure 1 shows the general idea of the proposed approach.

A. Transaction model

In this section a proposed transaction model is presented, which facilitates the configuration and representation at the agent level of the transactions, this model is based on concepts of graph theory.

Definition 1: A transaction $T = \{St_1, St_2, ..., St_n\}$ $n >= 1$ is a set of simple paths, i.e, paths that do not pass more than once through the same vertex, called sub-transactions.

Definition 2: A sub-transaction $St = \{sc_1, sc_2, ..., sc_m\}$ $m >= 1$ is a set of paths of length 1 called sections.

In this way, transactions can be viewed as linear sequences that are broken down into sections. In each section one step of the transaction is executed, for this, it is necessary to have associated information.

Definition 3: $IS = \langle tr\_id, st\_name, input\_data, incoming\_action, output\_data, compensation\_action,$

$state, next\rangle$ is a tuple of information associated with each section, where:

- tr_id A unique identifier of the transaction to which the section belongs.
- st_name A logical name of the sub-transaction to which the section belongs.
- $input\_data$ An associated data to be processed by the section.
- $incoming\_action$ A name of the microservice action to be executed in the section. The action receives $input\_data$.
- $output\_data$: data produced by $incoming\_action$.
- $compensation\_action$ A name of the microservice action to be executed if $incoming\_action$ has already been executed and something went wrong at any point in the transaction. The action receives $output\_data$.
- $state$ Is the execution status of the section, it is a value in the set $\{pendant, processing, processed, pendant\_compensation, processing\_compensation, processing\_compensation\}$.
- $next$ An agent that will continue with the sub-transaction st_name for the current transaction tr_id. It can also be established a link between the current sub-transaction st_name and a new new_st_name, this in case of needing to visit an agent again, that is, create a path that is not simple.

The proposed approach does not have specific guidelines as to how to represent or create $input\_data$ and $output\_data$, this depends on the microservice developers and the specific implementation of the model. For example, the use of a standard text format such as JSON may be preferred where developers may choose to use only some of the input data for their operations.

It is also interesting to note that the model allows the creation of sub-transactions that work in parallel, thanks to the fact that it is possible to link an existing sub-transaction with a new one through $next$, as long as there is no direct dependency between the sub-transactions, in which case a single sequence of sub-transactions can be used.

B. Configuration

Most information of a section can be generated dynamically by the agent layer, however, some data must be configured by the microservices developers: st_name, $incoming\_action$, $compensation\_action$ and $next$. A configuration format suited for the agents in the MAS is proposed, which has the purpose of representing the transactions in which the agents participate. The configuration is done per agent, and is, together with the actions to be executed, the only aspects referring to transactions for which the microservice developer must worry. The proposed format contains the following predicates:

- $incoming\_action(st\_name, action)$: sets the action $action$ to be executed in the microservice given the sub-transaction st_name. The action is executed using $input\_data$ and generates $output\_data$ as mentioned in section III-A.
- $compensation\_action(st\_name, action)$: sets the action $action$ to be executed in case there is a problem in the transaction and $state = processed$ for the corresponding section st_name. The action is executed using $output\_data$.
- $next(st\_name, agent)$: sets the next agent $agent$ to continue the sub-transaction st_name.
- $next(st\_name, new\_st\_name, agent)$: links an existing st_name sub-transaction with a new new_st_name and sets the $agent$ agent that will start with the new sub-transaction. This predicate is useful for creating non-simple paths in the transaction without ambiguity, i.e., when the next agent of a sub-transaction had already been visited.

As an example of configuration, consider a case where a customer wants to establish a purchase order on a product. Below is a possible configuration using the proposed format.

```
// Agent order
incoming_action(orderCreation, createPendingOrder).
compensation_action(orderCreation, deleteOrder).
next(orderCreation, customer).
incoming_action(orderApproval, approveOrder).
compensation_action(orderApproval, cancelOrder).
```

```
//Agent customer
incoming_action(orderCreation, reserveCredit).
compensation_action(orderCreation, cancelReservation).
next(orderCreation, orderApproval, order).
```

The workflow of the transaction described in the configuration can be easily represented by a diagram similar to a UML sequence diagram, which is shown in figure 2. Note that the figure only shows the normal flow of the transaction, i.e., without compensation actions. Likewise, for readability, the figure does not show the input and output data for each action. This easiness of representation is due to the high level management of transactions and shows the possibility of a tool that automatically generates the configuration files through a graphical interface where the main and alternate workflows (compensation actions) of the transaction are modeled.

C. Workflows

In addition to a transaction model and a form of configuration, it is necessary for the agent layer to perform various tasks that support transactions. In this section, from the perspective of the agent layer, the workflow of a transaction is presented, including the cases when the transaction is performed normally and when a failure is generated. The normal flow of a transaction is as follows:

1) A microservice communicates the start of the transaction to its associated agent, delivering input data $input\_data$ and the name of the initial sub-transaction st_name.

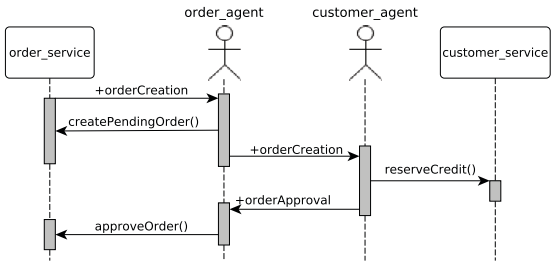

Fig. 2. Normal workflow of the transaction of the example of orders and clients. The " + " symbol is used to denote a new belief.

2) The agent determines which agents will participate in the transaction and which sub-transactions are part of the transaction. This information must be known by all agents.
3) A unique identifier tr_id is created for the transaction.
4) In each section the corresponding agent executes the appropriate $incoming\_action$, passing $input\_data$ as input and generating $output\_data$ as output.
5) Using the information of $next$, the agent of the section passes $output\_data$ and st_name to the next agent.
6) When a sub-transaction ends, the last agent at the far end communicates it to the other agents of the transaction. A sub-transaction ends when a $next$ in the current section is no longer associated, or when a new sub-transaction is linked in $next$.
7) When all sub-transactions end, the transaction ends.

The alternate compensation workflow only considers problems that could occur at the level of $incoming\_action$, for example, in the context of orders and clients, an error of this type may be that the client does not have enough credit, in this case the action fails and therefore the entire transaction. A possibility of handling errors derived from loss of connectivity and shutdown of agents is specified in the following section. The compensation workflow of a failure generated in an action is the following:

1) An $incoming\_action$ of a section fails and the associated agent perceives it.
2) The agent sends a message to the other agents of the transaction to stop their work related to the transaction, this in case there are parallel sub-transactions.
3) The agents consider the $state$ of their sections and execute the $compensation\_action$ if necessary, using the appropriate $output\_data$ for the section. The execution of the $compensation\_action$ is done in the reverse order of $incoming\_action$.
4) When an agent finishes executing his compensation actions, it communicates it to the other agents of the transaction.
5) When all the agents finish, the transaction ends.

D. Error handling strategies

In a distributed system, it is important to consider component-independent failure and establish appropriate error detection, tolerance and recovery strategies [5]. In this section, we propose a series of possible strategies to be implemented for handling errors, considering only problems that may arise in the agent layer since the problems derived from microservices can be handled through the alternate compensation workflow presented in the section III-C. For handling errors, two types of general errors are considered:

1) Loss of connectivity between agents. It refers to network problems, which can be temporary and affect the coordination between agents.
2) Shutdown of some agent. It groups problems such as the generation of an unhandled exception to hardware failures that cause the agent process to stop abruptly.

The most severe type of error is the shutdown of an agent, since it may require a manual restart and a more complex error recovery, however, from the agent's point of view, the effect of either of the two errors mentioned above is not being able to establish any communication with the affected agents. Furthermore, the recovery of dropped agents has common elements of error recovery derived from connectivity problems. In this way, the proposed error detection strategy is based only on loss of connectivity and a general error recovery strategy is proposed that applies to connectivity errors and partially to shutdown errors. Later, an extension is proposed for the recovery of shutdown errors. The strategy of detection, tolerance and general error recovery is as follows:

- As part of the agent layer, each agent maintains a periodic exchange of connectivity information with one or two agents. This connectivity is linear, in this way, the exchange can be from left to right and from right to left. Agents at each end only exchange data with an agent that gives information about the connectivity of all agents at the opposite end; while the other agents maintain an exchange with two other agents, the agent on their left receives connectivity information from the other agents on the left, while in turn they inform them about the connectivity of the agents that are on the right, this happens on the right side in reverse. The process described above is shown in the figure 3.
- With the connectivity information, agents can determine if it makes sense or not to start a new transaction or continue one in progress. This aspect can be seen as a form of fault tolerance.
- If an agent loses connectivity, this fact is reported to all agents by one of the agents waiting for a message from the affected agent. During this loss, it is possible that some agent has tried to contact the affected agent, this as part of the work of a transaction, if this is the case, then the transaction is paused.
- Non-affected agents wait for a grace period before making any error recovery, in case a reconnection occurs.

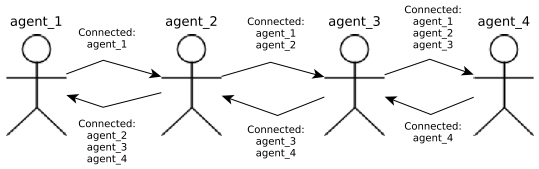

Fig. 3. Exchange of connectivity information between agents. Each agent shares its information with one or two agents to mitigate network load.

- During the grace period, connectivity messages are sent as normal.
- If the reconnection occurs during the grace period, then the agents that had yet to contact the affected agent restart their work.
- If the reconnection does not occur during the grace period, the agents with connectivity adjust their exchange of connectivity information with the available agents and continue working, all this as a measure of error tolerance. At the same time, they continue trying to contact the agents not reachable. Agents affected by incomplete transactions initiate a process similar to the one described in the alternate compensation flow mentioned in the section III-C, except that the transaction is not considered to be completed until eventually there is a reconnection and a complete cleaning can be done.

It is important to note that when a connectivity problem occurs, it is possible for several agents in the same network segment to lose connectivity with the agents of another segment, in the aforementioned strategy, each segment carries out the steps described above independently. The general strategy of error recovery before a reconnection that happens after the grace period is the following:

- When one or more agents are reconnected, an attempt is made to recreate the connectivity exchange as it was before the loss of connection.
- If all the agents that are part of some incomplete transaction are reconnected, the transaction is considered finished, considering that agents have already executed their compensation actions.

The extension of the aforementioned strategy for agent shutdowns is as follows:

- Each agent keeps a persistent log of the status of each of its transactions.
- When the agent restarts, it checks its log and executes the appropriate compensation actions.
- The agent reconnects with the other agents.

In order to process error recovery derived from agent shutdown reliably, it is necessary to have atomicity between the writing of the log and the performance of the actions that change the state of the transaction, this can be achieved in different ways at the implementation level, for example, using traditional local transactions.

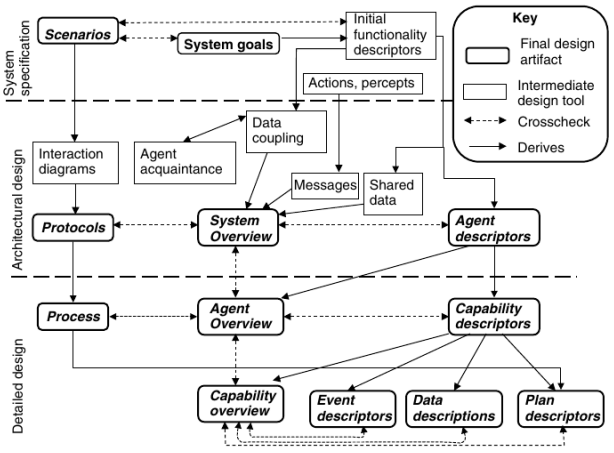

Fig. 4. The phases of the Prometheus methodology, adapted from [20]

IV. Methodology

The general models, workflow, and strategies described in section III need to be formalized in order to be properly implemented in a functional framework. For this, SagaMAS follows the Prometheus [19] methodology, a software developing methodology for MAS. Prometheus defines a detailed process for specifying, designing, implementing and testing/debugging Agent-Oriented software systems [2]. Aside from a detailed processes, it also defines a range of artifacts created in each phase of the process.

Prometheus consists of three phases, as shown in figure 4:

1) The system specification phase focuses on identifying the goals and basic functionalities of the system, along with inputs (percepts) and outputs (actions).
2) The architectural design phase uses the outputs from the previous phase to determine which agent types the system will contain and how they will interact.
3) The detailed design phase looks at the internals of each agent and how it will accomplish its tasks within the overall system.

The phases are applied iteratively, following a RUP [12] like scheme. Prometheus is intended to be useful and usable by industry developers, it is described in sufficient detail and it provides tool support for artifacts creation.

Being SagaMAS a still on development framework, in what follows, only the system specification phase is described, leaving the last two phases as future work.

A. System specification

This phase centers on the specification of goals, that can be seen as the requirements of the software system. Goals can be detailed using scenario descriptions. This phase also requires the specification of functionalities related to identified goals [20]. There is also a focus on how the agent system interfaces with the environment in which it is situated, in

terms of percepts that arrive, or can be obtained, from the environment, and actions that modify the environment.

Figure 5 depicts the goals diagram for SagaMAS, which groups and relates goals and sub-goals of the system.

It is worth mentioning that a lot of effort is being done in order to make SagaMAS easy to use for developers, thus the goal "User experience" as the top level goal of the diagram.

From the identified goals, functionalities are identified, which includes a grouping of related goals, as well as percepts, actions and data relevant to the behavior. Figure 6 shows the functionalities diagram for the SagaMAS system.

Each functionality is accompanied by a descriptor, which includes a short description, related goals, related actions, notes about the data required and produced by the functionality, and related triggers, i.e, information about what events or situations will cause activity to be initiated within the functionality. For space limitations, only an example descriptor for the deployment management functionality is presented next.

- **Deployment management functionality**
- **Description:** This functionality deals with deploying and initializing the agents in the distributed system from the start of the system or after a system failure.
- **Goals:** Deployment easiness
- **Actions:** Distribute agents
- **Triggers:** System start, Agent restart
- **Information used:** Deployment configuration, Transaction configuration
- **Information produced:** Connectivity log

Next, scenarios related to goals and functionalities can be created. Scenarios are complementary to goals as they show the sequences of steps that take place within the system. Scenarios are used to illustrate the normal and alternative running of the system. Possible steps are achieving a goal, performing an action, receiving a percept, or referring to another use case scenario. For convenience, just one of the scenario description of SagaMAS is presented, this scenario deals with connectivity issues between agents.

- **Connectivity loss scenario**
- **Trigger:** connectivity failure percept
- **Description:** when an agent fails to be located, the system waits for reconnection. In this scenario reconnection does not occur.
  1) GOAL: Error handling
  2) GOAL: Connectivity information
  3) ACTION: Request connectivity information
  4) OTHER: wait for reconnection
  5) GOAL: Rollback support

It is also useful to identify the percepts that the agents can get from their environment and other agents, such percepts may serve as triggers of scenarios. The identified percepts serve as an early interface descriptor, which it is further refined in the architectural design phase. The list of identified percepts for the SagaMAS systems is as follows:

- Transaction started
- Transaction rollback started
- Subtransaction finished
- Continue next subtransaction step
- Transaction finished
- Subtransaction finished
- Transaction rollback finished
- Connectivity failure
- Reconnection

V. Design and implementation choices

Even if the design phase of the development of SagaMAS is yet to be refined, some design and implementation choices have already been taken, these decisions may influence the final architectural design. In this section such choices are presented.

For the Multi-Agent development stack, the JaCaMo [3] framework has been chosen. JaCaMo is the result of the composition of three technologies for MAS: Jason [4] (taken as a proper name inspired by Greek mythology), CArtAgO [22] (Common ARTifact infrastructure for AGents Open environments), and MOISE [10] (Model of Organization for multI-agent SystEms).

Jason is an agent oriented programming language that entails the Belief-Desire-Intention (BDI) approach, it is based on the abstract language $AgentSpeak(L)$ [21]. Apart from its solid BDI theoretical foundations, the language offers several facilities for programming Java powered, communicative MAS. Communication in Jason is based on Speech Acts, as defined in KQML [7].

CArtAgO provides the means to program the environment, following an endogenous approach where the environment is part of the programmable system. In CArtAgO terms, the aspects that characterize a model for environment programming are the following [23]: 1) Action model: how to perform actions in the environment. 2) Perception model: how to retrieve information from the environment. 3) Environment computational model: how to represent the environment in computational terms. 4) Environment data model: how to share data between the agent and environment level to allow interoperability. 5) Environment distributed model: how to allow computational distributed environments. Aspects 1-3 are directly supported by artifacts [18], which are dynamical sets of computational entities that compose the environment and encapsulate services and tools for the agents. Artifacts are organized and situated in workspaces, which essentially are logical places (local or remote) where agents center their attention and work. Aspect 5 is supported by workspaces, but also partially by artifacts, as artifact actions can be executed remotely. Aspect 4, on the other hand, depends on the underlying agent programming language used and is not directly related to artifacts or workspaces.

MOISE provides the means to create agent organizations, which have the aim to control and direct agent autonomy in a general purpose system. To this end, it is possible to specify tree aspects: i) Structural, consisting on the different agent groups and roles that take part in the organization; ii) Functional, defined by social schemes, missions, and goals

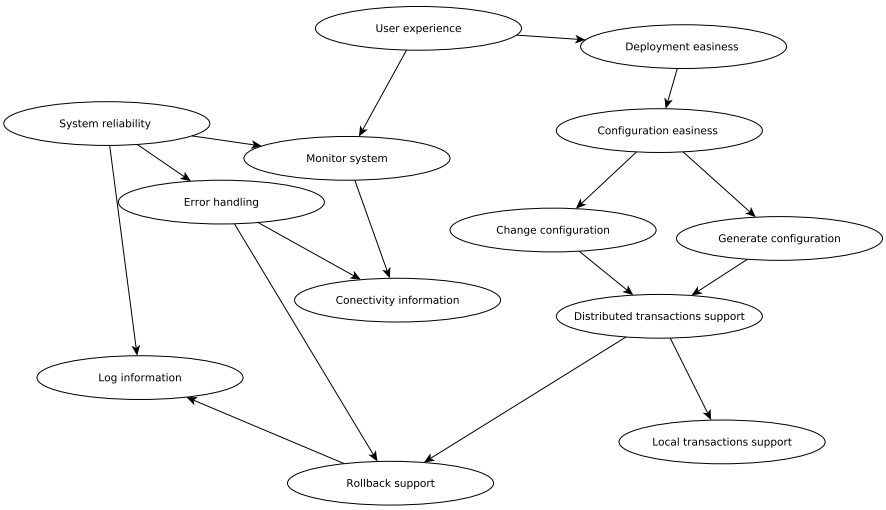

Fig. 5. Goals diagram for the SagaMAS system.

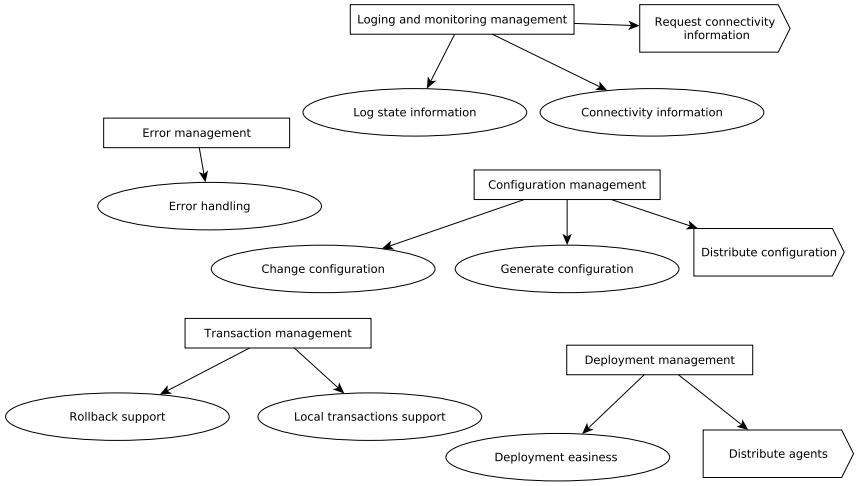

Fig. 6. Functionalities diagram for the SagaMAS system. Functionalities are depicted by rectangles, goals with ovals, and actions with a rectangle extended with a triangle pointing to the right. Arrows link functionalities to their goals and actions

which direct the agent behaviour toward organization ends; and finally iii) Normative, defined though norms that bind roles to missions, constraining agent's behaviour when entering a group and playing a certain role.

An important aspect of SagaMAS is how the autonomous agent layer is going to interact with the heterogeneous microservices, for this end, RESTful microservices using JSON as exchange format are chosen. This widely supported option gives, in our opinion, the most flexibility and heterogeneity. A deployment configuration may be created to state the API of each microservice in order for the correct generation of the $input\_data$ and $output\_data$ of each sub-transaction, as stated in section III-A. In the same regard, the agent layer will have a RESTful service to start a transaction, so the coupling between the agent layer and microservices stays as low as possible.

VI. Conclusions and future work

SagaMAS promotes a high level modeling of scalable and heterogeneous distributed transactions, providing a flexible definition of transaction, which can be represented as agent knowledge through the proposed configuration format. The use of MAS allows the definition of high level of workflows and error handling strategies, leaving in turn open opportunities to improve aspects of the process through more sophisticated negotiation and agent reasoning schemes that exploit techniques of Artificial intelligence.

From the software engineering point of view, SagaMAS promotes a low coupling between microservices, since the coordination of transactions is isolated in the agent layer. Likewise, the coupling between the microservice side and the agent layer remains low. From the point of view of microservices, there is no agent layer, except perhaps to solve things like the start of a transaction. In turn, the agent layer is generic, it can be adapted to any system based on microservices, so there is no domain dependency between both parties, thus allowing each part to evolve separately. This low coupling at all levels has the benefit of greatly facilitating the management of distributed transactions for developers, since they do not have to worry about establishing special code, such as for managing events, but rather the reliable and efficient management of the transaction is carried out by the agent layer. Likewise, there is no central point of failure or concentration of traffic, as in many other existing approaches discussed in section II.

The immediate future work is to follow and refine the next phases of the Prometheus methodology, realizing the architectural design and the detailed design of the system. Taking into account the design and implementation choices mentioned in section V a functional implementation of SagaMAS is envisaged, such implementation must be tested and debugged. Of particular interest is the assessment of scalability for which stress testing may be employed. Also, the inherent costs of adding an agent layer to microservices has to be measured and compared to existing approaches.

The adoption of BDI technologies, such as JaCaMo, opens possibilities to improve aspects of reasoning, negotiation and coordination between agents. One of the future objectives is to explore more sophisticated schemes in the handling of transactions, so that, for example, agents are able to reason whether or not to execute an operation on data that is being manipulated in another transaction, with the objective of mitigate possible problems derived from the use of eventual consistency.

REFERENCES

[1] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20, 2013.

[2] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. *Methodologies and software engineering for agent systems: the agent-oriented software engineering handbook*, volume 11. Springer Science & Business Media, 2006.

[3] Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761, 2013.

[4] Rafael H. Bordini, Jomi F. Hübner, and Mike Wooldridge. *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd, 2007.

[5] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[7] T. Finin et al. An overview of KQML: A knowledge query and manipulation language. Technical report, University of Maryland, CS Department, 1992.

[8] Hector Garcia-Molina and Kenneth Salem. *Sagas*, volume 16. ACM, 1987.

[9] Red Bull Media House and Martin Krasser. Eventuate. *URL: http://rbmhtechnology.github.io/eventuate/index.html [accessed: 2018-06-03]*, 2015.

[10] Jomi F Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20(3):369–400, 2010.

[11] Nicolai M Josuttis. *SOA in practice: the art of distributed system design*. " O'Reilly Media, Inc.", 2007.

[12] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[13] Rodney Limprecht. Microsoft transaction server. In *Compcon'97. Proceedings, IEEE*, pages 14–18. IEEE, 1997.

[14] Chayapol Moemeng, Vladimir Gorodetsky, Ziye Zuo, Yong Yang, and Chengqi Zhang. Agent-based distributed data mining: A survey. In *Data mining and multi-agent integration*, pages 47–58. Springer, 2009.

[15] C Mohan and Bruce Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 76–88. ACM, 1983.

[16] Álvaro F Moreira, Renata Vieira, Rafael H Bordini, et al. Extending the operational semantics of a bdi agent-oriented programming language for introducing speech-act based communication. *Lecture notes in computer science*, pages 135–154, 2004.

[17] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[18] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous agents and multi-agent systems*, 17(3):432–456, 2008.

[19] Lin Padgham and Michael Winikoff. Prometheus: A methodology for developing intelligent agents. In *International Workshop on Agent-Oriented Software Engineering*, pages 174–185. Springer, 2002.

[20] Lin Padgham and Michael Winikoff. *Developing intelligent agent systems: A practical guide*, volume 13. John Wiley & Sons, 2005.

[21] A.S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[22] A. Ricci, M. Viroli, and A. Omicini. Construenda est cartago: Toward an infrastructure for artifacts in MAS. *Cybernetics and systems*, 2:569–574, 2006.

[23] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[24] Chris Richardson. Microservice architecture patterns and best practices. *URL: http://microservices. io/index. html [accessed: 2018-03-17]*, 2016.

[25] Greg Young. Cqrs and event sourcing. feb. 2010. *URl: http://codebetter. com/gregy oung/2010/02/13/cqrs-and-event-sourcing*, 2010.