Swarm Communication – A Messaging Pattern Proposal for Dynamic Scalability in Cloud

Conference Paper · November 2013

DOI: 10.1109/HPCC.and.EUC.2013.277

3 authors:

L. Alboaie
Universitatea Alexandru Ioan Cuza
25 PUBLICATIONS **59** CITATIONS
SEE PROFILE

Alboaie Sînică
Axiologic Research
7 PUBLICATIONS **31** CITATIONS
SEE PROFILE

Andrei Panu
Universitatea Alexandru Ioan Cuza
5 PUBLICATIONS **2** CITATIONS
SEE PROFILE

2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

Swarm Communication – a Messaging Pattern Proposal for Dynamic Scalability in Cloud

Lenuta Alboaie
Faculty of Computer Science
Alexandru Ioan Cuza University
Iasi, Romania
Email: adria@info.uaic.ro

Sinica Alboaie
S.C. Axiologic SaaS S.R.L.
Iasi, Romania
Email: abss@axiologic.ro

Andrei Panu
Faculty of Computer Science
Alexandru Ioan Cuza University
Iasi, Romania
Email: andrei.panu@info.uaic.ro

***Abstract*—Programming for cloud systems looks deceptively similar with programming web and enterprise applications except that it is harder. The challenges that a few years ago were reserved only for distributed systems specialists from academia or from big internet companies are now coming to masses. The well known methods of dealing with concurrency and scalability by using threads should be replaced with asynchronous messages because vertical scalability is limited and expensive. Programming with asynchronous messages can be also a challenge and we present in this paper an improved method, called swarm communication. The proposal is based on the metaphor inspired by nature, and we think at messages as relatively smart beings visiting relatively non intelligent places. Through this article we describe this new method for decomposing complex applications in small services and we will present how dynamic scalability can be achieved in a swarm system.**

I. Introduction

With the emergence of cloud computing, which is a specialized form of distributed computing, the sophistication of service-oriented solutions has increased greatly and many solutions emerged for the integration problem.

For small and medium enterprise systems, the integration problem was solved through an ad-hoc middleware in which the entire communication was mediated by a node (server) which realized all the basic operations. For big enterprise systems, this solution proved to be unreliable and thus another approach was taken in the form of MOMs (message oriented middlewares) [1], [2], which introduced functionalities like message routing and message transformation. As the systems evolved, their complexity increased and new integration solutions were developed based on ESB (Enterprise Service Bus) [3], [4], which introduced support for orchestrating services. With the emergence of cloud computing, new integration necessities appeared, like commercial cloud systems that need interoperation with on-premise enterprise systems or complex systems with components that are not under the control of the same entity. In situations like these, integration can be done using service choreography.

In this article we propose the design and implementation of a messaging component which can be the base layer for an iPaaS (Integration Platform as a Service). This component uses as communications mechanisms principles from the choreography metaphor [5]. In section 2 we present a domain overview in which we made comparisons with existing systems at architectural level. In section 3 we present our proposed architecture and in section 4 we describe a technique which demonstrates how load balancing can be addressed in our proposal.

II. Domain Overview

In [5], [6] orchestration is seen like a system where a conductor coordinates every person in an orchestra. The coordinator (the intelligent node) is the one who indicates during a concert (achieving a task) the way each member in the orchestra (each node in a distributed system) must act (accomplish an action). Therefore we have a centralised system.

Choreography [5], [6] is seen like a space where dancers (the nodes in a distributed system) accomplish some actions according to a set of rules that they previously learned and now they collaborate in order to perform a dance (realise a task). In this situation the system is not centralised anymore.

Our proposal, called Swarm, aligns with this last scenario and also adds a new element: the nodes (dancers) that are part of the system act based on rules that they do not initially know, rules brought by smart messages that will visit each node. Metaphorically, we can see the dancers as being ”inspired” and acting according to the specifications from the messages. After finishing the task, they will return to the initial state or they will be ”inspired” again (by a new message). This represents a new approach. All the existing commercial and open source solutions that address the integration problem were developed based on the ESB architecture, like MuleESB [7], Apache ServiceMix [8], SwitchYard (former JBoss ESB) [9], OpenESB [10] TalendESB [11]. There are many similarities, but also differences between them, because they take different approaches to achieve the same goal. A comparison can be made regarding many key features of an ESB, like transports and connectors support (JMS, XMPP, AMQP, SOAP, TCP, FTP, JDBC etc.), security plugins, web services support (Axis, CXF, REST, WSDL etc.), transformation mechanisms (XSLT, Smooks etc.), BPM integration etc.

For example, OpenESB was designed and built from the ground up to conform strictly to the JBI (Java Business Integration) [12] standard. The adherence to this JCP-created [13] standard has some constraints when developing integration solutions and services. It requires that all components be managed within JBI containers and all follow JBI conventions, the implementation of containers is tightly coupled to WSDL descriptions of services, performing orchestrations without additional extensions can be done using only BPEL [14] and being an Oracle-backed project, OpenESB is tightly coupled with other development tools from Oracle, like NetBeans IDE and Java EE. Due to the general unpopularity, JBI is considered a deprecated standard, OpenESB remaining the only mature project built on it. ServiceMix, an open source ESB project from Apache, used to be also built from the ground up on JBI specification, but the latest iteration, version 4, was rewritten and is designed on top of the OSGi specification [15], which is a set of specifications that defines a dynamic component system for Java, providing a modular architecture for large-scale distributed systems. ServiceMix contains an engine that executes business processes and supports orchestrations using WS-BPEL standard. TalendESB is another solution built on the same core technologies like ServiceMix, but without JBI support.

MuleESB is a standards agnostic approach to integration. It provides support for JBI containers, but does not restrict developers to the JBI model. Instead, Mule uses a simple POJO and XML based architecture and configuration that greatly eases the learning process thanks to its similarity to Java. It enables orchestrations using BPEL or using a proprietary mechanism named Mule Flow, which automates integration processes and allows construction of sophisticated integration solutions.

SwitchYard is a lightweight service delivery framework providing full lifecycle support for developing, deploying, and managing service-oriented applications [9]. It is built based on the Service Component Architecture (SCA) [16] approach. At its core, it provides all the features that are typically associated with an ESB and adds many more, like support for Camel routing [17], BPMN tools, Drools rules enginesc̃itedrools and other tools for messaging, business process management and service orchestration. Also, new tooling that SwitchYard supports provides a better visual representation of the integration design. Using all the new features in combination with SwithcYard's visual representation makes adopting SwitchYard less complicated that adopting JBoss ESB.

With the emergence of cloud computing and the increasing need for greater scalability, ESB solutions have seen the next iteration in the form of iPaaS. CloudHubc̃itecloudhub is an enterprise-class integration platform as a service that connects SaaS (Software as a Service) and on-premise applications. It is designed from the ground up to be highly available and architecturally consists of three major components: a web console, a platform service cloud and an application cloud. The web console is the interaction point for the users, the platform cloud is the set of platform services (alerting, logging, load balancing etc.) and the application cloud, or the worker cloud, is an elastic cloud of MuleESB instances that run integration applications. Workday's Integration Cloud Platform [18] is another integration solution that is built with the same architectural ideas: the core component is an enterprise class ESB grid consisting of many ESB instances that are managed and monitored by another major component of the platform. There are many other iPaaS platforms with proprietary implementations, like SnapLogic Integration Platform [19], SoftwareAG Integration Live [20], Tieto iPaaS [21], Informatica iPaaS [22].

In this paper we propose a new paradigm regarding communication mechanisms and composability of services. It brings a new perspective compared to service orchestration and choreography and also to MOM and ESB systems. We call our approach swarm communication. The goal of swarm communication, as in case of service integration patterns from SOA [5], is to compose services' behaviours. Swarming, as we define it, is different from service orchestration because there is no central controller that manages the business logic and messaging sequence. It is also different from service choreography because the individual nodes that represent the participant services do not fully control the answers to requests, they do not know the entire business logic as in case of choreography. We can say that swarming is somewhere in the middle, with more smart messages and less intelligent service nodes. The logic is distributed in swarm descriptions and the effective execution and decisions take place in individual nodes. The lack of a central orchestrator makes a swarm based system inherently more scalable. By using swarm communication, a system will have the ability to dynamically scale, meaning it will be capable of handling variations in capacity requirements in an automated mode.

A swarm based system is similar in some respects to Storm [23], a distributed real time computation system. Storm reliably processes unbounded streams of data using a network of intelligent nodes, called *topology*. A Storm topology represents a complex multi-stage stream computation which runs indefinitely. A swarm system is doing real time processing for a flux of requests using *swarm descriptions*. Swarm descriptions are used to launch *swarms* that are visiting distributed nodes to perform computation. A swarm description can be seen as a "program" describing *swarm processes* that are executed when a request is made. Swarms can be integrated with existing queueing technologies, database technologies, or web services. A key difference between Storm topologies and swarm descriptions is that a swarm will always finish and will not generate more communication and computation, whereas a Storm topology processes messages forever or until it is killed.

To enable swarm communication we propose a description language, which depicts the flow of messages that occurs between nodes in an implicit and dynamic way. This language enables an intuitive description of communication patterns, based on intuitive primitives inspired from nature.

We define a swarm as a set of related messages with some basic intelligence. The functioning of a swarm system is different from the classical approach, in which "smart nodes" communicate with "dumb messages", and is based on an intuitive point of view: computer processes communicating by asynchronous messages are more like "dumb trees/flowers" visited by "smart swarms of bees". Reversing the perspective on communication through asynchronous messages is a novel approach compared to other solutions. The fact that the distributed nodes do not "communicate", instead they are visited by "families of intelligent messages" is very intuitive and has a powerful effect on managing complexity.

III. Swarming

In this section we present our communication pattern proposal. The swarm concept is similar to the swarming phenomenon found in nature, where a set of simple entities are

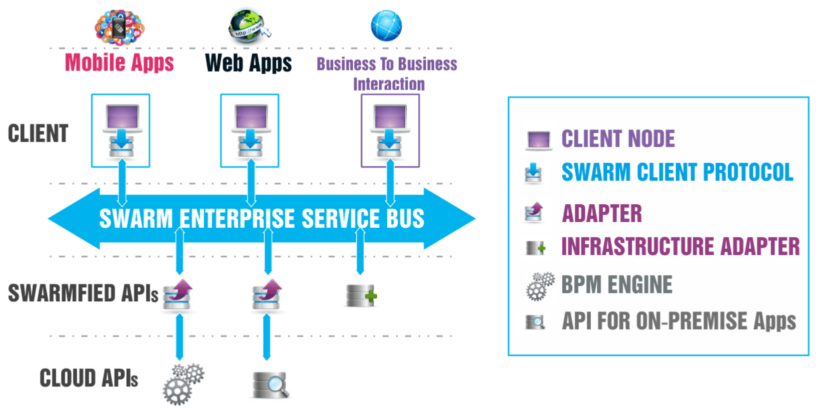

Fig. 1. Generic Swarm Architecture

collaborating and following simple rules are able to achieve a complex behaviour.

Swarm brings a new viewpoint on two key aspects of SOA: message exchange patterns and service integration patterns [5], [6]. Swarms keep the benefits of asynchronicity for message communication but also they decrease the complexity and the associated costs.

In practice we met many communication patterns (e.g. request/reply, request/reaction etc. [5]), but swarms bring something new: when a use-case appears, all related messages can be automatically grouped in a simple representation. This is a significant advantage to code maintenance and offers support for developer intuition regarding messages flow. When we mention ”use-case”, we envision that a communication use-case represents a set of complementary messages sent to solve a specific task. This set of complementary messages forms a swarm. In a distributed environment many communication cases appears and in our context we refer at all as cases of *swarming*.

The communication is performed between nodes of the distributed system. In our environment a *node* is a software entity that can receive or send swarm messages. From SOA perspective, a node includes an endpoint functionalities having a unique address, a specific contract and of course containing services, which provide specific functionalities [5]. Our concept of nodes is nesting all these aspects, making it easier for developers to mentally handle them.

In the swarm communication model, each node has a unique identifier that we usually refer as name of the node (*nodeName*). We distinguish between two types of nodes (see Figure 1): adapter nodes and client nodes. Adapter nodes are server side nodes that provide some services or APIs for existing applications (e.g. CRM, ERP etc.). The nodes that offer system core functionalities are called infrastructure adapters. Client nodes are logically connected to an adapter by communication protocols created over TCP sockets or other protocols.

For *nodeName* will have three possible values:

1) *wellKnownNode* - represented by infrastructure nodes;
2) *groups* - to enable different grouping of nodes we introduced the concept of *groups*. Each adapter can join to a number of groups by using the *join()* primitive. By joining to a group a node becomes accessible to nodes that know the group name but are not aware of its name. As notation, we set that group names will always start with character ”@”. Therefore, nodes can be grouped in *fleets* or groups of nodes with similar functionality or by geographic location or other criteria. Sending a swarm to a group with *swarm()* primitive means that one node is chosen and the execution will continue there. If the chosen node is down, another node is tried until the swarm succeeded to finish that phase;
3) *innerNodeName* - represents the name for inner nodes. Inner nodes are using resources from a normal node but they don’t have any visibility outside of that node. Sending a swarm to an inner node doesn’t produce network traffic. Sending a swarm to an inner node is exactly like sending the swarm to current node, and this feature can be used during development for comparing execution times between remote and local communication or when planning what services (adapters) are required.

The communication mechanism is accomplished through the nodes ”talk”: a node is talking to another node by sending a swarm. When we talk by a node sending a swarm we mean that it sends a message that is part of the set of swarm messages. Returning to our metaphor inspired by nature, the ”talk” is made not by words but with ”small beings” that have intelligence and take decisions instead of being just a method for information representation. In other words, in time, a swarm goes through a set of phases to accomplish a goal. Such phases contains applications code that change the internal state of the swarm or the state of the adapters in which the

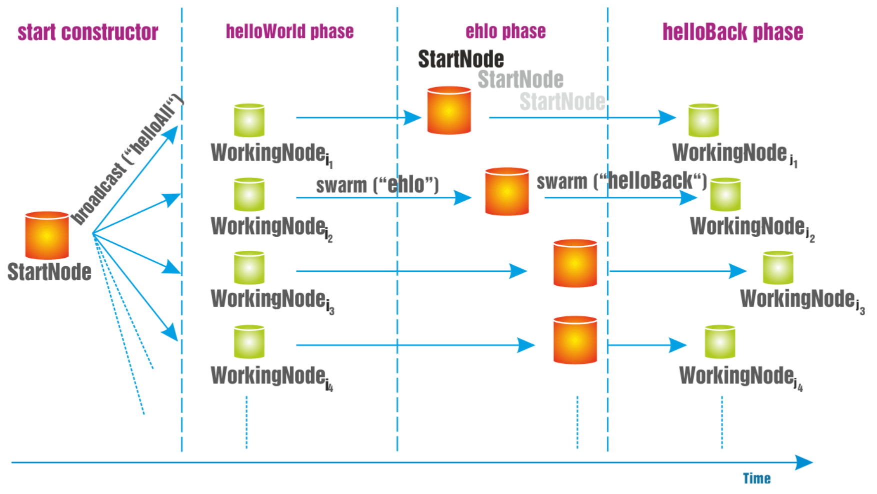

Fig. 2. HelloWorld swarm execution in time

swarm got executed. Change of nodes state is made by calls of adapter's APIs that we call *swarmified APIs* to differentiate from external APIs (see Figure 1).

It is important to notice that the phase's code is not part of the adapter, but code of the swarm itself, even if it is executed in the context of the adapter node. Except that having a name (phase name) and a node hint, a phase is the behaviour (code) that should happen (execute) when a message is received by a node (adapter).

For a better understanding, we will present a HelloWorld example (see Figure 2). We assume that we already started an arbitrary number of nodes with name *WorkingNode* and the following swarm is launched in the system:

```
vars:{
  message:"Hello World",
},

/* A constructor can be executed in any
   adapter node
   Below we obtain a unique identifier of
   the current adapter */
start:function(){
  this.startNode = thisAdapter.getUUID();
  this.broadcast("helloAll");
},

/* Phase that gets executed in all
   "WorkingNode" adapters nodes that
   are available */
helloAll:{
  node:"WorkingNode",
  code : function (){
    /* send the swarm back in the start
       node*/
    this.swarm("ehlo", this.startNode);
  }
},

/* ehlo phase executed in the node that
   started the swarm */
ehlo:{
  node:"*",  /* "*" is not a valid node
  name but it means that the second
  parameter of swarm and broadcast
  primitives should be a valid name and
  the node that execute this phase is
  explicitly declared when the swarm is
  swarming*/
  code : function (){
    /* each time the ehlo phase will send
       the swarm in another node */
    this.swarm("helloBack");
  }
},

/* phase executed in all "WorkingNode"
   nodes */
helloBack:{
  node:"WorkingNode",
  code : function (){
    /* use the console.log function
       (swarmified API) */
    console.log(this.message +
                "in node:"" +
                thisAdapter.getUUID());
    }
}
```

In implementation, a swarm description is a source code file written in a subset of JavaScript language. A swarm description is for a swarm what a class is for objects (class instances). The name of this file will give a common characteristics to all instances created on it and we refer at this characteristics as the type of the swarm.

We can see from our example that swarm description can contain four construction types: a swarm section called *vars*, *swarm phases*, *swarm meta variables* and *swarm functions*. The vars section in a swarm description is a manner through we document and initialize variables used by the swarm. For our example we initialize the variable *message:"Hello World"*. Because of the dynamic nature of JavaScript, you don't have to declare variables, but if you do, they will get initialized in all swarms instances. At runtime, a swarm is a regular JavaScript object and you can manually add or delete fields from this object, at any time. These fields at runtime correspond to swarm variables. The vars section is used as a template for creating these regular JavaScript objects. Only serializable types can be fields in a swarm object (e.g. no file descriptions or sockets). Only one vars section is allowed in a swarm description, but any number of variables can be initialized there. The vars section is optional but can be useful to initialize swarm variables for all constructors or for documentation purpose. Swarm meta variables are special variables that can be handled by programmers to influence or control the execution of the swarm in various ways. In our implementation we have the following meta variables: *swarmingType* – represents the swarm type; *currentPhase* – represents the name of current phase that is currently executed; *sessionId* – is the identifier of the current session. A sessionId is a token assigned to uniquely identify an active client in the system, and it can be used for actions as checking permission etc.

A swarm phase is another part of a swarm description. A swarm phase contains two parts: an optional indication of the node where the swarm should be executed and a required part represented by code (function). We identify two types of functions (both declared with an identical syntax): *normal functions* and *ctor functions* (similar to constructors in OOP languages). Ctor functions or ctors are functions that are called when a swarm is started. A ctor function will initialize the swarm variables and will start swarming in one or multiple phases. Ctors and functions have names visible only in the current swarm code.

In the code for *phases*, *functions* and *ctors* are available for programmers in this reference.

We will describe just those primitives that we have used in our use-cases:

- *swarm (phase name, [adapter—group name])* is a key primitive for understanding our communication model. We can view the swarm call as a fork of current swarm. A child swarm is born, but because a swarm is a collective entity, not an individual entity, we can consider this new child swarm as part of the current swarm. The state of current execution of the swarm, at the moment of the call, is serialized and a message containing this serialisation is sent to another node requesting execution of a specific phase to take place there. The node where this new child swarm will be born is decided by looking at the phase declaration or at second parameter. The parent swarm entity can continue to call the swarm primitive as many times as is necessary or eventually it will end the execution. If the second argument identify a group or multiple nodes with that name are concurrently in existence, one node is chosen and the child swarm is sent there. If the execution fails, another node is chosen (Round Robin strategy);
- *broadcast(phase name, group name)* - a broadcast primitive send the swarm towards all members of a group. There is no guarantee that all new nodes that recently joined the group are receiving the swarm.

Other swarms not exemplified here is *startSwarm(swarm type, ctor, args)* An entirely new swarm is created by calling *startSwarm()* primitive. The ctor will be called with the given arguments.

For our HelloWorld example, use of indices for WorkingNode in Figure 2 is suggesting that is not possible to predict the order of execution but all nodes are pass the *helloAll* and respectively *print* phase only once. The execution is starting in *start* constructor that sends the swarm using the broadcast primitive towards all nodes with the well known name *WorkingNode*. Each node will receive the swarm and execute *helloAll* phase that is just sending a swarm back to the start node in phase *ehlo*. In the *ehlo*, the launching node is sending the swarm in *print* phase that should be execute in only one node named *WorkingNode* (remember there we have launched multiple nodes with the name *WorkingNode*). Because swarm primitive is choosing another node each time, we can almost guarantee that each node will print *Hello world* message on its console. Alternatively to this implementation, broadcast can send messages to a group by modifying the use of *WorkingNode* to something like *@WorkingNodes* and joining all the nodes in *@WorkingNodes* group. From this example we omitted entirely the node implementation because the only swarmified API that this swarm is used from nodes is *console.log* and we can assume that every nodes have this function because of the Java Script hosting language.

IV. Dynamic Scalability with Swarms

Regarding scalability, the best practice is to architect for the worst load your application is likely to encounter. If this high load will happen during a specific event, then the system should be prepared for that event [24]. Cloud infrastructure like Amazon's EC2 can help by reducing the cost of expanding infrastructure when is needed, at request [25]. To use services as EC2, the system should be composed with stateless services that almost have no cohesion between them. On such systems one can double the nodes number with a very small degradation in performance per node added. This quality of being able to expand with almost no impact on the architecture is called *dynamic scalability*. Dynamic scalability is a form of horizontal scalability. The enterprise world has long dealt with the notion of horizontal scalability on scales so small as to be unrecognizable compared to today's cloud ready systems. One method for achieving this level of scalability is with messaging. The classic star topology that use a hub that intermediates messages processed by clients is a perfect example [26]. Dynamic scalability is achieved in these cases by merely adding new clients. Similar properties can be obtained within a swarm architecture. Swarms cannot ensure that services are stateless and have no cohesion between them. But what we can do is to ensure that communication primitives can be used to create dynamically scalable systems. The *swarm* primitive is inherently designed with scalability in mind, because it sends swarms using a Round Robin strategy. Round Robin is a mechanism used for load distribution, load balancing and fault-tolerance and is usually used for DNS balancing [27] but it is also appropriate for swarm based messaging. In this section we discuss an improved approach to overcome the limitations of the Round Robin method.

The swarm primitive alone may not be the best choice for load balancing on its own, since it merely alternates the order node names each time a swarm is sent. At the level of swarm primitive, there is no consideration for transaction time, server load or network congestion.

Load balancing with swarm primitive works best for swarms visiting nodes running on servers of equivalent capacity. Also, the running phases use similar amount of computational resources running in nodes located on servers of equivalent capacity. Thus it can ensure load distribution for all cases, but not load balancing in all possible cases.

Below we present a technique that can be extended to control the balancing when aspects like working time, server load, network congestion are considered. We start with a set of nodes having two types: *workers* and *balancers*. In the following, we depicted the prototype for worker's code:

```
/* standard node.js module loading , in
   our case "swarmutil" which contains
   the core implementation */
require('swarmutil').createAdapter(
                "Worker",
                onReadyCallback);

/* "WorkerManagement.js" swarm will be
   started after initialisation  */
function onReadyCallback(){
    startSwarm("WorkerManagement.js",
               "register",
               thisAdapter.getUIID());
}

/* performs a task that eventually takes
   a long time */
doWork = function(){
    ....
    return result;
}
```

At start, each worker is sending towards *Balancer* node a swarm, registering its availability for doing work. It is doing this using the *startSwarm* primitive. Balancers nodes prototype are exemplified in the following.

```
/* add in system a node with the well
   known name "Balancer" */
thisAdapter =
 require('swarmutil').createAdapter("Balancer");

/* initialise a global object containing
   algorithms and strategies for
   load balancing */
var theMagicChooser = new MagicChooser();

/* define chooseWorker functions for
   swarms visiting a "Balancer" node;
   swarmState is a reference to current
   swarm message containing the current
   values for all swarm variables and
   all swarm metadata */
chooseWorker =
 function(balacingStrategy, swarmState){
  if(balacingStrategy == undefined){
   balacingStrategy = "Round-Robin";
  }
  if(balacingStrategy == "Round-Robin"){
   return
     theMagicChooser.chooseRoundRobin();
  } else
  if(balacingStrategy == "random"){
   return
     theMagicChooser.chooseRandom();
  } else
  if(balacingStrategy == "enqueue"){
   /* save the state of the swarm for
     later dispatch return */
   theMagicChooser.enqueue(swarmState);
  }
  else{
   /* log an error or warning because
      an unknown balancing strategy
      was requested */
   logInfo("Unknown name" +
           balacingStrategy +
           ", choose default strategy.");
   return
     theMagicChooser.chooseRoundRobin();
    }
}
```

A balancer can be implemented in multiple ways. In our implementation we have a Balancer adapter containing two main functionalities: a global object called *theMagicChooser* and a function *chooseWorker*.

The object *theMagicChooser* provides a set of functions:

- *registerWorker(workerName)* – mark the node with name workerName as available for tasks. The worker identified by workerName will be added to the worker's pool.
- *status(workerName, workerStatus)* – inform theMagicChooser about the load status of a worker or about other useful informations for theMagicChooser to do its job. The swarm that is using the status function is beyond the scope of this article.
- *chooseRoundRoubin()* – use Round Robin strategy to choose the next worker
- *chooseRandom()* – choose randomly an worker from the pool of workers
- *startWork(nodeName)* – inform theMagicChooser that the worker with name nodeName started some work
- *endWork(nodeName)* – inform theMagicChooser that the worker with name nodeName finished his task
- *enqueue(swarm)* – choose a free worker (one that is not set busy by startWork) or relaunch the swarm given as parameter when a worker becomes free (by

calling endWork). The enqueue is returning false if no worker is known to be available otherwise it will return a worker name available for doing work.

The *WorkerManagement.js* swarm is the swarm used by workers to notify balancers of their availability and we briefly present below:

```
/* constructor "register" of
   "WorkerManagment.js" swarm */
register:function(workerName){
  /* remember the swarm variable
     "workerName" */
  this.workerName = workerName;
  /* send the swarm to Balancer in phase
     "doRegister" */
  this.swarm("doRegister");
},

// doRegister phase
doRegister:{
  node:"Balancer",
  code : function (){
    /* call registerWorker from
       theMagicChooser global object */
    theMagicChooser.registerWorker(
                       this.workerName);
  }
}
```

When work should be done in our system, this work should be requested by creating *WorkerRequests* swarms following the below template:

```
vars:{
  // default balancing strategy
  defaultBalancingStrategy:"Round-Robin"
},

// swarm constructor
requestWork:function(balacingStrategy) {
  if (balacingStrategy != undefined) {
    /* setting the default
       balancingStrategy */
    this.balacingStrategy =
        defaultBalancingStrategy;
  }
  // send the swarm in "Balancer" node
  this.swarm("doChooseWorker");
},

doChooseWorker:{
  node:"Balancer",
  code : function (){
    /* choose an available worker or put
       current swam in a queue if the
       current balancing strategy is
       "enqueue" and all nodes are busy.
       The swarms from this queue
       belonging to theMagicChooser will
       be restarted when an worker
       becomes available */
    this.selectedWorker = chooseWorker(
              this.balacingStrategy,
              this);

    /* can be false if balacingStrategy
       is "enqueue" */
    if (this.selectedWorker) {
    /* inform theMagicChooser that
       this.selectedWorker is busy */
    theMagicChooser.beginWork(
               this.selectedWorker);
    this.swarm("executeWork",
               this.selectedWorker);
    }
  }
},

executeWork:{
  node:"*",
  code : function (){
    /* call the doWork function from
       this Worker, e.g. operations as:
       database access, image
       processing etc. */
    this.result = doWork() ;
    this.swarm("taskDone");
  }
},

taskDone:{
  node:"Balancer"
  code : function (){
    /* inform theMagicChooser that
       this.selectedWorker is
       available */
    theMagicChooser.endWork(
               this.selectedWorker);
    /* send the result to requester (who
       started current swarm); the result
       of computation will be found in
       swarm variable "result" */
    this.home("done");
  }
}
```

WorkerRequest swarm has 3 phases with clear responsibilities:

- *doChooseWorker* – executed in Balancer
- *executeWork* – executed in a Worker
- *taskDone* – executed in Balancer after a task is completed

With this example we show the possibility of having control over scalability by creating a load balancer with multiple strategies. The *enqueue* strategy guarantees that only one task is requested for each worker and the requested tasks are added in a queue until a worker is available. *enqueue* strategy is useful to ensure that nothing wrong happens when the rate requests from tasks producers is higher and the rate of solving tasks by workers. In [3] this problem is resolved by using persistent message queues. We can also envision how the Balancer could request to other cloud service to start new workers when his queue is greater than a specific threshold, ensuring the dynamic scalability for a swarm based system.

V. Conclusions

This paper presents swarm communication, a new approach regarding communication and composability mechanisms for services in cloud systems. Our proposal brings a new perspective on two key aspects of SOA, message exchange patterns and service integration patterns. It represents a novel approach to using asynchronous messages and to composing services' behaviours. Swarming is different from normal message communication to some extents: a swarm based system is working with smart messages and less intelligent service nodes. Swarms are not just a set of related simple messages, they have a powerful mechanism for composing intelligent behaviours from simple behaviours. Each message has an associated code that is executed in the nodes that receive it. The entire communication in a swarm based system is based on an intuitive point of view, with the goal of reducing complexity: computer processes communicating by asynchronous messages are more like ”dumb trees/flowers” visited by ”smart swarms of bees” than ”smart nodes” communicating with ”dumb messages”.

In this paper we presented the architecture of a Swarm system that enables integration between heterogeneous components. To experiment with swarm communication we have developed an open source project, called SwarmESB [28], which is both an ESB and an early stage iPaaS. It is build using NodeJS [29] and Redis [30], solutions that proved to be very effective for developing such a system.

Although swarming represents a different perspective on how composability of services is done, having a distinct approach compared to the standard one taken by web service orchestration, it is re-using familiar techniques for many programmers.

Future research directions include testing the system's fault tolerance in the case of errors and implementing security mechanisms for messages.

References

[1] Q. Mahmoud, *Middleware for Communications*. Wiley, 2004.

[2] D. C. R. Curry and G. Lyons, “Extending message-oriented middleware using interception,” in *In Proc. of the 3rd Int. Workshop on Distributed Event-Based Systems*, 2004, pp. 32–37.

[3] D. Chappel, *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[4] D. Dirksen, *SOA Governance in Action: Rest and WS-* Architectures*. Manning Publication Co., 2012.

[5] A. Rotem-Gal-Oz, *SOA Patterns*. Manning Publication Co., 2012.

[6] T. Erl, *SOA Design Patterns*. Prentice Hall, 2009.

[7] (2013) Mule ESB. MuleSoft. [Online]. Available: http://www.mulesoft.org

[8] (2013) Apache ServiceMix. Apache Software Foundation. [Online]. Available: http://servicemix.apache.org

[9] (2013) JBoss SwitchYard. Apache Software Foundation. [Online]. Available: http://www.jboss.org/switchyard

[10] (2013) OpenESB. [Online]. Available: http://www.open-esb.net/

[11] (2013) TalendESB. Talend. [Online]. Available: http://www.talend.com/products/esb

[12] (2013) Java Business Integration. Java Community Process. [Online]. Available: http://jcp.org/en/jsr/detail?id=312

[13] (2013) Java Community Process Program. Java Community Process. [Online]. Available: http://www.jcp.org

[14] (2013) Web Services Business Process Execution Language (WSBPEL) TC. OASIS. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[15] (2013) OSGi Alliance Specifications. OSGi Alliance. [Online]. Available: http://www.osgi.org/Specifications/HomePage

[16] (2013) Service Component Architecture Specifications. OASIS. [Online]. Available: http://oasis-opencsa.org/sca

[17] (2013) Apache Camel. Apache Software Foundation. [Online]. Available: http://camel.apache.org

[18] (2013) Workday Integration Cloud Platform. WorkDay. [Online]. Available: http://www.workday.com/applications/integration_cloud/integration_cloud_platform.php

[19] (2013) snapLogic. snapLogic. [Online]. Available: http://www.snaplogic.com/

[20] (2013) Software AG Integration Live. Software AG. [Online]. Available: http://www.softwareag.com/live/integrationlive.asp

[21] (2013) Tieto iPaaS. Tieto. [Online]. Available: http://www.tieto.com/services/infrastructure-solutions-and-services/platform-service-paas/integration-platform-service

[22] (2013) Informatica iPaaS. Informatica Corporation. [Online]. Available: http://www.informaticacloud.com/products/integration-platform-as-a-service-ipaas.html

[23] (2013) Storm Distributed Realtime Computation System. [Online]. Available: http://storm-project.net

[24] J. B. R. Buyya and A. Goscinski, *Cloud Computing: Principles and Paradigms*. Wiley, 2011.

[25] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2007.

[26] V. Cardellini, M. Colajanni, and P. S. Yu, “Dynamic load balancing on web-server systems,” *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, May 1999. [Online]. Available: http://dx.doi.org/10.1109/4236.769420

[27] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2012.

[28] (2013) SwarmESB. [Online]. Available: https://github.com/salboaie/SwarmESB

[29] (2013) NodeJS. [Online]. Available: http://nodejs.org

[30] (2013) Redis. [Online]. Available: http://redis.io