ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

Technical Report: TR20130300 - The Science Cloud Case Study

Technical Description of Implementation

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **7.6.2010**

Lead contractor for deliverable: **LMU**
Author(s): **Philip Mayer (LMU), José Velasco (Zimory)**

Due date of deliverable: **March 21, 2013**
Actual submission date: **March 21th, 2013**
Revision: **V1**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

This document describes the technical implementation details of the Science Cloud Platform (SCP), which is a platform-as-a-service cloud providing the ability to execute applications in a robust manner while keeping to their SLAs.

The cloud consists of individual instances which may join or leave the cloud at will. Thus, the SCP is an exercise in volunteer computing and has no central coordinator. Individual instances of the cloud software work together to keep applications running, thus forming ensembles.

This document is a companion document to the general cloud overview given in [MKV12]. In this document, we discuss the cloud infrastructure in terms of layers; the main scenario in use, thoughs on application resilience, and the actual implementation of the cloud in terms of the p2p substrate Pastry.

Contents

1 Introduction

The idea of the science cloud is that of a peer-2-peer, voluntary computing based Platform-As-A-Service (PaaS) cloud. This means we want to

deploy and run user-defined applications on the connected web of machines which form the science cloud

With regard to the nodes in the cloud, we assume:

- Nodes may come and go with or without warning (voluntary computing / p2p)
- Node load may change based on outside criteria (voluntary computing)
- Nodes have vastly different hardware, which includes CPU speed, available memory and also additional hardware like specialized graphics processing etc. Also, a node may have different security levels (voluntary / p2p)

With regard to the applications, we assume that:

- An application has requirements on hardware, i.e. where it can and want to be run (CPU speed, available memory, other hardware) (cloud computing)
- An application is not a batch task. Rather, it has a user interface which is directly used by clients in a request-based fashion.

2 Cloud Infrastructure

The cloud infrastructure should provide the following features. We separate this description into three levels for clarity; this does not mean that the levels can be exchanged independently; in fact, they heavily depend on one another from an implementation side of view.

- *Network Level*: First of all, the nodes which form the science cloud need to know one another (at least partially), be able to route between themselves, and be stable under adverse conditions (i.e. nodes that are part of the science cloud leave, or new nodes are added). This means we need network resilience (self-healing), i.e. routing still needs to work under these conditions.
- *Data level*: When an app is deployed, the code needs (at least in principle) be available to all nodes which can possibly execute it; furthermore, application data needs to be stored in such a way that resuming an application, after a node which ran it failed, is possible. We thus need data storage with data redundancy, not only of immutable data (app code) but also of mutable data (app data).
- *Application level*: Finally, apps can only run on some machines (based on app requirements) so these must be found in the network and instructed to run an app (might be multiple nodes at once). The apps store their data on the data storage level. If apps need to coordinate, they can also do that via this level (i.e. a distributed database). If a node with a running instance goes down, it must be restarted (failover) on another, fitting node. We can call this application resilience. Furthermore, user requests to apps (request/response based, as in HTTP) must be routed from the requesting node (user node) to the app node (executing node).

3 Main Scenario

The main scenario of the science cloud is based on what the cloud is supposed to do, i.e. run, and continue running in the case of changing nodes and load, applications.

The document [MKV12] has listed three smaller scenarios which we combine here to a general scenario which describes how the cloud manages adaptation. On top of this basic scenario, other scenarios may be imagined which improve specific aspects such as how to distribute load based on particular kinds of data or how to improve response times.

The *basic cloud scenario* focuses on application resilience, load distribution and energy saving. In this scenario, we imagine apps being deployed in the cloud which need to be started on an appropriate node based on its SLA (requirements). The requirements may include things like CPU speed of the node to be run on, memory requirements, or similar things. Once the app is started, we can imagine that problems occur, such as that a node is no longer able to execute an app due to high load (in which case it must move the app somewhere else) or due to a complete node failure (in which case another node must realize this and take over). Also, a node may realize it is not used anymore and, if this ability is available due to the use of an IaaS solution, shut down. Finally, if an app is removed by a user, it must stop executing on the cloud.

4 Thoughts On Application Resilience

The first two levels listed above have been addressed in the literature. This is often done by employing a structured overlay network and a Distributed Hash Table (DHT), which comes in many variants. The fundamental idea is storing data in a key-value system. Each node and each piece of data has a key, and routing and storage takes place based on an idea of nearness between node key and data key. There is a certain structure to the nodes based on the keys (for example, a ring structure) which allows defining nearness, and which can be exploited for both routing and redundancy (for example, storing k redundant copies of some data in the k nodes with keys closest to the data key).

This structure is ideal for network and data resilience, but does not work for application resilience, since the nodes which store an application (based on key) might not be the ones able to execute it (due to multiple different hardware requirements, like CPU speed, memory, etc.). Thus, if the node (hardware) capacity is not in the overlay structure, it needs to be established on another layer on top of the existing nodes. This can be done by using a two-stage approach.

- *Stage 1*: for each app, one node is chosen as being responsible for its execution not necessarily executing the app itself, just being responsible that it is executed (Initiator). This node needs to be secure against failures, i.e. if it goes down another node needs to take its place. Using a DHT-like approach, this might be the node storing the app in which the app key is closest to the node key.

- *Stage 2*: this initiator node needs to find one or multiple nodes which can execute the app (Executor). This can be done using a voting- or bidding-like mechanism, for example the ContractNet (CNet) algorithm from the multi-agent domain. A communication channel is used to request bids for the app (based on app requirements); nodes which can execute the app reply with a bid. The initiator node selects the best node(s) for execution and continues watching these nodes (Observer).

The communication channel required for this approach can following a distributed publish/subscribe mechanism. Note that a communication channel does not mean global awareness. With a proper rout-

ing system in place, nodes still only need to know a subset of the closest nodes in the cloud. Still, this approach depends very much on how quickly the communication is re-established after node failures.

5 Science Cloud Implementation

The science cloud platform does not implement the first two levels discussed above (network and data), but reuses existing work from the literature, namely Pastry [RD01b] (for the structured overlay) and PAST [RD01a] (for the DHT). For both, an implementation exists [PD13]. In the case of PAST, we utilize gcPAST, which is a version of PAST which allows changes to already stored data. Tombstones are used for deleting data.

The third level, i.e. application resilience, follows the ContractNet idea for bidding laid out above [Fou13]. It uses another Pastry component for communication: The distributed group communication system SCRIBE [CDKR02] which uses the publish/subscribe mechanism. An implementation for scribe is available too [PD13].

Below, we sketch the algorithm for application failover. It is a variant of the ContractNET (CNet) algorithm used in multi-agent systems. This algorithm implements a bidding system: An initiator sends a request for bids (to perform some task); interested entities respond by bidding for the task; and the initiator selects one of them to perform the task.

In our case, we use continued application execution instead of tasks, thus we need to address observation as well; furthermore, we need to address the issue that the initiator itself may fail during observation. The following diagram shows the behavior of each node. For each application, we separate this by three states or modes:

- The basic node behavior (which is running on every node, always active)
- The initiator/observer mode (a node may enter this mode for one or multiple applications; basic behavior is still active)
- The execution mode (a node may enter this mode for one or multiple applications; basic behavior is still active).

It is important to note that a node will ALWAYS show the basic node behavior, and will additionally, optionally, also show multiple initiator/observer and execution modes depending on how many apps are initiated and executed.

We discuss each of the modes in turn. In the following figures, a question mark denotes an incoming network event (request by another node). An exclamation mark at the end of a line denotes an active operation (network send). A minus (-) denotes an internal action without network activity.

5.1 Basic Behaviour

The basic behavior is apparent on any node in the system. It may run in parallel to the other modes (which are for one app each).

The behavior is shown in Figure 1. Lets discuss those in clockwise order.

- If we get a request for bids i.e. some initiator is asking for nodes to execute an app we decide whether we want to bid based on our abilities and load. The bidding request will include the list of app requirements (CPU speed, memory,) which we can check against ourselves. If so, send a bid.

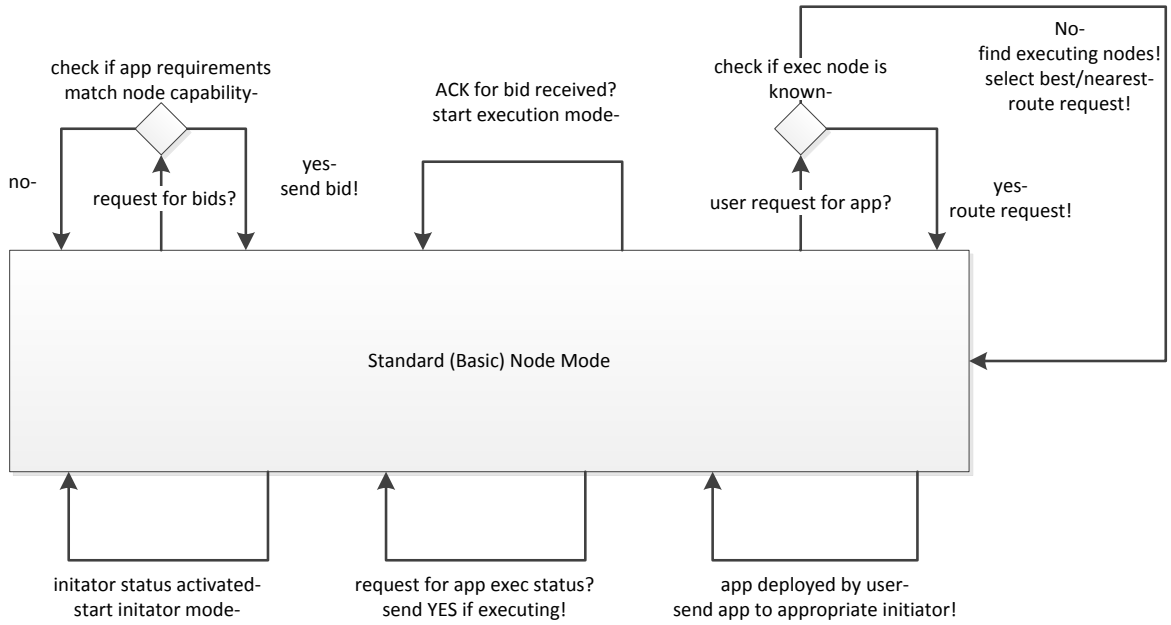

Figure 1: SCP Node App Basic Mode

- If we have sent a bid before and we now get an ACK, we can start executing the app i.e. one instance of the execution mode is started for this app in parallel.
- If our user (i.e. the user on the machine we are running) sends a request to an app in the cloud, we check whether we already know who executes this app (and/or have a session stored). If so, directly route the request. Else, find the executing nodes (via the initiator), select the best one for us, and then route the request.
- If an app is deployed by our own user, deploy it on the DHT (i.e. send to initiator)
- If we get a request asking us whether we execute an app (see observer mode below), we answer it truthfully
- If we notice that we should be initiator for an app (based on hash key nearness), we start an instance of the initiator/observer mode for this app in parallel.

5.2 Executing Mode

Executing mode is for one app at a time, though multiple apps may be running. The behavior is shown in Figure 2. Again, in clockwise order:

- We start this mode when we are made executor (by receiving an ACK for a bid)
- Every n seconds, we check that we still fulfill the requirements of this app, and that we are not overloaded. If we detect a problem, we inform the initiator and shut down.
- If we receive a user request, we let the running app handle it and return a response.

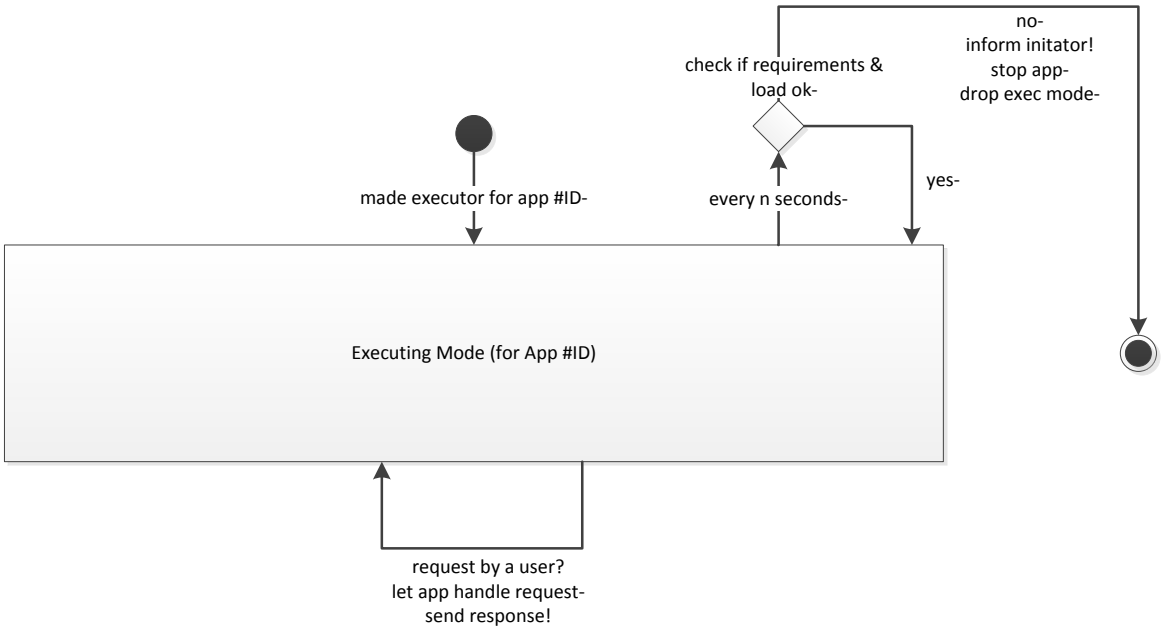

Figure 2: SCP Node App Executing Mode

5.3 Initiator/Observer Mode

Initiator/Observer mode is for one app at a time, though we might be responsible for multiple apps. The behavior is shown in Figure 3.

- We are made initiator for an app by detecting that we are the node with the closest key to the app key. This might happen on the first insert of the app or after neighboring nodes have failed.
- Every n seconds, we check that the app is running in the cloud. This is done by sending a request for app exec status via SCRIBE. If we do not receive a positive response, we send a request for bids (along with the app requirements) via SCRIBE, and wait for a determined amount of time
- If this amount of time (the bidding window) has elapsed, we select the best node(s) for execution. We send those nodes an ACK; all the others receive a NACK
- If we lose initiator status (for example, if the network is restructured) we stop being initiator.

6 Conclusion

This document has described the technical aspects of the science cloud case study of the ASCENS project. The aim of the case study is to provide a proving ground for ASCENS languages, methods, and tools in the cloud context.

The scenario has described the basic implementation on top of the p2p substrate Pastry and accompanying frameworks.

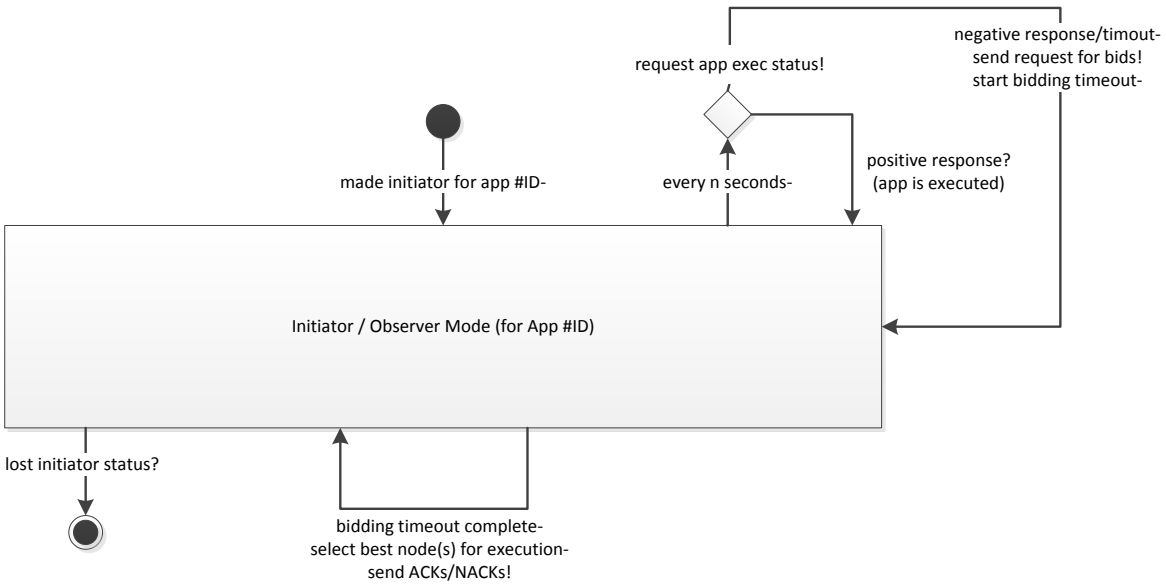

Figure 3: SCP Node App Initiator Mode

References

[CDKR02] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[Fou13] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. http://www.fipa.org/specs/fipa00029/SC00029H.html, March 2013.

[MKV12] Philip Mayer, Christian Kropiss, and José Velasco. Technical report tr20129500 - the science cloud case study - overview and scenarios. Technical report, Ludwig-Maximilians-Universität München, 2012.

[PD13] Jeff Hoye Sitaram Iyer Alan Mislove Animesh Nandi Ansley Post Atul Singh Miguel Castro Manuel Costa Anne-Marie Kermarrec Antony Rowstron Sitaram Iyer Dan Wallach Y. Charlie Hu Mike Jones Marvin Theimer Alex Wolman Ratul Mahajan Peter Druschel, Andreas Haeberlen. FreePastry. http://www.freepastry.org/, March 2013.

[RD01a] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 188–201. ACM, 2001.

[RD01b] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.