Delivering High Resilience in Designing Platform-as-a-Service Clouds

Qianhui Liang, Bu-Sung Lee

HP Laboratories
HPL-2011-75

Keyword(s):

cloud computing; component; constraint; data flow; dependency graph; PaaS; resilience; time stamp

Abstract:

Platform-as-a-Service (PaaS) clouds allow faster and more effective application development than traditional non-PaaS ways. One issue in designing PaaSs is how to make the development process deliver applications resilient to potential changes of the constraints. This is because any successful applications today must be as resilient as possible to dynamic external or internal constraining factors. Along this line, the first type of dynamic constraints we need to consider is the compatibility between possible components of the application. PaaSs must only engage compatible components to collaborate with each other in the same instance of applications. Other constraints include the environment that the application is running as well as the preferences of the users (or devices) that interact with the application. We present a data-flow based approach, for PaaS clouds, to designing cloud-based applications that are resilient to failures due to dynamic constraints on resources and on component compatibility. The uniqueness of our approach is the following: The procedure of building cloud-based applications is time-stamped. In this way, the composition of the application is updated anytime in accordance to the constraints in order to maximize the resilience of the application at that time. We have designed a graph structure called Instance Dependency Graphs (IDGs), and have used time-based IDGs to capture, analyze and optimize the resilience of the application. We present a case study to validate our approach.

External Posting Date: June 6, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: June 6, 2011 [Fulltext]
To be published and presented at IEEE Cloud 2011 July 4-9, 2011.

Delivering High Resilience in Designing Platform-as-a-Service Clouds

Qianhui Liang[1], Bu-Sung Lee[1,2]

[1]Cloud & Security Lab, HP Labs, Singapore
{qianhui.liang, francis.lee}@hp.com

[2]School of Computer Engineering,
Nanyang Technological University, Singapore
ebslee@ntu.edu.sg

Abstract:
Platform-as-a-Service (PaaS) clouds allow faster and more effective application development than traditional non-PaaS ways. One issue in designing PaaSs is how to make the development process deliver applications resilient to potential changes of the constraints. This is because any successful applications today must be as resilient as possible to dynamic external or internal constraining factors. Along this line, the first type of dynamic constraints we need to consider is the compatibility between possible components of the application. PaaSs must only engage compatible components to collaborate with each other in the same instance of applications. Other constraints include the environment that the application is running as well as the preferences of the users (or devices) that interact with the application. We present a data-flow based approach, for PaaS clouds, to designing cloud-based applications that are resilient to failures due to dynamic constraints on resources and on component compatibility. The uniqueness of our approach is the following: The procedure of building cloud-based applications is time-stamped. In this way, the composition of the application is updated anytime in accordance to the constraints in order to maximize the resilience of the application at that time. We have designed a graph structure called Instance Dependency Graphs (IDGs), and have used time-based IDGs to capture, analyze and optimize the resilience of the application. We present a case study to validate our approach.

Keywords- cloud computing; component; constraint; data flow; dependency graph; PaaS; resilience; time stamp

I. INTRODUCTION

One challenge that today's applications face is the dynamic constraints posted by external or internal factors of the applications. Here we generally define *constraints* as any imposed conditions that must be met for applications to function properly. (Constraints may include requirements, available computational resources, and users' profiles.) Constraints limit the performance and behavior of the applications and even determine if they are useful at all. Furthermore, the constraints are usually from complex sources and thus entail complex changes. The dynamism of the constraints is aggravated in clouds. Constraints may be on external or internal factors. Here, by external factors, we mean any elements that are not part of the applications (and its users) themselves. Internal factors are the elements that are part of the applications themselves.

Platform-as-a-Service (PaaS) clouds allow faster and more effective application development over the clouds by using available granular software components. To make the development process deliver applications resilient to potential changes on the constraints is a highly desired feature of such platforms. Resilient applications must deal with possible dynamic external or internal constraining factors. One type of such constraints is the compatibility between possible components of the application. PaaSs must only engage compatible components to collaborate with each other in the same instance of applications. Other types of constraints may regard the virtual machine or the device that the application is running as well as the preferences of the users (or devices) that interact with the application.

In designing a useful PaaS, a "handler" of potentially dynamic constraints needs to be built into applications by the platform that is used to develop them. Prior work on software integration, either in a loosely-coupled way or a tightly-coupled way, does not address this issue, as the first priority task to be solved then is the integration logic itself. QoS-based dynamic composition of (Web) services, for example, allows dynamic selection of components. However, in their solutions, optimization on composite services is designed to cater to the estimable (and semi-fixed) internal factors of the applications, such as the control flow, and more from the point view of a workflow engine. It does not provide the platform-layer support that is needed for the ever changing constraints such as those on the data flows of the application resulted from the compatibility of its components, which may not be estimable given the open-ended and uncontrolled sources of components in real world.

We claim that for PaaSs, we must design an approach to building applications that makes dynamic adjustments on the applications according to the dynamic constraints on their components. We refer to this capability *building applications resiliently*. The objective is to maximize the quality of the individual application and to optimize its usage under certain constraints at a particular time for its users. The potential promising impact of enabling such resilience in building applications is the following: Allow applications to be server-side and client-side running environment independent; Allow applications to be adaptive to data or metadata associated with the software applications; Ensure compatibility of component instances; Allow applications to be independent of platform-nativeness.

We study what kind of features is needed in PaaSs in order to help build applications that are resilient to

dynamic constraints. Our research results contribute to the issue of resilience in building applications in the following way:

- *To capture the execution progress of applications not only at the individual component level but also within the individual components*. Optimization of building cloud-based applications should not stay at the level of individual components as autonomous entities. In contrast to this, we provide a finer granular control of component instances that follows changes internal to the execution window of each component to be captured and accommodated. On one hand, we keep the promise that functionalities of components in a cloud-based application are provided as virtualized resources to its users and that users thus do not have to care about the implementation and platform of the components. More importantly, to achieve this transparency, the underneath PaaS keeps track of the problems (potentially) occur within the interval of component execution. In this way, the problems during execution are taken care of by the component composition function of the PaaS.
- *To understand the execution progress of applications in terms of changes on constraining factors.* The challenges of understanding the changes lie in finding a model that encompasses all possible relevant factors. We ensure that the composition function provided by the PaaS is based on a model that is aware of the changing constraints during the execution of components. It is also able to pay attention to all the various factors that cause the changes.
- *To formalize the resilience accounting for all constraining factors.* In this research, we have developed a technique of analyzing, calculating and comparing resilience of different optional components. The measures of resilience are calculated at the very beginning of the whole process. Then at various points in time, if any changes on any constraining factor in consideration have been detected, the measures of resilience shall be recalculated. We analyze the necessary changes on the application itself; switch to a different implementation of the component or different quality of the same component according to analysis results. This can happen at any time during the execution of a composite component.

In this paper, we deal with the requirement of resilience in designing PaaS clouds, i.e. to support building applications that takes into consideration of dynamic changes on constraints. We present our approach to meet this requirement. Especially, we pay attention to the compatibility constraint between interacting components and have designed our approach with a "linkage" mechanism embedded to handle the compatibility. To make the problem more explicit and clearly targeted, we narrow the resilience objective and only focus on resilience to component failures under dynamic constraints. However, this study is general and is applicable to other resilience objectives. The uniqueness of our approach is the following: The procedure of building cloud-based applications is time-stamped. In this way, the plan of the application is updated anytime in accordance to the constraints in order to maximize the resilience of the application at that time. We have designed a graph structure called Instance Dependency Graph (IDG), and have used time-based IDGs to capture, analyze and optimize the resilience of the component-based application. The "linkage" mechanism only allows compatible components to connect to each other thus ensuring the compatibility constraint is met. We present a case study to validate our approach.

The rest of the paper is organized as the following. In section 2, we give a running example to give an idea of the requirements that PaaSs have to meet. This running example will be used throughout the paper. In section 3, we first provide an overview of the problem of resilience in building cloud-based applications. Then we provide our solution. We describe the detailed model, i.e. Instance Dependence graph (IDG) and time-based IDGs as a basis of our solution in section 4. IDG is a data structure that allows capturing the dynamic execution progress of the cloud-based application and that can be used to analyze the changes and implement the changes. In section 5, we discuss problem quantification using time-based IDGs and present the solution calculation using the quantification. Section 6 is the related work.

II. RUNNING EXAMPLE

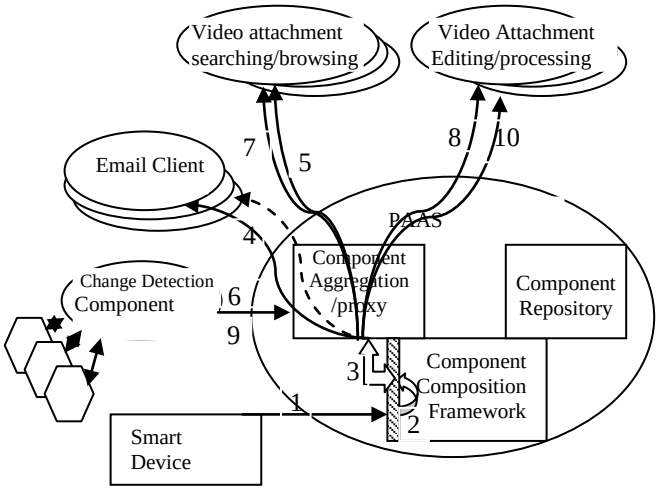

Fig. 1: PaaS and cloud-based application

In this section, we provide a brief description of a possible PaaS and how to build an example cloud-based application in the PaaS. The PaaS is composed of Component Composition Framework (CCF), which is responsible for creating an executable workflow instance of the application, Component Repository (CR), which is a repository of all registered components, and Component Aggregation (CA), which is the brokering service that enables categorizing and browsing the available components. The target application is to receive a user's request in browsing his/her emails and video attachments

(Video Attachment Searching and Browsing), and in editing selected video attachments (Video Attachment Editing and Processing). Upon getting triggered, the application receives the mobile user's interaction in the form of a request to use Email Client component, as label 1 in the figure. The PaaS CCF, through some internal processing, composes and selects the first few component instances according to the selection results based on the current constraint, shown as label 2 in the figure. In this case, component instances for all three functions are selected and planned for the application, which we denote as s_1_1, s_2_1 and s_3_1. (In the above notation, s_i_j denotes jth component instance of the ith component.) Then the CCF communicates with the CA as label 3 and invokes s_1_1 (an instance of the email client component) and successfully completes s_1_1, shown as label 4 in the figure. Here we differentiate component and component instance. An easy-to-understand analogy is class and class instantiation. In the rest of the text, we use component instance and instance interchangeably.

When the CCF goes onto the execution of s_2_1, it starts to load video frames for the user to watch, shown as label 5. The first attachment is a football video clip, which requires a high frame rate. After that, the framework starts to load the second clip which shows a museum exhibition. This does not require a high frame rate but instead a very high resolution of each frame. At this point in time, the CCF is notified of this change, shown as label 6, and reselects and invokes the component instance s_2_2, as in label 7.

Upon the completion of invocation of s_2_2, the framework continues to invoke s_3_1 to allow editing of the video, as label 8. During editing, the user stepped out of his car to enter his office building. His device starts to give low-battery warning, which is causing possible network connection problem with his cellular service provider. Meanwhile, the CCF is notified of this change, shown as label 9. The editing component needs to be switched from the original planned component instance (s_3_1) provided by his native mobile platform to the Web-based mobile-friendly component instance provided by his company (s_3_2), shown as label 10. The entire application ends as the successful completion of s_3_2.

III. Problem Overview and Formalization

As mentioned in the introduction section, the purpose of this research is to allow PaaS clouds to build into the application the resilience of changes on the constraining factors. The resilience of the application is a guarantee that the quality of the application remains high despite of the dynamic constraints.

We first give a definition of resilience of applications. In the presence of dynamic constraints, we define the capability of an application being resilient to possible execution faults as the following.

Definition 1 Resilience of Applications:
Resilience of a cloud-based application is its potential to accommodate dynamic changes on its constraining factors in order to maintain low failure rate by dynamically changing affected component instances accordingly.

In the above definition, we can see the measure of resilience is on how well the cloud-based application is made to respond to the changes on the constraining factors. The responding changes that the PaaS initiates on the cloud-based application are only to maintain a low failure rate. Such changes are realized by the changes of individual instances of some components.

To explain this idea, we can refer to the running example. Before the cloud-based application starts, three instances, i.e. s_1_1, s_2_1 and s_3_1, are pre-selected to provide the required functions of the application. Now let us assume this selection is fixed. When the changes on environment and on the content of the application occur, the cloud-based application cannot change accordingly and may very easily fail. In this case, this cloud-based application is considered not resilient. On the contrary, reselection can be allowed to take place as described in the running example. As a result, cloud-based application can change accordingly when the changes on the constraining factor occurs, thus making itself less failure-prone. Such a cloud-based application has good resilience.

Given the above definition, we can further derive a definition of the exact problem we shall be solving.

Definition 2 Resilience in Application Building for PaaS Clouds :
The problem of resilient application building by PaaS clouds is defined as finding incremental steps of component selection at various points in time for building the applications, such that at each step, one or more component instances that implement the required function(s) of the application and that (heuristically) allow(s) the highest resilience of the application at that point are (re)selected.

The above definition frames the problem of resilient application building as an incremental procedure of selecting suitable components for execution now or in the near future. The result of each step is also revocable. Some instances that have been selected to be executed may be replaced during the next iteration of reselection because of a new change on the constraining factors. The result of consecutive steps may overlap with each other. For example, when the nature of the input of the cloud-based application changes, a few consecutive selections may be triggered in order to (re)select different component instances for the same task. In this case, the same function will have multiple component instances mapped to it at various times, and each component provides better resilience under the content during a particular time frame.

If we take the example given in the previous section, resilient application building must compose the cloud-based email application by incrementally selecting component instances to allow the highest resilience. The

assignments of component instances to tasks of the application being built can be written as {task1:S1_1, task2:S2_1, task3:S3_1}, {task1:S1_1, task2:S2_1, task2:S2_2, task3:S3_1}, {task1:S1_1, task2:S2_1, task2:S2_2, task3:S3_1, task3: S3_2}, where each list (in a pair of braces) gives a composition of certain component instances during a particular time frame. We see that the composition gradually evolves and, thus qualifies a resilient application building.

The above problem definition emphasizes the aspect that some resource constraints as well as requirements from the users will only be gradually known nearer to the point when the component is to be executed. For example, context information that determines the resource constraints is only available when the sensors detect certain changes in the environment (due to the activity of users) and then derive the new constraints. This is also true the change is only known when a different type of data/content of the application is loaded by the component.

What naturally follows the current way of framing the problem is to delay the selection of the components. In particular, this is achieved by the following distinguishing mechanism: Each individual component of the cloud-based application is selected and/or reselected dynamically nearer to the point when (new) changes that trigger the selection are revealed.

IV. A Graphical Model for Resilient Application Building: IDG

Based on the discussions in the previous sections, we conclude it is important to include "change information" in any model we decide to use for building applications resiliently. Applications can be studied by a control-flow based approach or data-flow based approach. Here we focus on data-flows, as our research strategy is based on dynamic component instance (re)selections which are mostly affected and constrained by the quality and interoperability between collaborating components. Therefore, we propose to model applications using the data dependency relationships between components and the relevant constraining factors of individual component. This way, we are able to incorporate information about the changes that may happen over time.

To do this, we need to provide a basic model for showing the relationship of components within an application. For this purpose, we design IDG, or instance dependency graph. IDG is a direct acyclic graph that describes the possible input-output dependency relationships among the instances of the building blocks of applications. Each node in IDG is a component instance. If an edge is present from one node to another, data may possibly flow from the source component instance to the destination component instance. IDGs also conform to the compatibility constraints among interacting component instances. In other words, only compatible component instances are linked by the graph. Fig. 2 and Fig. 3 give two examples of IDGs for the running example.

IDGs can be built in two steps. First, we build a graphical model to describe the data dependency of components in a service-based application. In this step, we do not worry about the compatibility issue among the component instances. Each node in this graph is a component instead of a component instance. If an edge is present between two nodes, data may flow from the source to the destination. This graph is very similar to is service dependency graph (SDG) [15].

The difference is that in SDG, there are two types of nodes: service operation nodes and data entity nodes. Service operation nodes model abstract operations of components and data entity nodes model their input and output attributes. Edges in an SDG link the nodes of the input attributes to the nodes of the corresponding service operation or link a node of a service operation to the corresponding nodes of its output attributes. Fig. 4 shows the SDG for the running example. In this case, we only have nodes denoting components and edges denoting data flows between the components. Fig. 5 shows the graph in this case as a comparison to Fig. 4.

In the second step, we instantiate the component nodes and replace them with the corresponding component instances. It is not necessarily the case that any component that produces the data required by a consumer component can connect to it. In practice, it may not be the case due to interoperability issues between two components. A common problem on the components is the heterogeneity of their data formats. In the running example, attachment browsing component S_2_3 may not be able to open the attachment in the format output by email client component S_1_3 or S_2_3 needs an actual file and S_1_3 outputs a URL link or file descriptor of a file. Other problems like government regulations and competitor conflicts may also prevent one component working with another. This complicates the dependency relationship of components and demands employing IDG to solve the problem of resilient component composition.

Here we enforce a linkage mechanism to guarantee that the compatibility constraint is met. The linkage mechanism is the following: An edge in IDG indicates that two components conform to the dependency relationship defined and two components do not have an incompatibility issue and can work together. We notice the difference between Fig.2 and Fig. 3 is the presence of edge e(s_2_1,s_3_2). This edge is present in Fig. 2 but not in Fig. 3. Fig. 2 represents the case with no incompatibility between s_2_1 and s_3_2 and Fig. 3 represents the case with incompatibility between s_2_1 and s_3_2.

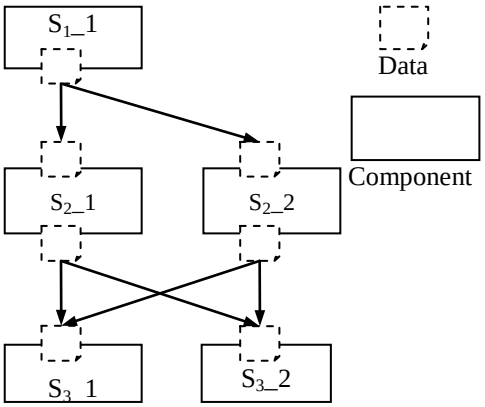

Fig. 2. IDG (a) for the running example

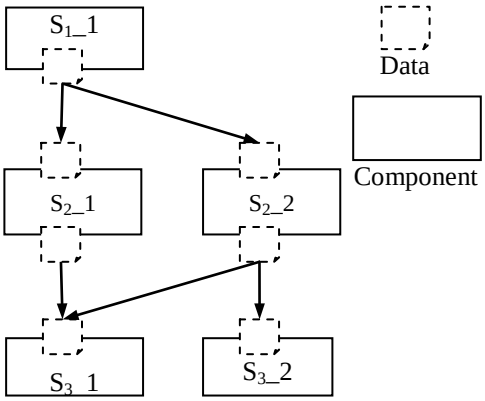

Fig. 3. IDG (b) for the running example

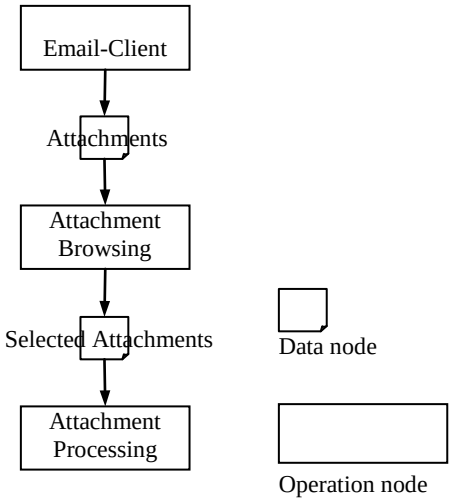

Fig. 4. SDG for the running example

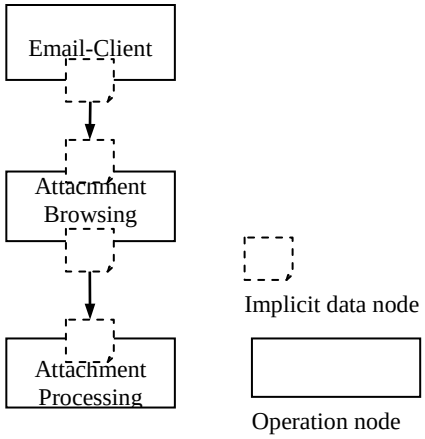

Fig. 5 Converted from SDG in Fig. 4

IDG enables capturing the change information by allowing the graph to have a time stamp denoting the status of its execution in terms of the time passed since its beginning and a location tag denoting where each component instance is located. We refer to it as time-based IDG. A cloud-based application is a sequence of incremental time-based IDGs that are composed of the actual components selected for implementing each functions required by the application. Each sub-graph in the sequence only contains the components that are picked to be invoked at a particular time during the execution of the component. Therefore, it is possible that at different points in time, the time-based IDGs representing the cloud-based application are different. Fig. 6 shows the sequence of incremental IDGs corresponding to the cloud-based application in the running example.

We can see in the sequence shown in Fig. 6, there are three time-based IDGs, each capturing the component selection progress made at distinct points in time. At time stamp t_1, which is right before the cloud-based application starts to execute, s_1_1, s_2_1, s_3_1 and the edges connecting them are included in IDG_{t_1}. At the next point in time, when S_1_1 is completed and the nature of the input to attachment browsing changes, IDG_{t_1} is updated to IDG_{t_2}. Lastly, after the first two tasks are completed and a change on the location of the user (into attachment processing) is detected, IDG_{t_2} is again updated to IDG_{t_3}. In the figures, dashed lines correspond to unselected and solid lines correspond selected. The numbers indicate non-failure rates of component instances.

V. Problem Quantification and Solution Calculation

In this section, we discuss in a more formalized way the calculation of the quantified resilience in order to determine which component instance to be planned in order to allow best resilience. Every time a different instance is to be invoked, the calculation we discuss below will have to be done.

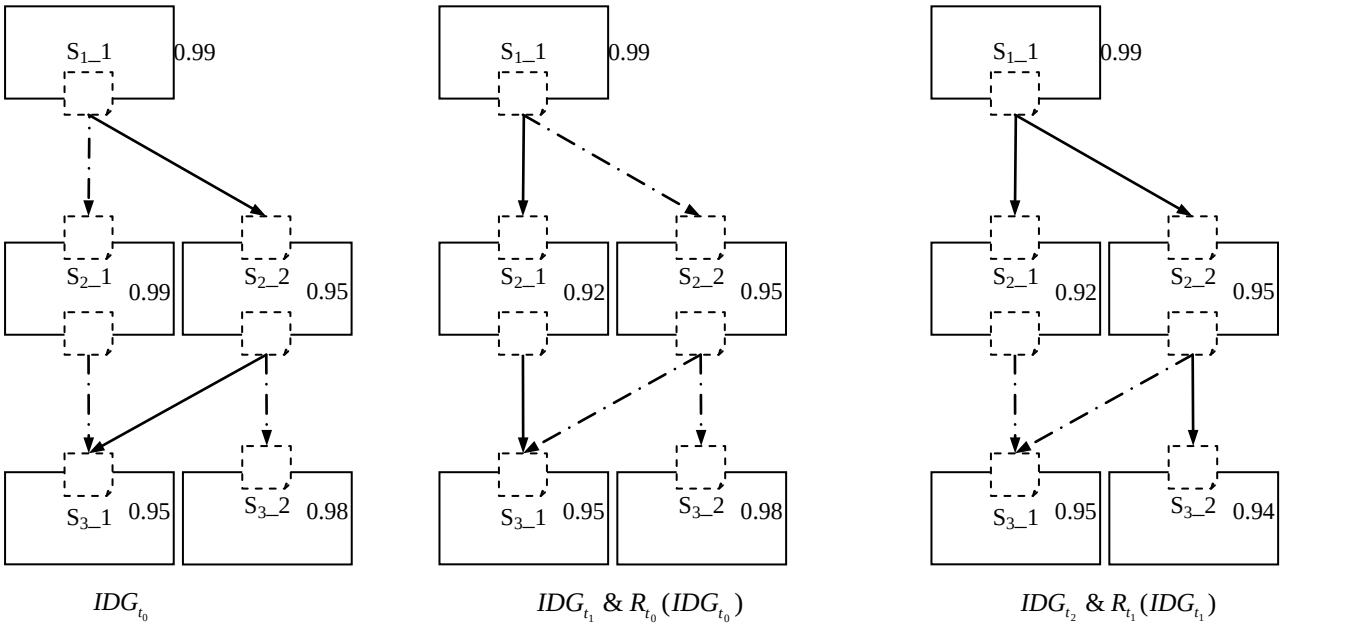

Fig. 6. Time-based IDGs in the running example at t_1, t_2 and t_3

A. Overview

Here is an overview of the approach before we go into more details: The formal quantified resilience can be made in a recursive way by defining the resilience of the cloud-based application (with a complete set of components it uses) by a partial set.

An instance at a particular point in time can be re-planned as a function of the resilience of the instance currently to be considered for selection, and those of the component instances that will possibly be scheduled to invoke one or more few steps ahead, including the component instance to be planned. In other words, select instance *s* such that $R_{t_k}(IDG_{t_k})$ is maximized. Here, in order to maximize $R(a)$, where *a* is the cloud-based application to be built, we will maximize $R_{t_k}(a)$. That is to find IDG_{t_k} that maximizes $R_{t_k}(IDG_{t_k})$ for all t_k. (When t_k =*K*, we have the following: $R_{t_k}(a) = R(a)$)

B. Details

The details are listed as the following: With the time-based IDG model, we derive a problem presentation of designing a resilient PaaS for building cloud applications in a formal way. In this research, this problem is cast as an incremental graph search problem. The problem presentation is given as the following:

Given a complete IDG of a cloud-based application $G(V,E)$ representing all valid component instance dependency relationships among possible components of the application, find a sequence of sub graphs $\{IDG_{t_i}\}$, such that IDG_{t_i} provides the required functions of the cloud-based application and the resilience of the cloud-based application, i.e. $R(a)$ is maximized.

Here we define $R(a)$ as a vector composed of the resilience of each sub graph in the sequence. It can be written as the following:

$$R_{t_k}(a) = \begin{pmatrix} R(IDG_{t_1}) \\ R(IDG_{t_2}) \\ \cdots \\ R(IDG_{t_k}) \end{pmatrix},$$

where *a* is the cloud-based application to be built, IDG_{t_k} is the *kth* time-based IDG in the sequence of time-based IDGs identified at moment of all t_k, and *k* = 1, …, *K*. *K* is the total number of time stamps before the entire application is completed. $R(IDG_{t_1})$ is the resilience of IDG_{t_k}. $R_{t_k}(a)$ is the resilience of *a* at time t_k. When t_k =*K*, we have the following:

$R_{t_k}(a) = R(a)$;

In order to calculate and compare the quantified resilience of an application being built in the platform, we have to first define the resilience of an individual or simple component. As mentioned, we focus in this paper on the resilience of a piece of software to possible failures. In this case, the resilience of a simple component instance can be defined as the probability that the component instance is successfully executed, in other words, the non-failure rate. The resilience of a component instance shall be affected by the constraining factors we discussed. Each

of the constraining factors is thus present as a parameter in the resilience formulation. We assume the failure rates of component instances can be obtained using the historical data of the component execution. The definition of the resilience of simple component instance *s* can be defined as:

$$R_{t_k}(s) = 1 - f_{t_k}(x,c,d)$$

where $R_{t_k}(s)$ is the resilience of component instance *s* at time t_k and $f_{t_k}(x,c,d)$ is the failure rate of *s* at time t_k with current user location *x*, content type *c* and deployment *d*.

Calculation of $R(IDG_{t_k})$ can be done in a recursive way. Basically, we start from the component instances of the first component (task) in the graph. We can use the resilience of a single component with the current instantiation. Then we will gradually enlarge the collection of tasks according to the partial order defined. We look at the resilience of the optional collection of component instances that implement the selected tasks. This procedure repeats until finally all the tasks of $R(IDG_{t_k})$ have been considered. The procedure can be summarized as the following formula, i.e. the resilience of the *IDG* at moment t_k is calculated by "transplanting" (using function Γ) the resilience of the newly added component instances (i.e. $\Delta(R(IDG_{t_k}\_l), R(IDG_{t_k}\_l-1))$ onto the resilience of the part of time-based *IDG* that keeps unchanged, i.e. $R(IDG_{t_k}\_l-1)$.

$$R(IDG_{t_k}\_l) = \Gamma(R(IDG_{t_k}\_l-1), \Delta(R(IDG_{t_k}\_l), R(IDG_{t_k}\_l-1)))$$

We consider three ways that a new component instance s_{new} can interact with a component *s* in the current *IDG*, s_{new} is in parallel with *s*; s_{new} is an optional replacement to *s*; and *s* sequentially follows *s*. The calculation of resilience of the $IDG_{t_k}\_l$ can be rewritten as the following:

$$R(IDG_{t_k}\_l) = R(IDG_{t_{k-1}}) * R_{t_k}(s_{new})$$

if s_{new} sequentially follows *s*;

$$= R(IDG_{t_{k-1}} \setminus s) * \min(R_{t_k}(s_{new}), R_{t_k}(s))$$ if s_{new} is in parallel with *s;*

$$= R(IDG_{t_{k-1}} \setminus s) * \exp(R_{t_k}(s_{new}), R_{t_k}(s))$$ if s_{new} is an optional replacement to *s*;

Here $R(IDG_{t_{k-1}} \setminus s)$ refers to the time-based IDG at moment t_{k-1} without *s* and $\exp(R_{t_k}(s_{new}), R_{t_k}(s))$ refers to the outcome of resilience of s_{new} and resilience of *s* under the probability that each option can occur.

With the above model defined, we are able to calculate IDG_{t_k} and, to select the one that has the maximum resilience IDG_{t_k} for all t_k. An illustration of the calculation of IDG_{t_k} for all t_k for the running example at time t_1, t_2, t_3 and t_4 are shown in table I.

VI. Related Work

We categorize related work of our research into those in the service arena and in the cloud arena. A collection of related topics in services computing include service composition, matching and adaptation [1][2][3][5][7][6]. For example, Hamadi et al. propose a service composition approach using a Petri net-based algebra. This algebra is designed to model control-flows, as a necessary constituent of reliable Web service composition process [3]. Canfora et al. proposes a QoS-aware binding approach based on Genetic Algorithms. The approach includes a feature for early run-time re-binding whenever the actual QoS deviates from initial estimates [19]. The above approaches can be characterized as re-estimation of the overall QoS of the composite service when the non-deterministic control flow constructs become known. Compared to their work, we focus on data-flows instead of the control-flows and one of the issues we address is the compatibility of components within the data flows.

A matchmaking algorithm that discovers and reports the relationships among terms describing service capabilities was described in papers such as [8]. Semantic matchmaking uses a matching engine or applies matching rules for terms from different descriptions. A semantic service discovery approach was also reported for efficiently finding services in given contexts in pervasive computing environments. Compared to our work, none of the above focused on facilitating building applications in a resilient way within clouds.

Research investigations also show promising results from Web service compatibility and substitution. This includes approaches where there is no automated

	$R(IDG_{t_i})$	$R(IDG'_{t_i})$	$R(IDG^*_{t_i})$	$IDG^*_{t_i}$
t_1	0.99*0.95=0.9405	0.95*(0.95+(1-0.95)*0.98)=0.93906	0.9405	V={s1_1,s2_1,s3_1} E={(s1_1,s2_1), (s2_1,s3_1)}
t_2	0.92*0.95=0.893	0.95*(0.95+(1-0.95)*0.98)=0.93906	0.93906	V={s1_1,s2_1,S2_2,s3_1} E={(s1_1,s2_1),(s1_1,s2_2),(S2_2,s3_2)}
t_3	0.95	0.98	0.98	V={s1_1,s2_1,S2_2,s3_1} E={(s1_1,s2_1),(s1_1,s2_2),(S2_2,s3_2)}
t_4	0.95	0.94	0.95	V={s1_1,s2_1,S2_2,s3_1} E={(s1_1,s2_1),(s1_1,s2_2),(S2_2,s3_2),(S2_2,s3_1)}

Table I. Incremental resilience of IDG_{t_i} in the running example at t_1, t_2 and t_3 and t_4

discovery of replacement services and there is neither automated way of handling dynamic context in optimizing certain quality criterion. For example, Taher et al. reported an approach to deploying communities of Web services. A community promotes the dynamic binding of Web services through a common interface, known as Open Service Connectivity. [9]. Becker et al. reported a method to automatically determine whether two services are backward compatible according to their descriptions. They then describe a case study to illustrate how to leverage version compatibility information in a SOA environment and the initial performance overheads of doing so. In comparison to the above research, our approach focuses on accommodating changes on the component instances that comprise the application to be built.

Related work in the cloud arena is mostly on distributed computing and grid computing, where a large body of work is presented on on fault tolerance, such as [11][12][13][14]. Most of the work is aimed at studies on the fault types or replication algorithms and fault-tolerant architectural designs to tackle various fault types in different environment such as synchronous and asynchronous. Comparing to such work, our research is cloud-oriented where instances of application components are distributed in a wider scale (as the Internet) and we are more concerned about the equivalence in functionality and the compatibility of them working with each other. Our focus is how to make use such of such component instances to enable building resilient applications.

VII. CONCLUSION

With PaaS clouds' promises of low-cost, easy and ubiquitous access of computing platforms and solution stacks, people are looking for a whole set of facilitations in building and deploying software applications and services in the clouds. In this paper, we discuss the issue of delivering resilience by PaaS clouds. In particular, we address the need of an enabler to build resilient cloud-based applications in PaaS clouds and we propose our solution using a graph structure called time-based IDG for dynamically maximizing the resilience of the application being built/deployed. The contribution of the paper is to satisfy the rapidly growing needs in developing resilient applications that can adapt to changing context such as that characterizing the mobile setting. We are able to take into considerations both external and internal factors that may change dynamically.

We are currently looking into a deeper study on related issues to switching component instances to enable resilience. These include deeper understanding of what are the factors and the rationales of determining when the changes are possible and what the changes are. In addition to that, house-keeping in the platform is usually needed after such switching is enacted.

It follows that the resilience of applications also relies on the transitional support in handling possible partial results of the components by the platform. We see this as an important aspect that needs to be studied in order to make sure the applications never become invalid under no circumstances.

ACKNOWLEDGMENT

The authors would like to thank all the reviewers for their helpful comments. We especially appreciate some very detailed comments, which are really good suggestions for us to improve the paper.

REFERENCES

[1]. O. Moser, F. Rosenberg and S. Dustdar, "Non-Intrusive Monitoring and Service Adaption for WS-BPEL," WWW 2008.
[2]. K. Yang, R. Steele, "An Ontology Mediated Web Service Aggregation Hub." Web Intelligence 2007, pp. 572-576.
[3]. Rachid Hamadi and Boualem Benatallah. 2003. A Petri net-based model for web service composition. In Proceedings of the 14th Australasian database conference - Volume 17 (ADC '03), pp.191-200.
[4]. S. B. Mokhtar, D. Preuveneers, N. Georgantas, V. Issarny, Y. Berbers, "EASY: Efficient semAntic Service discovery in pervasive computing environments with QoS and context support," Journal of Systems and Software 81(5): 785-808 (2008).
[5]. Q. Liang, X. Wu, H. C. Lau, "Optimizing Service Systems Based on A[7]pplication-Level QoS," IEEE T. Services Computing 2(2): 108-121 (2009)
[6]. Peter Leong, Chunyan Miao, Bu-Sung Lee, "A survey of agent-oriented software engineering for service-oriented computing," Int. J. Web Eng. Technol. 4(3): 367-385 (2008).
[7]. O. Moser, F. Rosenberg and S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL," www 2008.
[8]. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic Matching of Web Services Capabilities," The 1st International Semantic Web Conference, June, 2002.
[9]. D. Athanasopoulos, A. V. Zarras, V. Issarny, "Service Substitution Revisited," 24th IEEE/ACM International Conference on Automated Software Engineering - ASE 2009.
[10]. Dejun Jiang, Guillaume Pierre, Chi-Hung Chi: Autonomous resource provisioning for multi-service web applications. WWW 2010: 471-480
[11]. S. Basu, L. B. Costa, F. V. Brasileiro, S. Banerjee, P. Sharma, Sung-Ju Lee: NodeWiz: Fault-tolerant grid information service. Peer-to-Peer Networking and Applications 2(4): 348-366 (2009).
[12]. Stephen Frechette, A Proxy-Network Based Overlay Topology Resistant to DoS Attacks and Partitioning, Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 16, p.283.1, April 04-08, 2005.
[13]. A Graph Model for Fault-Tolerant Computing Systems, J. P. Hayes , IEEE Transactions on Computers archive, Volume 25 Issue 9, September 1976.
[14]. M. Castro, B. Liskov, Proceeding, Practical Byzantine fault tolerance, OSDI '99 Proceedings of the third symposium on Operating systems design and implementation.
[15]. Q. Liang, Stanley Y. W. Su, "AND/OR Graph and Search Algorithm for Discovering Composite Web Services," Int. J. Web Service Res. 2(4): 48-67 (2005).
[16]. K. Becker, A. Lopes, D. S. Milojicic, J. Pruyne, S. Singhal: Automatically Determining Compatibility of Evolving Services. ICWS 2008: 161-168.
[17]. Dual Transitions Petri Net based Modelling Technique for Embedded Systems Specification (2001) by Mauricio Varea , Bashir Al-Hashimi In Proceedings of the 4 th Proc. Design, Automation and Test in Europe.
[18]. http://www.mefosyloma.fr/pdf/j2005-03-18/BerndtFarwer.pdf.
[19]. G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. 2008. A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81, 10 (October 2008), 1754-1769.