Towards a SLA-based Approach to Handle Service Disruptions

Lionel Touseau [1], Didier Donsez [1], Walter Rudametkin [1,2]
[1] *University of Grenoble, LIG laboratory, ADELE team*
[2] *Bull SAS, Echirolles*
{firstname.lastname}@imag.fr

Abstract

Service-orientation enables cooperation between multiple organizations and has become a solution of choice to tackle the complexity of ubiquitous computing. The very nature of ubiquitous applications implies a need for dynamic solutions. Service oriented computing provides support for these dynamic applications. However, current solutions overlook important aspects, particularly when dynamically substituting services. Some applications may be mission-critical and therefore the disruption of a particular service could be harmful. Other applications may tolerate the disappearance of a service if the service returns within a predefined amount of time. Hence, guarantees regarding availability of services that compose an application are required. In this paper we propose to take into account service disruptions through service level agreements for dynamic service-oriented applications.
Keywords. *Dynamic SOC, service disruption, SLA, mission critical, OSGi*

1. Introduction

The success of Web Services has popularized the Service-Oriented Approach (SOA). Service-oriented computing (SOC) paradigms are becoming the basis for systems integration and are used for constructing applications in many varying domains. Service-oriented computing (SOC) has indeed revealed itself to be an appropriate solution to tackle the complexity of ubiquitous systems [1,2].

In these systems, where the architecture continuously evolves, smart devices are modeled as service consumers or service providers that offer their functionalities in the form of services. Moreover, most applications are dynamic and therefore connections and disconnections of devices must be taken into consideration [3]. This aspect, which is not directly related to the functionality of the application itself, is usually handled by developers and is a real burden. Devices, represented as services, composing an ad-hoc network are not known in advance, and the environment is likely to evolve at anytime. For instance, a device can be out of order, either temporarily or definitively, for many different reasons. It can be defective, run out of energy, undergo a physical maintenance or a software update. At the service level, any one of these phenomena will result in the unavailability of the required service and consequently in a *service disruption*.

Dynamic service-oriented computing, where services are introduced to or removed from the execution environment at runtime, provides mechanisms that allow services to be published and discovered dynamically. Thus, developers can rely on dynamic SOC to build applications that can adapt to new situations. In order to spare developers from the tedious and error-prone work, most current solutions [4,5,6] propose to mask service disruptions and to substitute the leaving service automatically using another service that provides the same contract, if one is available.

However, this solution is oblivious to cases where service disruptions are either acceptable or not harmful to the application. Not all disruptions are unexpected phenomena that must be considered as errors. A server can undergo a predictable maintenance or a service can be down for a few seconds because of a network disconnection. Moreover, switching providers can be costly. Not only is there the cost of unbinding and rebinding the services, but there is also the loss of state or context information.

Another point that needs to be considered is the multi-organizational aspect being introduced into service platforms. Service-oriented applications are likely to be composed of services managed by different organizations thanks to the loose coupling offered by SOA. In these situations, the consumer service, controlled by one organization, does not necessarily have control over the life-cycle of a service delivered by another organization. Furthermore, some applications may be mission-critical and non-tolerant to excessive service interruptions. That is why service consumers require guarantees regarding service disruptions and service availability. In this paper we will focus on service oriented platforms used in home and building automation domains [7,8]. In this context, services are generally deployed by different providers on local gateways, which are remotely operated by ISPs (e.g., embedded gateways in set-top boxes) or electric companies (e.g., electricity meters).

This paper presents an approach based on Service Level Agreements (SLAs) to express and to handle service disruptions in dynamic service-oriented applications. The remainder of the paper is structured as follows. Section 2 underlines some issues regarding service disruptions in dynamic service-oriented computing. Section 3 introduces service level agreements and then describes our dynamic SLA-based approach for handling service disruptions. A D-SLA Manager, which has been implemented on the OSGi framework [9] in a building automation application, is presented in Section 4. Section 5 discusses related work, and finally, Section 6 concludes this paper with perspectives and future work.

2. Service disruptions related issues in dynamic SOC

While static SOC is not flexible enough to build service-oriented applications in dynamic environments, dynamic SOC can be too permissive when dealing with service disruptions.

2.1. Service-oriented computing

The Service-oriented approach, which has been popularized by the standardization of Web Services, consists of the use and reuse of a functional entity: the service. A service is described by a contract that is independent from its implementations. Beugnard defined four levels of contract [10] that range from the syntactical level (e.g., CORBA IDLs or Java interfaces) to the Quality of Service (QoS) level (e.g., extended WSDL descriptor or OSGi service properties). The remainder of this article focuses on the fourth one, the QoS level contract. After being published by its provider, a service can be discovered and invoked by service consumers. Loose coupling is thus enabled and allows consumers to use services without knowing the details of their implementations. This way, cross-organizational interactions are facilitated.

2.2. Dynamic service-oriented computing

Dynamic SOC should not be confused with late-binding. Late-binding is the mechanism inherent in SOC that enables loose coupling by allowing service implementations to be chosen at runtime. Actually, at selection time, the service consumer only knows the service specification, not the implementation details which depend on the service instances published by service providers. The binding between the consumer and the service instance occurs at the latest possible moment, that is, when the consumer actually needs the service for the first time.

In dynamic SOC, services can be registered and unregistered at anytime, and service consumers can be notified of these changes. A service provider can therefore be dynamically substituted by another if the provider disappears or if another provider offers better contract conditions [11]. Let's consider a building automation example. A building decides to renew its smoke detectors which are getting outdated, as illustrated on figure 1. A new implementation of the *SmokeDetector* service is deployed along with the new sensors. Then the old ones are physically and logically uninstalled. At this time the fire detection application will automatically detect the removal and bind to the new service providers that represent the new sensors.

However a dynamic SOC approach is not always the best solution. The first problem when developing dynamic service-oriented applications is to take into account the dynamism itself. Service providers may become available and unavailable at anytime in an unpredictable way. Following the separation of concerns principles, it is not suitable to have code managing dynamism mixed in with the functional code. That is the reason why the management of bindings should be made as transparent as possible to developers. They should not have to deal with service arrivals and service removals, and should be able to program as if the service was always there, without worrying about dynamic availability concerns [4].

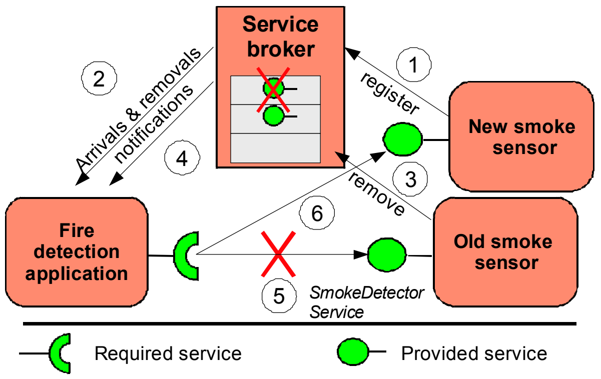

Figure 1. Service substitution scenario

Some research have led to solutions that consist in automating bindings [3,4,5,6]. Generally it means that a service provider which is removed from the execution environment will be replaced by another one, later or immediately, if possible. As a result, when a service disruption occurs there are three possibilities depending on the binding policies.

- The choice of the service provider is statically predefined and the consumer is stopped until this particular provider returns.
- The identity of the leaving provider does not matter and the latter can be replaced by another one as long as a compatible service is provided.
- No other provider is available for the same service and the consumer is stopped until a compatible service is published.

Yet, our study underlines a lack in these policies.

2.3 Limitations

Although some service disruptions are unpredictable (e.g., a device accidentally unplugged), others may be negligible or predictable enough to be acceptable (e.g., a scheduled server maintenance operation). From a software engineering point of view, switching providers can be an expensive operation. First, if dynamism is not handled orthogonally, developing a dynamic service-oriented application implies extra-functional lines of code that must foresee the cases when services are not available. Yet, most developers are used to and prefer programming in a static way. Second, depending on the service platform, unbinding and rebinding actions are more or less costly. Especially when an architecture is "sparkling" because of services being continuously removed and redeployed. Third, even if the substitute provided service is the same, it still may differ from the previous service from a quality point of view, that is, the QoS of extra-functional properties may differ from one service to another. Last but not least, a substitution may become tricky as soon as service usage consists of a sequence of service calls. All contextual or session information kept by the service provider would be lost. For example, long-term transactions are commonly made of multiple invocations.

That is the reason why, depending on the situation, it may be preferable to wait for the service to come back, especially if the disruption is expected and if the service provider can return within a reasonable time. Still, the provider could have left *ad vitam aeternam* or for too long a period and should not be waited for in vain. Consequently in the case of a service consumer that chooses to wait for its provider, the waiting time should not exceed a predefined limit.

3. Towards a SLA for the regulation of service disruptions

From the previous observation we can deduce a need for guarantees on services regarding service disruptions. Moreover, mission-critical applications must ensure high availability while remaining dynamic, that is, their downtime must not exceed a certain threshold. This is the case of most healthcare applications (e.g., heart rate monitoring) or applications in the energy industry [12,13]. Furthermore, in SOA, services are likely to be managed by independent organizations. Consequently, for a component belonging to an organization, it is not always possible to control the life cycle of components managed by other organizations, and this unpredictable nature hinders the adoption of dynamic multi-organizational service-oriented applications. This justifies that consumer components in these applications may require guarantees on the availability of their service providers. One way to express these guarantees is at the service contract level, through service level agreements (SLAs).

After a brief presentation of service level agreements major principles, these principles will be applied to address the issues raised in the previous paragraph.

3.1. Service level agreements overview

A service level agreement (SLA) is an agreement where the level of a provided service is formally defined. The agreement describes terms regarding service usage and service delivery upon which signatory parties have agreed. Generally a SLA contains the following:

- The agreement context: signatory parties, generally the consumer and the provider, and possible third parties entrusted to enforce the agreement, an expiration date and any other relevant information.
- A description of the provided service including functional and non-functional aspects such as quality of service.
- Obligations of each party, which are mainly domain-specific.
- Policies: penalties incurred if a term is not respected and compensation for the service usage.

A SLA becomes valid once it has been signed by the contracting parties after a negotiation process. Although this article does not develop this aspect, so far, we have distinguished three levels of negotiation which range from simple service provider selection to customizable contracts and complex negotiation processes. At runtime, the compliance is monitored through service level management (SLM). This management, which consists basically in monitoring and reacting towards agreement violations, is usually performed by third-parties for reliability reasons, since there might be no mutual trust in case of multi-organizational interactions.

3.2. D-SLA: SLA for dynamic SOC

From the limitations expressed about dynamic SOC and the conclusions drawn upon the requirements for a proper handling of service disruptions in a multi-organizational context, it is possible to define our D-SLA. D-SLA is actually a SLA for dynamic SOC that focuses on service disruption concerns. In the following sub-sections we define its content, and the agreement life-cycle, from its negotiation to its termination.

3.2.1. Content. First, parties should be uniquely and persistently identifiable. If one were to leave the application, it needs a unique identifier in order to be recognized and then be able to resume its activity when it comes back. This identifier could be derived from platform-specific identifiers (e.g., service.pid for OSGi, UDN for UPnP or the device UUID and the Bluetooth Device Address (BDA) for Bluetooth services).

Then, to characterize service interruptions, three criteria were judged to be necessary:

- *Maximum service disruption time*: the time elapsed between the service interruption and the return of the service provider.
- *Maximum accumulated service disruption on a sliding time-window*: total unavailability time on a certain period.
- *Time between two service disruptions*: fixing a minimum uptime avoids "sparkling" architectures where services continuously appear and disappear, disturbing the global application.

In addition to the involved parties, the expiration date, and the contract terms, a D-SLA also declares the policies used by the SLM system to define the actions that will be taken if a contract were to be violated, for instance.

3.2.2. Agreement negotiation. For the time being, complex negotiation protocols are not considered. SLA establishment is restricted to a simple service selection depending on the criteria listed above. The service is therefore selected according to the consumer requirements and the contract proposal published by the provider. Nevertheless we propose an innovative approach that considers service providers which are not present in the system at the selection time, but which will probably return soon. Thus, a client can choose to wait for a provider instead of using one already present. Such a selection can be performed according to past activities and service contracts of providers whose histories have been recorded.

The example shown in figure 2 depicts one service consumer, three service providers and their historical uptime. Only two properties are considered: the *maximum service disruption time*, Δ, and the *maximum accumulated disruption time*, $\sum\Delta$. The consumer cannot bear a service disruption Δ greater than 10 seconds and more than 10 minutes of unavailability per day. The three providers available in the system propose different guarantees. Only provider A and provider C match the consumer's requirements, since provider B cannot ensure a satisfying uptime as shown in the availability diagram. At t0, the moment when the consumer needs the service, only providers A and B are available. Although it could bind to provider A, the consumer could also wait for provider C to come back because it has slightly better uptime guarantees.

3.2.3. Service Level Management. In dynamic SOC it is possible to be notified of changes in the service registry and to make components aware of service arrivals and departures. Therefore by listening to these events, a service certifier can easily monitor the state of a particular service provider along with its compliance to the D-SLA clauses.

At runtime, unlike common practice, the disconnection of a service provider is not considered an error. Instead the service usage is suspended until the provider, uniquely identified, returns. If this does not happen within the allowed disruption time or if the accumulated interruption time goes beyond the limits of the agreement, the corresponding clause is broken. If a clause is broken, actions must be taken depending on the policies defined in the agreement. These actions could be as simple as replacing the missing provider and terminating the contract, but they could also be more complex, such as, putting the service provider on a blacklist and ignoring it for future selections, charging it with penalty fees, or even decreasing their reliability rate if providers are marked with trust indicators. In a context of dynamic renegotiation, it is also conceivable for a provider to adjust its guarantees and to renegotiate the agreement.

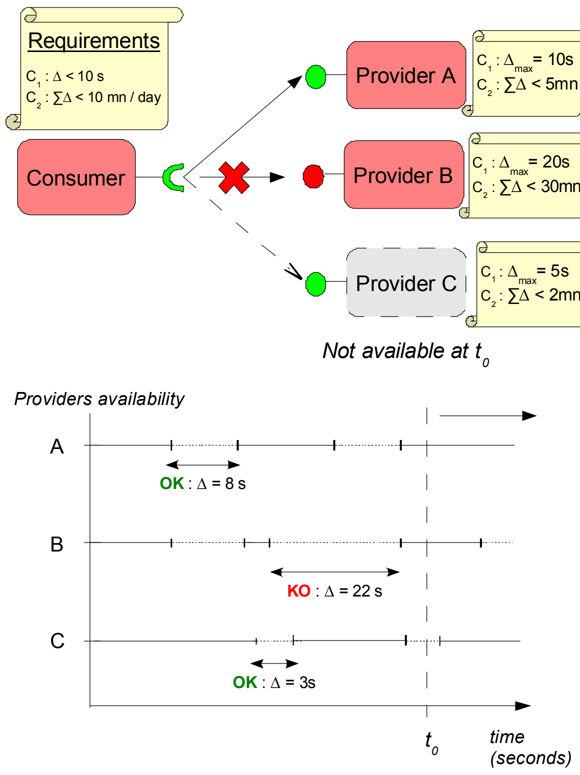

Figure 2. Service selection according to service disruption criteria and historical providers' availability

3.2.4. SLA termination. The agreement is terminated when the expiration date is reached, or, once one of the parties does not comply anymore with the agreement terms. This means in our case that a service disruption threshold has been exceeded and that a policy implied the agreement termination for this violation. When the contract is terminated, third parties declared in the contract can stop their monitoring activities.

4. Implementation and validation

To test and experiment our approach, a D-SLA Manager has been developed on top of the OSGi service platform [9]. This DSLA Manager is then examined through a fire alert application example.

4.1. DSLA Manager for the OSGi platform

To implement the DSLA Manager we chose the OSGi service platform for it appeared to be the best candidate for home and building automation service gateways. Although this dynamic service platform is a centralized execution environment, it still enables reasoning on local proxies of remote services. Local representatives can reify remote devices and Web services thanks to bridges between OSGi and other technologies like UPnP [14], DPWS [15] and WS technologies. Dynamism is delegated to the iPOJO framework [5], an extensible service-oriented component model for OSGi. This framework relies on dependency injection [16] to manage dynamic bindings and injects pieces of code called handlers in POJOs to deal with extra-functional concerns.

Figure 3 shows the DSLA Manager and its internal structure. The DSLA Manager keeps track of the contracted SLAs written in a XML syntax and is in charge of all SLM activities regarding these agreements. It includes one utility service and three handlers. Components just have to declare the handlers they want to use with the correct meta-data and their instances will be automatically configured by these handlers.

The first handler, the DSLAConsumerHandler is used by service consumers as an extension of the iPOJO DependencyHandler. In addition to the provided dynamic binding's management, this handler implements our policy, which means that it can freeze a service call when a service provider is no longer available. This same handler is then notified by the SLM handler if the provider returns or if the legal disruption duration expressed in the agreement is exceeded. Afterwards it either resumes the service call, or aborts it and look for another available service provider.

The DSLAProviderHandler adds some information at the service registration time, in particular a persistent identifier, that is, a value for the OSGi service.pid property, and service disruption duration properties necessary for the negotiation.

The DisruptionsLogger monitors and records disruptions of involved components. Thence, its DisruptionsLogService and the gathered information can be used to improve negotiations on availability criteria, and can assist SLM components with their monitoring activities.

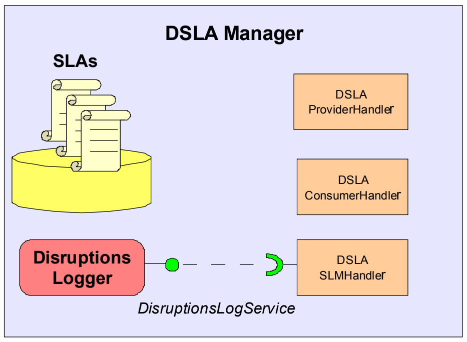

Figure 3. DSLA Manager structure

The SLM handler acts as a compliance monitor. Not only does it monitor service disruptions using the DisruptionsLogger but it also reacts by notifying involved parties and applying penalties. Regarding security and performances, the monitoring is not intrusive since it is done using service event listeners.

4.2. Building automation example: fire system

Consider a fire alarm system in a building. Each floor is equipped with smoke sensors able to detect a fire. The data is aggregated and sent to a central fire system. If a fire is detected, the system will perform several actions like activate the alarm and send an alert to the fire station. Figure 4 shows a service-oriented view of the system. Smoke sensors and the central fire system communicate using the producer/consumer pattern [17]. For clarity's sake, aggregation services and others devices such as fire doors or sprinklers do not appear in this figure. The alarm and the fire station alert system are respectively represented by an AlarmService and an AlertService that sends alerts to the fire station. In this scenario it is understandable that the unavailability of the AlertService would be harmful in case of fire. This justifies the need for an agreement between the provider of the alert service and the central fire system of the building. Agreements could also be established between smoke sensors and the central system since a defective sensor would not be able to detect a fire and deliver the information to the system. But, for simplicity we just focus on the agreement involving the fire station.

Technically, the central fire system, the AlarmService that controls the alarms and the AlertService that remotely calls the fire station, are all deployed on the same OSGi gateway.

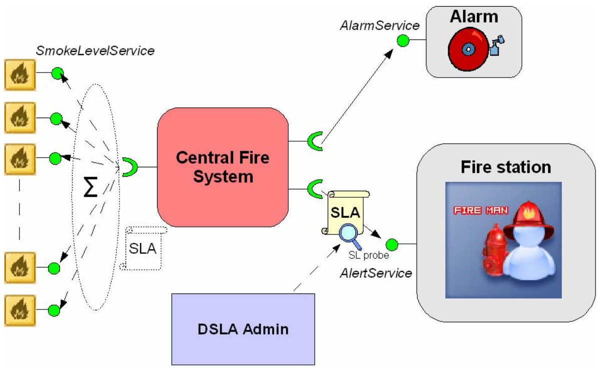

Figure 4. Fire alert system

In this example, the alert service must not be disrupted for more than 1 minute. If the AlertService provider is interrupted for more than 1 minute it will have to compensate the consumer and will not be able to charge it. Figure 5 shows the iPOJO meta-data describing the central fire application and the AlertService which respectively declare their DSLA ConsumerHandler and DSLA ProviderHandler.

Consumer: Central Fire System

```
<?xml version="1.0" encoding="UTF-8"?>
<ipojo
xmlns:dsla="fr.liglab.adele.dsla.ipojo.handlers">
  <!-- declaration of component type and instances-->
  <component
        classname="firecentral.FireCentralApp">
    <requires field="m_smokeServices"
              policy="dynamic-priority" />
    <requires field="m_alarmService"/>
    <callback transition="validate" method="start"/>
    <callback transition="invalidate" method="stop"/>
    <!--  declare our handlers -->
    <dsla:requires field="m_alertService"
                 PID="fire.system.central"
                 maxServiceDisruption="60000"
                 maxCumulatedServiceDisruption="300000"
                   period="1"/>
  </component>
  <!-- Declare an instance -->
  <instance
          component="firecentral.FireCentralApp"
          name="CentralFireSystem"/>
</ipojo>
```

Provider: Fire Station Alert Service

```
<?xml version="1.0" encoding="UTF-8"?>
<ipojo
xmlns:dsla="fr.liglab.adele.dsla.ipojo.handlers">
  <!-- declaration of component type and instances-->
  <component classname=

"firestation.alertservice.impl.AlertServiceImpl">
    <provides interface=

"dsla.fireapp.alertservice.AlertService" />
    <callback transition="validate" method="start"/>
    <callback transition="invalidate" method="stop"/>
    <!--  declare our handlers -->
 <dsla:provides PID="firestation.alertservice.provider"
                maxServiceDisruption="60000"
                maxCumulatedServiceDisruption="300000"
                period="day" />
    <dsla:SLM agreementField="m_agreementID"
             violationPolicy="TerminationWithNoCharge">
  </component>
  <!-- Declare an instance -->
  <instancecomponent=
       "firestation.alertservice.impl.AlertServiceImpl"
  name="FireStationAlertService"/>
</ipojo>
```

Figure 5. Sample of the fire central and the alert service component descriptors

5. Related work

Through this paper we demonstrated that there are features lacking in both static and dynamic SOC. In the first case service disruptions are simply considered as errors and stop the application, whereas in the second case a disruption is made as transparent as possible by dynamic substitution regardless of the context. Web Services which have been initially designed for long-term transactions and workflow processes are the best representatives of static SOC. Despite the late-binding characteristic, dynamic applications cannot be designed without any service registry. The same limitation arises for the Service Component Architecture (SCA) initiative [18]. It is yet possible to add dynamism in these models. For instance, the Device Profile for Web Services [15] enables dynamic service-oriented computing over ad-hoc network thanks to WS-Discovery and WS-Eventing protocols. In a different way, the well-known Eclipse IDE which is based on the OSGi Equinox platform [19] does not fully exploit its dynamic potential. When a new plugin or service is deployed, the whole environment must be restarted. In the field of dynamic SOA, several component models have aimed at handling dynamism concerns in order to facilitate the development of dynamic service-based applications. On the OSGi platform, the first was *ServiceBinder* [4] that later became *Declarative Services* in the 4th release of the specification. Spring-Dynamic Modules [6] is another component model that uses XML descriptions to automate bindings in the Spring framework. The iPOJO [5] DependencyHandler follows the same principles. Components declare their service dependencies and according to these declarations, component bindings are automated dynamically at runtime, although it is still possible to declare static mandatory services that cannot be substituted. Regarding distributed dynamic service platforms, UPnP [14] and DPWS [15] allow dynamic publication and discovery of remote services.

However, these models do not prevent "sparkling" architectures that change continuously without ensuring a minimum steadiness, nor do they take into account the fact that a service disruption can be a normal phenomenon and that in certain cases it is not worth substituting a service provider that might come back within an acceptable delay. Our proposition is a compromise between these two visions. Dynamism is supported but in some cases, a consumer would be willing to keep the same service provider even after a disruption.

Moreover, these component models do not provide any control on the service selection, except for filters on service properties. Context ranking selection [11] proposes a refined service selection by extending the Declarative Services model with definition of preferences depending on the context. In a different way, probabilistic selections [20] in disconnected and mobile ad-hoc networks take into account the disruption probability of available services in order to select the services to bind and to invoke. This approach could extend our selection model that considers providers potentially available in the future. The DisruptionLogger, which keeps a history of service availabilities, could infer and compute disruption probabilities that could help in the service selection process. However, in [20] there are no considerations for services that surpass the expected duration of unavailability.

Finally, this paper does not aim at defining new SLA formalisms or impose a SLM framework. WSLA [21] and its successor, the WS-Agreement specification [22], or even other SLA formalisms such as SLAng [23] already exist and are satisfactory. Besides, to cope with other SLAs we plan to rely on the WS-Agreement framework for future implementations through MDE approaches. For the same reason, we did not consider semantic matching issues for contract negotiation since it is already addressed by other research [24].

6. Conclusion and future work

This article has presented issues raised by dynamic service-oriented architectures, an efficient way to build machine-to-machine applications, which are likely to cross organizational boundaries. As a result, services come and go in an unpredictable way. Yet, some healthcare or security applications are mission-critical and cannot afford permanent or lasting service disruptions. This paper has described a way to transparently handle service disruptions through service-level agreements and service-level management. An implementation of our approach has been developed on the OSGi framework since we focused on the home and building automation contexts, but for other domains it might be implemented on SCA or Spring frameworks (e.g., for dynamic application servers [25]).

Several points that have not been deeply investigated in this position paper, often for space reasons, will be the subject of future work. This is the case of SLM activities other than monitoring. For instance, we did not go through policies and mechanisms of penalties and compensations. Neither did we detail the agreement creation after the negotiation process.

Besides, several issues that have not been mentioned remain open. This paper was essentially about synchronous request/response connections but in future work we will consider other types of connections [26], particularly the producer/consumer pattern. Sensor-based applications have become a major element in the building automation domain, and most sensor communications are based on data streams or events. We are currently investigating D-SLA for those communication patterns in the context of the

ASPIRE project [27]. This project, funded by European Community, develops an open-source middleware for applications involving passive and active Radio Frequency Identification [28] (RFID) exchanges.

In addition, it is worth noticing that the service provider could also require guarantees upon the consumer's availability because obligations may concern both parties. For instance, if a consumer leaves in the middle of a transaction operation, the provider cannot afford to keep the client session forever. As a consequence, service providers should be able to perform admission control on their users, and thus, only cooperate with D-SLA-aware service consumers.

Finally, in the negotiation process, other quality-of-service properties should be taken into account along with service disruption criteria. Therefore, our D-SLA Manager should be extended and it should conform to SLA recognized standards like WS-Agreements in order to support other kinds of agreements.

7. References

[1] M. Weiser, “The computer for the 21st century”, Scientific American, 265(3):66-75, September 1991.

[2] International Telecommunication Union, “The Internet of Things”, Executive Summary, ITU Internet Reports, November 2005.

[3] L. Touseau, H. Cervantes, D. Donsez, “An Architecture Description Language for Dynamic Sensor-Based Applications”, in 5th IEEE Consumer Communications & Networking Conference (CCNC 2008), Las Vegas, Nevada, January 2008.

[4] H. Cervantes and R. S. Hal, “Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model”, International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May 2004

[5] C. Escoffier, R. S. Hall, P. Lalanda, “iPOJO: an Extensible Service-Oriented Component Framework”, IEEE International Conference on Services Computing, 2007. (SCC 2007), 9-13 July. Salt Lake City (Ut)

[6] Spring DM for OSGi Specification (v0.7), 2006, http://www.springframework.org/osgi/specification.

[7] D. Marples, S. Moyer, “Home Networking and Appliances”, in Diane Cook, Sajal Das, Smart Environments: Technologies, Protocols and Applications, Wiley, 2004

[8] D. Snoonian, “Smart Building”, IEEE Spectrum, August 2003.

[9] OSGi ^{TM}, “OSGi Service Platform Specification, Release 4”, Available online at http://www.osgi.org

[10] A. Beugnard, J-M. Jézéquel, N. Plouzeau, “Making Components Contract Aware”, IEEE Computer 32(7): 38-45, 1999

[11] A. Bottaro, R. S. Hall, “Dynamic Contextual Service Ranking”, 6th International Symposium on Software Composition (SC 2007), Braga, Portugal, March 2007

[12] C. Marin, P. Lalanda and D. Donsez, “A MDE approach for power services development”, International Conference on Service Oriented Computing (ICSOC), Amsterdam, December 2005.

[13] A. Chazalet, P. Lalanda, “A Meta-Model Approach for the Deployment of Services-oriented Applications”, Proc. 5th IEEE International Conference on Services (SCC'07), Salt Lake City, USA, July 2007.

[14] The UPnP Forum, http://www.upnp.org

[15] F. Jammes, A. Mensch, H. Smit, “Service-oriented device communications using the devices profile for web services”, MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, 2005

[16] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern” http://martinfowler.com/articles/injection.html,, 2004.

[17] N. Nillson, “Connecting Producers and Consumers”, position paper at OOPSLA Worshop on References Architectures and Patterns for Pervasive Computing, 27 October 2003, Anaheim, CA, USA

[18] Service Component Architecture (SCA) specification, http://www.ibm.com/developerworks/library/specification/ws-sca/, 2006

[19] Equinox OSGi platform, http://www.eclipse.org/equinox/

[20] N. Le Sommer, “A Framework for Service Provision in Intermittently Connected Mobile Ad hoc Networks”, in 8th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WOWMOM 2007), Helsinki, Finland, June 2007.

[21] H. Ludwig, A. Keller, A. Dan, R.P. King and R. Franck, “WSLA Language Specification, Version 1.0”, 2003

[22] Grid Resource Allocation Agreement Protocol (GRAAP) WG, “WS-Agreement Specification”, March 2007, http://forge.gridforum.org/sf/projects/graap-wg

[23] D. Lamanna, J. Skene and W. Emmerich, “SLAng: A Language for Defining Service Level Agreements”, FTDCS, 2003

[24] N. Oldham, K. Verma, A. Sheth., and F. Hakimpour, “Semantic WS-agreement partner selection”. In Proceedings of the 15th International Conference on World Wide Web (WWW '06), Edinburgh, Scotland, May 23 - 26, 2006

[25] M. Desertot, D. Donsez, P. Lalanda, “A Dynamic Service-Oriented Implementation for Java EE Servers”, IEEE SCC 2006, Chicago, USA, September 18-22, 2006

[26] N.R. Mehta, N. Medvidovic, S. Phadke, “Towards a taxonomy of software connectors”, in the Proceedings of the 22nd International Conference on on Software Engineering (ICSE), June 4-11, 2000, Limerick Ireland. ACM (178-187)

[27] ASPIRE Project (FP7-215417) website, http://www.fp7-aspire.eu/

[28] Sandip Lahiri: RFID Sourcebook. Pub. IBM Press, August 2005; Pages: 304, ISBN 0131851373