

Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing

W. T. Tsai, Weiwei Song, Ray Paul*, Zhibin Cao, Hai Huang
Department of Computer Science and Engineering
Arizona State University
Tempe AZ, 85287 USA
**Department of Defense, Washington DC*
Email: {wtsai, weiwei.song, zhibin.cao, hai}@asu.edu

Abstract

Web services (WS) received significant attention recently because services can be searched, bound, and executed at runtime over the Internet. This paper proposes a dynamic reconfiguration framework based on the WS architecture so that critical computations such as atomic transactions can be continued in spite of the unavailability or failures in the participating services or networks. This framework proposes a new service specification to define services and applications in a WS system. As an extension to the existing WS interface specification, the new specification adds the ability to include the scenario specification based on ACDATE (Actors, Conditions, Data, Actions, Timing, and Events), and to enforce the system constraints such as reliability, security, and performance. Using the new specification, this paper proposes a runtime dynamic reconfiguration tool that consists of several distributed agents to reconfigure participating parties to provide a reliable, secure, interoperable and integrated application. These distributed agents will perform various runtime verifications, auditing and reconfiguration. One of the interesting properties of the framework is that it is designed to adapt to change easily and the changes are verified and applied at runtime.

Keywords: Service Oriented Architecture, Dynamic Reconfigurable Services, Scenario Specifications, Constraint Specification, and Distributed Agents.

1. Introduction

Traditionally, many studies [1][2][8][10] have been focused on exploiting the software architecture specifications such as Architecture Description Language (ADL) to represent the configuration and connection of system components. ADL is an architecture language that formally specifies the structure of the components and their connections. Based on ADL, several dynamic architecture

description languages have been developed with corresponding language tools such as Rapide [7]. While the former focuses more on the static perspective of system and system component architecture, the latter is ideally to express the dynamic transitional relationship among the working tasks.

Besides the approach based on the software architecture, there are other wide variety of studies on runtime dynamic reconfiguration. These include the approaches exploiting the middle ware object oriented architecture such as CORBA, and augmenting of the operation systems and compilers technologies. In recent DARPA programs on survivable systems, several research activities are reported. Wells et al proposed a dynamic reconfiguration system based on an object services architecture in [17], where the reliable configuration model is built to connect the components, in addition, a utility model is used to optimize reconfiguration decisions given that the same function may be implemented by multiple services. Hiltunen et al proposed a dynamic reconfiguration framework in [5] where an event-driven middleware layer is used to reconfigure the system. In [7], Knight, Sullivan, Elder and Wang discussed several survivability issues and proposed a hierarchical survivable architecture.

Web services (WS) based systems [18] received significant attention recently as major IT companies such as IBM and Microsoft are pushing for this new distributed computing paradigm. WS has significant advantages over traditional approaches because services are located, bound, and executed at runtime over the Internet using standard protocols such as UDDI, WSDL, and SOAP. Since it is relatively easy to perform system integration, applications developed in this paradigm can be viewed to form a loosely coupled

architecture that provides maximum flexibility in terms of system structure and evolution. Furthermore, it makes the system more dependable because sub-systems can be removed from or added into the environment without changing the overall architecture [4]. This kind of architecture also maximizes system reusability because legacy systems can be wrapped and reused without significant changes.

On the other hand, although WS can be located, bound and executed at runtime and over the Internet, once bound, the application will always call the same service unless another round of service relocating and rebinding are performed. Furthermore, there are times when deployed services need to be upgraded. Another weakness of the existing WS systems is that real-time and dependability requirements are not taken into consideration.

Based on our previous work [3][12-16], this paper proposes a new framework that extends the existing WS to address the problems mentioned above. The new framework can perform true dynamic reconfiguration for WS, i.e., the system will automatically reconfigure participating services at runtime in case of service unavailability, network congestion, overload, security intrusion, software and hardware failures. This framework is based on a new specification technique which is an extension of the existing WS interface specification. Current WS specification is based on WSDL. An extension to WSDL has been recently in [12] to enhance reliability of WS through verification. This paper specifies a service by its ISC (Interfaces, Scenarios, Constraints) specifications. The Interface in the ISC represents the existing WSDL specifications and their extensions. The Scenarios in the ISC represent the operational scenarios of the service based on the ACDATE (Actors, Conditions, Data, Actions, Timing, and Events) model [13]. The ACDATE is the building blocks for semi-formal system scenarios, and once a system is specified using ACDATE, it is possible to perform various automated simulation without any simulation programming, and furthermore, the system can be subject to various formal analyses such as model checking [15] and Event Tree Analysis (ETA) [14]. The Constraints of the ISC specify system constraints such as timing constraints, reliability constraints and security constraints. The ISC specifications are used by the proposed dynamic reconfiguration tool to reconfigure WS in case of system failures or overload.

Based these research results, this paper proposed the Dynamic Reconfiguration Service (DRS) framework for improving the dependability of the

WS systems. Before any service can be added into the DRS framework, it is important that the service satisfied the dependability attributes. In the DRS framework, each service must be certified before it can participate in the application. The verification is done by the check-in and check-out mechanism described in [13][16]. This process is triggered each time the service is changed.

Our approach is based on the same spirit of utilizing a software architecture approach to address the dynamic reconfiguration of the Services Oriented Architecture SOA. We explored the feasibility of developing new service specification technique ISC which is an extension of WSDL to specify the static and dynamic structure of services. And based on the ISC, a runtime distributed dynamic reconfiguration tool has been designed and implemented.

The ISC specifications are used by the DRS to verify that any new services joining the application satisfied the application's requirements and constraints. In the proposed framework, services and applications are organized as services in a hierarchical directory tree. The DRS performs the service registration/de-registration, lookup, verification, binding, execution, monitoring at runtime, and the re-selection and re-binding in case of failures or overload.

Furthermore, the DRS itself is implemented as a critical service with redundancy. One DRS can be replaced by another DRS in case of failure at runtime, and thus removing single point of failure.

The rest of the paper is organized as follows. Section 2 gives an overview of SOA and the ISC specifications and explains how ISC can be used to specify the scenarios of reconfiguration. Section 3 describes the system architecture of DRS and its distributed agents. Section 4 elaborates the dynamic reconfiguration mechanism within the DRS framework. Section 5 presents the experiments of the DRS with a sample illustration. Section 6 concludes the paper.

2. Services-Oriented Architecture and ISC

The SOA extends WS by requiring each service publishing its ISC specifications in addition to its WSDL specifications. Like WS, the SOA organizes each sub-system as a service, and each sub-system interoperates with each other via standard protocols. The differences between WS and SOA are that the entire system, including all of its internally layers, is organized as a service architecture, rather than

just at the application level like WS. In other words, the system will become inherently survivable in case of system failures because each layer of the system can be considered a loosely coupled architecture with services. Furthermore, each service will be specified using the ISC (Interfaces, Scenarios, and Constraints) convention defined as follows:

- **Interface Specification**
 - Input/Output parameters
 - Communication protocols
 - Interfaces with other sub-systems
- **Scenario Specification**
 - Specify the service scenarios using the ACDATE model
 - Specify Interactions with Other Sub-systems
- **Constraint Specification**
 - Based on the scenario model and ACDATE model
 - Specify the properties of the services must include, such as
 - Reliability
 - Availability
 - Security
 - Timing
 - Concurrency
 - Performance
 - Sequence
 - Safety
 - These constraints must be addressed at runtime to assure dependable computing.

Note that a service can be formed by several other sub-services. In this case, the composite service has its own ISC specifications, and each of its sub-services also has its own individual ISC specifications, as shown in Figure 1.

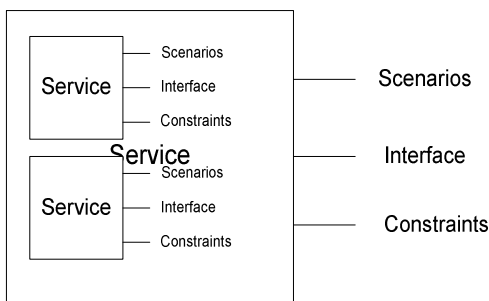


Figure 1 A Composite Service

Once the ISC specifications are available, it is possible to perform various static and dynamic analyses such as completeness analysis, consistency analysis, state analysis, pattern analysis, usage analysis, security analysis, and safety analysis as discussed in [14], either at compile time or runtime.

For instance, a real-time message exchange service

is a program that can forward incoming messages from and to the registered parties. It also blocks any unauthorized messages that are trying to reach a registered party, and prevents a classified or sensitive message from being sent to any unauthorized party. The following shows some aspects of its ISC specifications:

- Interface Specification:
 - Register/deregister: Participants register or deregister to the service with the information of identification, address, authorization level etc,
 - Update registering information,
 - Messages receiving and forwarding.
- Scenarios Specification:

The scenarios can be used to describe a service functional behavior and non-functional constraints. One functional scenario for this service could be:

Begin:
While (Event: Receiving)
Action: Check the Authorization levels
If (Condition: Authorized)
Action: Forward Message
ELSE
Action: Discard the Message

End

- Constraints Specification:
 - Timing constraints: Message forwarding should not take more than T seconds after it receives the message at any time. This constraint is represented as:
 $t(\text{event. Forwarding}) \leq t(\text{event. Receiving}) + T$, where $t(\text{event})$ is the event occurring time.
 - Security Constraints: an unauthorized message cannot be sent to a registered party, and a classified message cannot be sent to an unauthorized party:
 $L(\text{Actor. Message}) \leq L(\text{Actor. Receiver})$, where $L(\text{Actor})$ means the authorization levels of a specified Actor or Participants inside of the system.

The ISC specifications can now be used to verify and audit various system properties at runtime.

In the following two sections, we present the major research work performed in this paper, the proposed DRS framework.

3. Architecture of DRS and its Components

A DRS uses the ISC specification to configure

the participating services and to form the application. It monitors the runtime behavior and performs dynamic reconfiguration in case of service unavailability, overload, and system failures, to ensure the quality of services.

In the SOA, every participating unit, including DRS, is a service and each service is treated the same. As a service, the DRS provides the following functions:

- Dynamic service lookup, service publication, service binding, and service profiling,
- Registration and de-registration,
- Runtime services verification including constraint verification such as security verification, interoperability checking, and performance monitoring, and
- Dynamic Service reconfiguration, which means by changing the ISC definition for a service, it changes the behavior of a service runtime and even the DRS framework itself, since it is also a service of services.

Multiple DRSs can exist at each layer of the system including

- Application layer: Services at this layer provide application-oriented services.
- System layer: Services at this layer provide platform-related services such as resources allocation, file management and system-level security and monitoring.
- Infrastructure layer: Services at this layer manage service creation, scheduling, and deletion.
- Network layer: Services at this layer handles various communication protocol stacks.

In this way, a service at one layer actually uses services at the lower layer to perform its computation. Because each layer now is a loosely coupled architecture of decentralized services, each layer is self survivable, which in turn makes the entire system survivable.

At each layer, multiple DRSs, forming a DRS cloud, may exist so that in case a DRS of a given layer fails, the other backup DRSs can take over the assigned tasks to continue the operation. Thus, DRSs at each layer must communicate and synchronize with each other to ensure dependable computing. Figure 2 shows the layered architecture of DRS and the overall system infrastructure. A DRS is a service with several sub-services:

- Service Directory (SD): This stores and organizes services in a hierarchical tree with internal tree node representing a group of related services.
- Standard Service Naming Directory (SSND): This stores all the names of services registered

in an alphabetical order.

- Proxy Agents: An Proxy Agent (PA) is responsible for interoperability and integration between DRS, services and their clients. In addition, it also enforces security accessing control.

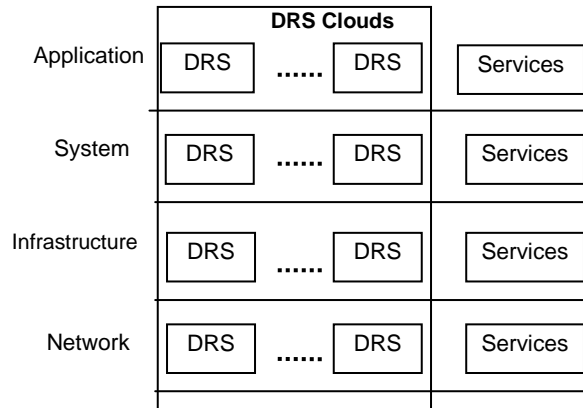


Figure 2 Layered Architecture of DRSs

- Auditing Agents: An Audit Agent (AA) monitors and checks the performance and user concerned properties of the participating services at runtime and updates their profiles.

The internal architecture of a single DRS is shown in Figure 3. The three key components, the service directory, proxy agents, auditing agents, and their interactions with other components are elaborated in the following subsections.

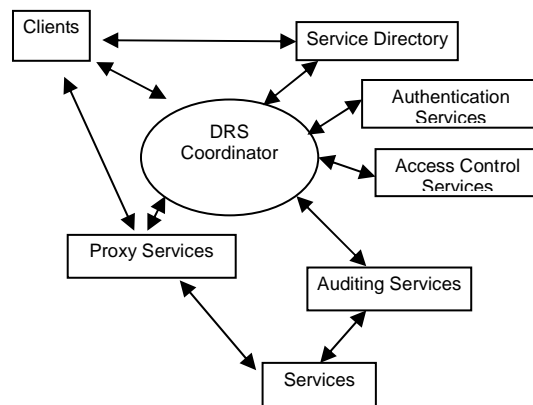


Figure 3 the Architecture of a Single DRS

3.1 Service Directory

An SD provides services lookup, services publication, services registering/de-registering, services evaluation and ranking, and runtime verification. For reliability sake, a single DRS can

have multiple SDs, and thus they need to synchronize with each other, and must back up in case of system failures. Furthermore, each SD keeps a cache of recently used services for efficient access, and each SD keeps track of a list of quality services in case of user queries. Currently, both the SD and SSND are implemented using LDAP (Lightweight Directory Accessing Protocol) [6].

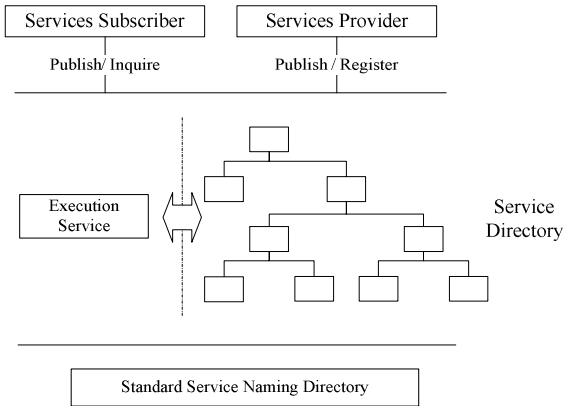


Figure 4 Service Directory Tree

A DRS needs to interact with Service Provider (SP), Service Subscriber (SS), SD, and SSND. Figure 4 shows these relationships, where the Execution Service represents all the execution parts of the DRS including runtime verification and registration. Each service in the SD is specified using the ISC format. As summarized in section 2. The SD also maintains the related assurance materials including test cases, evaluation routines, history usage patterns, and feedbacks associated with each service. The verification code associated with the service can be used to ensure reliability, interoperability, security, safety, and performance.

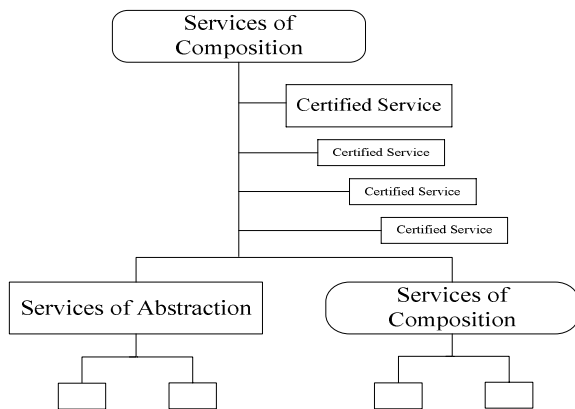


Figure 5 Hierarchical Tree of SD

The SD organizes services as a hierarchy of services. Each node in the tree can be one of the three types: *composition*, *abstraction*, and *certified*. Figure 5 shows a sample hierarchical tree of a SD. A composite service, denoted as a rounded rectangle, is a service that is an integration of its child services. An abstract service, denoted as a regular rectangle, represents an interface only, and all its child services satisfy the stated ISC requirements. A certified service does not have any child node, thus it cannot be decomposed.

The functions of the SD are to provide service publication, lookup, and certification for both Services Subscribers (SSs) and Services Providers (SPs). The mechanisms and operational processes of service publication and registration are described below:

- An SS sends in its service request to the DRS. The DRS will locate the needed services by matching the stated requirements with the ISC specification of registered services. The DRS may return multiple services that satisfied the stated requirements and let the SS decide which one to call.
- An SS may actually request creation of an abstract node in the SD to an DRS. The abstract node will contain the stated requirements, and the DRS may publish the requirements to all the SPs. In this way, any SP that can actually supply the required services can request its services to be attached to the published abstract node. The DRS may actually evaluate the submitted services by SPs before accepting it by using the test scripts in the ISC specifications supplied by the SS.
- Whenever an SP wishes to register a service with the DRS, the service will be evaluated by the DRS before it can be published to the SS. Specifically, if the service wishes to join an existing node in the SD, it will be evaluated by the test scripts and evaluation routines associated with the node. The service must pass these tests successfully before it can be accepted by the SD node. This is needed for quality assurance [13][16]. An SP may submit a new service and request a new node in the SD to be created. It can do so by also supplying the related ISC specification with the node. The submitted service will still be evaluated before it can be accepted by the DRS.
- A DRS keeps track of the status of various online services including their performance and will initiate a dynamic reconfiguration if it

detects a service failure or overload to the concerned SSs or SPs.

- A DRS keeps a list of best services at its cache based on user feedback and status report from participating agents. The DRS may actually publish the list to all SSs and SPs so that the information stored can be useful for both SSs and SPs in making their decisions.
- If an AA detects the failure of a service, it will be reported to the DRS, and the corresponding service will be marked unavailable. If a service is marked unavailable for an extended period of time, it will be de-listed from the SD.

In summary, a SSND is an integrated component in a DRS, and it contains the complete set of registered services in alphabetical order, and it will be updated whenever a service is registered or de-listed.

3.2 Proxy Agents

A PA coordinates between services and their clients at runtime. Due to its role as a broker, it is also a place where access control can be enforced. A PA is created whenever an abstract node in the SD is created. It will be activated whenever one of the services under the abstract node is called. A PA may consist of one or multiple Service Proxies, Access Control Lists, and Proxy Scenario Lists. Because all the services under the abstract node provide the same functionality, each can replace each other whenever

there is a need of reconfiguration. The functions of a PA are briefly described as follows.

- The interface of PA is defined by the corresponding abstract node in the SD. In each abstract node there is one corresponding PA.
- When a client initiates a service request, the DRS will return the address of corresponding PA to the client instead of the address of a service directly. In other words, the client interacts with the PA only.
- Each registered service must interoperate with its corresponding PA.
- PA responds to any service invocation by invoking the corresponding service proxy scenario and checks with the accessing control list in the mean time. A proxy scenario is a scenario given by service providers that instruct the PA how to map the standard interface to its proprietary counterparts. Once defined, it provides a universal form for service accessing. By redefining the proxy scenario for a service, it changes the way service will be accessed automatically as long as it confirms with the abstract service definition.

Now consider an example of the proxy scenario. Assume we have several Echo Web Services which always echo back whatever input it gets over the network. However, they have different interface definition. They are “1)String echo(string input)”,

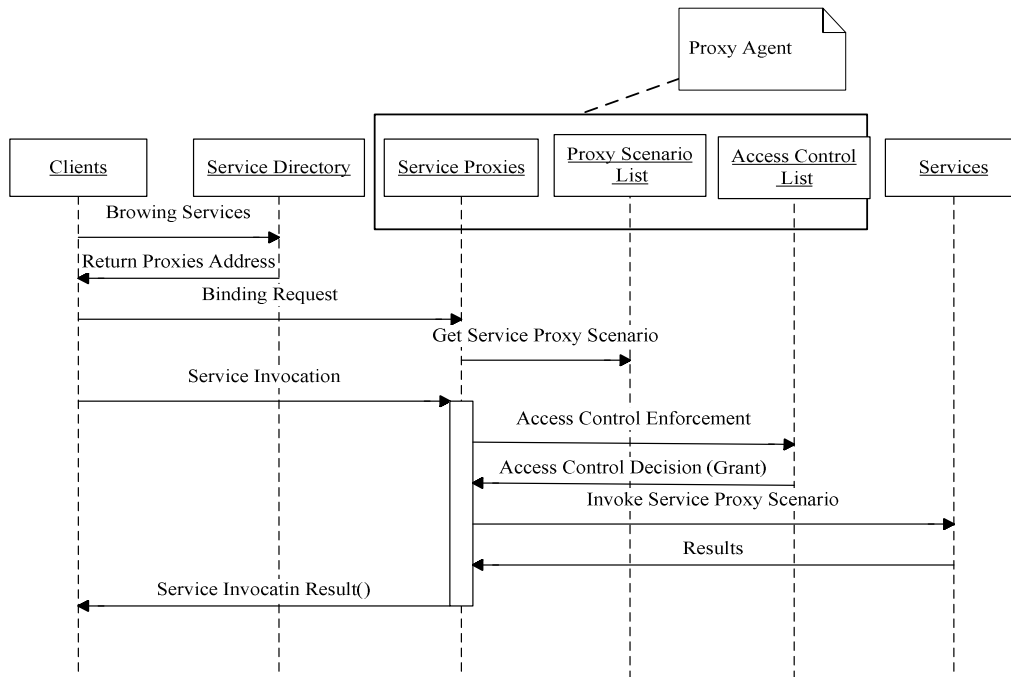


Figure 6 Interaction of Service Invocation via Proxy Agent

“2)String echome(string input)”, “3)String echoback(string input)”, respectively. So, the DRS will define an abstract service dedicated for this type of service “Echo” and have an standard interface definition as “string echo(string input)” which is the only interface for the client application. The proxy scenario for each of the services can be defined as:

ProxyScenario_EchoService1_echo

Action: echo

ProxyScenario_EchoService2_echo

Action: echome

ProxyScenario_EchoService3_echo

Action: echoback

The PAs in the DRS framework provide flexibility, interoperability, service integration, and service reconfigurations. Figure 6 shows the interaction of service invocation via proxy agents.

3.3 Auditing Agents

An AA monitors the status of the participating services at runtime. Each participating service will have at least one AA, but an AA may audit multiple services. An AA has the following behaviors:

- An AA is created whenever there is at least one service is bounded to a client through a PA;
- An AA monitors the status and performances of services, and notifies the DRS in case of failures or overload.
- An AA is also responsible for creating and maintaining a profile for each active service to keep track of its performance and usage patterns.
- Standard auditing scenarios are defined for each type of services and SPs can provide additional auditing scenarios and data for better decision making. Two common auditing scenarios are:
 - Hello Auditing Scenario: this detects the liveliness of the concerned service periodically
 - Threshold Auditing Scenario: this is used for performance or other constraints-related audits. When the reported data are outside the normal range, the AA will report immediately to the DRS to initiate a reconfiguration. For example, an AA for a video multicast service will inform the DRS if the packets/frames lost rate exceeds a certain limit.

Figure 7 shows a defined auditing scenario for a service that will send a Hello message periodically to the server application and wait for its reply to arrive within a certain time. If the service fails to respond for three consecutive times within the time window, the AA will notify the DRS that service is temporally unavailable (due to network congestion

or service failure, or overloaded) and needs to be switched off. Once this scenario is defined, it is exported to an AA for execution.

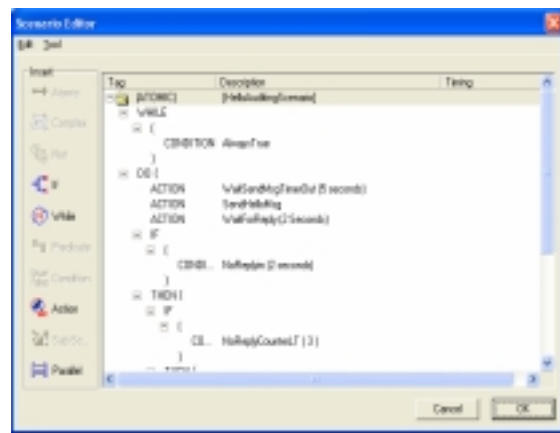


Figure 7 Auditing Scenario

4. Reconfiguration

A DRS will temporarily remove a service from active duty when it detects that the service is not responsive. The DRS will then assign the related tasks to another service to continue the operation at runtime. Dynamic reconfiguration can be initiated in at least two circumstances:

- Initiated by clients

An SS can initiate a dynamic reconfiguration by requesting its tasks to be performed by another service listed in the same SD. If SS does not identify a specific service in the SD node, the DRS may choose another service with the minimum recent workload. If all the registered services are working in an overloaded mode, the DRS may look up other services registered at another DRS.

- Initiated by the DRS

Once an AA detects that the concerned service is not working properly, it will inform the DRS immediately. If the service does not recover from the situation within a predetermined time span, the DRS will remove the service from the SD and transfer its tasks to another service in the same SD node.

The DRS will then notify the PA to complete the reconfiguration. Figure 8 shows an example of the interactions among PAs, DRScordinator and AAs during dynamic reconfiguration.

Furthermore, it is possible that participating agents such as AAs and PAs may fail, and even the DRS may also fail or become overloaded. This problem is addressed by redundant DRSs and

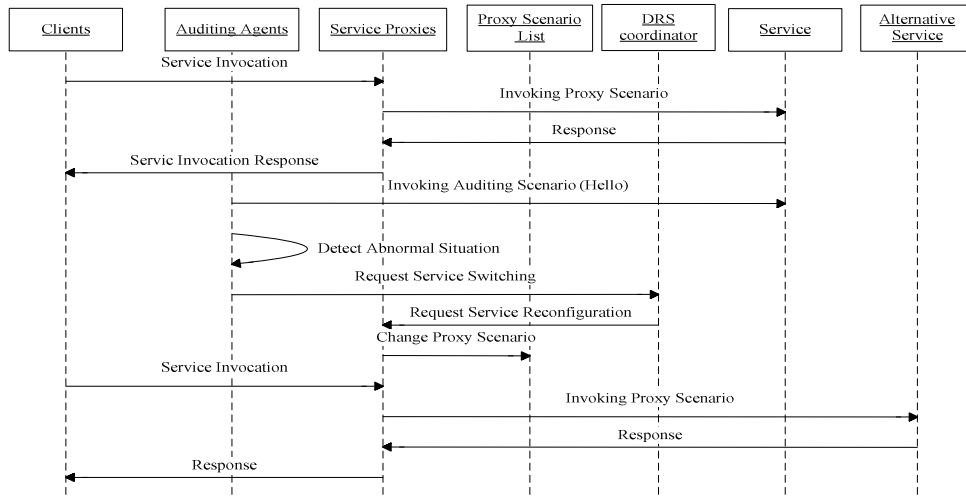


Figure 8 Interaction of Dynamic Service Switching

periodic review by the DRS on AAs and PAs. Specifically, the DRS monitors the status of AAs periodically, and if a certain AA does not respond within a certain time span, the DRS may initiate a new AA to replace the one that failed. Similarly, the DRS also monitors the status of PAs periodically, and if a certain PA fails to respond within a time period, the DRS will initiate a replacement PA. Note that a PA is also a service, and thus it has its corresponding AA, and when a PA fails, its AA will report immediately to the DRS. Similarly, the DRS is also a service, and its AA will report the failure of the DRS to another DRS in case of failure.

In our design, DRS is the core of fault-tolerance. Like all fault-tolerant systems, we face the dilemma: who is watching the watch dog? Basically, we can tolerate fail-stop failures of DRS through redundancy. However, we have to assume that DRS may not have malicious failure mode that sabotages the operations of the system. This is a reasonable assumption because the DRS is a system agent that may not be installed by clients.

5. Illustration and Experiments

In this section, we illustrate the process of dynamic service reconfiguration in the scenario of service failure and how the DRS framework and the software are evaluated to determine their effectiveness and performance. We developed the DRS framework and its software using the .NET platform. The .NET platform is chosen because it provides many features related to the SOA and it is service-oriented.

The experiment is based on the Travel Agent case study described in [9]. The TA is designed to allow the users to plan and reserve trips over the web. A high-level structure of TA is shown in Figure 9. The TA system is developed on the Web services architecture, where Flight reservation, Hotel booking and Car rental components are implemented as the web services. In this experiment, several external events including service failures, disconnected network are generated to evaluate the DRS. These external events are generated so that the DRS needs to keep on reconfiguring the system in response, and the DRS is evaluated according to speed of reconfiguration.

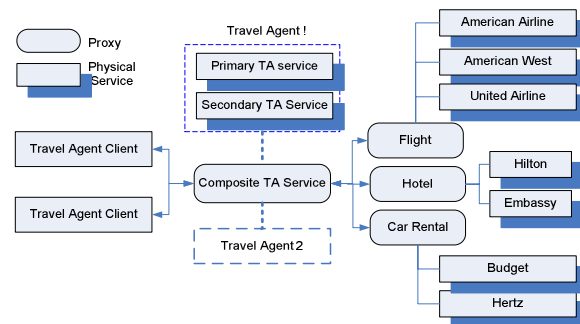


Figure 9 Travel Agent Web Services

First, the service directory definition for the TA application is shown as Figure 10. For each reservation service, multiple alternative certified

services are available in the Service Directory.

Second, a client binds to the service proxy for the TA Web services and it in turn binds to other sub-services. It invokes the TA Web services periodically via standard TA interface defined in [9] from service proxy. Then the service proxy invokes the corresponding proxy scenario, for example "PSTA_booking", which simply maps the signature of booking to that of the chosen TA Web services in this case it is TAWebService2. In the meantime, Auditing Agent invokes the Hello Auditing Scenario periodically to monitor the runtime status of all TA services.

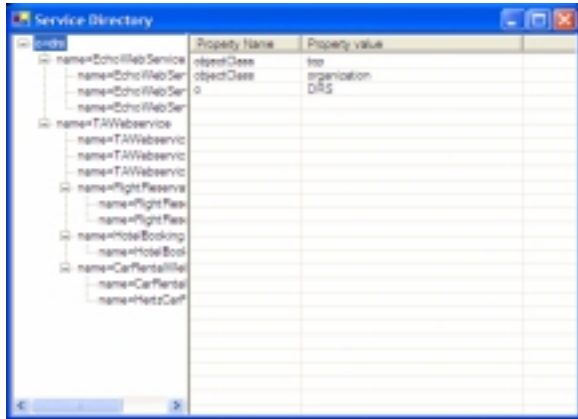


Figure 10 Service directories with defined service tree

Third, a service failure has been detected by AA for TAWebService2. It is reported to the DRS coordinator immediately. DRS coordinator chooses one service from the remaining normal running certified TA Web services, notifies the Service Proxy to reconfigure proxy scenario, and points to new service, which in this case it is TAWebService3. Figure 11 shows the failure detection interface.

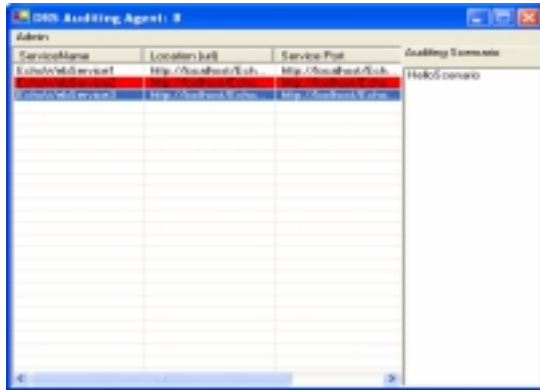


Figure 11 Failure Detection by Auditing Agents

Fourth, a new service request comes in, and it is redirected to the new TA Web Service without the awareness of clients and the rebinding of service at client side. In addition, if the auditing agent detects that the original service is back to normal, it is up to the DRS to decide whether the switched-off connection from clients need to be switched back. In this case, a jittering scenario is needed to handle the possible frequent service up and down in a short time period.

The experiments ran three TA Web services, each on a separate PC. In addition, several instances of client application and one instance for each of DRS-coordinator, auditing agent and proxy agent ran on an additional machine. All the PCs used have the same configurations. The timeout of auditing is set up for every 3 seconds. The client invokes the TA services through proxy services every 5 seconds. We measured the responsiveness of DRS based on dozens of trials. The experiments shows that on average it took about 3.253 seconds to detect and confirm a failure, and it took only 0.197 second to complete the reconfiguration.

Also note that the detection time is close to the auditing time interval which is set to 3 seconds. In other words, it took additional 0.253 second to detect a failure after the auditing interval. If the auditing period is smaller, it is possible to reduce the detection time but this will add communication overhead.

Further implementation and experiments are being performed including the study of the appropriateness of real-time reconfiguration decisions including utility of resources and available services is being underway.

6. Conclusion

This paper proposed a services-oriented dynamic reconfigurable framework for dependable distributed computing. In this framework, every process is a service, either an atomic service or a composite service. Furthermore, each service is uniformly defined using the same ISC service specification technique. This work also explored the feasibility of utilizing the ISC specification technique that we proposed in our previous work. ISC is an extension of WSDL to represent the dynamic reconfiguration model of SOA, The DRS framework is an extension of the existing WS architecture. In both cases we added many runtime features, including verification, auditing and reconfiguration functions, into the system. A framework has been built and experiments on the

framework proved the feasibility and effectiveness of our approach.

Reference:

- [1] R. Allen and D. Garlan. "Formalizing Architectural Connection", Proc. of IEEE International Conference on Software Engineering, 1994, pp. 71-80.
- [2] M. Aksit and Z. Choukair, "Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision", Proc. of IEEE International Conference on Distributed Computing Systems Workshops, 2003, pp. 84 -89.
- [3] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, "Scenario-based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing", Proc. of IEEE WORDS, 2002, pp. 140-151.
- [4] D. Cotroneo, C. Di Flora, S. Russo, "Improving Dependability of Service Oriented Architectures for Pervasive Computing", Proc. of WORDS, 2003, pp. 74-81.
- [5] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong. "Survivability through Customization and Adaptability: The Cactus Approach", DARPA Information Survivability Conference and Exposition (DISCEX 2000), 2000, pp. 294-307.
- [6] T. Howes and M. Smith, LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol, Macmillan Technical Publishing, 1997.
- [7] J. C. Knight, K. J. Sullivan, M. C. Elder, and C. Wang, "Survivability Architectures: Issues and Approaches", DARPA Information Survivability Conference and Exposition-Volume 2 (DISCEX 2000), 2000, pp. 1157-1171.
- [8] D. C. Luckham, "Specification and Analysis of System Architecture Using Rapide", IEEE Transactions on Software Engineering 21 (4), 1995, pp. 336-355.
- [9] P. Periorellis, J. Dobson, *The Travel Agent Case Study*, DSoS Project, IST-1999-11585, 2001.
- [10] J. M. Portilo, "The Polyolith Software Bus", ACM TOPLAS (16) 1, 1994, pp.151-174.
- [11] A. Troelsen, C# and the .NET Platform, Addison Wesley, Reading, MA, 2001.
- [12] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing", Proc. of IEEE HASE, 2002, pp. 171-172.
- [13] W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, "Verification of Web Services Using an Enhanced UDDI Server", Proc. of IEEE WORDS, 2003, pp. 131-138.
- [14] W. T. Tsai, C. Fan, R. Paul, and L. Yu, "Automated Event Tree Analysis from Scenario Specifications", Proc. of IEEE ISSRE, 2003, pp.240-241.
- [15] W. T. Tsai, L. Yu, R. Paul, C. Fan, and X. Liu, "Rapid Scenario-Based Simulation and Model Checking for Embedded System", Proc. of International Conference on SEA, 2003, pp 568-573.
- [16] W. T. Tsai, R. Paul, and L. Yu, A. Saimi, and Z. Cao, "Scenario-Based Web Service Testing with Distributed Agents", IEICE Transaction on Information and System, 2003, v.E86-D, no. 10, pp.2130-2144.
- [17] D. Wells, S. Ford, D. Langworthy, and N. Wells. "Software Survivability", DARPA Information Survivability Conference and Exposition-Volume 2 (DISCEX 2000), 2000, pp. 1241-1255.
- [18] W3C, Web Services Architecture Working Group. <http://www.w3.org/2002/ws/arch/>.