An Expressive Approach to Distributed Applications Dynamic Adaptation

Abdullah O. Al-Zaghameem
Technische Universität Berlin, Software Engineering Department
Berlin, D-10587, Germany

Abstract
Dynamically adaptable distributed applications need to be composed in an expressive and modular fashion due to the complexity of these applications. This paper discusses the shortcomings of recent approaches to achieve this goal, in particular the aspect-oriented programming approaches. It addresses the requirements for consistent and modular dynamic adaptation of applications, while improving their modularity. Then, the Remote Role-Playing (RRP) concept is presented as a new promising programming technique, which aims at employing the separation of crosscutting concerns in distributed applications dynamically at runtime in a modular and consistent manner with high degree of expressivity. The paper introduces the DOT/J framework which implements the RRP. The feasibility of the DOT/J approach and its advantage over other approaches is demonstrated through a case study.
Keywords: *dynamically adaptable applications, distributed applications, distributed-AOP, remote role-playing, dynamic aspects weaving.*

1. Introduction

The growing complexity of distributed applications, as well as changes in their execution environments, demands for applications that are more adaptable and easy to compose, evolve, and maintain. Currently, the demand for developing dynamically adaptable distributed applications is highly increased as the environments on which these applications execute become more mobile and changeable. Causes to this change might include variation in network bandwidth and network topology, and the desire to employ new algorithms in legacy applications. The dynamic adaptation of distributed application objects is the process of enabling these objects to change their behaviors dynamically at runtime as a response to changes in their execution environment. Lately, the Aspect-Oriented Programming (AOP) [1] technique has been employed in distributed programming due to its prosperity to improve applications' modularity. It allows separating those crosscutting concerns that are tangled and/or scattered in application code, and capturing them in *aspects*. For this purpose, several distributed AOP approaches like AWED [2], JAC [3], DJcutter [4], etc. have been developed. These approaches enable application objects to adapt through replacing their methods with new code segments called *advices* at specific points designated by remote *pointcuts*.
From a perspective of the AOP, the dynamic adaptation of distributed applications imposes weaving aspects dynamically at runtime. The current distributed AOP models lack supporting for consistent dynamic aspects weaving [5]. This lack has emerged primarily from the necessity to weave aspects into distributed application objects *atomically* [6]. Besides, in these approaches there is no explicit representation of the context in which aspects are applied, which reduces their expressivity to specify in a clear and understandable manner how distributed applications can adapt and when? In addition, application developers do not have the proper mechanisms to control the effects of aspects on application objects dynamically; because most of these approaches do not allow aspect instances to be accessed explicitly.

Object Teams [7], on the other hand, is a programming model that implements the collaboration-based (role-based) design [8, 9, 10 and 11] for the object-oriented languages. It employs the AOP concepts to separate the collaborations that crosscut application core classes. Object Teams (OT for short) captures collaborations in modules called "*teams*", and the participation of application objects inside these teams within modules called "*roles*." A team module acts as the context in which roles are *played by* application objects. A specific role could be played by application objects (called base objects) of a specific class type through binding them through the "playedBy" relationship. Application developers can declare roles to comprise new functionalities, which are considered as extensions of the behavior of player objects. This allows the behavior of player objects to be changed, which is a key feature to achieve objects' behavioral adaptation.

This paper maps the fundamentals of the OT model to distributed environments. Through employing the expressive "playedBy" relationship in distributed computing, the expressivity of distributed application adaptation will be enhanced in the name of role-playing; because the "playedBy" relationship is readable and understandable without developers being familiar with the OT's infrastructure. Thus, the modularity of distributed adaptable applications will be enhanced, and distributed aspects (roles) could be controlled and managed at runtime in a modular way. For example, aspects effects on base objects could be controlled by using specific constraints called guard predicates [12] at different abstraction levels. Furthermore, the applicability of aspects is governed by the dynamic activation and deactivation of team instances at runtime [7, 12].

This paper is organized as follows: in Section 2, a quick survey of the dynamic adaptation literature will be presented. In Section 3, the requirements of the dynamic adaptation of distributed application will be addressed. Section 4 introduces the Object Teams model and highlights its fundamental

features especially the expressive "playedBy" relationship. The Remote Role-Playing concept will be introduced in Section 5. Section 6 presents the DOT/J framework. A case study to demonstrate the RRP technique in dynamic adaptation of distributed applications will be established in Section 7. In Section 8 the related works will be discussed, and Section 9 concludes.

2. A Brief Survey of the Dynamic Adaptation of Applications

In general, adaptability of object-oriented applications is the ability of their objects to modify their behaviors and/or structures to adapt to changes in their execution environments. The adaptation of applications aims to empower them to obtain a desirable level of performance (or to enhance it) [13], or to improve the objects' adequacy to execute new tasks and conform to new requirements. The dynamic adaptation of applications (or the *runtime system evolution* as denoted by Taylor *et. al*. [15]), on the other hand, is the capability of their objects to adapt at runtime without the need to re-engineer the source code of these objects or interrupt application execution. This definition of adaptation should be applicable for distributed applications as well.

However, a more comprehensive definition for the behavioral adaptation of distributed application objects could be formulated as follows:

> "The dynamic adaptation of the distributed object **Obj** is its capability to adapt to changes in its execution environment by changing its behavior through replacing a specific functionality (i.e. method) with new one, satisfying the following conditions:
>
> - Adaptation must be performed dynamically at runtime.
> - **Obj** must keep its original functionality (i.e. its original behavior) and be able to reclaim it if the cause of adaptation has no longer applied.
> - Consistency of **Obj** and the entire application must be ensured before and after adaptation."

As denoted by Ben-Shaul *et. al*. [16], dynamic adaptation of applications should be considered at two levels:

Adaptability of the individual application objects: at this level, adaptability means changing the behavior of application objects through providing them with new functionalities. These functionalities should empower them to behave differently. An interpretation of adaptation from this perspective is to modify the architecture of objects' classes. This has been achieved in several approaches at different granularities of modification. That is, approaches like PROSE [17] can replace entire aspects with new ones at runtime. In iPOJO [18], Plain Old Java Object (POJO) components could be glued to base components through "handlers" to compose service-oriented applications. Others like HADAS [19] enable application components to perform the change through the meta-methods they inherit from the parent component of all HADAS components. Thus, new methods could be added and other existing methods could be removed at runtime from that component.

In the distributed-AOP scope, approaches like DJcutter [4] (which extends AspectJ [14] with remote pointcuts and remote advice execution) support so-called inter-type declarations, which enable application developers to perform "internal structure modifications" like injecting new methods and fields into application target classes, and "compositional structure modifications" like defining new interfaces for the target class. Thus, the structure of target classes is modified. Though, this can cause several problems like field and methods ambiguities, especially when several aspects target the same base class. For example, inter-type declaring a public method in a specific target class can cause compile-time conflict if that class already implements a method with the same signature [20].

Other AOP approaches like Lasagne [6] and DandyJ [5] have introduced a dynamic structural adaptation of objects at runtime by using dynamic aspect weaving mechanisms, which allow developers to weave and unweave aspects from running applications [21]. Practically, problems of consistency of aspects and base objects could be occurred during aspects weaving and unweaving if aspects have inter-dependencies with each other.

Aspects with Explicit Distribution (AWED) [2] is a distributed AOP language that extends JAsCo [22] (a dynamic AOP language tailored for the component-based models). AWED has proposed remote pointcuts definitions and a mechanism to execute advice codes at remote hosts. In this way, distributed application objects can change their behavior through executing new code segments, which are confined in constructs called *hooks*. A hook is a generic pointcut definition that could be reused to trap different base functionalities (methods). The advices of a specific hook are woven dynamically into application objects at runtime via *connector* constructs. A connecter can bind at runtime exactly a base method to a specific hook instance. This allows the dynamic adaptation of base objects in the sense new advices could be woven at runtime to replace the old base methods. In this regard, JAsCo enables application developers to create *combination strategies* to filter the list of applicable hooks at a specific point of execution, which is important to control, for example, adaptation priorities. This technique, however, might not be applicable in distributed applications due to the uncontrolled nature of distributed environments. For example, if a combination strategy in AWED is applied to remove the hook X from hooks list if the hook Y is already there, and to add X if Y does not exist, then a deformation in application functionality will occur if hook Y has been initiated at host H1 but (due to congestion) not yet initiated at H2. In this case, the hook X will be initiated at H2 while Y is still applied at H1. A primary reason for this inconsistency is that AWED simulates remote advice executions by executing advices of local aspect copies deployed on each host.

Adaptability of application objects' inter-relationship: adaptation at this level targets the contextual changes of application objects like adjusting objects' locations due to migration over the network from one node to another. Therefore, the adaptation affects mainly the communication environment of the interacting objects and not the objects themselves (unless the contextual changes are part of the objects' states). This research focuses on the first level.

3. The Requirements of Dynamic Adaptation

In adaptation of distributed applications, Fransisco *et. al.* [13] recommended that developers should consider, in addition to the implementation of distributed application functional behavior, some other underlying issues like monitoring of resources usage and application-specific interactions, specify which environment elements should be monitored, how to detect such environmental changes, and which software adaptations should be handled and when?

In the scope of AOP, a key concept to achieve the dynamic adaptation of distributed applications is the dynamic weaving of aspects. Therefore, new essential requirements have emerged. These requirements include:

- *Atomicity*. As defined by Truyen and Joosen [6], atomicity is the process of adding or removing the mutually dependent functionalities in an "all or nothing" fashion. That is, with application objects dispersed over different nodes, a specific aspect must be woven into all target objects at the same time. Otherwise, it must not be woven in any of them at all. Current distributed-AOP approaches lack constructs that support consistent and atomic weaving of aspects [5].
- *Consistency*. Here it means the consistency of individual distributed application objects and the entire application after dynamic adaptation has been applied.
- *Dynamicity*. That is, distributed application objects must be able to adapt at runtime without stopping their execution.
- *Continuity*. Application objects must be able to adapt by changing their behaviors even further.
- *Expressivity*. Current AOP approaches have introduced "aspect" as the top-level unit of modularity to solve the problem of separation of crosscutting concerns. Thus, to exploit aspects in applications, developers pay most of their efforts in formulating pointcut expressions, and the way to glue the associated advice codes to base objects when joinpoints are matched. Thus, there is no clear relationship between aspects and application base objects could be designated especially when a single aspect affects several base objects in the name of reusability. This relationship is not clear due to lack of these approaches an explicit representation for the context where aspects are applied. From a dynamic adaptation perspective, this makes the process of expressing adaptation via aspects hard and/or incomprehensible.

4. An Overview of the Object Teams Model

The origin of the OT model could be ascribed to the collaboration-based design, which describes a methodology for decomposing object-oriented applications into a set of classes and a set of collaborations [11]. The diagram shown in Figure 1 (a) illustrates how this design could be organized in a two-dimensional composition: in the vertical dimension application base classes, and collaborations in the horizontal dimension. The OT model benefits from the observation that collaborations are crosscutting application classes at specific parts of their behaviors. Then, it applies the AOP concepts to separate and modularize this "crosscutting" as follows: collaborations are captured in "team" modules, and the intersections between a specific collaboration and application classes are captured in "roles." Consequently, the design of Collaboration1 shown in Figure 1 (a) could be redesigned in OT as shown in Figure 1 (b).

4.1 Improving Applications' Modularity

The OT model presents a new concept for applications' modularity via so-called *role-playing*. Let us take a simple example to demonstrate this in the context of applications adaptation. In Figure 2, **Client** class is a *base class* of a distributed application; it can send text messages to a specific server (not shown in the figure). Suppose that a simple encryption mechanism needs to be applied so that client objects encrypt their messages before they are sent. In the OT model, this could be represented as a *role* of **Client** class which it can play in a specific team; say **EncryptionTeam**. As shown in the figure, **Client** class is bound to the **EncryptedClient** role via the "playedBy" relationship. Note the high expressivity of this relationship. The **EncryptionTeam** team can declare attributes and methods. Likewise, the **EncryptedClient** role can declare

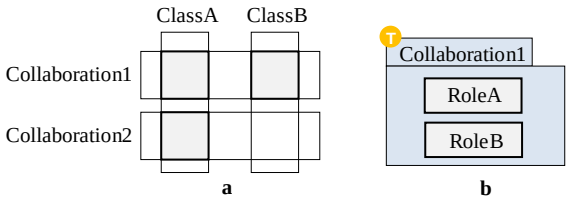

Fig. 1 A collaboration-based design in (a), and in (b) a representation of Collaboration1 in the OT model.

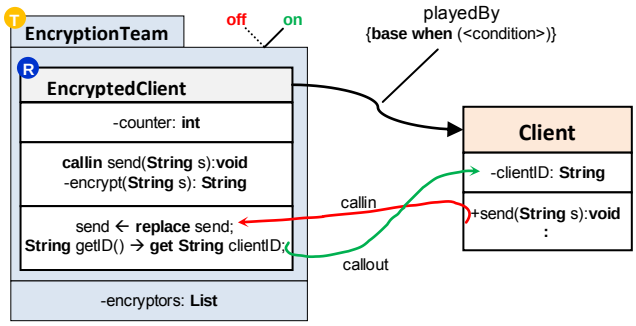

Fig. 2 A simple role-playing design in the OT model.

its own attributes and methods, which gives roles a kind of autonomy. This autonomy, though, is dominated in the OT model by the *role confinement rule* [12]; which states that roles could not be existed outside the boundaries of their enclosing team. Thus, the **EncryptedClient** role is drawn inside the **EncryptionTeam** team.

The "playedBy" relationship involves two types of binding between a role and its base class, namely: the callin method binding and the callout method binding. A callin binding enables base object to *call into* role instance a specific method either *after*, *before*, or in *replacement* of a call to one of its methods. For example, the callin expression {send ← **replace** send;}, shown in Figure 2 in the role class diagram, instructs a **Client** object to call the method send of the role instance it will be bound to *instead* of its original send method. This concept motivates adapting application objects through enabling them to play roles in teams. The role's methods that are bound via replacement callin expressions are called *callin methods*, and must be designated by the keyword **callin** as shown in the figure.

The other binding type is the callout binding which indicates that a role instance can declare a method (which is not available locally) by *calling out* to a method in its associated base object. In the OT model, three sub-types of callout bindings could be declared: a callout to a base method, a callout to get a specific field value (called field getter), and a callout to set a value of a specific base field (called field setter). For example, the callout expression {**String** getID() → **get String** clientID;} dispatches the calls made to the method getID on a role instance to get the current value of the clientID attribute of its base object.

4.2 Constraining the Role-playing Process

To control the process of role-playing and the effects of role's callins on application base classes, OT has been equipped with two mechanisms to accomplish this task:

Teams' activation/deactivation*:* This mechanism indicates that the declared "playedBy" relationships in a specific team are considered applicable if, and only if, an instance of that team has been activated. Contrariwise, the role-playing process will not take place if the team instance is deactivated. In Figure 2, the team activation/deactivation process takes the shape of an On/Off switch. In practice, this facility describes how could developers control when application objects could adapt if, for example, the teams' activation/deactivation status has been linked to the conditions of execution environment changes.

Guard Predicates are conditional expressions that could be attached along the "playedBy" relationship, role methods, or callin and callout expressions. The idea is simple, yet powerful; if the "playedBy" relationship is augmented with a guard predicate expression (see Figure 2), then binding base objects and role instances will not take place in the team instance if the expression evaluates to "false"; even if the team instance found to be activated at the binding-time.

If the guard predicate expression access any of base object's fields or call any of its methods, then it is called *base guard-predicate*. In this case, the keyword **base** is used as a placeholder for the bound base object.

4.3 The OT/J Programming Language

OT/J [12] is the programming language that implements the OT model in Java. Developers can use the keyword `team` to declare team classes. Roles, on the other hand, did not require a special keyword to declare them; rather, they are normal inner classes. In this way, any class or interface to be declared inside a specific team class is considered role.

5. The Remote Role-Playing

This section introduces a new programming concept called the Remote Role-Playing (RRP). The RRP aims at enabling objects of distributed applications to play different roles dynamically at runtime and remotely from any application node in a transparent fashion. This provides a modular and expressive mechanism for dynamic adaptation of distributed applications. For this purpose, the Distributed Object Teams for Java (DOT/J) [23] framework has been developed. The DOT/J framework implements the RRP concepts by extending the OT/J infrastructure with load-time transformation library and distributed runtime system.

5.1 Why Remote Role-Playing?

Consider that several client objects have been deployed on different network hosts. An encryption/decryption mechanism could to be represented as a role-playing process as discussed earlier. But, employing the "playedBy" relationship at the hosts on which client objects reside imposes the deployment of a copy of **EncryptionTeam** team instance at each of these hosts. This technique is adopted often by most distributed-AOP approaches. Anyway, the deployment of team instance copies in this way can cause the following problems:

The activation and deactivation of team instance copies (and in general their states) must be synchronized at all nodes. In the context of dynamic aspect weaving, this is important so that client base objects can play roles *atomically*. That is, if the team copy deployed on host H1 has been activated first, then all other team copies *must* be activated simultaneously at an accurate time of execution. This may result in plain-text messages to be sent by other clients at different hosts if the team copies deployed on these hosts have not yet activated (for some reasons like network congestion). Moreover, the encryption process (i.e. playing the **EncryptedClient** role) must be sparked after the decryption process has been ensured to be turned on, and not before. In addition, it will put extra

charges to preserve team copies consistent, which complicates the entire role-playing process.

Re-Engineer Team classes. A small change to the team class imposes recompiling and redeploying new team copies on all hosts, which might lead to functionality violation if a new team copy has not been deployed at a specific host unintentionally.

Fracture in Roles' Inter-relationships. This problem could be encountered when a team class encloses two associated roles, which are played by two different base classes. Thus, this situation could not be employed in a distributed environment because instances of the first role will be bound to their base objects at a specific host, and the instances of the second role will be bound at another host (on which a team copy has been deployed).

Therefore, the RRP aims at enabling the deployed client objects on all hosts to play the **EncryptedClient** role in the same way they would play it locally (i.e. preserving the semantics of the "playedBy" relationship). Practically, to overcome the aforementioned problems, a single team instance needs to be deployed. Thus, the activation and deactivation of that team instance will simultaneously affect the role-playing process. That is, roles will be played atomically by application base objects.

5.2 The Requirements of RRP

A deep look at the local "playedBy" relationship reveals that in order to enable a specific base object to play a role in a specific team instance, first that team instance should bind the base object to a role instance. In the binding process, role instance preserves an *immutable* reference to base object; thus role instance can issue callouts. To perform role's callins from the base object, an immutable reference for the team instance must be maintained; because role instances could be accessed only by the enclosing team as the *role confinement rule* said.

In case of separated base and team instances, only *remote references* could be used. This calls for *reformulating* role classes so that their instances can handle callouts via the remote references of *remote* base objects. Likewise, base and team objects should be taught how to perform role's callins via remote references. Also, base guard predicates must be evaluated in team instances using bases' remote references.

From a perspective of dynamic adaptation, application base objects should be enabled to play new roles at runtime. Therefore, a mechanism to enable developers to employ new teams without interrupting application execution or violating base objects integrity is needed. Another mechanism is required to enable base objects to allocate the new employed teams properly, and to play new roles accurately.

The current implementation of OT/J prevents base objects to play new roles at runtime. This is because OT/J weaves roles into application base classes at load-time, which results in a cohesive role-base link that could not be changed without reloading application classes in order to weave new roles.

Finally, current OT/J version allows the binding of roles to base interfaces if and only if these roles declare only callout bindings [12]. But distributed objects of object-oriented distributed applications mainly represented by *remote interfaces* [24, 25]. Thus, binding roles to remote interfaces at the source code level will result in partial binding between their instances at runtime; hence behavioral adaptation becomes impossible. The next section will introduce the DOT/J framework [23], which fulfills the requirements of the RRP and resolves the obstacles mentioned above.

6. The DOT/J Framework

In the preliminary paper [23], DOT/J has been introduced as an extension to the OT/J language into distributed applications. This section re-introduces it as a realization of the RRP.

6.1 Prerequisites

A key concept to realize the RRP is to *replace* the local "playedBy" relationship, which binds local role and base classes, with a remote relationship that can bind remote role and base objects. This replacement includes the following:

- *Adapt application base objects to play roles remotely*. That is, convert all application base classes, which are involved in RRP relationships, into RRP-ready classes. This conversion must not violate the original functionality of the objects (hereafter remote base objects) generated from these base classes (hereafter remote base classes). At the same time, it should allow remote base objects to play new roles dynamically. This means the RRP-ready conversion should be generic as much as possible in order to enable any (*unanticipated*) role to be played; hence, an accurate and continuous dynamic adaptation is achieved.
- *Adapt remote roles to be played remotely*. This involves mainly replacing the local features of role-playing in these roles (hereafter remote roles) with remote features. More specifically, remote roles should hold remote references to their remote base objects, and issue callouts accurately via these references.
- *Adapt teams to facilitate the RRP*. First, any team class that encloses at least one remote role, or the one which its instances need to be accessed remotely, is called *remote team class*. To enable remote base objects to execute the callins of their remote roles inside a remote team instance, an immutable remote reference for that team must be obtained by base objects. Before a remote base object can play roles, it should be first bound to role instance in a process called *Lifting* [12]; when a base object enters the boundaries of team, it is lifted to a corresponding role instance. But, since only remote references of remote base objects are available, remote team instances must be *taught* to handle the lifting process in different way.

6.2 The DOT/J Infrastructure

The diagram shown in Figure 3 demonstrates the general structure of the DOT/J framework. The DOT/J framework consists mainly of two sub-systems:

The DOT/J Transformation Layer (DTL): is responsible for converting remote base and remote team classes (including remote roles) into "RRP-ready" classes. To perform this, the DTL extends the Object Teams Runtime Environment (OTRE) of OT/J by three load-time bytecode transformers. The first one called the Remote Base Bytecode Transformer (RBBT), which is dedicated to transform the bytecode of remote base classes. The second one is dedicated to transform the bytecode of remote team classes and called Remote Teams Bytecode Transformer (RTBT). The last one is the Remote Role Bytecode Transformer (RRBT), which transforms remote role classes. The transformation process "equips" each of remote base and team classes with the necessary toolkits that enable their objects to communicate and exercise the RRP activities at runtime. For example, the RBBT injects into each base method a trap to dispatch the calls of methods to the Methods Dispatcher (MsD). The MsD is a central *call-by-reflection* method which receives the trapped calls of a specific base method and checks for any remote callin bindings associated to this base method.
The DOT/J framework uses the Java-RMI middleware [24] to establish the communication between remote base and remote team objects. Therefore, the DTL transformers weave into the classes of these objects the required code which enables them to export as remote objects according to Java-RMI disciplines.

The DOT/J Runtime System (DRS)*:* helps remote base objects and remote team instances to join and establish the necessary remote communication before the RRP takes place. More precisely, it registers transparently and automatically all the remote references (called *stubs*) of remote team instances, besides other supportive and contextual information about roles like the declared remote callins. The DRS is responsible for replicating this information at every application node via a reliable mechanism. The DRS subsystem comprises a constitutive component called the Distributed Objects and Teams Manager (DOTM), which acts as a registry of remote team instances. A DOTM component must be executed at

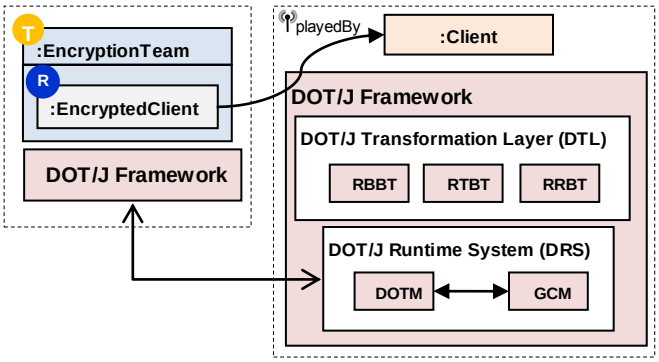

Fig. 3 The general structure of DOT/J framework.

a
```
1. <DOTJ>
2.     <RemoteTeam class="a.b.c.EncryptionTeam">
3.         <RemoteRole name="EncryptedClient"/>
4.     </RemoteTeam>
5. </DOTJ>
```

b
```
1.     <RemoteBases>
2.         <Base class="x.y.z.Client"/>
3.     </RemoteBases>
```

Fig. 4 Labeling the **EncryptionTeam** team as remote team class in (a), and in (b) the **Client** class is labeled as remote base class.

each application node. The DOTM can communicate with the remote base objects deployed on its local host. Thus, it can *notify* them with the registration of any new remote team instances. Also, it provides these base objects with the Remote Teams List (RTL), which includes all information required by these objects to *spark* the RRP process, upon their requests transparently. The DOTMs in a distributed application communicate with each other via the Group Communication Manager (GCM), which is using the JGroups system [30]; a reliable multicast groups communication system. The GCM establishes a communication protocol between DOTM components to replicate the RTL and keep the *role-playing map* consistent and up-to-date all the time.

6.3 Mapping Application Classes to the DOT/J Framework

In order to enable the DTL transformers to recognize which of application classes involved in RRP relationships, a simple XML-based language has been used to *label* these classes as remote classes. For example, the XML code shown in Figure 4 (a) illustrates how the **EncryptionTeam** team and the **EncryptedClient** role could be labeled, respectively, as remote team and remote role classes. Likewise, Figure 4 (b) illustrates how **Client** base class could be labeled as remote base class. When application classes are loaded into the JVM for execution, the DTL reads the XML file and intercepts the loading of remote classes, and then transforms their bytecode.

7. Case Study: Dynamic Encryption/Decryption in a Client/Server Messaging Application

Consider the object-oriented client-server messaging application shown in Figure 5. Assume that client and server objects are using Java sockets to establish communication. Client objects can send their plain-text messages via the "`send`" method. A server object is created for each client in a separate thread to receive, process, and respond to the

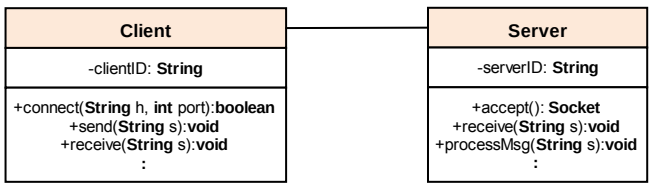

Fig. 5 A diagram of simple Client/Server messaging application.

messages sent by that client. After a while, and for security reasons, client and server objects need to apply a specific encryption/decryption mechanism. That is, a client object encrypts its messages before sending, and the dedicated server object must decrypt them before manipulation.

The AOP technique proposes to represent encryption and decryption (and others like fragmentation/defragmentation [26]) as crosscutting concerns [6, 27]. But, the actual challenge is in the employment of this requirement (i.e. the encryption/decryption concern) in the application *dynamically* and *consistently* at client and server. As mentioned by Michihiro *et al.* [5] weaving aspects dynamically in an executing application is a complicated and error-prone with the current "dynamic-AOP" approaches; because aspects need to be woven into multiple hosts atomically. Other interoperability problems like the unintended aspect effects, the partial weaving of aspects and unknown aspect assumption [17] could be emerged if the current distributed-AOP approaches are used.

To illustrate this issue, consider that the client-server application in Figure 5 has been deployed as shown in Figure 6, in which three client objects have been deployed each at a separate host. The dashed rectangles represent the physical distribution of objects. In this case, the encryption concern must be woven into the three client objects atomically and in a consistent manner. Furthermore, client objects should not start encrypting their messages before the decryption concern has already been employed. Otherwise, the application functionality will be seriously deformed. This section demonstrates how these problems could be resolved by using the DOT/J approach; in particular the RRP.

7.1 Modularize Application Adaptation in OT/J

In OT/J, the adaptation of application objects could be represented via role-playing in very expressive fashion. As mentioned earlier, the encryption/decryption requirement could be expressed as roles of teams; see Figure 7. The figure shows the **Client** class playing the role **EncryptedClient** in the **EncryptionTeam** team. Likewise, the server base class plays the **DecryptedServer** role inside the **DecryptionTeam** team. The key concept in DOT/J is to enable the deployed client objects shown in Figure 6 to play their **EncryptedClient** role *atomically*; i.e. all clients play the role simultaneously or none at all. At the same time, the encryption concern must not be *woven* and "put to the service" before the weaving and employing of decryption concern has been verified. This could be achieved in DOT/J via two modular steps:

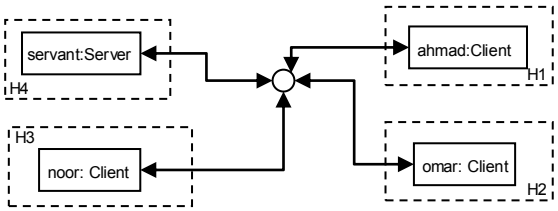

Fig. 6 Deploying client and server objects on a distributed environment.

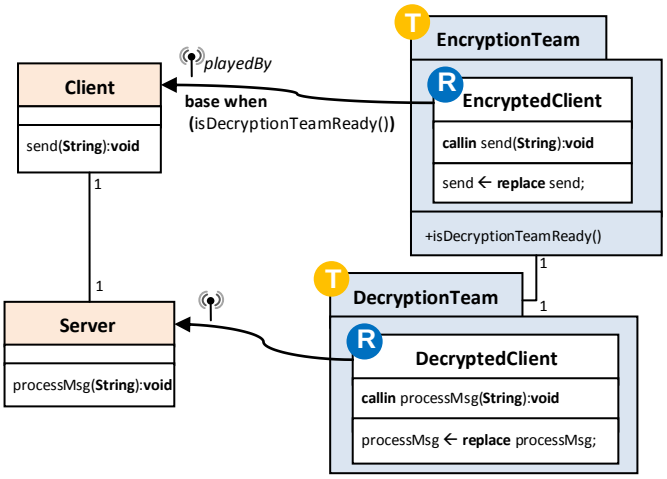

Fig. 7 Adapting the client-server messaging application through remote role-playing.

- Declaring a Base Guard Predicate (BGP) at the "playedBy" relationship in the **EncryptionTeam** team that ensures the deployment (weaving) and activation (employment) of a team instance of type **DecryptionTeam** in the application. Thus, the BGP shown in Figure 7 has been declared. The method "`isDecryptionTeamReady`" will be called just before any of client objects start playing the role.
- The **DecryptionTeam** team instance must be *activated* and *deactivated* by the **EncryptionTeam** team instance only. This means that when an instance of **EncryptionTeam** is activated, then the **DecryptionTeam** team instance (if it has been already deployed) will be activated. The same procedure will be applied in case of deactivation. This ensures that no message will be treated incorrectly by client and server objects.

The BGP will constraint the applicability of playing the **EncryptedClient** role; hence it controls the encryption callin (the advice), and determines when client objects can adapt. That is, if the BGP expression has evaluated to "false," then clients will not encrypt their messages. The callin declaration shown in the **EncryptedClient** role diagram of Figure 7 causes (when the encryption team instance is activated) all calls to the client object's method "`send`" to be intercepted, and then dispatches the control flow (via the MsD), along with the message, to the "`send`" method of role instance. At this point, the role instance encrypts the "dispatched" message, and then returns the control flow to the base client to *proceed* sending the encrypted message via a *base call*. To illustrate this, Figure 8 shows the implementation of the "`send`" callin method in the **EncryptedClient** role. At line 3, the message is encrypted, and then it will be sent via the original "`send`" method of the base object via the base-call in line 4.

```
1.  callin void send(String s)
2.  {
3.     String new_s = encrypt(s);
4.     base.send(new_s);
5.  }
```

Fig. 8 The implementation of "send" callin method.

7.2 Execution Scenarios and Performance Analysis

To verify and evaluate the dynamic adaptation of distributed applications by using the DOT/J framework, the distributed client/server application has been executed over a LAN network consists of two laptops; the first is "Sony-VAIO" with Intel® Core™2 Duo CPU 2.1GHz and is running Win7 Pro 32-bit SP1. The other is "Toshiba-Satellite" with Pentium dual-Core CPU 2.0 GHz and is running Win7 Pro 64-bit SP1. Both laptops have 4GB of RAM. The experiment has been implemented using OTDT v1.3.3 (which is augmented by the DOT/J subsystems) over Eclipse IDE v3.5.2 and Java 1.6.

First, the overload of using DOT/J framework has been measured by executing the client-server application with the DOT/J (i.e. the application is running over the DOT/J framework) and without. In both cases, a client object has sent 1000 messages to the server in a rate of 1message/100ms. The server application has been executed on the "Toshiba" laptop and the client application on the "Sony" laptop. The runtime values of this experiment have been recorded and represented in the charts shown in Figure 9. The average runtime value for sending the 1000 messages without using the DOT/J framework was 1.134ms, and with using the DOT/J framework was 1.882ms. This means that executing the application over the DOT/J framework adds – in the average – only 0.748ms as an overload at each sending call.

7.3 A Dynamic Adaptation Scenario

The actual dynamic adaptation experiment has been established as follows: on the "Sony" laptop, a client application is executed, and on the "Toshiba" laptop the server application. Before the client start sending its messages, a **DecryptionTeam** team instance has been deployed (in a separate application) on the "Toshiba" laptop. Then, a team instance of type **EncryptionTeam** is deployed on the "Sony" laptop (also in a separate application). After that, the client object starts sending 1000 messages in a rate of 1message/1ms (in order to evaluate the DOT/J framework in an intensive execution environment).

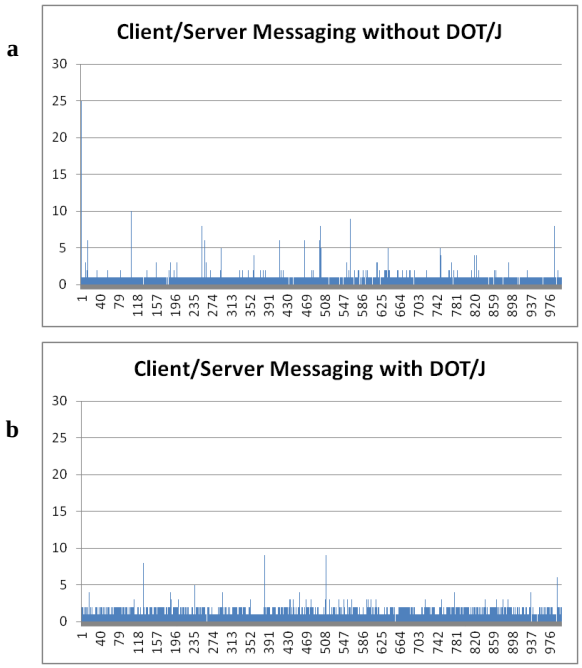

Fig. 9 Executing the client-server messaging application to measure the overhead of using DOT/J framework: in (a) without DOT/J, and in (b) with using DOT/J.

All messages have been encrypted and then sent. When arrived to the server, they have been decrypted and processed properly as if they have not been encrypted. The runtime to perform a complete *send-call* (RT_{send}) is calculated as follows:

$$RT_{send} = MsD_{time} + Enc_{time} + Send_{time} + Dec_{time} + Process_{time} + Ack_{time} \quad (1)$$

Where:

- MsD_{time}: is the time required to *trap* and *dispatch* the call of "`send`" method on client object to the Methods Dispatcher (MsD) and invoke the role's callin. On the first call to base "`send`" method, this time includes the runtime required to grip the Remote Teams List (RTL) from the DOTM.
- Enc_{time}: is the time needed to encrypt the message in hand by the role instance.
- $Send_{time}$: is the time needed send the encrypted message to the server (via the base call in line 4 of Figure 8).
- Dec_{time}: is the time required to *trap* the "`processMsg`" method of server object and decrypt the received message. Again, this time includes the runtime required to fetch the RTL for the **Server** class from its local DOTM at the first call to "`processMsg`" method.
- $Process_{time}$: is the time needed by server object to manipulate the message.
- Ack_{time}: the time required by client object to receive the server's acknowledgment for each sent message.

The runtime values of this experiment have been recorded and represented in the chart shown in Figure 10. The average runtime for sending the 1000 messages with the encryption/decryption mechanism is 7.04ms. This value indicates that the efficiency of DOT/J is higher than other recent distributed-AOP approaches like DandyJ [5], which has registered an average time of 49.6ms (a latency of 27ms between client and server nodes is included) for the same experiment. Regarding the development of experiment, it is

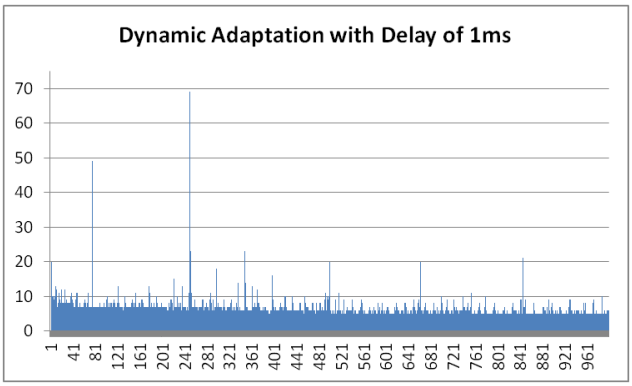

Fig. 10 Applying the dynamic adaptation of client-server messaging application in a delay time of 1ms.

clear that the application shown in Figure 7 is very readable. Besides, it requires implementing two teams and two roles; whereas in DandyJ, for example, more than six "dynamic aspects" are required to implement the same concerns. In AWED, for example, it is impossible to synchronize and control the employment of the encryption and decryption concerns as in DOT/J. Finally, DOT/J has the advantage of *transparency* over DandyJ, which uses explicit pointcut designators to determine the hosts on which joinpoints should be matched.

8. Related Works

In addition to the approaches discussed in Section 2, several approaches in the literature have been presented in the context of dynamic adaptation of applications. This section discusses some of them. A Two-Phased Aspect-Oriented (TPAO) solution to dynamic adaptation has been presented in [27]. The first phase (which takes place at the development time) identifies the points of adaptation in the application code. The second phase encompasses the activities concerning the actual adaptation, which includes: checking of adaptation's conditions and adding or removing the code of adaptation. This phase takes place at runtime. Anyway, one of the major difficulties for achieving an AOP solution for dynamic adaptation is how to make the existing application adapt-ready, i.e. to extend application so that new functionalities can be loaded and unloaded at runtime dynamically.

The problem of dynamic aspect weaving adopted in the TPAO approach is that old and new aspects could be overlapped during adaptation, which leads to unpredictable and/or undesirable behavior [28]. In DOT/J, the process of Remote Role-Playing (RRP) is controlled through the teams' activation/deactivation mechanisms, which is governed by activation precedence mechanism as well. Thus, application objects will always play their roles in the accurate order; therefore, always adapt precisely. Furthermore, dynamic adaptation of distributed applications like the one discussed in this paper could not be implemented with this and alike approaches; because synchronous aspects employment is not supported.

TRAP/J [29] is a software tool for enabling developers to add new adaptable behaviors to existing Java applications transparently without the need of their source code. It could be considered as an enhanced version of the TPAO approach. However, it operates in two phases as well. In the first phase, TRAP uses aspects to provide the necessary *hooks* to realize runtime re-composition of the application, and to produce adapt-ready program. At the second phase, new behaviors can be introduced via interfaces to the adaptable classes, which are *wrapper* versions of the original ones.

The problem in TRAP/J and alike is that they cannot support the unanticipated adaptation of applications. That is, continuity of adaptation is not fulfilled. In addition, to enable the adaptation of applications by changing functionalities that are not considered previously, application execution must be interrupted, and phase one must be repeated. Moreover, TRAP/J support adaptation of single applications and not distributed applications. Furthermore, it is not clear if application consistency is guaranteed in TRAP/J; because no mechanism is mentioned for resolving aspects precedence during the weaving process. Finally, wrapping the original classes could break the application architecture, and restricts the approach capabilities.

DyReS [26] is a Java-based framework for distributed dynamic AOP. It offers coordination support for distributed adaptation in aspect-oriented middleware. DyReS observes that coordinating the weaving and unweaving of multiple inter-dependent aspects is verbose and error-prone task because structural integrity and global state consistency need to be ensured. However, DyReS does not support remote pointcuts, and the use of XML – to describe how to control aspects weaving – reduces the expressive power of the framework and decreases the degree of transparency with respect to dynamic adaptation.

The using of XML description to control the dynamic weaving of aspects in DyReS cannot be installed during runtime [5]; rather, it must be installed statically on all nodes before the actual execution of application. The XML description of aspects weaving is used in DOT/J, however, only to map remote teams, roles, and base classes *once* to the framework. In fact, remote roles need to be mapped only on the node they were declared at. Furthermore, DOT/J does not require base-to-role bindings in the XML description. Thus, DOT/J maintains transparent deployment and employment of remote roles. Also, in DOT/J, new remote team instances can enroll dynamically during application execution. Thus, it supports a true dynamic weaving and employment of roles without stopping application execution or reloading its classes.

9. Conclusions

This paper has presented the DOT/J framework, which is used to propose a new approach for realizing the dynamic adaptation of distributed applications in modular and expressive fashion. The key concept introduced is the Remote Role Playing (RRP), which exploits the features of the Object Teams model in the distributed computing. The paper has demonstrated how the RRP can improve the modularity of distributed adaptable applications. The dynamic weaving of aspects at runtime, which is adopted by current distributed-AOP approaches, could lead to consistency problems if aspects have not been woven atomically. In the DOT/J approach, application base objects either play the same role simultaneously or none at all. The results of the experiments developed in this research reveal a promising approach, which could enhance adaptable applications modularity and improves their composition expressivity. This makes them easy to evolve and adapt.

ACKNOWLEDGMENT

The author would like to express his sincere thanks to Sadiq Abd Elall for his efforts in preparing the experiments established in this research.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, and *et al.* "Aspect-oriented programming." In Proceedings of the ECOOP'97. Springer-Verlag LNCS 1241, Finland, pp. 220-242. (1997).

[2] L. Navarro, M. Sudholt, W Vanderperren, and *et. al.* "Explicitly distributed AOP using AWED." In Proceedings of AOSD'06. ACM, NewYork, NY, USA, pp. 51-62. (2006).

[3] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. "JAC: A flexible solution for aspect-oriented programming in Java." In Proceedings of Reflection'01, volume 2192 of LNCS. Springer-Verlag, Sept. (2001).

[4] M. Nishizawa, S. Chiba, and M. Tatsubori. "Remote pointcut - a language construct for distributed AOP." In Proc. of AOSD04. ACM Press, (2004).

[5] M. Horie, S. Morita, and S. Chiba. "Distributed dynamic weaving is a crosscutting concern." In Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11). ACM, New York, NY, USA, pp.1353-1360, (2011).

[6] E. Truyen and W. Joosen. "Run-Time and Atomic Weaving of Distributed Aspects." In Transactions on Aspect-Oriented Software Development II, pp. 147–181. Springer, (2006).

[7] S. Herrmann. "Object teams: Improving modularity for crosscutting collaborations." In Procs. of Net.ObjectDays. Springer, pp. 248-264. (2002).

[8] M. VanHilst and D. Notkin. "Using role components in implement collaboration-based designs." In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96). ACM, New York, NY, USA, vol. 31, no. 10, pp.359-369, October (1996).

[9] H. Zhu. "Role mechanisms in collaborative systems." International Journal of Production Research, Vol. 44, No. 1, pp. 181–193, January (2006).

[10] M. VanHilst and D. Notkin. "Using C++ Templates to Implement Role-Based Designs." In Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS '96), Kokichi Futatsugi and Satoshi Matsuoka (Eds.). Springer-Verlag, London, UK, pp.22-37, (1996).

[11] M. Mezini and K Lieberherr. "Adaptive plug-and-play components for evolutionary software development." In Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98). ACM, New York, NY, USA, pp. 97-116, October (1998).

[12] S. Herrmann, C. Hundt, and M. Mosconi. "OT/J Language Definition V1.3." Technical Universitäte Berlin. Object Teams home-page: http://www.objectteams.org, (2009)

[13] F. J. S. Silva, M. Endler, and F. Kon. "A Framework for Building Adaptive Distributed Applications." In The 2nd Workshop on Reflective and Adaptive Middleware, Rio de Janeiro, Brazil, (2003).

[14] G. Kiczales, E. Hilsdale, J. Hugunin, and *et al.* "An Overview of AspectJ." In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, pp. 327-353. (2001).

[15] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation." Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009, pp. 171 – 180, September (2009).

[16] I. Ben-Shaul, H. Gazit, O. Holder, B. Lavva. "Dynamic self adaptation in distributed systems." In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, Springer, Heidelberg, vol. 1936, pp. 134–142. (2001).

[17] A. Popovici, T.Gross, and G. Alonso. "Dynamic weaving for aspect-oriented programming." In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, pp.141-147. (2000).

[18] C. Escoffier, R. S. Hall, and P. Lalanda. "iPOJO: an extensible service-oriented component framework." IEEE International Conference on Services Computing, IEEE Computer Society,July, 2007, pp. 474–48, (2007).

[19] I. Ben-Shaul, O. Holder, and B. Lavva. *Dynamic adaptation and deployment of distributed components in Hadas*. IEEE Transactions on Software Engineering, vol. 27, no. 9, pp. 769 – 787, September (2001).

[20] The AspectJ™ Programming Guide, website: (last visited: May, 2012) http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[21] N. Loughran, N.Parlavantzas, M.Pinto, P. Sánchez, M. Webster, A. Colyer, "Survey of Aspect-Oriented Middleware." AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-10. (2005).

[22] D. Suve, W. Vanderperren, and V. Jonckers, "Jasco: an aspect-oriented approach tailored for component based software development." In Proceedings of AOSD'3. ACM Press, NY. U.S.A., pp.21-29. (2003).

[23] A. Al-Zaghameem. "Extending the model of ObjectTeams/Java programming language to distributed environments." In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '10). ACM, New York, NY, USA, (2010).

[24] Sun Microsystems. "Java Remote Method Invocation Specification V1.5.0." Sun Microsystems Inc., Santa Clara, California,U.S.A. (2004).

[25] O. M. Group. "IDL to Java Language Mapping, Version 1.3." OMG, Needham, MA 02494, U.S.A., (2008).

[26] E. Truyen, N. Janssens, F. Sanen, and W. Joosen. "Support for Distributed Adaptations in Aspect-Oriented Middleware." The 7th International Conference on Aspect-Oriented Software Development AOSD'08, (2008)

[27] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. "An aspect-oriented approach to dynamic adaptation." In Proceedings of the first workshop on Self-healing systems (WOSS '02), David Garlan, Jeff Kramer, and Alexander Wolf (Eds.). ACM, New York, NY, USA, pp. 85-92, (2002).

[28] K. Biyani and S. Kulkarni. "Assurance of dynamic adaptation in distributed systems." Journal of Parallel Distributed Computing, Vol.68, pp.1097–1112, (2008).

[29] S. Sadjadi, P. McKinley, B. Cheng, and R. Stirewalt. "TRAP/J: Transparent Generation of Adaptable Java Programs." On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE Lecture Notes in Computer Science, Springer Berlin / Heidelberg (Lecture Notes in Computer Science) vol. 3291/2004, pp.1243 – 1261, (2004).

[30] B. Ban. "JGroups, a toolkit for reliable multicast communication." http://www.jgroups.org/ , 2002. (Last visited: May, 2012).

Abdullah O. Al-Zaghameem. Currently a Ph.D. student in Technische Universität Berlin. He received his B.Sc in Computer Science in 1999 and M.Sc in 2007 from University of Jordan. His master degree was in applying genetic algorithms to estimate the parameters of affine transformation from MRI brain images. He was employed in Tafila Technical University as a lecturer in computer science department from 1999 to 2008. His current research interests include: distributed AOP and collaboration-based computing.