Using Co-ordinated Atomic Actions for Building Complex Web Applications: A Learning Experience

A.F. Zorzo,
Faculty of Informatics
Pontifical Catholic University of RS - Brazil
zorzo@inf.pucrs.br

P. Periorellis, A. Romanovsky
Centre for Software Reliability
University of Newcastle upon Tyne - UK
{panos, alexander.romanovsky}@ncl.ac.uk

Abstract

This paper discusses some of the typical characteristics of modern Web applications and analyses some of the problems the developers of such systems have to face. One of such types of applications are integrated Web applications, i.e. applications that integrate several independent Web services. The paper focuses on providing software fault tolerance for such systems. The solution we put forward employs the concept of Co-ordinated Atomic (CA) actions for structuring such applications and for providing fault tolerance using exception handling. The paper discusses important design and implementation decisions we have made while developing a Travel Agency (TA) case study and attempts to generalise them to allow CA actions to be easily applied for building dependable Web applications.

1. Introduction

The use of Web applications by several people has become very common in the past years. Hence, the number of such applications has considerably increased, and the same person can use several different Web applications during a short period of time to achieve his goal. Usually, the user controls several interactions with all the Web applications in an *ad hoc* way. These interactions can be very complex concurrent activities. In some cases these concurrent activities may be working together, i.e. *cooperating*; in other cases the activities can be completely *independent* or may be essentially independent though needing to *compete* for shared common system resources. In practice, different kinds of concurrency might co-exist in a complex activity that thus will require a general supporting mechanism for controlling and coordinating this type of activity.

In this paper, we use the concept of Co-ordinated Atomic (CA) actions [1] for structuring such activities and for providing fault tolerance using exception handling. The paper discusses important design and implementation decisions we have made while developing a Travel Agency (TA) case study and attempts to generalise them to allow CA actions to be easily applied for building dependable Web applications that integrate several Web services that a user might want to access.

In particular, the paper discusses how to enclose the client side code and the server side code in one framework in such a way that single or concurrent errors detected on those sides are dealt co-operatively by both sides and that CA actions can have participants executing on both sides. Following the conventional way the Web services are implemented, our framework is based on a centralised component offering Web services to a number of clients (although the whole service can be a distributed application running, for example, on a cluster). An example of such services is the TA application, which allows integration of several other Web services available on the Internet. Usually Web services have a client side that uses a web browser to send HTTP requests to a Web application. But it is clearly much more convenient to implement the application logic on the server side using Java RMI or some other technologies which are not oriented towards the Web. This heterogeneity creates a problem that we had to solve. In this paper we show how the HTTP requests are transformed into remote method invocations to specific Java objects implementing most of the framework features, e.g. a special method for concurrent exception resolution [2], and the application logic. In order to transform a HTTP request into RMI calls we use the Java Server Pages (JSP) [3] technology, although other technologies like Active Server Pages (ASP) [4] could also be used. Another important solution we have used allowed us to deal with the statelessness of the method calls on the server side. The approach we are using makes it possible for a CA action to include sequences of HTTP requests issued by the client side. To conclude, the ultimate aim of this investigation conducted within European IST DSoS project (IST-1999-11585) [5] is to develop an advanced framework for employing CA actions for building complex Web applications.

2. Co-ordinated Atomic Actions

The Co-ordinated Atomic (CA) action [1] is a general mechanism for co-ordinating multi-threaded interactions and ensuring consistent access to objects (resources) in the presence of concurrency and potential faults. It can be regarded as providing a programming discipline for nested multi-threaded transactions that in addition

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

supports implicit co-ordination of a number of co-operating activities and very general exception handling facilities. The scheme is directly suitable for handling situations in which hardware and software faults have not been masked by the underlying transaction mechanism but have instead been reported to the application level, and/or at which there are application-level abnormal situations that have to be handled.

A CA action involves multiple co-operating roles that, among other things, must agree on the action outcome. There are four possible kinds of outcome: normal, exceptional, abort and failure. A CA action terminates normally if it is able to satisfy its post-conditions. If a CA action does not terminate normally, then each role must *signal* an exception to indicate the outcome. The roles should agree about the outcome so each role should signal the same exception. If an exception is *raised* during the execution of a CA action, this triggers a process of exception handling. Depending on how successfully the CA action can recover from the exception, it may still terminate normally or otherwise exceptionally. If error recovery is not possible, the CA action may attempt to rollback the state of external objects and signal abort. If the rollback is unsuccessful, then the CA action signals failure.

If a CA action terminates exceptionally (i.e. with exceptional, abort or failure outcome), the corresponding exception is raised in the enclosing context. CA actions can be nested and this means that an action which terminates by signalling an exception is effectively passing on the responsibility for exception handling to the enclosing CA action.

Our experience in developing Web applications as TA shows that there are situations in which the canonical CA actions have to be modified for practical reasons and to reduce the complexity the system designers have to deal with while applying this fault tolerance scheme. For example, canonical action nesting is defined in such a way that a subset of participants of the containing action takes part in a nested action. This is a straightforward rule that guarantees absence of information smuggling and facilitates the action support. Sometimes, as we will show in the next sections, we have to apply another type of CA action, i.e. CA actions that are executed as a method call in which the body has several threads forked and joined when the action starts and completes. All forked threads are involved in co-operative exception handling when any of them raises an exception. If there are several concurrent exceptions they are resolved in the way this is done in the canonical CA action scheme as described above. Such method call either returns a result or signals an interface exception to the containing action. It is not difficult to see that such CA actions have all main properties of the CA actions with respect to fault tolerance and complexity encapsulation because there is no information smuggling outside such actions. Actions allowing this type of nesting can be freely mixed with the canonical CA actions, as indeed has been done in this paper.

3. Travel Agency Case Study

To demonstrate how CA actions can be used to build Web applications, we have chosen a very typical system, a Web Travel Agency, which, as our analysis shows, has main characteristics of many real-life Web applications. We assume that there is a number of Web services in place that make it possible for the client to book some parts of trips (e.g. a hotel room, a car, a flight). Therefore, the goal of the exercise is to apply fault tolerance techniques in building a new service that allows the client to book whole journeys. By doing this we will be building a new emerging service, which none of the existing services is capable of delivering individually [6].

The main challenges related to provision of fault tolerance of the integrated Web applications are as follows. The legacy components are Web servers that are controlled by different organisations and are not developed for integration, because of this there is often not enough information which the integrators might need (including, for example, component complete and correct specification). Another consequence of this is that system integrators have to treat these components as black boxes that can only be accessed via standard interfaces. With respect to the dependability of the integrated application there are two factors to be taken into account: a well-known fact that the quality of many Web services is very low [7] and absence of evidence supporting any reasonable claims about their reliability. While integrating dependable Web applications is important to realise that it is impossible to develop or rely on features for locking Web services and for aborting (sequences of) operations on them. Another set of the problems specific for such systems is related to the Internet as the only communication media and the only environment in which composed systems operate. Web services are autonomous entities oriented mainly towards interactions with clients and they often take liberty to send replies that do not exactly fit the requests as a way of helping the clients or promoting their service. Moreover, because of their nature they offer a very specific type of interface suitable for browsing only (HTML interfaces). It is a well-known fact that the Internet is not a very reliable media and that there is a high number of Internet-specific faults such as delays, lost requests, services switched down (because of either their faults or regular shutdowns) [7] [8]. Integrated applications of the TA type have to meet high dependability requirements including consistency of money transfers and clients' satisfaction. One more problem that the designers of such systems have to deal

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

with is that they have to preserve the right level of abstraction while composing the system. Such Web applications are typically built using complex composite middleware consisting of several levels with an ability to deal with exceptions at different levels, so there is a need for a unified approach and for a proper exception handling encapsulation. One more characteristic worth mentioning here is the fact that people are involved in execution of such systems and they can both cause errors and be involved in recovery; in the context of TA clients, the integrated system support and the legacy component support can be included into consideration.

Our choice of the fault tolerance and structuring techniques to be used is defined by these characteristics. In our design of TA case study we will be developing and applying the techniques that allow system integrators to meet high dependability requirements by incorporating measures for disciplined tolerance to the faults of several types. First of all, TA should tolerate errors caused by hardware failures in communication (mainly delays) or in legacy components (mainly crashes), which should not cause failures of the whole TA. Secondly, the client's mistakes and client side machine crashes should be tolerated without affecting either TA or the legacy components. Thirdly, TA should tolerate situations when legacy components cannot provide the required service or when they behave abnormally. Besides, the clients should be informed about the situations when the machines on which TA is executed crash and these crashes should not affect the legacy components. The design should guarantee that all components, including legacy servers, TA and clients stay in a consistent known state even when faults happen.

4. Structuring Web Applications

Normally, Web applications are divide in two parts: a client side and a server side. The client side executes on the client's machine and usually gathers information from the user to be sent to the server. The server side is responsible for using the users input, processing and then returning the result to the client computer. The TA structure has a similar structure, which is typical for many Web services [6]. The major difference is that in the TA system, requests from the client are passed to legacy components (this is not seen by the client) (see Figure 1). Note that a client can be accessing a legacy component directly, therefore the legacy component will see our TA as a client. In order to provide a client with some level of fault tolerance, our approach focuses on employing application-level fault tolerance by means of structured exception handling. In designing TA we employ the CA actions concept.

The overall TA execution, with respect to each client, is structured using CA actions. TA is a complex concurrent distributed application with a considerable number of exceptions to be handled. Several interacting components of different types are to be involved in this execution and there is a need in consistent co-operative application-specific handling of all abnormal situations. All information exchanged between client and the TA is realised inside CA actions via shared local objects. Because the client thread is either executing in the client side or on the TA side, a special shared object is used. This object is a way of signalling exceptions to the client side if the TA side raises an exception that has to be handled by all participants of an action.

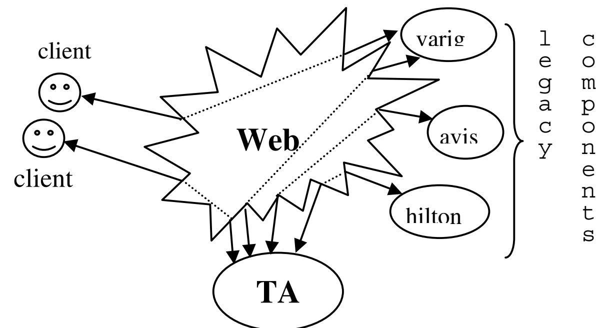

Figure 1. Architecture of Travel Agency

4.1. Structured System Design Using CA actions

In our design [9], every time a client connects to the TA a special CA action is started on the TA side (this CA action can be executed on the same place the TA server is executing or in a special computer used to host the clients actions). This special action encloses all activities that the client executes, even if these activities are executed on the client side. This action is called ***session*** action. The ***session*** action finishes when the client logs off or crashes.

The ***session*** action is composed of three co-operating roles, which are executed by participants represented as concurrent co-operating threads: client controller, TA CS (Client Side) controller and TA SS (Server Side) controller. The first participant is mainly located on the client computer, and is responsible for interacting with the user. The client participant is responsible for gathering input from the user, sending this information via HTTP (JSP) to the ***session*** action, and exhibiting the result to the user. During this process, the client thread can be seen as a thread that executes partially in the client side and partially on the server side. The remaining two participants are executed on the TA computer (or the computer destined to execute the client's ***session*** action). These last two threads are created when a client logs into TA. Introducing such threads allows us to make the system structure cleaner, to reduce the design complexity by separating concerns and to improve system performance. For example, one of the responsibilities of the TA CS is to monitor the client side, while the TA SS is responsible for distributing the client requests between

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

the legacy systems. As discussed in Section 2, this ***session*** action is not a canonical CA action, i.e. the threads that will execute each of the roles of the CA action are forked when the ***session*** action is started by the client controller thread (see Figure 2).

After the session action has started, the client can choose the activity among the following: checking availability of a trip, booking a trip, cancelling a trip, and paying for a trip. They correspond to four actions that are nested into the ***session*** action: the ***availability***, ***booking***, ***cancellation*** and ***payment*** actions. These four actions have the same three participants as the containing ***session*** action, i.e. they are canonical CA actions. The client may choose to perform any of these actions in any possible order but within a restriction imposed by the menu presented to him (e.g. it is not possible to cancel a trip if it has not been booked before). One of the possible scenarios is shown in Figure 2. In the figure, we represent the client participant informing the TA CS and TA SS controllers that he wants to execute the ***availability*** action and when this has finished (supposing that he is content with the choices he got) he informs the other participants to execute the ***booking*** action.

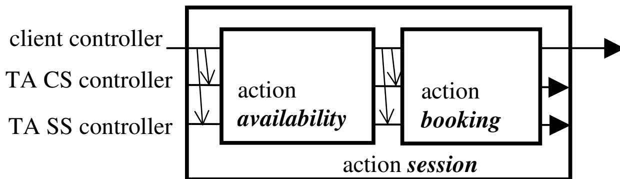

Figure 2. Valid execution of the *session* action

If any of those four actions is not able to deliver the service required, it completes abnormally and propagates an interface exception to the ***session*** action. When possible all three participants of this action are involved in handling of such exception. Note that when the ***availability*** action completes without exceptions it produces a normal result consisting of a description of a number of trips meeting all client's requirements: in the scenario shown in Figure 2 the client chooses one of these trips and proceeds with booking. The trip choices are sent to the client when the client controller is executing inside the ***availability*** action.

Let us consider now the internal structure of the ***availability*** action. In our design it has two nested actions (Figure 3): the ***request*** action and the ***consult_services*** action. They implement distributed browser access to the TA service. Within the ***request*** action client's information is passed from the client computer to the TA server and checked. If during this checking the TA CS controller finds that some part of the information is incorrect (e.g. city name, days of travel, length of the stay, etc.) it raises a corresponding internal exception in the action to alert the client and to advise him to correct this information. After such correction the action continues. If the TA server is down or crashes, the corresponding action is aborted and an external exception is propagated to the ***availability*** action level. This action is aborted in its turn and an external exception is signalled to the ***session*** level to inform the client and to advise him to close the session. If one of these two actions (***request*** or ***consult_services***) detects that the client is not on-line or his computer crashes, the action itself and the containing action ***availability*** are aborted, and the ***session*** action completes.

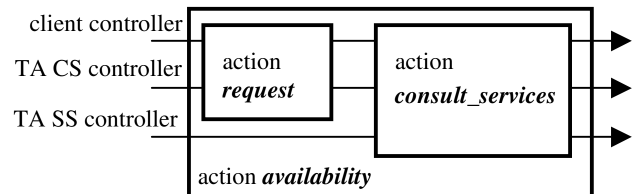

Figure 3. Structure of the *availability* action

As mentioned in Section 2, sometimes we need a special type of CA action that is initiated by only one thread but that have several threads inside. The TA SS controller, one of the participants of the ***consult_services*** action, activates the ***compose_trips*** action that is designed as a CA action of this type (Figure 4). CA action ***compose_trips*** has four cooperating participants: the ct controller (a service thread coordinating the execution of the remaining three participants) and three participants: flight, car and hotel, which are responsible for providing respective information for composing the whole trip. The arrow from the client participant to the TASS controller represents the sending of the information from the client to the TASS controller. This information is later passed to the ct controller participant in the ***compose_trips*** action, which will split the information into details of the flight, car and hotel. The ct controller passes then this information to the respective participants, which access their legacy components.

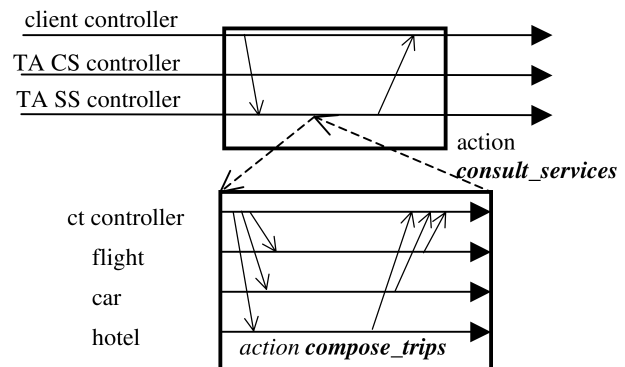

Figure 4. Action *compose_trips* is nested in action *consult_services*

If any of these participants raises an exception all of them are involved in cooperative handling. For example, if there is no car available for the date of travel the ct

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

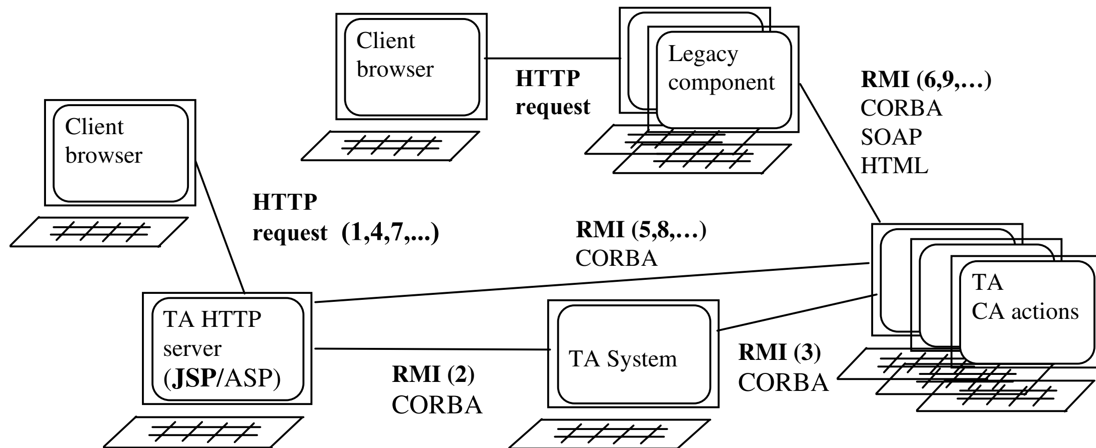

Figure 5. General implementation structure

controller may decide to find another airport nearest to the destination city, or to check a more expensive or cheaper option for car rental, or to search for the hotels offering car rental. When handling is not possible at the level of action ***compose_trips*** a corresponding exception is propagated to the TA SS controller and raised in all participants of action ***consult_services***.

5. Implementation

The previous section has shown how we have structured all activities that are executed by the TA in order to access legacy components using the CA action concept. In this section we discuss the implementation details, the technologies used and the architecture of the whole system. Figure 5 shows the architecture of our system and possible technologies that could be used for exchanging information between client, TA and legacy systems. In our implementation we have used the technologies written in bold font in the figure.

The first step (**1**) is executed by the client when he sends an HTTP request to the HTTP server on the TA side. When the TA HTTP server receives the request it executes JSP code associated with the Web page the client was trying to access (note in the figure that we are using JSP but could have used ASP instead). This JSP code is interleaved with Java code that makes an RMI call (**2**) to the TA system, which is responsible for creating the ***session*** action as described in the previous section. All the CA actions are created as remote objects, via an RMI call (**3**), on a different machine. Note that this remote machine could be a set of computers, or a cluster of computers, and the CA actions are distributed among these computers. After the ***session*** action has been started, the client receives back an identifier to this action, and a new Web page containing a set of options he can access in the TA. This identifier is send back to the client via a cookie, which is stored in the client's machine.

All the steps described above are related to the first access the user makes to our TA. Once he has got a Web page back, the whole structure for checking trip availability, or executing any of the actions described in the previous section is ready. Therefore, if the user now wants to check availability for a specific trip, he fills a form in an HTML page and the information on this form is sent via an HTTP request (**4**) to the TA HTTP server. The TA HTTP server executes then the JSP code that sends this information to the ***session*** action via an RMI call (**5**). In the ***session*** action, as described in the previous section, the client's request is passed to the legacy components (**6**). When all legacy components, i.e. hotel, airline and car rental Web sites, have returned the availability, this information is send back to the client, and the whole process can be started again (**7,8,9,…**). As can be seen in Figure 5, the CA actions access the legacy component via the RMI protocol, but we could have used any of the other technologies shown in the figure, i.e. SOAP, CORBA, etc. One important feature in our design is that each access to legacy components is wrapped into a special code implementing a kind of plug-in (or driver). This plug-in provides always the same interface to the CA actions independently of the technology used by the legacy system. Employing such remote protective wrappers is an important design decision that allows us to separate a number of lower-level and routine activities from the main TA logic. Moreover, although in our current implementation we use synchronous calls to legacy components, using such wrappers will allow us to deal with asynchronous calls as well.

5.1 HTTP Server and JSP Code

One of the important features of the HTTP server is that it is *stateless*, i.e. it does not keep state between calls. This is very important when one wants to deal with faults of the HTTP server. In the event of the HTTP server crashing, then a new HTTP server can take over the job of

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

the crashed server. This feature is also valid for the TA system. All the information needed about the client is kept, in our system, in a cookie that is set when the client first connects to the TA Web site and downloads the HTML page shown in Figure 7. In this figure, we show how HTML and JSP code are interleaved. The Java code is shown between lines 2 and 12. The Java code is responsible for accessing the TA system (line 6), creating a new ***session*** action in the TA (line 7), and creating a cookie to store information about the ***session*** action. The information stored in the cookie will be used every time the client wants to execute a nested action in the ***session*** action, for example ***availability*** action. The client, via JSP code also uses this information to access local shared objects that will serve as a communication channel between the client and the other participants of the ***session*** action. The HTML code shows a set of options to the client, for example an option for making a reservation (line 13), or an option to cancel a reservation (line 14).

```
01: <html> … <body>
02: <%@ page import="TravelAgency"%>
03: <%@ page import="java.rmi.*"%>
04: <%@ page // other Java imports …
05: <% try {
06:      TravelAgency ta = (TravelAgency) Naming.lookup("rmi://address/TA");
07:      int numb = ta.createAction();
08:      Cookie c = new Cookie (request.getRemoteHost(), Integer.toString(numb));
09:
10:      response.addCookie(c);
11:      response.setContentType("text/html"); ...
12:    catch ( … ) { … }   %>
13: <p align="center"><a href="sos_ta.htm">Make A Reservation</a></p>
14: <p align="center"><a href="soscancel.htm">Cancel a Reservation</a></p> ...
15: </body> </html>
```

Figure 7. HTML and JSP code for the start menu of the TA

```
01: public TASS(String n, drip.Manager mgr, drip.Manager leader) throws RemoteException {
02:      super(mgr, leader, n);
03:
04:      // Create the compose_trips CA action.
05:      // Create managers. Parameters: manager name, DMI name
06:      drip2.Manager mgr1   = new drip2.ManagerImpl("mgr1","compose_trips");
07:      drip2.Manager mgr2   = new drip2.ManagerImpl("mgr2","compose_trips");
08:      drip2.Manager mgr3   = new drip2.ManagerImpl("mgr3","compose_trips");
09:      drip2.Manager mgr4   = new drip2.ManagerImpl("mgr4","compose_trips");
10:
11:      // Create roles: Parameters: role name, role manager, leader manager
12:      ctComposeTrips       = new compose_trips.CT    ("ct",     mgr1, mgr1);
13:      flightComposeTrips   = new compose_trips.Flight("flight", mgr2, mgr1);
14:      carComposeTrips      = new compose_trips.Car   ("car",    mgr3, mgr1);
15:      hotelComposeTrips    = new compose_trips.Hotel ("hotel",  mgr4, mgr1);
16: }
```

Figure 8. Java code for creating the *compose_trips* action

```
01: public void body(Object list[]) throws Exception, RemoteException {
02:   try {
03:     RemoteQueue rqIn  = (RemoteQueue) list[0], rqOut = (RemoteQueue) list[1];
04:     BreakRequest request = new BreakRequest((triprequest) rqIn.get());
05:
06:     ctCarQueue.put(request.cr);
07:     ctFlightQueue.put(request.fr);
08:     ctHotelQueue.put(request.hr);
09:     waitAnswers.synchronize();
10:
11:     Flight fl[]   = (Flight[]) ctFlightQueue.get();
12:     Hotel hl[]    = (Hotel[])ctHotelQueue.get();
13:     Car cl[]      = (Car[])ctCarQueue.get();
14:     Trip trips[] = Compose.combine(fl,hl,cl);
15:
16:     rqOut.put(trips);
17:   } catch (Exception e) { throw e;  }
18: }
```

Figure 9. Java code of the ct controller role of the *compose_trips* action

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

Another important feature that can be seen in Figure 7 is the Java exception handling code used to inform the client when the TA system is not available (line 12). Note that the HTTP server can be active but the TA system can be down. They execute on different machines as shown in Figure 5.

5.2 CA Actions

The implementation of the CA actions is realised using an object-oriented framework [10] developed in Java RMI. In this framework, CA actions are implemented with two types of objects: manager and role. As described in Section 2, each CA action is composed of a set of roles. A role object contains the code that will be executed by a participant of the CA action. A manager object is responsible for controlling the execution of a role object. The set of managers control as CA action protocols, i.e. synchronisation upon entry, synchronisation upon exit, concurrent exception resolution, testing of the pre and post-condition. For the complete description of the framework see [10]. This framework was extended to support the type of CA actions described in Section 2. Although this extension was enough to implement CA actions, it could not be applied directly to the system we have implemented. The major reason is that the code for the client participant is split into two parts: one running on the server machine and another running on the client machine. We solve that by allowing the thread that executes the client role to execute in the client's browser, therefore we consider the thread executing in the client's browser as being inside the action. The TACS participant is responsible for controlling the client, and may use timeout mechanism to detect when the client is not running anymore.

Figure 8 shows the constructor of the TASS role object of the ***consult_services*** action. This object is responsible for creating and starting the ***compose_trips*** action. The creation of the ***compose_trips*** action is divided into two parts. First the set of manager objects is created (lines 6 to 9). Second, the set of role objects is created (lines 12 to 15). Each manager is created with a name and the name of the action it belongs to, while each role object is created with a name, a manager that will control the role, and a manager that is the leader[1] of the manager of this role. All these objects are remote objects and could be executing on different machines. One important feature that can be identified in Figure 8 is the different managers that are used in the ***compose_trips*** and ***consult_services***: drip.Manager (line 1) and drip2.Manager (lines 6 to 9). The former is used to create a canonical CA action. The latter is used to create the modified CA action as described in Section 2.

Figure 9 shows the main code of the ct controller role of the compose_***trips*** action. This role receives references to two remote objects that are used to receive (`rqIn`) and send (`rqOut`) (line 3) information from/to the client via the JSP code (see Figure 4). Line 4 shows how the client request is received from the client via the remote object, and how this request is split into separate requests to the legacy components. The next step is to pass this information (line 6 to 8) to the roles that will access the legacy components. This sending of information is realised via local objects. After sending this information, the ct controller has to wait the other roles to receive the required information back from the legacy components (line 9). When car, flight and hotel roles have got back availability from the legacy components, the ct controller receives this information (lines 11 to 13), combines this data, and sends it back to the client via the remote object `rqOut` (line 16). Figure 4 represents the Java code from Figure 9. Note that if any exception is raised during the execution of ct controller role, this exception is caught and thrown to the manager object that controls this role (line 24). The manager object then informs its leader about the exception and the exception resolution algorithm is executed.

6. Concluding Remarks

Our experience in the past years has shown that CA actions provide a powerful support structuring mechanism for several different types of applications, for example, control software for different types of production cells (fault-tolerant [11], not fault-tolerant [12], and real time [13]), or for complex GAMMA computation [14]. In this paper we have discussed and shown how CA actions can be used for structuring and implementing integrated Web applications. This case study differs from the previous ones because the application area has a number of very specific characteristics, which required some adjustments in the way CA actions are used. For example, heterogeneity and complexity of the environment, autonomy and legacy of the Web servers, and needs to explicitly deal with node crashes and communication delays. One of such problems is that legacy components, i.e. existing Web services, are not controlled by system integrators and, due to this, the main means of system recovery is application-specific exception handling. The situation is complicated by the fact that only weak assumptions can be made of the behaviour of such components. It is becoming clear to the specialists in the field that ACID transactions cannot be used for such purposes; this is why more flexible techniques are being developed [15]. CA actions clearly offer a more general approach that allows developers to deal with co-operative and competitive concurrency, and to employ application-specific and component-specific exception handling in a

[1] The leader is responsible for controlling all protocols of the CA action.

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.

disciplined and structured way. Another relevant characteristic of CA actions is their ability to support structuring and fault tolerance of the complex systems that include non-software entities such as human beings, devices, money, goods, documents, etc. Because of their very nature, activities involving such entities become long-lived and the abort semantics is not applicable. CA actions keep all information under control and allow different types of application-specific recovery to be programmed using exception handling [1] [16]. CA actions, being a general design concept, are not attached to deal with any specific type of faults, although in each particular case study the fault assumptions have to be clearly stated. In this practical work we do not consider Byzantine faults (this is where the main focus of research on consensus is [17]), but we mainly deal with environmental faults, software design faults and hardware crashes with fail-stop semantics.

With respect to the implementation of CA actions, one important aspect that had to be dealt with in this paper was the separation of the code of one of the participants of a CA action. The client participant code was split between the TA computer and the Web browser computer. Controlling this type of separation was very complex. We solved this by having some kind of watchdog that would check, via time-out, whether the client was down or not. Another point, as explained in Section 2, was the extension of the Java framework [10] to allow for a new special type of CA action.

We would like to conclude by saying that in many practical situations it makes sense to apply specific structuring techniques tailored for particular needs: we refer here to employing a special type of CA actions and splitting an action participant into two parts executed at different machines. Another important conclusion is that the solution proposed for developing a unified service interface and several wrappers oriented towards accessing existing Web services using different technologies (WSDL [18], SOAP [19], other XML-based techniques, CORBA, etc.) offer a very flexible and dynamic way of dealing with ever-growing number of technologies.

Acknowledgments. This work is supported by European IST DSoS Project (IST-1999-11585). A. Zorzo is a researcher supported by the Brazilian agency CNPq (350277/2000-1). We would like to thank B. Randell, C. Jones, and V. Issarny for the fruitful discussions.

References

[1] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault Tolerance in Concurrent Object-oriented Software through Co-ordinated Error Recovery. In *FTCS-25*, California, USA. IEEE CS Press, 499-509, 1995.

[2] R.H. Campbel, B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, 12(8), 811-826, 1986.

[3] Sun. *Java Server Pages*. http://java.sun.com/products/jsp/

[4] Microsoft. *Active Server Pages*. http://www.asp.net.

[5] *Dependable System of Systems*. European IST (1999-11585) http://www.newcastle.research.ec.org/dsos/

[6] P. Periorellis, J.E. Dobson. Case Study Problem Analysis. The Travel Agency Problem. Technical Deliverable CS1. Dependable Systems of Systems Project. University of Newcastle upon Tyne. 37 p.

[7] M. Kalyanakrishnan, R.K. Iyer, J.U. Patel. Reliability of Internet hosts: a case study from the end user's perspective. *Computer Networks*, 31, 47–57, 1999.

[8] C. Labovitz, A. Ahuja, F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *FTCS-29*, Madison, USA. IEEE CS Press, 1999.

[9] A. Romanovsky, P. Periorellis, A.F. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. *ISADS 2003* (to appear).

[10] A.F. Zorzo, R.J. Stroud. An Object-Oriented Framework for Dependable Multiparty Interactions. In OOPSLA-99. *ACM Sigplan Notices*, 34(10), 435-446, 1999.

[11] J. Xu, B. Randell, A. Romanovsky, R. Stroud, E. Canver, A. Zorzo, F. Henke. Rigorous Development of a Safety-critical System based on Co-ordinated Atomic Actions. In *FTCS-29*, Madison, USA. IEEE CS Press, 68-75, 1999.

[12] A.F. Zorzo, A. Romanovsky, B. Randell, J. Xu, R.J. Stroud, I.S. Welch. Using Coordinated Atomic Actions to Design Safety-Critical Systems: A Production Cell Case Study. *Software: Practice and Experience*, 29(8), 677-697, 1999.

[13] A. Zorzo, L. Cassol, A. Nodari, A. Oliveira, R. Morais. Long Term Scheduler for Real Time Industrial Installations. In *3rd Congress of Logic Applied to Technology*, São Paulo, Brazil, 2002.

[14] A. Romanovsky, A. Zorzo. Co-ordinated Atomic Actions as a Technique for Implementing *Distributed* GAMMA Computation. *Journal of Systems Architecture – Special Issue on New Trends in Programming.* 45(9):79-95, 1999.

[15] J. Webber, V. Corrales, M. Little, S. Parastatidis. Making web services work. Application Development Advisor November-December, 68-71, 2001. http://www.appdevadvisor.co.uk.

[16] A. Romanovsky. Coordinated Atomic Actions: How to Remain ACID in the Modern World. *ACM Software Eng. Notes*, 26, 2, 66-68, 2001.

[17] L. Lamport, R. Shostak, M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4, 3, 382-401, 1982.

[18] World Wide Web Consortium. *Web Service Definition Language*. http://www.w3.org/TR/wsdl

[19] World Wide Web Consortium. *Simple Object Access Protocol*. http://www.w3.org/TR/SOAP

Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems ISBN 0-7695-1929-6/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on June 10,2022 at 17:13:20 UTC from IEEE Xplore. Restrictions apply.