Structuring Integrated Web Applications for Fault Tolerance

Alexander Romanovsky, Panos Periorellis
School of Computing Science
University of Newcastle upon Tyne, UK
{alexander.romanovsky, panos.periorellis}@ncl.ac.uk

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

Abstract

This paper shows how modern structuring techniques can be employed in integrating complex web applications such as Travel Agency systems. The main challenges the developers of such systems face are dealing with legacy web services and incorporating means for tolerating errors. Because of the very nature of such systems, exception handling is the main recovery technique to be applied in their development. We employ Coordinated Atomic actions to allow disciplined handling of such abnormal situations by recursively structuring the integrated system and by associating handlers with such actions. We use protective wrappers in such a way that each operation on legacy components is transformed into an atomic action with a well-defined interface. To accommodate a combined use of several ready-made environments (such as communication packages, services and run-time supports), we employ a multilevel exception handling. We believe that these techniques are generally applicable for both: structuring integrated web applications and providing their fault tolerance.

1. Introduction

Many researchers and practitioners realise that, to build complex fault-tolerant applications, proper system structuring is indispensable. It not only makes it possible to deal with complexity of modern applications, but also allows fault tolerance measures to be associated with the system structure and helps apply them in a disciplined fashion. This is why one of the main requirements to any application-level fault tolerance technique is its recursiveness [12]; recovery blocks, exception handling, atomic actions are recursive fault tolerance schemes. Each level of system structure contains errors and is responsible for dealing with them. If fault tolerance measures fail at this level, then the responsibility for recovery is passed on to the upper level (usually an attempt is made to leave the erroneous level in a consistent known state to facilitate the recovery at the upper level).

In this paper, our focus is on exception handling as the main and the most general application-specific technique [3]. It allows system designers to build applications capable of tolerating several types of faults, including software (design) faults, exceptions propagated from the run-time support and from the hardware level, environmental faults, operators' mistakes. At each system structure level, it is important to distinguish between internal exceptions specific to the level implementation and external exceptions specified in the level interface. Exceptions of these two types serve different purposes and are used in different ways. External exceptions are part of the level interface and are signalled to the upper level to inform it about the failure of the underlying level to deliver the required service. Internal exceptions are to be handled locally; both they and their handling are hidden from the upper level.

There are different ways in which systems can be structured to achieve fault tolerance via exception handling. This depends on many factors including design paradigms, computational models, application requirements, types of structuring units available in libraries, programming language used, etc. The general pattern common to all of them is captured by the concept of Idealized Fault Tolerance Components [8], which encapsulates and separates normal and abnormal activity of each structuring unit (abnormal activity represents provision of fault tolerance by exception handling) in such a way, that when faults cannot be tolerated inside such a unit, it signals an exception; otherwise it delivers a normal response. Sequential systems are typically built either as a multilayer structure or as a set of (nested) components (e.g. procedures, classes). Associating exception handling with component nesting is a straightforward task, and many practical languages incorporate sequential exception handling.

Concurrent systems require different structuring mechanisms to capture their specific characteristics. To this end atomic actions were proposed [4] as structuring units to be used for developing cooperative concurrent systems and for providing disciplined exception handling in such systems. The concept of Coordinated Atomic (CA) actions was introduced as a generalisation of atomic actions which allows objects (resources, servers) to be shared by different actions while guaranteeing transactional (atomicity, consistency, isolation, durability - ACID) properties of (nested) action access to such *transactional* objects [17]. Both atomic actions and CA actions are defined by a number of action participants (e.g. threads, objects) coming together to cooperate and to achieve a common goal; in the CA action context each of such participants is described as playing a role in an action. If any of them detects an error when in an action, all participants are involved in cooperative exception handling. CA actions usually have several outcomes to allow developers to report different situations when the required service cannot be fully provided. These are used, for example, when all-or-nothing semantics cannot be guaranteed. Actions can be nested, and if participants of a nested action are not able to handle an exception, an external exception is propagated to the containing action. To guarantee action atomicity, all action participants enter and leave actions at the same (logical) time; they do not exchange information with any processes outside the action scope.

2. *Travel Agency Case Study*

Modern web applications are typically built by integration of existing web services. Their developers face a number of serious challenges, one of which is providing high level of dependability of the composed applications. Such applications have to deal with a big number of abnormal situations, belonging to different types and often happening concurrently, in an adequate and effective way, satisfying the client' expectations. These characteristics of modern web applications are becoming increasingly important as our society puts more and more reliance on e-services of different types. It is our intention in this paper to investigate how modern fault tolerance techniques can be applied in a disciplined and systematic fashion to guarantee fault tolerance of the applications of this particular type. To demonstrate our approach we have chosen a very typical system, a web Travel Agency (TA), which, as our analysis shows, has main characteristics of many real-life applications of this type. 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), so 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 [10].

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 which 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 [6] and absence of evidence supporting any reasonable claims about their reliability. While integrating dependable web applications it 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) [6, 9]. 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 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 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 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.

3. *Assumptions*

Development of an integrated system, in general, and provision of its fault tolerance, in particular, is not possible without understanding the assumptions we can make about the legacy components and about the environment in which the system will operate.

We assume that component systems are black boxes with known call interfaces. TA handles exceptions that they can propagate and it can access component systems during exception handling only through those interfaces. Exceptions that these systems propagate to TA can be caused by many reasons, for example, by erroneous, incomplete or insufficient input data that TA sends to these component systems during calls.

We assume that legacy components fail in a fail-stop fashion and that such crashes can be detected by timeouts. It is further assumed that legacy components have all-or-nothing semantics for each individual request (call) TA sends them. Another assumption is that if TA crashes without receiving a result for a request that was sent to a legacy component this request gets cancelled. It is assumed that messages can be neither lost nor corrupted. We further assume that each legacy component identifies the requests from TA using an id that it sends to TA the first time TA contacts the component.

Our assumption is that the legacy components can be concurrently accessed (using TA or directly) by a number of clients, and that they execute individual requests when they are delivered one at a time (or, as if they were executed one at a time). Clearly we cannot assume that TA can lock those components.

4. Fault Analysis

TA system should be able to tolerate the situations when legacy services are down or crash. These errors can be detected by timeouts or by catching exceptions signalled by the underlying communication and middleware software. In a similar way TA will be able to detect client computer crashes.

Mistakes made by clients can be tolerated after they are detected by TA. One more type of abnormal situations is application exceptions propagated from legacy components (e.g. trip is not available or credit card is not valid). Any other problem that the underlying software (OS, middleware, communication packages) detects and propagates as an exception to the application level will be handled at the application level as well.

Another possible source of errors are misbehaving legacy components. Dealing with them is complicated by the fact that TA integrators do not have complete or correct specification of such services. Possible solutions rely on developing protective wrappers incorporating executable assertions; such assertions are built using several sources of information [11]. The wrappers signal an exception when they detect an abnormal behaviour of a legacy component (we do not discuss the issues of developing such protectors any further in the paper, as it is a separate strand of our research).

Error recovery at the level of TA has several important characteristics. First of all, it is clear that simple abort is not applicable here because the system is built out of legacy components that do not have abort semantics (actually web services often have a very complicated cancellation policies) and because humans are involved in its operation. This is why we need application-level exception handling as the main means of recovery which supports moving TA and its components into a known consistent state and continuous delivery of the service. Another typical characteristic is that several components have to be involved in cooperative recovery because several of them are always involved in execution of any request coming from the client. One more complication is the fact that several exceptions can happen concurrently in such systems and they have to be dealt with properly without ignoring any of these exceptions. In the systems like this human beings (i.e. clients) have to be involved in handling of many abnormal situations.

5. Design

5.1. General architecture

The general architecture of the TA (Figure 1) is typical for many web services [10]. There is a dedicated TA server (or, servers) that can be accessed by clients via the Internet. When a client accesses the TA service, some part of TA is dynamically loaded to the client computer and a session starts. While executing client's requests within a session TA is split into two parts: TA server side (SS) and TA client side (CS). TA CS provides a web front-end to the client and performs initial checks of the information she inputs. TA CS and TA SS are executed on the TA server. Web requests from the client are passed from TA CS to the TA SS, where they are executed and from where the existing legacy components are accessed (this is effectively hidden from the client). A number of clients can access TA concurrently: TA creates a copy of TA CS and a session for each client.

Our principle approach to achieving application-level fault tolerance in such a system is by structured exception handling. As we have discussed in Section 1, the choice of the structuring technique depends on many factors. In designing TA we employ two such techniques: CA actions and layering.

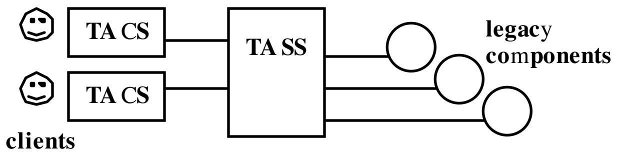

Figure 1. Architecture of Travel Agency

The overall TA execution 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 cooperative application-specific handling of all abnormal situations. Layering is used for structuring the execution of individual web requests passed between TA SS and TA CS. Such requests encompass complex activity at different levels with well-defined interfaces: we use several middleware services to offer clients a standard browser interface. An exception of each level is to be either handled or transformed to an exception of the above level and propagated further.

5.2. Structured system design using CA actions

In our design, execution of the entire client session is structured as a top-level CA action encompassing all possible activity the client may wish to perform. This action has a specific and important task because it encapsulates all TA execution with respect to one particular client. As a top-level action it cannot propagate any exception outside because action-level exception handling outside such action is not applicable. When the client decides to complete the session she informs TA by logging off and the session action completes. In our design this action has three cooperating participants represented as concurrent cooperating threads: client controller, TA CS controller and TA SS controller. The first participant is located on the client computer; it is a client's proxy and is responsible for interacting with the client. The remaining two participants are executed on the TA computer. These threads are (logically) 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.

Inside the session action 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 which 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. The client may choose to perform any of these actions in any possible order but within a restriction imposed by the menu presented to her (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.

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. Let us consider the availability action that has the following interface exceptions: no trip available, TA is down (this exception is propagated to only the client controller as the remaining two participants of action session are down), client timeout (when client is not responding for a predefined period of time – this exception is propagated to only two participants of the session action), client site is down (this exception is propagated to the TA SS controller and TA CS controller), new offers (this exception is raised to inform the client about the following situation: TA is not able to find exact matches to all client's requirements and it offers several "similar" trips that the client might like). 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.

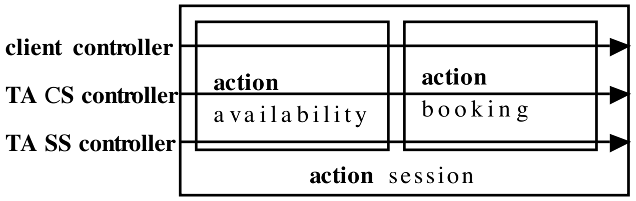

Figure 2. Top-level view on TA structuring

Let us consider now the internal structure of the availability action. In our design it has two nested actions: request and consult_services. 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 her 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 her to close the session. If one of these two actions (request or consult_services) detects that the client is not online or her computer crashes, the action itself and the containing action availability are aborted, and the session action completes.

Our analysis shows that there are situations when the canonical atomic action scheme (e.g. from [4, 8]) has to be modified for practical reasons and to reduce the complexity the system designers have to deal with while applying this fault tolerance scheme. 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. In our design of TA we use another type of atomic action: these actions are executed as a method call the body of which has several threads forked and joined when the action starts and completes. All forked threads are involved in cooperative 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 atomic action scheme [4]. Such method call either returns a result or signals an interface exception to the containing action. It is not difficult to see that such atomic actions have all main properties of the atomic actions with respect to fault tolerance and complexity encapsulation because there is no information smuggling outside such actions. Actions with this type of nesting can be freely mixed with the canonical atomic actions. There are two action schemes that have some properties of these atomic actions. In Argus [7] a method call can have a number of internal threads forked and joined inside, such method has transactional properties and it can signal exceptions to the caller context. This approach is not suitable for programming cooperative systems because internal threads handle their exceptions separately (which is not applicable for the cooperative systems the essence of which is that all action participants are always involved in dealing with any abnormal situation). In the Concurrent Recovery Block scheme [5] each alternate of the recovery block [14] is designed as a set of cooperating processes that are forked and joined inside it. If such alternate cannot ensure the acceptance test it is aborted and the following alternate starts. Although the conversation scheme does not support forward error recovery by cooperative exception handling this scheme is suitable for designing cooperative systems.

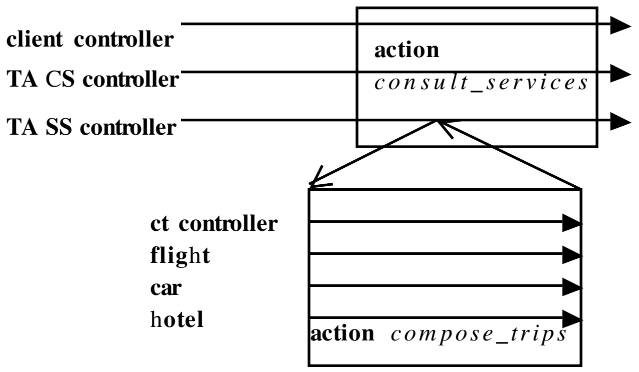

Figure 3. Action *compose_trips* is nested in action *consult_services*

The TA SS controller, a participant of action *consult_services*, calls method *compose_trips* that is designed as an atomic action of this type (Figure 3). The execution of this method is an atomic operation executed by the TA SS controller process. Atomic 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 trips.

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 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*.

5.3. *Actions accessing legacy components*

We use the *flight_availability* action to show how low-level actions – the *flight_availability*, *hotel_availability*, *car_availability* actions – are built. The flight process participating in action *compose_trips* is responsible for collecting all information about flights. To do this it calls method *flight_availability*, constructed as a nested action with a number of participants forked and joined inside. This solution allows us to completely separate all actual access to legacy components from action *compose_trips* and to make it transparent to this action: this includes dealing with component crashes, composing and filtering information coming from different legacy components, etc.

Each participant of action *flight_availability* (except for the fa controller) is responsible for interaction with only of legacy component (an airline booking service). In our design the controller collects all information coming from the rest of action participants, filters and sorts it, and manages the whole action. Each action participant accessing a legacy component (e.g. KLM, BA, AF) implements protective functions by wrapping each request (Section 4): all information going to and from the legacy componpent is checked using executable assertions reflecting the TA integrators' view on the correct behaviour of TA and of the legacy component [10]. In addition, it implements timeouts to detect communication problems (delays, traffic jams, lost connections) or legacy component failures (crashes, overloading, etc.). In our design each remote request issued by the action participant is structured as a simple atomic action which incorporates the error detection features discussed above and allows local handling of some situations typical for the Internet: analysis in [6] shows that for many web servers it makes sense to re-try after a short delay. If there is no reply after the second attempt, an exception is propagated to the action level.

We consider two types of internal action exceptions: exceptions signalled by individual component systems and exceptions detected at the level of actions. The component systems can return the following exceptions: no flights available, the component server is down. Possible ways of (cooperative) handling are ask for more expensive tickets, check spelling, check flights to the nearest airport, use another (redundant) service, ask the remaining services for more flights (if the action aims at providing a sufficient number of flights to the higher level action). If handling is not possible, an exception is signalled to the *compose_trips* action.

Exceptions of the second type are caused by the fact that component systems can return data that although correct from the component system point of view, do not exactly fit the requests. For example, some airlines regardless of the date you specify return offers a day or two before or after the specified date. These replies are filtered by the fa controller, and action participants may cooperatively decide to issue additional requests to some of the component servers.

5.4. Action request: multilevel exception handling

The **request** action includes typical activity accompanying remote information passing between the client and the TA CS. This information goes through multiple service and communication levels with known interfaces. Each of these levels represents a distinct environment, so error detection and exception handling are possible at each of them. In spite of the fact that these levels do not follow the same exception handling policy and often are not developed for use in dependable systems, it is important for TA to impose the unified approach to exception handling at all these levels: at each of them all exceptions thrown by the underlying level have to be caught and, if handling is not possible, signalled in the new context as the exceptions of the next level. These exceptions are parameterised to carry additional information facilitating exception handling at the higher level. Within this action the TA CS controller and the client handle exceptions of different types:

• exceptions caused by client's mistakes: the TA CS controller verifies the requests, checking, for example, spelling, correctness of travel dates, consistency of information given by the client, completeness of the request from the client, etc.

• all exceptions propagated from the underlying levels, including communication delays, crashes of the client machine or the TA server.

Note that the **consult_services** action contains similar information exchange between the client and the TA CS controller: it is incorporated in a nested action, which we do not discuss here as it is very similar to the **request** action.

5.5. Exception handling in other actions

In this section we briefly discuss several issues related to exception handling in the **booking**, **cancellation** and **payment** actions. In the first action, the validity of the credit card is checked; after that TA makes sure that there is sufficient money to cover the booking that each particular component delivers. If one of these checks detects a problem, a corresponding exception is propagated to the **booking** (or **payment**) action. This situation is handled cooperatively: the client is involved in this handling, she is asked to check the information about the credit card or/and, if possible, to introduce information about another credit card. TA incorporates some checks of the driving licence to ensure the car rental service; one of the ways to handling the invalid licence situation is by offering the client a taxi service.

Cancellation of a partially booked trip when some of its part (e.g. a hotel room) becomes unavailable after their availability was checked in the previous action is a serious issue in designing the **booking** action. This abnormal situation is handled in the following way. First, an attempt is made to find and book a replacement. If this is not possible, the booked parts of the trip have to be cancelled. Understanding the cancellation policies of different legacy servers is a very important issue. In our first design we use only refundable types of bookings. Another possibility that we have analysed is to use changeable bookings and to always complete the **booking** action with a booked trip should partial booking happen: this option requires the client involvement because some of the requirements of the trip may have to be sacrificed. In our future research we plan to develop more sophisticated exception handling techniques which will rely on using some of the money which TA makes for paying the cancellation fee (this require a very thorough analysis of the frequency with which the partial bookings occur).

6. Implementation

Our prototype implementation uses a distributed centralised solution, which is typical of such applications (Section 5.1). A centralised component (TA SS) offers web services to a number of clients via TA CSs (Figure 4). In addition, it exposes a call interface (RMI in our case) to allow TA to be further integrated. Clients access TA using web browsers in such a way that the http requests are transformed into RMI calls to the Java classes implementing the TA logic. These client requests are first passed to a Java server page (JSP) which provides a communication layer between TA CS and TA SS. This approach allows us to separate the TA application logic from dealing with TA interfacing. We are using a web technology defined by Sun in which JSPs are HTML documents that are interleaved with Java code, which in our case incorporate RMI calls to TA SS. The interface of TA SS includes methods checkname, cancel, book, checkAvailability, etc. Method checkname, for instance, checks whether a client has registered with the TA or not, returning the booking description when the client has registered.

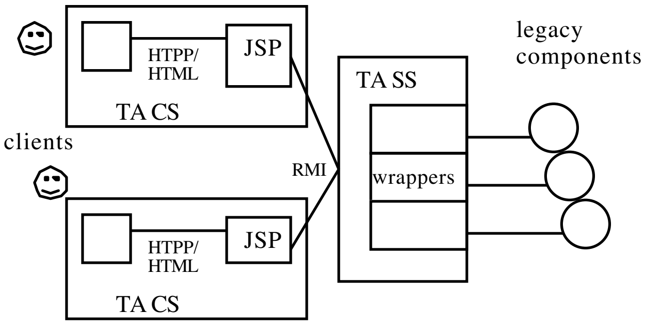

Figure 4. General implementation structure

TA SS maintains a time counter to throw a client-timeout exception if the client leaves the system without quitting. As we have explained before it is our assumption that the legacy components provide both their own web interface to the clients and RMI interfaces. Analysis of the latter is an important part of TA integration: among other things it allows the integrator to incorporate handling of the exceptions that the components can propagate.

In our design each access to legacy components issued by TA SS is wrapped into a special code implementing all functions described in Section 5.3 in such a way that each request represents an atomic action with a well-defined interface incorporating a rich set of interface exceptions. 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 SS 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.

CA actions are the main structuring technique we have applied in developing the TA. Several general CA action supports have been developed by now. Because of the fact that we had to apply a number of ready-made technologies and packages, and because of the heterogeneity of the environment we could not employ the same CA action support for implementing the whole TA system. In particular, the high level actions, such as session and availability have participants that are not explicitly programmed as processes (or threads) and are executed in different environments: the client controller, the TA CS controller and the TA SS controller. In our implementation of these actions we have used the design proposed in Section 5 and the CA action principles as the guidelines without employing any general CA action support (it is clearly in our plan to use the experience gained to develop such a support). The rest of the CA actions, including compose_trips and the nested ones, have been programmed using a distributed Java RMI framework support for CA actions [18]. This framework allows the implementation of the set of roles that compose the CA action as distributed objects, therefore the whole TA SS is designed as a distributed object system.

Our decision to distribute the role objects of each action of the TA system is important for improving the overall system performance in the situations when the TA is accessed by several clients at the same time. Using a distributed solution facilitates the distribution of the load when several clients are accessing the TA: different session actions can be executed on different TA machines leaving the control of the system on one machine. This solution will allow us to employ any known dynamic load balancing algorithms.

Section 5.4 describes a general structure of the request action; in our environment each HTTP request from the client is transported (via TCP/IP) to our server to be processed by a JSP servlet implementing the Java servlet interface: it takes as an input the HTTP requests and outputs HTML formatted code. Once the request is received from the servlet we remotely call an appropriate method to process the Java request further. The output of the call (RMI reply) is formatted in HTML by the servlet and returned to the client to be displayed as a web page. We have performed a detailed analysis of all exceptions which each of these levels can signal and for each level we have classified them into two categories: ones that can be handled (or, an attempt can be made to handle them) and ones that cannot be handled, in which case some interface exceptions are to be propagated to a higher level (note that we need a mapping here because exceptions of the higher level have to be expressed in terms of this level). In accordance with our approach no exceptions can be passed uncaught between levels. For example, in addition to the standard exception RemoteException we introduced a number of specific exceptions to allow a more focused handling to be executed and better diagnostic information to be reported to the client. In particular, we have introduced the following three exceptions which are signalled to the request action when the predefined timeouts expire: the TA SS is down, the client machine is down, the client timeout.

7. *Discussion*

One of our assumptions (Section 3) is that the designers have a complete description of legacy component call interfaces at their disposal. At present only a few web servers offer call interfaces to allow their integration; there are many reasons why their owners are reluctant to make them available for general public, one of which is the absence of standard technologies. This is why in our study we have been using only services which mimic the functionality of the existing web services. In spite of this, we believe that our research is an important contribution to the field. First of all, service trading is a very active area of development and standardisation, and it is only matter of time before call interfaces are offered for integration. Secondly, ours are general web-specific solutions which we believe will be useful when there are more services offering call interfaces. In particular, modern e-applications are built under the assumption that component call interfaces are known, so our approaches are directly applicable in this area.

Our second assumption is that legacy component crashes can be detected by timeouts, the approach which is used in most existing systems. The disadvantages of such solution are well known (see for example, [2). In response to this, several companies have been developing services that guarantee eventual message delivery in spite of network outages and node failures. IBM MQI [1] is an example of such a service.

8. *Conclusions*

This study shows that CA actions provide an optimal and powerful support for structuring integrated web applications. First of all, in such systems 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 which allows developers to deal with cooperative and competitive concurrency, and to employ application-specific and component-specific exception handling in a disciplined and structured way. Another relevant characteristic of CA actions is their ability to support structuring and fault tolerance of the complex systems which 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 [17, 13].

We realise that the results presented are preliminary in the sense that more work is needed to develop supports in a number of standard emerging web technologies (such as WSDL, SOAP, other XML-based techniques, etc.). It is our belief that it is important to apply known structuring and fault tolerance techniques as early as possible without waiting for such technologies to become mature and widely accepted. This allows the community to gain important experience and to become ready for the future developments in the area. We have chosen to use the Sun approach and made a number of assumptions, some of which may not be easy to guarantee in real-life situations.

This work has once again demonstrated the generality and power of the concept of CA actions, which have been successfully used in a number of applications before (see, for example, [16, 18]). This case study differs a lot from them, though, because the application area has a number of very specific characteristics (heterogeneity and complexity of the environment, autonomy and legacy of the web servers, needs to explicitly deal with node crashes and communication delays) that required serious adjustments in the way CA actions are used.

Another important conclusion is that in many practical situations it makes sense to apply specific structuring techniques tailored for particular needs: we refer here to protective wrapping discussed in Sections 5.3 and 6, and multilevel exception handling in action request (Sections 5.4 and 6). It is clearly difficult to apply the same structuring technique to integrating complex systems that incorporate legacy components of different types. Very often it makes sense to apply several of them in combination. In the TA case study we needed an extended CA action scheme to represent the overall structure of the system, multilevel exception handling to deal with a combined use of several ready-made environments, and protective wrapping to provide fault tolerance at the level of individual calls of legacy components. We believe a combined use of several fault tolerance and structuring techniques is an important direction of future research in application-level fault tolerance.

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

References

1. B. Blakeley, H. Harris, R. Lewis. Messaging & Queuing Using the MQI. McGraw Hill. 1995

2. N. Bowen, D. Sturman, T.T. Liu. Towards Continuous Availability of Internet Services through Availability Domains. Proc. of DSN'2000, 559-566. 2000

3. F. Cristian. Exception Handling and Tolerance of Software Faults. In Lyu, M.R. (ed.): Software Fault Tolerance. Wiley, 81-108. 1995

4. R.H. Campbell, B. Randell. Error Recovery in Asynchronous Systems. IEEE TSE-12, 8. 1986

5. K.H. Kim. Approaches to Mechanization of the Conversation Scheme Based on Monitors. IEEE TSE-8. 1982.

6. 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.

7. B. Liskov. Distributed Programming in Argus. CACM, 31, 3. 1988.

8. P.A. Lee, T. Anderson. Fault Tolerance: Principles and Practice. Springer-Verlag. 1991

9. C. Labovitz, A. Ahuja, F. Jahanian. Experimental Study of Internet Stability and Backbone Failures. Proc. of FTCS-29, Wisconsin. 1999.

10. P. Periorellis, J.E. Dobson. Case Study Problem Analysis. The Travel Agency Problem. Technical Deliverable CS1. Dependable Systems of Systems Project. 37 p. 2001.

11. P. Popov, S. Riddle, A. Romanovsky, L. Strigini. On Systematic Design of Protectors for Employing OTS Items. In Proc. of Euromicro-27. Poland, 22-29. 2001.

12. B. Randell. Recursive Structured Distributed Computing Systems. Proc. of the 3rd Symp. on Reliability in Distributed Software and Database Systems. Florida, 3-11. 1983.

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

14. B. Randell, J. Xu. The Evolution of the Recovery Block Concept. In Lyu, M.R. (ed.): Software Fault Tolerance. Wiley, 1-20. 1995.

15. J. Webber, V. Corrales, M. Little, S. Parastatidis. Making web services work. Application Development Advisor. November-December, 68-71. 2001.

16. J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver, F. von Henke. Rigorous development of an embedded fault-tolerant system based on Coordinated Atomic actions. IEEE TC-51, 2. 2002.

17. J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. Proc of FTCS-25, California, 499-509. 1995.

18. A.F. Zorzo, R.J. Stroud. An object-oriented framework for dependable multiparty interactions. Proc. of OOPSLA'99, ACM Sigplan Notices, 34, 10, 435-446. 1999.