Backward-Forward Transaction Service Design Pattern

Zdravko Roško
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zrosko@yahoo.hr

Abstract. *Two models of handling transactions; backward based transactions and forward based transaction compensation, need to be integrated into a model which can be used in distributed transactions, including web service transaction model. Backward based atomic transactions handling ACID (Atomicity, Consistency, Isolation, and Durability), two phase commit (2PC) and resource locking is not suitable for long-lived transaction or loosely-coupled nature and autonomy of web services. This paper presents the Backward-Forward Transaction Service Design Pattern which combines backward based transactions and forward based transaction compensation that can be used independently of the EJB, MTS or other technology supporting the two models.*

Keywords. Transaction, Pattern, Compensating Service Transaction, ACID, EJB, Data Access Object, Web Service, EIS.

1 Introduction

As more businesses migrate towards an e-business, integration with existing internal and external enterprise information systems (EIS) becomes the key to success for many businesses. Many companies need to integrate their existing EIS systems with new type of client applications, such as; mobile, web-based, web services, etc., as well as to provide a support for business to business (B2B) transactions (Figure 1).

A *business transaction* is an interaction in the real world, usually between an enterprise and a person, where something is exchanged [1]. For example, if an automated payment is made, the amount must be either both, withdrawn from first account and added to the second one, or nothing should happen. In case of a failure preventing transaction completion, the partially executed transaction must be "rolled back" by the transaction supporting system.

The general term "Transaction" has been introduced by Gray [2] and is defined by the four properties contained in the ACID acronym: Atomicity, Consistency, Isolation, and Durability. These properties guarantee that a system is maintained in a consistent state, even as transactions are executed within it concurrently. This includes the situations where one or more transactions fail to commit [3].

As new transactional applications have emerged, the limitations of classical transactions have been recognized and are well known. To handle transaction processing within these application domains, a number of alternative or extended transaction models have been suggested [4].

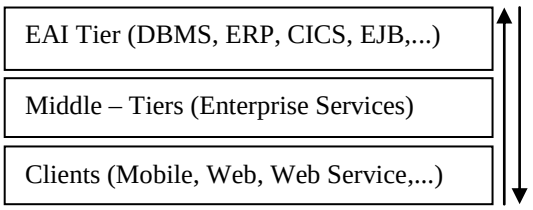

Figure 1. EIS environment

J2EE Connector architecture (JCA) part of Java 2 Platform, Enterprise Edition (J2EE) 1.3, specifies a standard architecture for accessing resources in diverse EIS, including ERP systems and mainframe transaction processing systems such as SAP R/3, IBM CICS, legacy applications and non-relational database systems. Currently JDBC Data Access API provides easy integration with relational database systems for Java applications. In a similar manner, the Connector Architecture simplifies integration of Java applications with heterogeneous EIS systems [5].
This paper presents the transaction model to be used outside of the technology supporting the two models

of transaction services (backward/forward). The model can service transactions within the client-server or three-tier application server environment. Backward-Forward Transaction Service (BFTS) presented here, is a compound design pattern which, among other features, addresses the missing elements from JCA including the following:

- Plain Old Java Object (POJO) transaction management implementation (not using J2EE application server).
- Compensating Service Transaction (CTS) [6] support.
- Client side transaction management in the diverse EIS environment.

The advantages of BFTS are:

- It provides a unified client programming model for accessing any back-end transactional system.
- It can execute within an application server context or as a stand-alone process.
- Support for connection pooling of diverse beck-end systems (JDBC, JMS, EJB, CICS, etc.)
- Support for transaction management within application server context or as a stand-alone process.

In the integration of a systems, the main concern is the unification of the design decisions and their assumptions [7]. The contribution of this paper is threefold and is concerned by unification of the design for transaction management in heterogeneous environment. First, the problem with the context of EIS perspective is presented. Second, it explains the solution which includes backward (ACID) and forward CTS transaction management. Third, Java implementation of BFTS is presented for showing a practical use of the pattern.

2 Context

A business transactions requiring the access to multiple types of data source systems (RDBMS, Web Services, legacy systems, ERPs, EJBs, CORBA services, and so forth), scale up for high performance, avoidance of errors due to concurrent operations, must be Atomic, Consistent, Isolated and Durable (ACID). They also require a uniform client programming model. BFTS design pattern is a compound pattern assembled from Sovereign Value Object [8], Dynamic Data Access Object [9], Pooling [10], Adapter [11], Command, Factory Method, Singleton, Façade, Proxy [11], and serves as a transaction client programming model.

BFTS solves a distinct problem, and not just a combination of problems of its contained patterns, including: connection handling, flat transaction handling, transaction timeout, multi-user concurrency, external transaction access (JTA, EJB, J2C) and Compensating Service Transaction [6].

Figure 2. shows BFTS Use Case for managing "Backward Transaction Service" to handle classic ACID type of transaction, and for managing "Long-lived Transaction Service" on the client and on the server side. Server side transaction service collects the transaction posts in the memory or in the persistence store such as RDBMS. "Forward Transaction Compensation Service" uses ACID as a mechanism for handling a collection of transaction "undo" actions in case a rollback is required for long-running transaction.

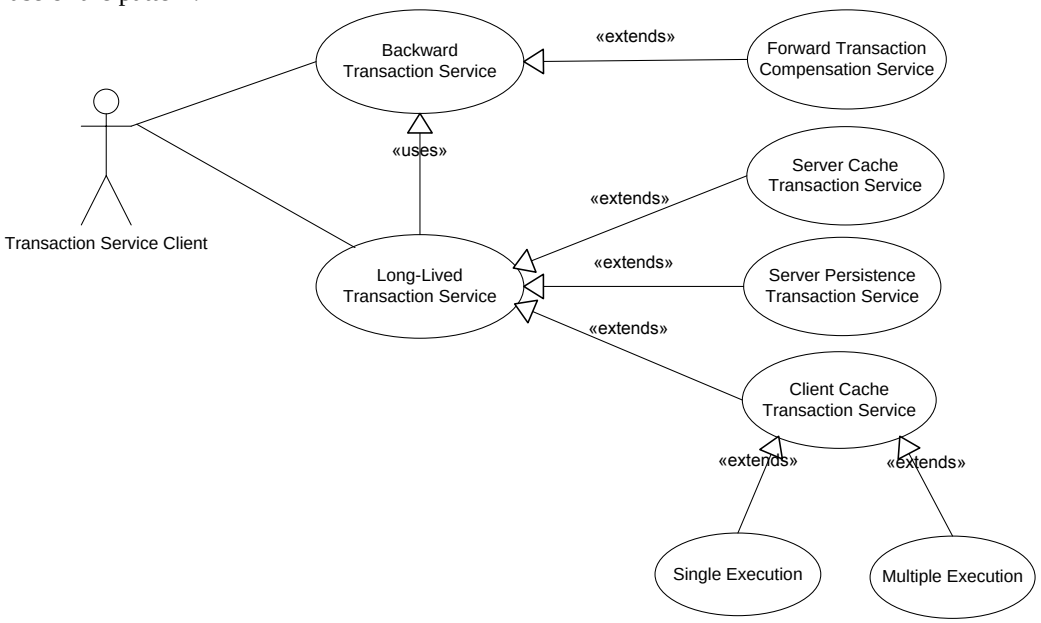

Figure 2. BFTS Use Case

3 Implementation

Transaction can exist within any Java application outside of the J2EE/JEE container. It also can exist and be used on the client or on the server environment execution. The service is configurable by simple setup parameters. Configuration of the transaction service is used to change the behavior of the service. Figure 3 shows the backward-based POJO classic ACID transaction environment which includes different beck-end systems, supported by the pattern, where *persistence* hides any data source system capable of supporting transactions (RDBMS, EJB, JMS, SAP, CICS, etc.).

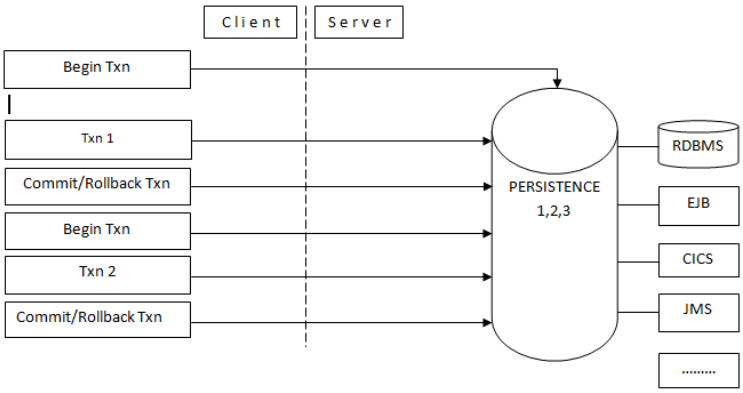

Figure 3. Classic ACID transaction

3.1 Compensating Service Transaction

When a global client transaction, composed of a number of smaller units of works, encounters failure condition, runtime exception is raised and termination of the global transaction is required. In case the global transaction is long-running service, it can severely degrade performance and can also reduce the scalability of the system. Therefore, instead of looking the resources, the global transaction is supplemented with compensating logic, which does not require maintenance of global transaction original state, or lock resources for the duration of the global transaction. Compensating actions are used as the extensions to the global transaction units of work via addition of "undo "capabilities. The actions are stored to the server or client cache, or to the database, and later used for rollback, in case a global transaction exception is raised. Compensating actions are restored in the form of Sovereign Value Objects [8] containing all the properties "undo" actions require (business data, target component and target service name) in order to execute the action as a command [11] to restore unlocked resources to the global transaction original state.

The actions can be designed to execute the compensating service with additional functions, such as sending out a notification about failure to the system administrator.

Application programmer needs to prepare "undo" logic for each service, where the compensation action is required. The "undo" logic is prepared in the form of an action (SVO).

Action is returned as an output parameter from the business component service to be cached and eventually used for compensation in case of failure of the global transaction.

3.2 Client side transaction management

To create a new long-lived transaction on the client, its representation must be stored at the client transaction collection, until the global transaction commit or rollback is called. The server is called at the end of client transaction, and the collection of the transaction data and actions is sent to the server at once, to be committed as one ACID transaction.

BFTS supports a sequential server calling after global transaction commits or rollback is called, but it cannot, however, guarantee the data consistency in case the commit or rollback fails in the middle of the process. The transaction can be roll backed to ignore all posting to the collection, in case a failure preventing the transaction completion.

The server side of the transaction service handles all other tasks such as connection *management* to many types of data sources, server side transaction coordination, security and logging.

Figure 4 shows client side transaction caching in case the client needs a full control of transaction compensation "undo" actions. The usage of this option requires a specific configuration parameters setup.

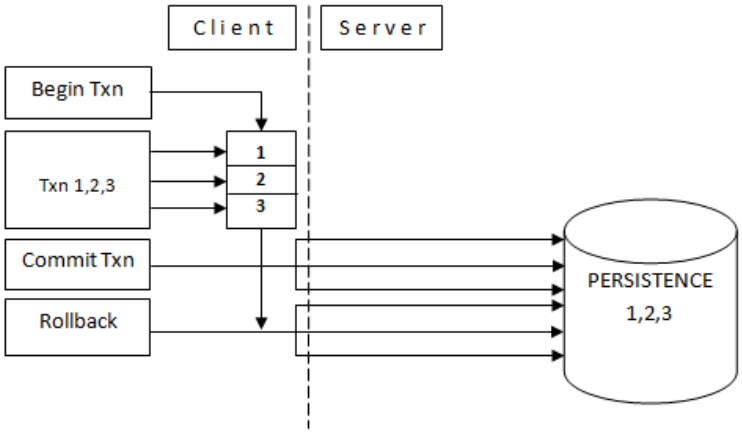

Figure 4. Client side transaction service

3.3 Server side transaction management

BFTS supports transaction data caching at the server memory cache or within the relational database. Client initiates a global transaction by calling the server to execute a business operation. Each client call to the server is saved and used later at the time a *commit* or *rollback* is requested from the client.
The transaction compensation data, in the form of "undo" actions, is also saved and used in the case a global transaction calls the rollback after the *commit* is done. Figure 5 shows the server side caching context.

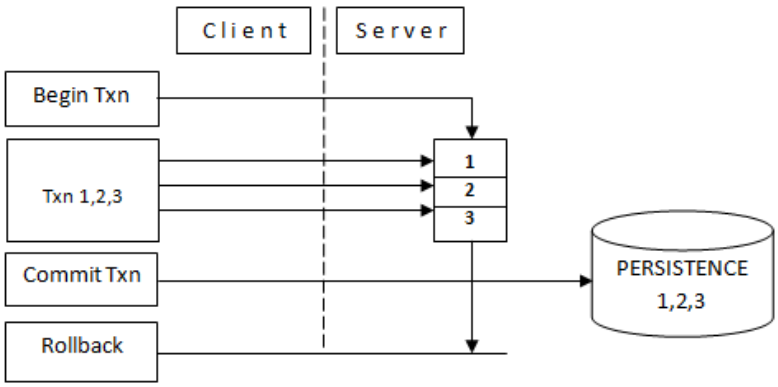

Figure 5. Server side transaction service

3.4 Structures

BFTS handles backward (ACID) transactions by using the Dynamic Data Access Object (DDAO) to transparently access business data from various beck-end systems.
Figure 6. shows the class diagram for BFTS backward transaction service.
The classes included are:

- J2EETransactionFactory handles transaction through various stages during its life cycle. The factory creates transaction per client request and uses the thread identifier as a unique key in multi-user environment. During its lifetime, a transaction passes through the following stages: No Transaction, Active, Committed, Rolled Back.
- J2EETransaction is an implementation of the *flat* transaction model which assumes that units of work are at the same hierarchy. The changes that are done as part of this transaction are either committed or aborted. This model suffers from a few limitations specific to a flat transaction models.
  J2EETransaction holds the connection pool of currently used connections to various back-end systems. The connections are extracted from J2EEConnectionPool and used during the lifetime of the transaction and returned back to the pool after commit or rollback of the transaction is done.

- J2EEConnection is an abstract class to represent various back-end systems such as: RDBMS, CICS, JMS, AS400, etc. and is used to handle logical unit of work across multiple type of data sources. It can be used as an abstract base class for implementation of any kind of back-end connection capable of supporting transaction management.
- J2EEConnectionJDBC is a concrete implementation of the abstract J2EEConnection and is used to handle all connection related work to JDBC data sources such as Oracle, SQL Server, My SQL, Sybase, DB2, etc.
- J2EEConnectionCICS is a concrete implementation of the abstract J2EEConnection and is used to handle all connection related work to CICS back-end data sources.
- J2EEConnectionJMS is a concrete implementation of the abstract J2EEConnection and is used to handle all connection related work to MQ and other JMS back-end data sources.

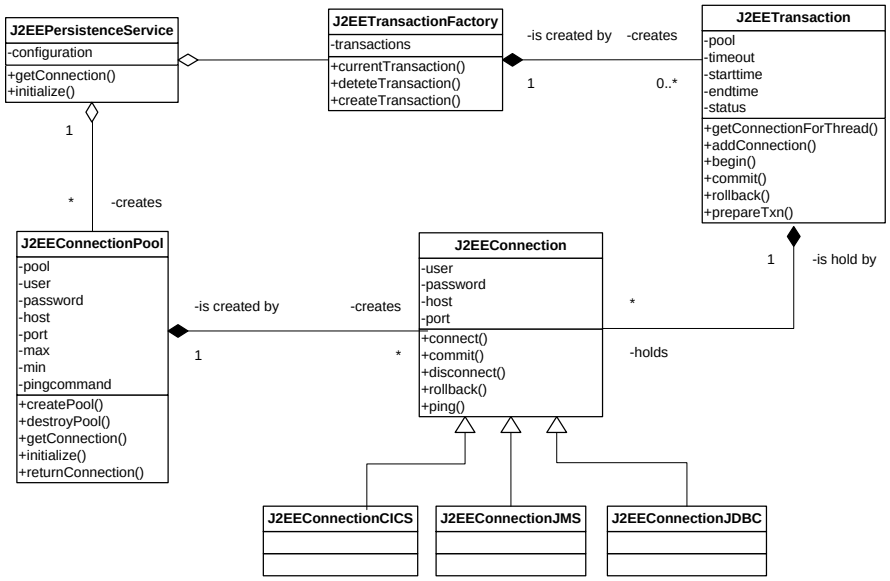

Figure 6. Backward based transaction

BFTS also handles forward based transaction service (Figure 7).

The classes included are:

- J2EETransactionCompensationFactory is used by the client to create and hold the transaction with its state.
- J2EETransactionCompensation is a client side service used to *begin, commit, rollback, add Step,* and to keep the transaction and compensation data and actions locally cached before they are sent to the server for processing.
- J2EETransactionCompensationService which handles multiple user transaction contexts by using transaction token generated by J2EETransactionCompensation on the client or by itself in case the client does not provide one at the start of a transaction. The class handles *begin, commit, rollback, rollback Forward* private operations called by the main *service* operation in order to handle the transaction. Compensation actions are executed in time descending order.
- J2EETransactionJdbc is a DDAO used to handle the persistence of the transaction data to the database.
- J2EECompensationJdbc is a DDAO used to handle the persistence of the compensation data to the database.
- J2EEApplicationController is a main entry point to the transaction processing. It can be configured to handle transaction management using JTA, EJB or BFTS transaction context.
- J2EETransactionCompensationFacade interface is the client API to BFTS component. The interface is implemented at the server (logical) side and on the client side. Server façade can be used by a client administration console to view the current state of transactions and compensations.

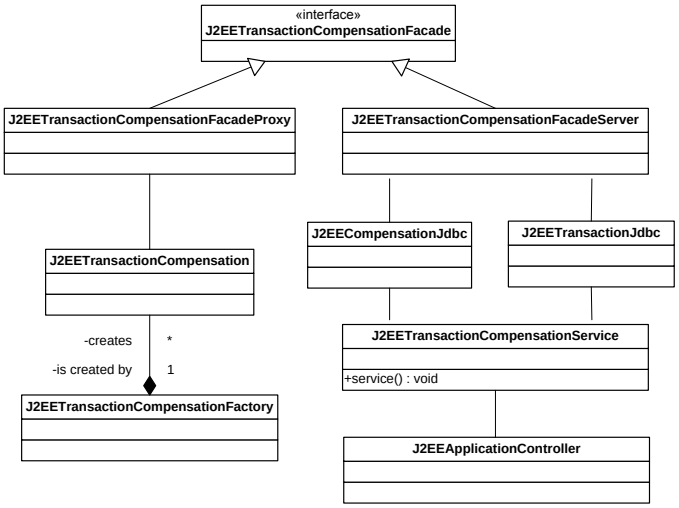

Figure 7. Forward based transaction

3.5 Dynamics

Figure 8. shows the message flow between the objects within the transaction context on the server. J2EEApplication controller is called by client to do a business transaction. The client can be a separated process or could be a fat-client within the same process as the logical server. The controller starts a transaction by calling *begin* operation on J2EETransaction which triggers the creation of the transaction by its factory. After the creation of the transaction for "current" thread, each connection to any back-end system, required by J2EEDataAccessObjects, is moved from the connection pool to the transaction object. At the end of the business logic, the transaction is either committed or rolled back by calling various beck-end systems implemented *commit* or *rollback* operations.

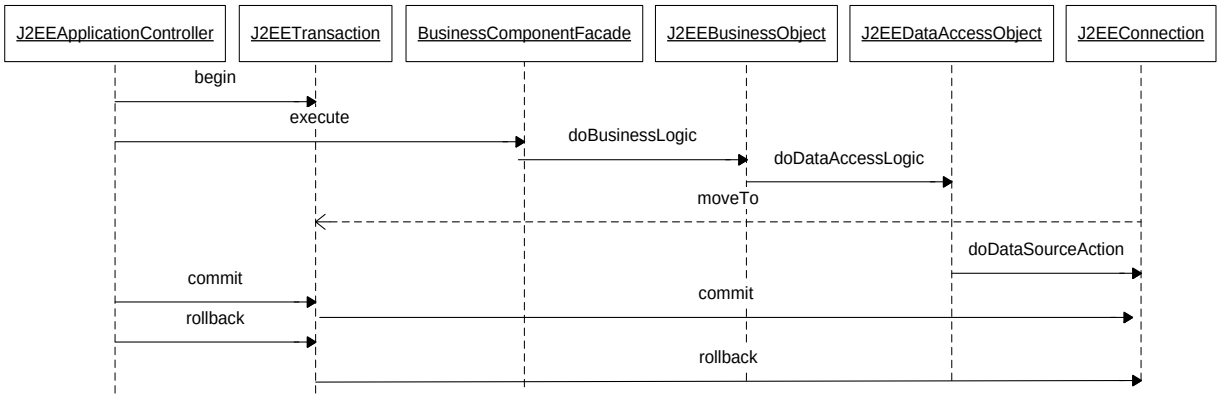

Figure 8. Server transaction

Figure 9. shows the message flow between the objects within the transaction context on the client in case a transaction compensation service is used.
Transaction compensation actions, with its data, are collected whether on the client or on the server. Client application is not aware of transaction handling mechanism because of its transparency. Client application initiates the transaction and calls a typical business component to execute business operations. The compensating transaction service handles the rest of the transaction work, such as collection of actions and compensations, and does the *commit* or *rollback* of the transaction or compensation actions.

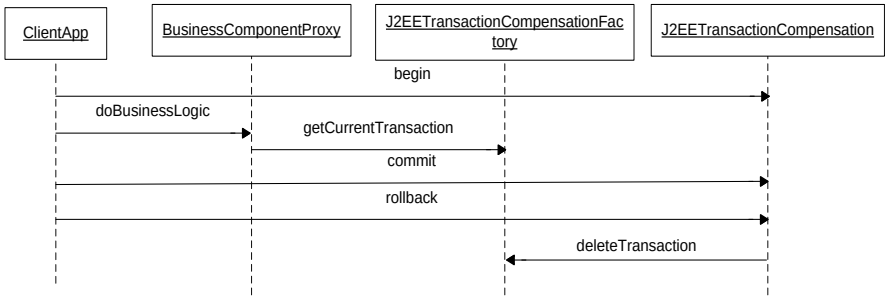

Figure 9. Client transaction compensation

4 Example Use Case

The Telecoms Equipment Inventory (TEI) Web services have been used by Physical Network Inventory (PNI) client application to provide centralized access to all processing and data associated with the telecoms equipment. TEI business objects and data are used by many other concurent applications, meaning that no long-running data or object looking is permitted. The PNI is a Graphical User Interface (GUI) application used to draw buildings and other locations associated with the telecoms equipment.

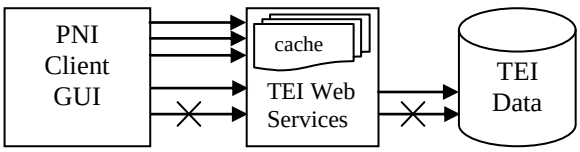

Figure 10. PNI Use Case

PNI holds a bulk of application data, being posted to TEI either at once after the local validation is passed or randomly after user completes each part of work. However the PNI logical unit of work, composed of many small parts, should be treated as a one transaction on the TEI side. PNI transaction is a set of related Web services that may need to be cancelled **after** they were executed. To handle this requirement a solution has to provide an „undo“ logic for each Web service called by PNI. PNI clients use BFTS to start a transaction, send multiple messages to TEI, and finally commit or rollback the transaction (Figure 10). Instead of having implemented one giant ACID transaction, a compensation-based approach treats each web service invocation as one ACID transaction, commited as soon as it has executed. The lock of resouces is reduced, but it means that cancellation has to be done by executing a separate ACID transaction that logically cancels the work after it was commited.

Global Transaction BEGIN	SmallWorld Transaction Begin
	BFTS Transaction Begin

Magik /command 1
Magik /command 2
status = AbcFacadeProxy.addEquipment(XML)
if status = OK
  status = AbcFacadeProxy.modifyLocation(XML)
  if status = OK
  status = AbcFacadeProxy.addLocation(XML)
if status = OK
begin
  Magik /command 3
  Magik /command 4
end
message = status /* OK-success, else it contains transaction step id */
/* Client with Exceptions handling does not need to check the status for OK */
/* The context of the transaction is owned by client program (assuming single threaded client*/	

Global Transaction COMMIT	BFTS Transaction Commit
	SmallWorld Transaction Commit
Global Transaction ROLLBACK	BFTS Transaction Rollback
	SmallWorld Transaction Rollback

Figure 11. Pseudo code Magik/Java

Figure 11. shows the pseudo code of a typical client usage of the BFTS transaction service. The client is using Magik programming language, accessing SmallWorld object database, and calls BFTS to access back-end server composed of JDBC, JMS and EJB systems. All data sources, including the object database, are part of a global transaction.

Conclusion

In this paper, BFTS pattern for handling long-running forward transactions, combined with backward based ACID transaction management, is introduced. The pattern can be used to build transactions and reliable transaction compositions with Web services and other distributed technologies. BFTS is used to capture transaction messages, save them, save the "undo" message for each transaction, and handle transaction logic toward any kind of back-end system (EJB, JDBC, CICS, JMS, etc.). BFTS does not require any container as a prerequisite.

Further, the BFTS follows a clean separation of concerns, having transactional properties isolated from other aspects of business logic. The design is also open for to accommodate other technologies other than Web services.

There are, however, some issues that need to be further explored, such as creation of "undo" logic for complex business operations, high-availability caching of transaction messages, business processes orchestration, fail-over and load balancing related issues.

References

[1] Philip A. Bernstein, Eric Newcomer. Principles of Transaction Processing For the System Professional. Morgan Kaufmann Publishers, page 2, 1997.

[2] Gray, J. & Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1993.

[3] Alexandros Marinos, RETRO – A (hopefully) RESTful Transaction Model. University of Surrey, 2009.

[4] Thomas Strandenaes, Randi Karlsen. Transaction Compensation in Web Services. Department of Computer Science, University of Tromso, 2002.

[5] http://java.sun.com/javaee/overview/whitepapers/connector.jsp, 2010.

[6] Thomas Erl. SOA Design Patterns, pages 632-638. Prentice Hall, Boston, 2009.

[7] Software Architecture as a Set of Architectural Design Decisions, Anton Jansen, Jan Bosch,2005, IEEE Computer Society Washington, DC, USA.

[8] Zdravko Roško: Sovereign Value Object, Facultyof Organization and Informatics University of Zagreb, IIS 2007.

[9] Zdravko Roško: Dynamic Data Access Object, Faculty of Organization and Informatics University of Zagreb, IIS 2008.

[10] Michael Kircher, Prashant Jain. Pattern-Oriented Software Architecture, page 97. John Wiley &Sons, 2004.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing company, pages 139, 233, 207, 185, 107, 127, 1995.