IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 November 2022, accepted 22 November 2022, date of publication 24 November 2022,
date of current version 1 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3224759

TUTORIAL

Fundamentals of Transaction Management in Enterprise Application Architectures

ANTONIO NAVARRO
Dpto. Ingeniería del Software e Inteligencia Artificial, Universidad Complutense de Madrid, 28040 Madrid, Spain
e-mail: anavarro@fdi.ucm.es

ABSTRACT Transaction management is a key issue in the development of enterprise application. During the payment of purchases, when dealing with bank operations or when making hotel reservations, transactions are everywhere. Curricula recommendations mainly consider specific knowledge units for transactions in the context of information management knowledge area. Thus, from a curricular point of view, transactions are usually related to Database Management Systems (DBMS). However, in the development of enterprise applications, designers and programmers use frameworks that manage transactions from the business tier. Therefore, there is a significant gap between the concept of transaction usually presented in degree courses and the real use of transactions made during the development of enterprise applications. There are excellent books that provide detailed descriptions of the transactional management in enterprise application from the business tier, but these are detailed and complex books beyond the reach of most students and, what is worse, beyond the reach of those lecturers without a significant background both in DBMS and enterprise application architectures. This paper provides a core of knowledge distilled from these books, as well as some examples of transactional architectures used in the grade software engineering courses taught by the author. The main goal is to describe in detail a knowledge unit focused on what I have called *service transactions* that helps to fill the gap between the learning outcomes provided in university courses and the use of transactions made in enterprise application development.

INDEX TERMS Service transaction, n-tier architecture, database transaction processing, transaction processing monitor, local transaction, global transaction, X/Open XA, JTA.

I. INTRODUCTION

Nowadays, terms like *artificial intelligence*, *big data* and *blockchain* are in the focus of society, industry, and academia. However, there is a significant number of *enterprise applications* [1], [2] that manage the business processes of key entities for society such as hospitals, banks and financial trading, or everyday activities such as online shopping. These applications are not in the present focus of attention, but without them much of the world's progress would not have happened. At the core of these applications lies *transactional management*. *Transaction* is a concept usually bound to *Database Management Systems* (DBMS): a database transaction symbolizes a unit of work performed within a database management system (or similar system) against a database and treated in a coherent and reliable way independently of other transactions. A transaction generally represents any change in a database. A database transaction must be *Atomic*, *Consistent*, *Isolated* and *Durable* (ACID) [3].

DBMSs and, specifically, *Relational* DBMS (RDBMSs) lie in the core of data management of enterprise applications because they handle the management of highly structured data very well in a concurrent and transactional environment. Transactions are so important in enterprise applications that many of the features found in application servers have their basis in transaction processing monitors [4].

The wide use of RDBMs in enterprise applications have promoted the development of automatic object-relational mappings in most platforms and frameworks for the development of enterprise applications such as JEE [5], Microsoft .NET [6] or Spring [7].

These mappings deal with both the persistence of objects and the transactional management of the operations that use them. Transactional management is so widely used in these

The associate editor coordinating the review of this manuscript and approving it for publication was James Harland.

operations that its presence is assumed in each operation and declaratively managed by platforms and frameworks. This management has to be explicitly configured but once done, it is implicitly performed by the platforms and frameworks, and programmers are excused from writing a code for transaction management. Therefore, this is called *declarative transaction management*.

From a technical perspective, it has taken too much effort and time for transaction processing systems to evolve to their current level of performance, fault tolerance and reliability [4], and declarative transaction management is a very useful tool for enterprise application development.

However, from a pedagogical perspective, there is a problem that has gone unnoticed for years: in curricula recommendations, transactions are only considered from an *information systems point of view*, and the *software engineering point of view* of transactions has remained unattended.

What is the main difference between the information systems and software engineering points of view? Information systems approach transactions from a DBMS point of view. Software engineering approaches transactions from the business logic point of view. Both of them are concerned with ACID properties, but information system transactions are about *database transactions* and software engineering transactions are about *service transactions*. For example, if a user makes a money transfer in a bank, the service transaction takes care of the amount of money that has to be transferred and has to check whether or not it is possible, which will end the service transaction with commit or rollback. And this is made with total isolation of the underlying DBMS in the resource tier. Database transaction takes care of providing an ACID management of the operations against the DMBS in accordance with what the service transactions expects. This is not an academic or professional rule for distinguishing transactions, but rather a convention used in this paper.

Not only curricula recommendations have failed to attend to the software engineering point of view of transactions, but even the most modern software engineering books such as [8] do not pay attention to this point of view of transactions.

A main question arises: what are the key elements of the software engineering point of view of transactions? Two: (i) service transactions; and (ii) the infrastructure that has to be provided in order to deal with the management (programmatic or declarative) of these service transactions.

Precisely, this paper focuses on the definition of service transactions and describes a simple solution for teaching how to build the necessary infrastructure for dealing with service transactions in enterprise applications. Three solutions are described in this paper according to the amount of resource managers and the presence of remote services. For the sake of conciseness, this paper focusses on *ACID transactions* and does not consider other *extended transactions models* such as *long-running transactions* [4], [9].

Because this paper has a pedagogical motivation (i.e., teaching transactions from a software engineering point of view) and no technical motivation (i.e., improving present infrastructure for dealing with service transactions), it defines a knowledge unit (KU) in the format of ACM/IEEE curricula recommendation providing core-tier1, core-tier 2 and elective topics and learning outcomes. This paper does not discuss whether or not present ACM/IEEE curricula recommendations should be revised in order to include this new knowledge unit. It only uses the ACM/IEEE format for defining academic knowledge about service transactions.

Using this curricular structure, core-tier1, core-tier2 and elective topics contents are provided in this paper with the frameworks defined by the author.

Thus, the paper is divided into the following sections. Section II reviews the related work and provides a basic vocabulary for transaction management. Section III defines a knowledge unit (KU) in the format of ACM/IEEE curricula recommendation. Section IV focusses on KU's core-tier1 contents (simple transaction management for local transactions without remote services). Section V focusses on KU's core-tier2 contents (simple transaction management for global transactions without remote services). Section VI focusses on KU's elective contents (transaction management for web services). Finally, Section VII presents conclusions and future work. Notwithstanding the complexity of the issue analysed in this work, and the paper's final length, an aim of conciseness is a cross-cutting concern in the paper in order to keep it within a *reasonable* size. Thus, for the sake of conciseness, the paper stresses the positive behaviour of transaction management, omitting the key mechanisms for failure recovering and transaction compensation.

II. RELATED WORK

This section gathers knowledge from different heterogenous domains. All of them are necessary because this paper deals with a technical computing science problem from an educational perspective. Thus, this section considers: curricula recommendations, information systems literature, software engineering literature, global transactions literature, literature about platforms and frameworks for the development of enterprise applications, documentation of frameworks for global transaction management and cloud computing literature. Finally, it provides a list of key terms extracted from the material analysed.

It is important to note that the classification provided in this section is not exclusive. Thus, for example, the same reference could be indexed under software engineering and global transactions. However, I have chosen an exclusive indexing in order to simplify this paper.

A. CURRICULA RECOMMENDATIONS

Curricula recommendations and bodies of knowledge can be a good starting point for lecturers willing to include some computer science-related concept in their courses.

ACM/IEEE have different curricula recommendations focused on computer science-related disciplines [10]. *ACM/IEEE Computer Science Curricula 2013*, CS2013, [11] considers different knowledge areas (KA), which are topical

areas of study in computing, such as *Computational Science* (CN), *Information Management* (IM), *Parallel and Distributed Computing* (PD), *Social Issues and Professional Practice* (SP) or *Software Engineering* (SE). They are divided into *knowledge units* (KU) with *core-tier1* (mandatory), *core-tier2* (important) and *elective* contents to be included in degree programs. Each type of content has a number of associated lecturer hours. The recommendations include learning outcomes that students should acquire as a result of understanding the knowledge areas. Each learning outcome has an associated level of mastery: *familiarity* (the student understands the concept), *usage* (the student is able to apply the concept in a concrete way) and *assessment* (the student is able to understand the concept from multiple viewpoints). In this paper I only consider those recommendations that are more closely related to the subject of this paper.

CS2013 provides curriculum guidelines for undergraduate degree programs in computer science. Transactions are considered in the following knowledge units:

- CN/ Data, Information and Knowledge. Elective content with the following learning outcome:
  - List and describe the reports, transactions, and other processing needed for a computational science application. [Familiarity]
- IM/Database systems. Core-tier2 content with the following learning outcome:
  - Describe the most common designs for core database system components including the query optimizer, query executor, storage manager, access methods, and transaction processor. [Familiarity]
- IM/Transaction processing. Elective content with the following learning outcomes:
  - Create a transaction by embedding SQL into an application program. [Usage]
  - Describe the issues specific to efficient transaction execution. [Familiarity]
  - Choose the proper isolation level for implementing a specified transaction protocol. [Assessment]
  - Identify appropriate transaction boundaries in application programs. [Assessment]
- IM/Distributed databases. Elective content with the following learning outcome:
  - Explain how the two-phase commit protocol is used to deal with committing a transaction that accesses databases stored on multiple nodes. [Familiarity]
- IM/Physical database design. Core-tier2 content with the following learning outcome:
  - Explain how physical database design affects database transaction efficiency. [Familiarity]
- PD/Communication and Coordination. Cores-tier2 contents with no specific learning outcomes.
- SE/Software design. Elective content with the following learning outcomes:
  - Apply component-oriented approaches to the design of a range of software, such as using components for concurrency and transactions, for reliable communication services, for database interaction including services for remote query and database management, or for secure communication and access. [Usage]
- SP/Privacy and civil liberties. Core-tier1 contents with the following learning outcomes:
  - Evaluate solutions to privacy threats in transactional databases and data warehouses. [Assessment]

Therefore, CS2013 basically considers transaction processing from an information management point of view, with mainly core2-tier and elective content. Only the elective KUs IM/Transaction processing, IM/Distributed databases and SE/Software Design slightly put the stress on service transactions and global transactions. It is important to remark that the learning outcome *create a transaction embedding SQL into an application program* is not enough to guarantee the teaching of service transactions as Section IV.C highlights.

ACM/IEEE *Competency Model for Undergraduate Programs in Information Systems 2020*, IS2020, [12] follows a slightly different approach to CS2013, translating the courses described in the previous curriculum guidelines published in 2010 into competency areas. In any case, its coverage of the concept of transaction is less detailed than CS2013. Interestingly, IS2020 removes the previous core course focused on *Enterprise Architecture*, which could be the item most related to the subject discussed in this paper.

ACM/IEEE *Software Engineering 2014*, SE2014, [13] provides curriculum guidelines for undergraduate degree programs in Software Engineering. The recommendation does not consider transactions in any knowledge unit.

Thus, considering CS2013, IS2020, and SE2014, only CS2013 mentions a few elective topics related to service transactions. This is the reason why I think that this paper is a contribution to computing education. I do not propose to extend ACM/IEEE curricula recommendations, but I think that this paper puts the stress on an interesting issue that has not been sufficiently taken into account by the academic world.

B. INFORMATION SYSTEMS LITERATURE

Transactional information systems are a key issue in software development. Therefore, there is a huge number of excellent books about transactional information systems. Choosing some references from this list can be very complex. In this paper [14], [15] are chosen as prototypical examples of what to find in this category. These books provide a wide view of DBMS, focusing on Relational DBMS (RDBMS), and putting the stress on the information systems point of view of transaction. Thus, they do not consider enterprise architectures or the management of transactions from the business

tier, because from a database point of view, this issue lies outside their scope.

There are fewer books that focus more specifically on transaction processing. In this paper [16], [17], [18] are chosen as prototypical examples of what to find in this category. These books study in more detail how RDBMS implement transactions, maintaining the ACID properties in a concurrent environment. This is a very complex issue that must be addressed by RDBMS designers and developers, but which lies outside the scope of this paper.

Therefore, information systems literature about transactions provides a necessary background for understanding and implementing service transactions, but totally ignores them.

C. SOFTWARE ENGINEERING LITERATURE

This paper focuses on the software engineering point of view of transaction management. Therefore, software engineering literature has been analysed.

The *Guide to the Software Engineering Body of Knowledge, SWEBOK V3.0* [19] describes generally accepted knowledge about software engineering. SWEBOK does not pay special attention to transactions either, and only considers them from a collateral point of view: 1.7.2. Trademarks and 14.1 Parallel and distributed computing overview.

This paper does not consider general purpose books about software engineering, because they provide a horizontal vision of the discipline, and does not delve into the details of transaction management. On the contrary, [1], [2] are key references for understanding the development of enterprise application, including transaction management. Reference [1] has an excellent chapter about concurrency and transactions (Chapter 5), and defines several patterns for transaction management, such as *transaction scrip*t and *unit of work*. Reference [2] gathers several patterns defined in [1] into the *domain store pattern*, which deals with the persistence of *business objects* [2] with dynamic load, in a transactional and concurrent application.

These two books teach three important issues:

- How to build n-tier enterprise applications from a pattern-based point of view, which underlies the main frameworks and platforms for enterprise application development.
- Application services are responsible for implementing the business logic and, therefore, they are responsible for starting and finishing transactions.
- How to build a simple transaction manager which deals with local transactions. *Local transactions* are those created and committed against a single resource manager (e.g., a RDBMS) [4].

However, they do not cover global transactions and, to some extent (specially [2]), pre-suppose certain knowledge about how application servers (and therefore transaction processing monitors) work.

D. GLOBAL TRANSACTIONS LITERATURE

Unlike local transactions, *global transactions* are those created and committed against several participants such as resource managers (e.g., several RDBMS) [20]. Global transactions are key elements of enterprise applications and professionals focusing on the development of these applications must understand global transactions in depth.

The literature about global transactions comprises a single category on its own. This literature lies in an intersection between information systems and software engineering literature, because, without previous knowledge about information systems and the architecture of enterprise applications, it is not possible to understand this literature.

The *X/Open XA Specification* [20] defines a distributed transaction processing model that envisages three software components: *application programs*, which define transaction boundaries and specify actions that constitute a transaction; *resource managers* (e.g., RDBMSs), which provide access to shared resources; and *transaction managers*, which assign identifiers to transactions, monitor their progress, and take responsibility for transaction completion and failure recovery. The kernel of the specification is the *XA interface*: the bidirectional interface between a transaction manager and a resource manager. X/Open XA is the core specification for global transactions and it is necessary to understand it before understanding advanced transaction processing presented in the rest of literature within this category. However, the single standard does not provide a detailed overview of how to build applications which deal with global transactions.

The X/Open XA specification is platform-agnostic, and this type of literature is a very good candidate for inclusion in curricular recommendation. On the opposite side, the book [21] focusses on transactions for the JEE platform and Spring framework. This is a concise and practical book that defines three types of transaction models: *local* (the programmer manages transactions using connections to RDBMS), *programmatic* (the programmer manages transactions using some type of transaction interface provided by a framework/platform) and *declarative* (the programmer does not write a code to manage transactions and the framework/container manages them). The book also introduces XA transaction processing and some specific patterns for transactions. It is a very useful book if the reader has some type of background about global transaction processing. Otherwise, it is hard to understand the inner infrastructure that underlies the code examples provided in it.

References [4] and [22] are excellent books for understanding the inner infrastructure that [21] does not describe. References [4] and [22] present the key concepts about global transaction processing from two different point of views. Reference [22] presents the concepts in a platform-agnostic way, and Reference [4] presents these concepts for the JEE platform. Both are excellent books and the key concepts depicted in this paper are mainly extracted from them. However, they do not include the specific transactional frameworks included

in this paper. These basic frameworks have been defined by the author of this paper for educational purposes only.

It is important to remark that declarative transaction management is the preferred method for transaction management because it is simpler, as it hides the complexities of transaction management, and separates transactional behaviour from business logic [21], [23]. In addition, declarative transaction management is usually integrated with some framework which implements a domain store pattern for the persistence of business objects, such as *Jakarta Persistence API*, JPA [24]. However, in the material provided in this paper explicit programmatic transaction management is made from application services in order to teach students the complexity that underlies transaction management.

E. LITERATURE ABOUT PLATFORMS AND FRAMEWORKS FOR THE DEVELOPMENT OF ENTERPRISE APPLICATIONS

The literature about platforms and frameworks for the development of enterprise application architectures includes some chapters about transaction management in these platforms and frameworks. There is a wide variety of literature for these frameworks and platforms and, in the case of information systems, this is a small selection from each one of them [23], [25], [26].

These books use the power provided by frameworks/platforms for the management of transactions. Therefore, although these books provide detailed examples about transaction management, the inner mechanisms used by the frameworks/platforms remain hidden to the programmer. Without hesitation, this is an advantage for senior programmers, but students in undergraduate courses are unable to see the complexity behind service transaction processing. This problem is not only present in the context of transaction management but inherent to the use of frameworks. For example, programmers can use the *Jakarta Server Faces* framework [27], which hides the complexity of implementing a *model-view-controller* [28]. But if programmers have never manually programmed a controller which maps incoming interface events in services and services responses in views, they will never understand the true essence of the model-view-controller pattern.

F. DOCUMENTATION OF FRAMEWORKS FOR GLOBAL TRANSACTION MANAGEMENT

Frameworks for global transaction management provide an interesting documentation that can help to understand the management of global transactions. In the context of Java, the key API for transaction management is *Jakarta Transactions* (JTA). *Atomikos* [29], *Bitronix* [30] or *Narayana* [31] are good examples of JTA implementations and their documentations provide interesting examples of how to deal with global transactions.

However, in general, these are technical documents, explicitly focussed on the programming platform (JEE in the previous examples). Thus, they are more focussed on describing how to configure and use the frameworks than on providing a pedagogical description of the global transaction management.

G. CLOUD COMPUTING LITERATURE

During the last years, cloud computing platforms such as Amazon Web Services, Google Cloud Platform or MS Azure have gained the focus of the market for the development of enterprise applications [32]. Enterprise applications built on these services usually rely on *microservices* [33], which use some type of *eventual consistency model* for dealing with transactional management. In this model, a business operation consists of a series of separate steps. While these steps are being performed, the overall view of the system state might be inconsistent, but when the operation has been completed and all of the steps have been executed, the systems should become consistent again [34]. This type of transactional management is analysed in Section VI. It is well worth mentioning the proposal made by [35] for a concurrency control protocol which allows greater concurrency across multiple microservices. However, this is an isolated approach that has not been generally adopted by the industry.

Cloud computing also includes other types of transactions which lie beyond the scope of this paper. Some of these approaches are closer to the underlying cloud infrastructure than to enterprise applications. For example, Reference [36] defines a *Virtual Machine Interface* (VMI) platform for transactional modification. Reference [37] presents a scalable protocol for transaction management in a key-value-based multiversion data storage system supporting partial replication. Reference [38] proposes a unified and comprehensive *Remote Direct Memory Access* (RDMA)-enabled distributed transaction processing framework supporting multiple concurrency control protocols.

Other transaction-related bibliography in cloud computing is focused on data stores not specifically intended for enterprise applications. Reference [39] presents an approach for transaction management in a cloud-based database. Reference [40] proposes a concurrency control method to support transaction processing capability for *Cloud Data Management Systems* (CDMS).

Finally, there is a range of cloud-based data stores from the academy which include transactional management. Reference [41] analyses these data stores, which are not considered in this paper because they are not being adopted by the industry for the development of enterprise applications.

H. KEY TERMS

Vocabularies (i.e., lists of key terms) are key parts of specific domains. Thus, coming into contact with the vocabulary is a key issue for understanding a domain. At a first glance, vocabularies are difficult to understand, but as the knowledge about a domain grows, vocabularies are better understood, and reciprocally, vocabularies improve domain knowledge. In addition, vocabularies identify a group of terms which are key in the understanding of a specific domain. This section defines some basic vocabulary used in service

transaction processing. Terms are presented from the more basic to the more advanced ones, which need basic terms to be understood. Some terms about multitier architecture are also included.

- *Enterprise application*: applications with the following characteristics: a significant amount of persistent data, which is accessed concurrently, a significant amount of user interface screens and complex business logic [1].
- *Business logic*: any logic associated with providing some service. This includes all logic related to processing, workflow, business rules, data and so forth [2].
- *Business object*: a class that separates business data and logic using an object model [2].
- *Application service*: a class that centralizes and aggregates behaviour to provide a uniform service layer [2].
- *Remote Procedure Call* (RPC): a programming mechanism that enables a program in one process to invoke a program in another process using an ordinary procedure call, as if the two programs were executing in the same address space [22].
- *Session façade*: a class that encapsulates business-tier components and exposes a coarse-grained service to remote clients using Remote Procedure Calls [2].
- *Web service broker*: a class that exposes and brokers one or more services using XML and web protocols [2].
- *Service activator*: a class that receives asynchronous requests and invokes one or more business services [2].
- *Business service*: In this paper, I consider business services to include any class which implements business logic. For the sake of simplicity, business services can be considered application services, because they implement business logic whether applications are directly invoked by local processes or are remotely invoked from other classes which access them using session façades, web service brokers and/or service activators. A business object can implement business logic, but if this logic is invoked by actors, I am prone to include it withing application services.
- *Application server*: a server that hosts applications [42].
- *Container*: the logical partitions of an application server [42].
- *Transaction*: a complete unit of work [43].
- *ACID transaction*: Atomic, Consistent, Isolated and Durable transaction [3].
- *Long-running transaction*: transaction that cannot be synchronously resolved [4], [9].
- *Extended transaction models*: transactions beyond the ACID paradigm (e.g., long-running transactions) [4].
- *Business transaction*: an interaction in the real world, usually between an enterprise and a person or another enterprise, in which something is exchanged [22].
- *Service Transaction* (ST): a transaction started and finished from a business service, in particular from an application service. This concept is further developed in Section V.A.
- *Transaction demarcation*: a mechanism that offers the application programmer commands to start, commit and abort a transaction [4].
- *Transaction bracketing*: transaction demarcation [22].
- *Transaction attribute*: mechanisms in object-oriented languages which allow one to define an implicit/declarative transaction demarcation in methods [4].
- *Transaction composability*: the ability of different transactions to become part of the same transaction with a behaviour consistent with the business logic implemented. This is an adaptation of the term *composability problem* defined in [22].
- *Compensating transaction*: a transaction that reverses the effect of an already committed transaction [22].
- *Connection passing*: a technique that establishes a database connection at the higher-level method (e.g., an application service's method), and passes the connection into the DAOs [21].
- *Application Program* (AP): the program that defines transactions and accesses resources withing transaction boundaries [43]. In this paper, application services are the core of application programs.
- *Resource Manager* (RM): a software that manages a certain part of the computer's shared resources. RDMBS or print servers are good examples of RMs [43].
- *Transaction Manager* (TM): a software component that assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery [43].
- *RM transaction*: a transaction performed by a RM. This concept is further developed in Section V.A.
- *X/Open Distributed Transaction Processing (DTP) Model*: a model in which APs that use resources from a set of RMs call TMs to structure transactions [20], [43], [44].
- *Local transaction*: a transaction created and committed against a single RM [4].
- *Global transaction*: a transaction that involves many RMs in a single unit of work [20], [43], [44].
- *Distributed transaction*: a global transaction [20], [43], [44], [45].
- *Nonglobal transaction*: a transaction in which, if two or more RMs are involved, there is no coordinated commitment between them [44].
- *Transaction Processing Monitor* (TPM): a product that supports the development of transactional applications/systems [4]. A TPM coordinates the flow of transaction requests between the client processes (APs) that issue requests and the back-end servers that process them. Basically, a TPM coordinates transactions that require the services of several different types of back-end processes, such as application servers and RMs distributed over a network. A TM is usually provided by a TPM [45].

- *Transaction context*: common information propagated between components involved in a transaction distributed among several machines [20].
- *Transaction branch*: A global transaction has one or more transaction branches (or branches). A branch is a part of the work in support of a global transaction for which the TM and the RM engage in a separate but coordinated transaction commitment protocol. Each of the RM's internal units of work in support of a global transaction is part of exactly one branch [20]. Reference [46] delves into this concept: a transaction branch is the part of the work between a TM and an RM that supports the global transaction. A global transaction could have multiple transaction branches when multiple RMs are accessed through one or more application processes that are coordinated by the TM.
- *Transaction identifier*: Each transaction has a unique transaction identifier (ID), which is assigned when the transaction is started (by a TM or a transactional RM). It is used to tell the RM on which transaction's behalf the access is being made. The RM needs this information to enforce the ACID properties. Transaction ID is usually hidden in a transaction context. There are two major types of transaction IDs: *global* (more than one RM participates in the transaction) and *local* (IDs assigned by RMs and correlated to the global transaction ID) [22].
- *XID*: a key used to identify a transaction branch [20]. It contains the global transaction identifier and a branch qualifier. The AP defines the start and end of a global transaction by calling the TM. The TM assigns an identifier to the global transaction and informs each RM of the XID on behalf of which the RM is doing the work [20]. Reference [46] delves into this concept: XIDs are assigned by the TM to identify both the global transaction, and the specific branch within an RM.
- *Exception handler*: Application programs that bracket transactions must provide error handling for two types of exceptions: transaction failures (e.g., a rollback) and system failures (e.g., an unsolicited abort by zero division). For each type of exception, the application should specify an *exception handler*, which is a program that executes after the system recovers from the error [22].
- *Savepoint*: a point in the program in which the application saves its whole state, generally by issuing a *savepoint command*, which tells the database system and other resource managers to mark this point in their execution, so that they can return their resources to this state later, if asked to do so [22].
- *Two-phase commit*: A commit encompassing two phases [20]:
  - Phase 1: RMs are requested to inform whether or not they can commit.
  - Phase 2: according to the result of phase one, RMs are requested to commit or abort.

  Usually, the TM is responsible for running the two-phase commit protocol, performing both the coordinator and participant functions for a group of transactions (interposition). It usually runs a two-phase commit for all transactions that execute on its machine. To do this, it communicates with RMs on its own machine and with TMs on other machines [22].
- *Coordinator*: a component that runs the two-phase commit protocol on behalf of the transaction. The coordinator receives the commit or abort request from the application program and drives the execution of the protocol [22].
- *Participant*: an RM involved in a two-phase commit transaction [22].
- *Interposition*: the use of coordinators as subordinate coordinators/participants of an ongoing transaction. Each domain (machine) that imports a transaction context may create a subordinate coordinator that enrols with the imported coordinator as though it were a participant. Any participants that are required to enrol in the transaction within this domain actually enrol with the subordinate coordinator [4]
- *XA interface*: an interface that enables the TM to structure the work of RMs into global transactions and coordinate the global transaction's completion and recovery (in case of failure) [20]. AKA *TM-RM interface* [44].
- *TX interface*: an interface that allows the application to delimit the global transactions. AKA *AP-TM interface* for global transaction demarcation [44]. An application program starts and completes all transaction control operations through the TM using the TX interface [45].
- *Native interface*: an interface that enables the AP to access shared resources (e.g., JDBC). AKA *AP-RM interface* [44].
- *Communication Resource Manager (CRM)*: a component that controls communication between distributed applications [44].
- *Communication Protocol Stack*: the component that provides the underling communication services used by distributed applications and supported by CRMs [44].
- *XA+ interface*: an interface that supports the global transaction information flow across TM domains. A TM uses the XA+ interface to communicate with remote TMs participating in global transactions. AKA *TM-CRM interface* [44].
- *Object Transaction Service (OTS)*: an interoperable transaction service specified by the Object Management Group (OMG)) [47], [48] in CORBA.
- *Java Transaction Service* (JTS): this is the Java language mapping of the CORBA OTS 1.1 Specification [4], [21]. Programmers are unaware of JTS. It is not mandated by JEE; it is mandatory for the interoperability of distributed transactions between heterogeneous implementations.
- *Java Transaction API* (JTA): this is the Java API for transaction management [4], [21]. Programmers use JTA for demarcating transactions. JTS is the underlying transaction service on which JTA is based.

- *Eventual consistency* [34]: model of consistency where the system can be temporary inconsistent but it is enforced to become consistent in a point in the future.

III. PROPOSED KNOWLEDGE UNIT

Section II.A describes the division of ACM/IEEE curricula recommendations into KAs and KUs. This section provides a KU, which describes the contents about service transactions not covered by curricular recommendation such as CS2013 or SE2014. The aim of this paper is not to propose the modification of such recommendations. It only uses the CS2013 KU format to describe the contents, learning outcomes and lecturer hours that I propose.

However, if this KU were a part of CS2013, it could be included within SE KA, and it would then build directly on the foundation provided by IM/Transaction processing and SE/Software Design KUs. The proposed KU could have a slight conflict with the KU IM/Distributed databases that should be considered when preparing specific curricula. Section IV.C deals with this topic.

If this KU were a part of SE2014, it could be included within Software Design (DES) KA, and it would then build directly on the foundation provided by Computing Essentials/Database fundamentals DES/Architectural design, and DES/Detailed design.

The inclusion of this KU in IS2020 is more complex due to the structure of IS2020. However, its learning outcomes could be included as competencies within the Application Development and Programming area of the Development competency realm.

For the description of this KU, I use the CS2013 format because it is more detailed than the formats used by other recommendations. The knowledge area is omitted, because, as I have already mentioned, it would be SE in CS2013, DES in CS2014 and Development competency realm in IS2020. The lecturer hours for core-tier1 contents are directly calculated from my experience teaching this subject during ten academic years. The lecturer hours for core-tier2 and elective contents are estimated from my experience with core-tier1 subjects.

Service transactions

Topics:

[Core-Tier1]

- Data persistence in concurrent environments. Optimistic and pessimistic concurrency control.
- Application programs and resource managers.
- Resource manager transactions and service transactions.
- Transaction demarcation.
- Transaction managers.

[Core-Tier2]

- Global transactions, transaction context and distributed transaction processing. XOpen/XA.
- Transaction branch, global transaction identifier and XID.
- Two-phase commit.
- Transaction managers for two-phase commit.
- Platform-specific frameworks for global transaction management.

[Elective]

- Web services and transactions.
- Persistence APIs. Transaction management.

Learning outcomes:

[Core-Tier1] 12 hours (6 theory + 6 practice)

1. Explain the problems related to data persistence in concurrent environments. [Familiarity]
2. Use optimistic and pessimistic concurrency control to avoid problems related to data persistence in concurrent environments. [Assessment]
3. Explain the roles played by application programs, resource managers and transaction managers. [Familiarity]
4. Explain the difference between those transactions related to resource managers and those transactions managed by elements belonging to the business tier. [Familiarity]
5. Use transactional demarcation in application services for managing transactions in the business tier. [Usage]
6. Build a transaction manager for applications dealing with a single resource manager and no remote services. [Usage]
7. Build an n-tier application using service transactions. [Usage]

[Core-Tier2] 9 hours (5 theory + 4 practice)

1. Explain the difference between local and global transactions. [Familiarity]
2. Describe the main elements of the XOpen/XA standard. [Familiarity]
3. Describe the two-phase commit. [Familiarity]
4. Use transactional demarcation in application services for managing global transactions in the business tier. [Usage]
5. Build a transaction manager for applications dealing with several resource managers and no remote services. [Usage]
6. Build a n-tier application using global service transactions with two-phase commit. [Usage]
7. Describe some platform-specific frameworks for distributed transaction processing. [Familiarity]

[Elective] 4 hours (2 theory + 2 practice)

1. Explain transaction management in web services. [Familiarity]
2. Explain how to use a persistence framework for dealing with data persistence in concurrent transactional environments. [Familiarity]

In the courses taught by the author of this paper, core-tier1 contents are explained and JPA is also explained and used instead of the core-tier2 contents described in this KU. An incremental approach is chosen: in a project done in the second course, a simple enterprise application with no transaction management is made. In the first part of the project done in the third course, the simple enterprise application

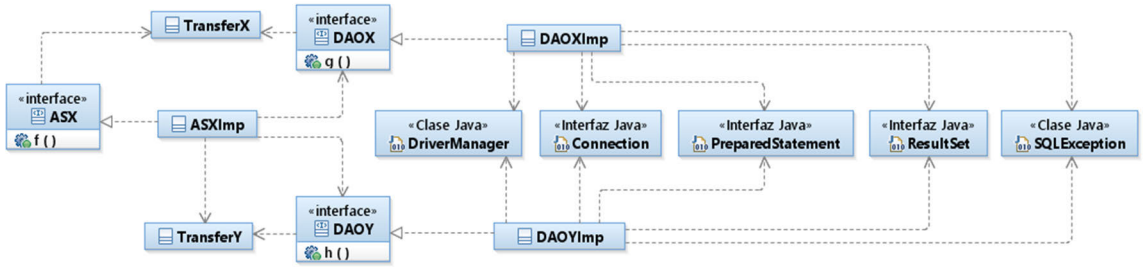


FIGURE 1. Basic multitier application based only on transfers, application services and DAOs.

is enhanced with the transactional management explained in core-tier1 contents. Finally, in the second part of the project made in the third course, the application is enlarged with some modules that use JPA for dealing with the persistence and transaction management. This enforces the presence of two sets of tables (in the same or different schema): one for those accessed from the non-JPA components and the other for those accessed for the JPA components. This characteristic is used to illustrate the need for a two-phase commit protocol, but in practice, it is never implemented because there is no business logic that simultaneously involves tables managed and non-managed by JPA.

In order to keep the KU defined in this paper within a reasonable time range, no persistence APIs are included in it, and they are only mentioned as elective contents. If a persistence API (e.g., JPA) is included with the intent of using it in core-tier1 and core-tier2 contents, the duration of the KU should be double.

The next sections describe the main concepts and references for teaching core-tier1, core-tier2 and elective contents. I have chosen to embody these contents in the context of a specific platform. Due to the restrictions in my school and the programming knowledge of the students, this platform is JEE. Between JEE standards and Spring, I have chosen JEE standards because they are supported by more vendors. In any case, other platforms such as .NET or Spring could be used instead with no significant changes.

IV. CORE-TIER1. SERVICE TRANSACTIONS FOR ONE RM AND NO REMOTE SERVICES

This section describes in detail core-tier1 contents for the proposed KU. As the title indicates, these contents are focused on providing students with a detailed view of service transactions which only involves one resource manager and does not consider remote services. The simple transaction manager presented in this section has been defined by the author taking into account the ideas suggested by [1] and [2] and has been in use for, at least, ten academic courses about software engineering taught by the author. This transaction manager has also been used in the development of the Virtual Campus of the Universidad Complutense de Madrid [49].

The section begins with a basic multitier architecture that the students must already have mastered before presenting the concept of service transaction. Then it provides important references for teaching the main concepts related to local service transactions. Finally, it describes the transaction manager used by the author in his courses.

A. BASIC MULTITIER ARCHITECTURE

The description of multitier architecture lies outside the scope of this paper and can be fully described in two catalogues of design patterns [1], [2]. A brief description and a design used in the development of the Virtual Campus of the Universidad Complutense de Madrid can be found in [49].

Before presenting the concepts of local service transactions, the students should be able to build an application using the following patterns: some type of *controller* (closer to the concept presented in [28] than to the dual *front/application controller* presented in [2]), *transfers* [2], *transfer object assembler* (TOA) [2], *application services* (ASs) [2] and *data access objects* (DAOs) [2].

One of the most underestimated issues in multitier books is the mapping of the domain relationships to DAOs and transfers. In order to avoid problems with dynamic loading, it is better not to include pointers between transfers. Thus, if there is a one to many relationship (e.g., `Department-Employee`), the transfer of the side with multiplicity N holds the identifier of the side one (e.g., `TEmployee` should have `departmentId`), and its DAO should have an operation to read them (e.g., `EmployeeDAO::readEmployeesBy DepartmentId (departmentId:int) :TEmploy ee[*])`. This is applicable to many to many relationship also. If there are attributes belonging to the relationship (e.g., the `quantity` of `Products` in an `Order`) the solution is straightforward because there is an intermediate class `(OrderLine)` which holds two many to one relationships with both ends `(Order and Product)`. If there are not these attributes (e.g., `Ingredient-Drink)`, there is no need for such a intermediate class, and the DAOs of each end can have a function to read the N elements of the other end (e.g., `IngredientDAO:: readIngredientsByDrinkId(drinkId:int):TI ngredient[*]` and `DrinkDAO::readDrinks By IngreditentId(ingredientId:int): TDrink [*])`. An intermediate DAO can be provided in order to deal with the binding and unbinding of both ends

(e.g., `Ingredient_DrinkDAO::bind(ingredientId: int, drinkId:int):int`). If a use case requires to get several related elements (e.g., a department with its employees) a TOA can be used to generate a transfer gathering this information (e.g a `TDepAndEmployees` with a `TDepartment` and several TEmployees).

In this application, no transactional management is made, either from a business or integration tier, and if a RDBMS is used (students can use a plain file system if they wish), this is supposed to be with the auto-commit feature on. The main goal is to facilitate the teaching of multitier application to those students who might not be familiar with relational databases.

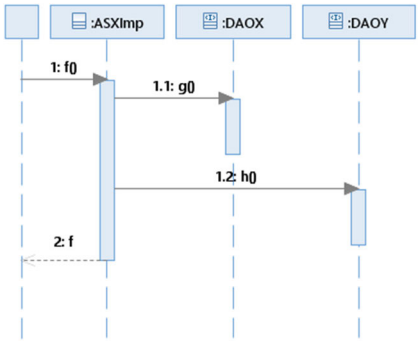


FIGURE 2. Idealized interaction between application services and DAO objects whose classes are defined in Fig. 1.

Although courses should be as general as possible, students are expected to build programs and some programming language has to be used. As I have previously mentioned, Java is the chosen language in the author's courses. Fig. 1 provides a UML class diagram for the business and integration tiers of application that the students are able to build because its architecture has been presented in the previous academic year. Fig. 2 provides a UML sequence diagram for a prototypical interaction between application services and DAOs. Note that in Fig. 1 a RDMBS underlies the integration tier and JDBC [50] is used to connect it with the application program. In order to avoid a problem of future transaction composability, application services are not allowed to invoke other application services. In this case, only one RDBMS is used, and JDBC `Connections` are created and closed by DAOs using the `DriverManager` class. I am aware of the fact that this is not very efficient, but at this point of the learning process, the search for efficiency is not a goal. In the event of students not being familiar with JDBC [51], it is a very simple and useful reference.

Multitier architecture patterns focus on logical tiers, but these tiers have to be mapped to physical tiers. Thus, a multitier application could be deployed in a machine (whether physical or virtual) that holds the presentation tier (e.g., a servlet container), the business and integration tiers (e.g., a Java application) and the resource tier (e.g., a SGBDR). However, at least three machines are usually present in the deployment of a multitier application: a web server/servlet container for the presentation tier, an application container for the business and integration tiers and a database server for the resource tier. If several machines are used for the deployment of the business logic or the deployment of the databases, then global transactions arise whenever elements of the same tier deployed in different machines become involved in a transaction.

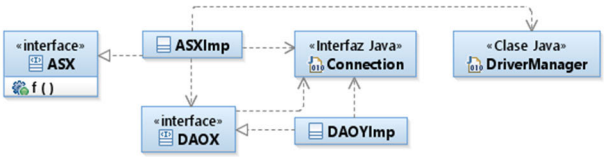


FIGURE 3. Simple but undesirable way to manage local transactions through connections in applications services.

B. LOCAL SERVICE TRANSACTION CONCEPTS

Taking into account the basic multitier application presented in figures 1 and 2, the goal of the core-tier1 contents is to introduce students to the definition and use of service transactions in multitier architecture application that they already know. At this point, it is also necessary that students have knowledge of relational databases and their access from applications using some type of object-database connectivity such as JDBC. Therefore, it is assumed that they are able to build applications like those described in Fig. 1, based on RDBMSs and not on Java-accessed files.

Before talking about service transactions, it is necessary to present fundamental concepts about concurrency problems, isolation, optimistic and pessimistic concurrency control, and to be sure that students are already familiar with RDBMS transactions and their ACID properties. All these concepts are very well described and presented in Chapter 5 of [1]. After that, *business objects* and *domain store* patterns [2] can be presented. The main idea is to keep the application simple, without business objects (whose dynamic load increases the complexity), but taking into account the ideas about transaction managers presented in the domain store pattern (which indeed is based on patterns presented in [1]) to be able to build a transaction manager that can deal with service transactions bound to the control thread. The next section covers the building of such a transaction manager. This transaction manager, and those defined in Section V and described in Section VI are elements belonging to the integration tier, which are used in the business tier.

C. INCREMENTAL DEFINITION OF A TRANSACTION MANAGER FOR LOCAL SERVICE TRANSACTIONS WITH NO REMOTE SERVICES

If I had to implement a commercial Java application that manages persistent data in a single RM, I would choose JPA as the persistence mechanism, and the `Jakarta.persistence.EntityTransaction` interface for the management of local transaction. However, this type of implementation hides significant details related

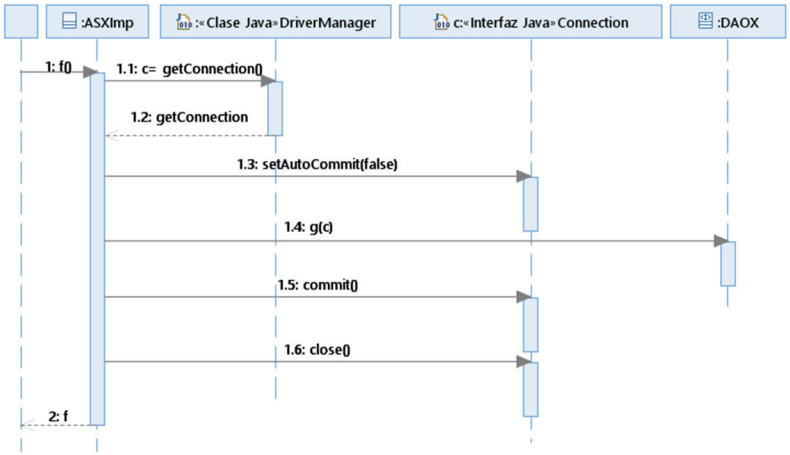


FIGURE 4. Transaction demarcation using object instances of classes depicted in Fig 3. Note the necessity for the application service to pass the connection to the DAO.

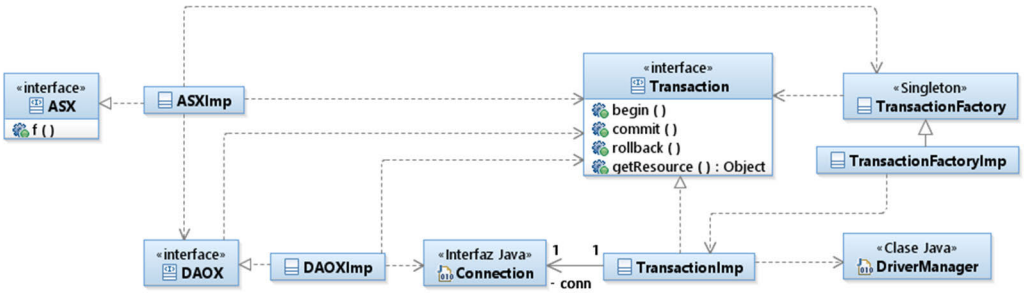


FIGURE 5. Basic local service transaction.

with the integration tier that students should be aware of. Therefore, this section provides incremental examples for justifying the introduction of a transaction manager that can deal with service transactions in a multithreaded environment. To do this, four solutions are provided to the students, discussing the limitations of each solution. After that, in the courses taught by the author, JPA and its transactional management is presented to the students.

The first example is an easy way to make students understand how application services can manage transactions: no wrappers are used, and application services manage transactions using JDBC `Connections` which are passed to DAOs. In this way, the *transaction demarcation* [4], [22] is done by application services using the interface `Connection`. This is easily understandable for students, but it couples the business tier to the integration tier, what is totally unacceptable. In addition, application services are forced to pass the connection to the DAOs. Figures 3 and 4 show the main concepts of this first undesirable solution.

The second example decouples application services from connections introducing a simple interface for hiding the JDBC `Connection`. This interface is the basis for service transactions and is called `Transaction`. Its implementation holds the connection to the single RDBMS that is used. In this way, `Transaction/TransactionImp` acts as an *adapter* [52] which wraps the JDBC `Connection` in a JDBC-agnostic coating. Thus, `TransactionImp` forwards the received commit/rollback to the underlying `Connection`. I know that this solution is very different from those found in the implementation of JTA, but at this point of learning, only the presentation of service transaction is intended. Fig. 5 and Fig. 6 shows this solution. Note that, in this solution, the application service is decoupled from the JDBC `Connection`, but it is still forced to explicitly pass the transaction as a parameter to the DAOs. The DAOs use the operation `Transaction::getResource()` to obtain the `Connection` (which is returned as `Object` for hiding it from the application services). To fully decouple application services from the transaction's implementation, an *abstract factory* is used [52].

The third example goes one step further, introducing a mechanism that prevents application services to pass transactions to DAOs: a very simple transaction manager that only manages one transaction with a single connection to a RDBMS and is intended for use in single-threaded environments. This transaction manager is implemented as a *singleton* [52]. When DAOs need this transaction, request it from the singleton transaction manager. Figures 7, 8 and 9 illustrate this extremely basic transaction

manager. `Transaction/TransactionImp` plays the same role as in Fig. 8, but now the commit/rollback also has to delete the transaction from the transaction manager.

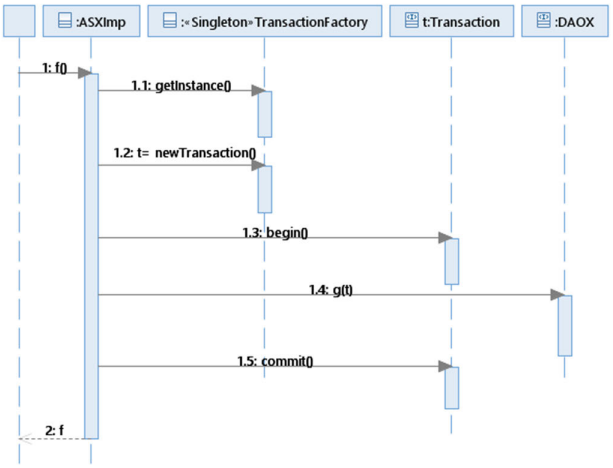


FIGURE 6. Transaction demarcation using object instances of classes depicted in Fig. 5. Note the necessity for the application service to pass the transaction to the DAO.

The last step enhances the transaction manager to be used in multithreaded applications, holding one transaction per thread of execution [1], [2]. No remote services are allowed and only one RDBMS can be used in the same transaction. As in the previous case, this transaction is mainly a simple wrapper for a JDBC `Connection`. Therefore, this transaction is only capable to deal with one local resource manager but allows a multitier application to include service transaction in a multithreaded environment as in the Virtual Campus of the Universidad Complutense de Madrid [49]. The binding of threads and transactions is made using a `ConcurrentHashMap` because it helps students to visualize the assignation of transactions to threads. However, `ThreadLocal` could be used instead [53]. Figures 10 and 11 illustrate this simple transaction manager. No sequence diagram is provided for the management of transactions from the application service because it is the same as that depicted in Fig. 8.

I am aware that the transaction manager depicted in Fig. 10 is not similar to JTA transaction managers. This solution is valid for multithreaded applications with only one RDBMS in a transaction and no remote services. However, the idea is to keep the design as simple and possible, and to some extent, similar to the use that application services make of JPA local transaction management (my students will see JPA later in the course). At this point, students are able to include service transactional management in the basic multitier architecture depicted in Fig. 1 according to the solution depicted in figures 10 and 11. Note that this solution can be used with both optimistic and pessimistic strategies for the management of the concurrency, because this strategy is implemented in the DAOs. However, according to the author's experience, a pessimistic strategy is easier to implement, including the sentence `SELECT ...FOR UPDATE` in the read methods of the DAOs and forcing application services needed to block any row in the database, to read it before. In any case, DAOs never commit or rollback transactions, as application services are responsible for the implementation of the business rules (i.e., application services are responsible for transaction demarcation).

This strategy does not permit *transaction composability* [22]. Therefore, the methods of application services cannot invoke the methods of application services. A very basic transaction composability can be achieved by including a Boolean attribute in the transaction which indicates whether or not any composed transaction has made a rollback, as well as an integer attribute that takes into account the number of begins performed. Each time a transaction is started in the same thread, as a result of a nested invocation from one application service to another, the counter increments. Whenever a commit/rollback is invoked, the counter decrements. If a rollback is invoked, the Boolean attribute is set to false. When the counter is at zero, the transaction commits if the Boolean attribute is true and a commit is invoked, and it is otherwise with rollbacks. To some extent, this is equivalent to a *required* behaviour [23] in application service methods, and consistent with the solution for transaction composability provided by [22]. It is not declaratively configurable, but it is simple enough to be understood by students with no previous knowledge of service transactions.

It is important to note that the solution provided in figures 3 and 4 covers the CS2013 learning outcome *create a transaction embedding SQL into an application program*, but the solution provokes a high coupling between the business and integration tiers, which it ruins the modularity, and therefore, the maintainability of applications. Thus, the service transaction introduced in the solution provided in figures 10 and 11 is necessary for transaction management from the business tier without coupling it with the resource tier.

Finally, in the author's courses, JPA is presented to students once they have implemented and used the transaction mechanism depicted in figures 10 and 11. In this way, they can value the work performed by object-relational mapping frameworks such as JPA at both levels, persistence, and transaction management. Special attention is paid to concurrency and locking in JPA, especially with optimistic locking [24].

V. CORE-TIER2. SERVICE TRANSACTIONS FOR SEVERAL RMs AND NO REMOTE SERVICES

Section IV provides students with a simple transaction manager able to deal with service transactions in a multithreaded environment with only one RDBMS per transaction and no remote services. However, these transactions are simple adapters of single database connections that are unable to deal with the inclusion of more than one resource manager involved in the transaction. This section enhances the simple transaction manager presented in figures 10 and 11 in order to make it work with several resource managers. As in the former case, no remote services are considered.

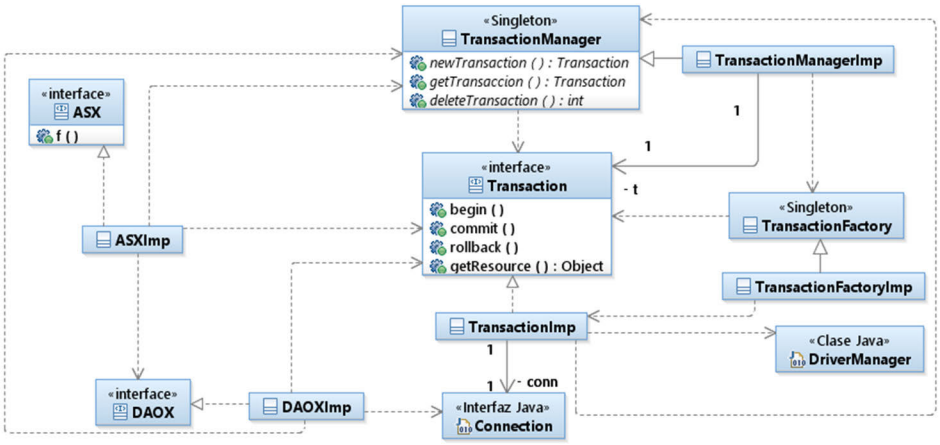


FIGURE 7. Basic local service transaction and transaction manager for single-threaded applications and single RDBMS.

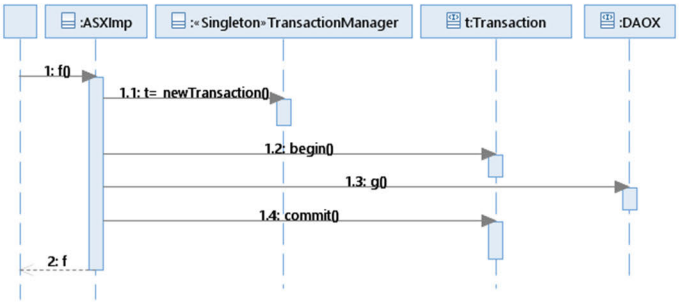


FIGURE 8. Transaction demarcation using object instances of classes depicted in Fig. 7. The transaction manager avoids the need of the application service to pass the transaction to the DAO.

The section provides important references for teaching the main concepts related to core-tier2 contents, but before considering these concepts, I think that reviewing the different classifications that are made for transactions in literature can be very interesting for both students and readers of this paper. Only the three first classification items are presented in the author's courses as well as service transactions. Finally, the section describes a simple transaction manager proposed by the author for dealing with several resource managers with no remote services.

A. CLASSIFICATION

Different authors consider transactions from different point of views. This section revises the classification of transactions made in the literature.

1) BUSINESS VS. SYSTEM TRANSACTIONS

According to [1], *business* transactions are those related to the use case, while *system* transactions are those performed by the RM and transaction monitors.

For example, let us suppose that a user wants to buy an item with the following process: the user can select it, add it to the shopping basket and finally pay for it. The user conceives the three operations associated to purchases (selection, addition to the basket and payment) as a single business transaction, but, usually, the application has performed three systems transactions against the RM.

2) REQUEST VS LONG TRANSACTIONS

According to [1] *request* transactions are those that are started and finished within one request, while *long* transactions span multiple requests. For example, the purchasing process described above is a long transaction. User registration in an application is a request transaction because the user provides the data, and they are processed in the request sent to the server with no need of further requests (if no problems found).

How do business/system and request/long transactions coexist?

- Business-request transaction: implemented by a single system transaction.
- Business-long transaction: very difficult to implement in practice. Simulated by system transactions as much as by requests.
- System-request transaction: implemented by a single system transaction.
- System-long transaction: the application is forced to hold the system transaction through several requests.

3) LOCAL VS GLOBAL TRANSACTIONS

Local transactions are those created and committed against a single RM [4] while *global* transactions involve many RMs in

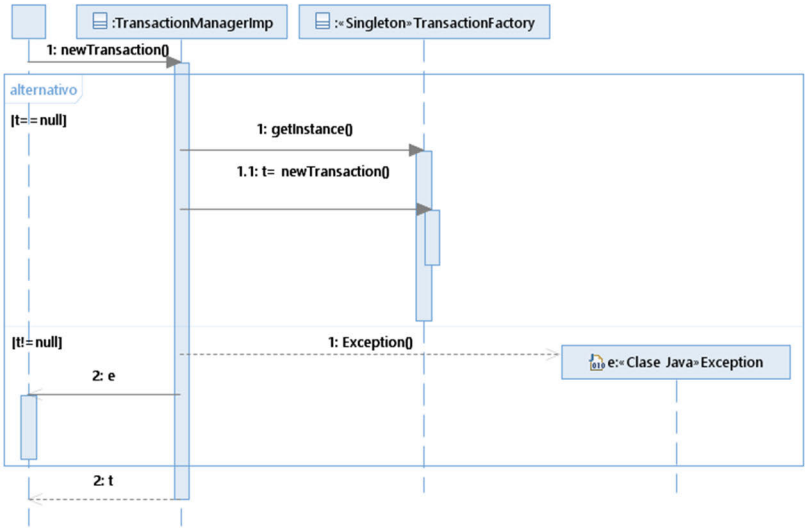


FIGURE 9. This basic transaction manager only holds one transaction.

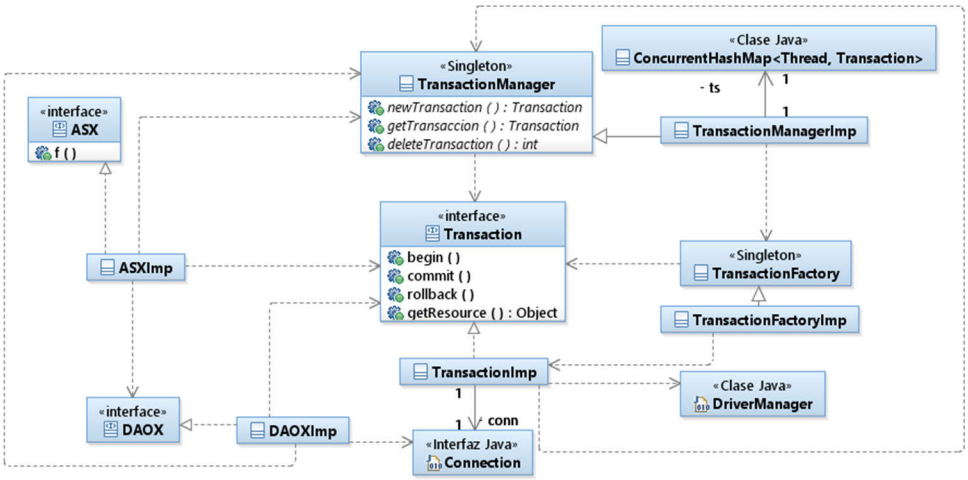


FIGURE 10. Basic local service transaction manager for multithreaded applications and no remote services.

a single unit of work [20]. The concept of *non-global* transaction can be assimilated to the concept of local transaction: a transaction in which, if two or more RMs are involved, there is no coordinated commitment between them [44].

In practice these RMs can be invoked from the same AP or from different APs. How do single/several RMs invoked from single/several APs coexist in the same unit of work?

- One RM invoked from one AP involves a local transaction.
- One RM invoked from several APs involves a global transaction if these APs are remote, and usually, one local transaction if these APs are not remote.
- Several RMs always involve a global transaction.

4) UNCOMPOSED VS. COMPOSED TRANSACTIONS

In this paper I define *transaction composability* as the ability of different transactions to become part of the same transaction with a behaviour consistent with the business logic implemented. This is an adaptation of the term *composability problem* defined in [22].

According to this, I define *uncomposed* transactions as those in which there are no two consecutive begins without an intermediate commit/rollback and *composed* transactions such as those where two begins can be produced without an intermediate commit/rollback.

In the *session façade* pattern [2], *coarse-grained transaction control* and *fine-grained control are considered*. In the coarse-grained control, business services do not demarcate transactions and they are contained within one single service that starts the transaction, calls these business services and closes the transaction. In the fine-grained control, each business service demarcates its own transaction, and there is no enclosing service. Thus, uncomposed/composed transactions are, to some extent, similar to those of coarse-grained/fine-grained transaction control.

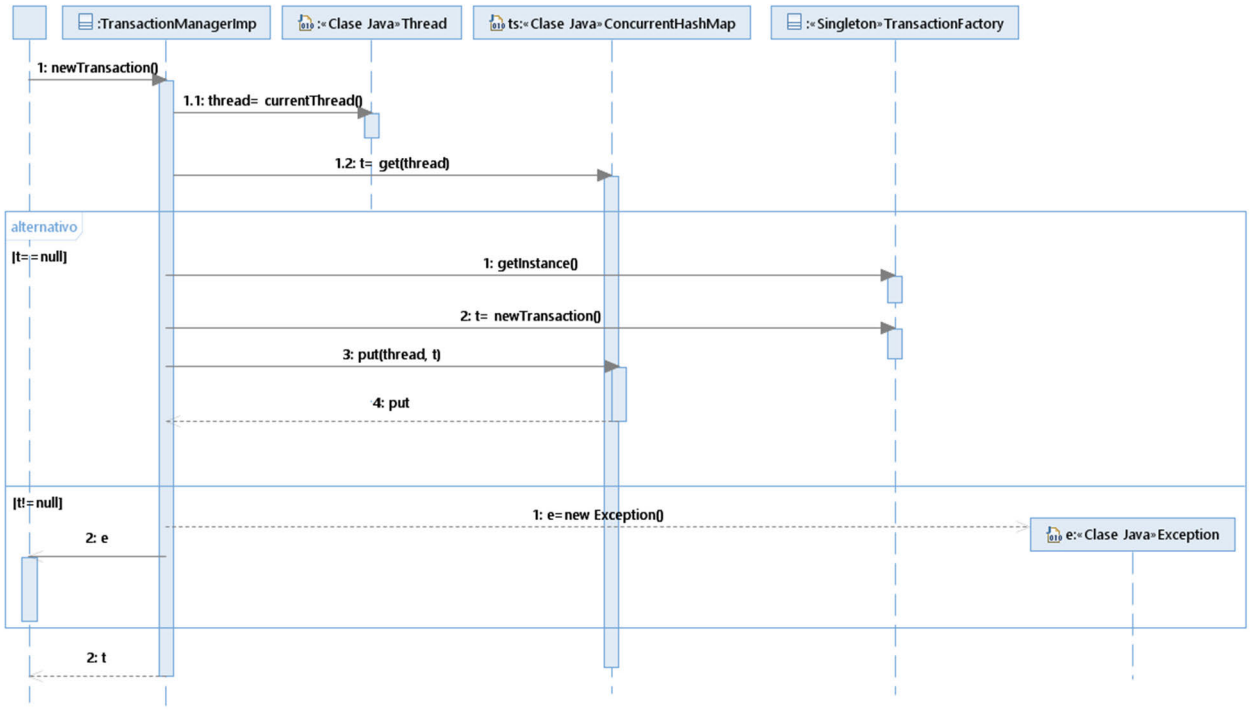


FIGURE 11. This basic transaction manager is able to hold one transaction per thread.

Uncomposed transactions avoid the problem of transaction composability. However, in object-oriented programming, it is common to have collaborative transactions that are managed with a *transaction attribute* that indicates the transactional behaviour of a method [22]. In JEE this behaviour can be: `NotSupported`, `Supports`, `Required`, `RequiresNew`, `Mandatory` and `Never` [23]. A method is *top-level* if it causes a new transaction to be started, i.e., the top-level method is tagged with `RequiresNew` or `Required`, and its caller is not executing in a transaction [22]. Generally speaking, a transaction commits when its top-level method terminates without an error. If it throws an exception during its execution, then its transaction aborts. To avoid the checking of each operation involved in a transaction, before an exception is thrown, an involved method can mark the transaction context as `setRollbackOnly`, which provokes the rollback of the top-level and all the methods involved in the transaction [21].

5) FLAT VS. NESTED TRANSACTIONS

When a composed transaction is made, it is possible to have flat or nested transactions. *Flat* transactions are those run independently, and *nested* transactions are those in which a tree hierarchy is made [22].

Although nested transactions are appealing from an application programming perspective, they are not supported in many commercial products [22]. Thus, in *Jakarta Transactions*, support for nested transactions is not required [54].

For example. Let us suppose that in JEE there is an outer method that starts a transaction (T1) and calls for an inner method which starts another transaction (T2). Let us suppose that the inner method that started T2 did so because it is marked as `RequiresNew`. What is the difference between the JEE flat/`RequiresNew` behaviour and the nested behaviour?

- The behaviour is quite straightforward if both, T1 and T2, commit or rollback.
- If T2 rollbacks, the behaviour of T1 is not conditioned and can do whatever it wants in both cases, flat or nested.
- The difference arises when T2 commits and T1 rollbacks [55]:
  - In the flat/`RequiresNew` scenario, T2 stays committed, because it is independent of T1.
  - In the nested scenario, T2 should rollback, because it is nested to T1.

OTS provides more information about nested transactions.

6) CHAINED VS. UNCHAINED

If every operation is always executed within a transaction, which only has to be committed, thus provoking the start of another transaction, we say that the transactions are *chained*. On the contrary, if operations can be executed without transactions, which have to be started and committed, we say that the transactions are *unchained* [22].

The transaction attribute used in object-oriented programming makes use of chained transactions.

7) CHECKED VS. UNCHECKED

In commercial applications it is normal to have several threads involved within a transaction. Checking is about thread and transaction synchronization.

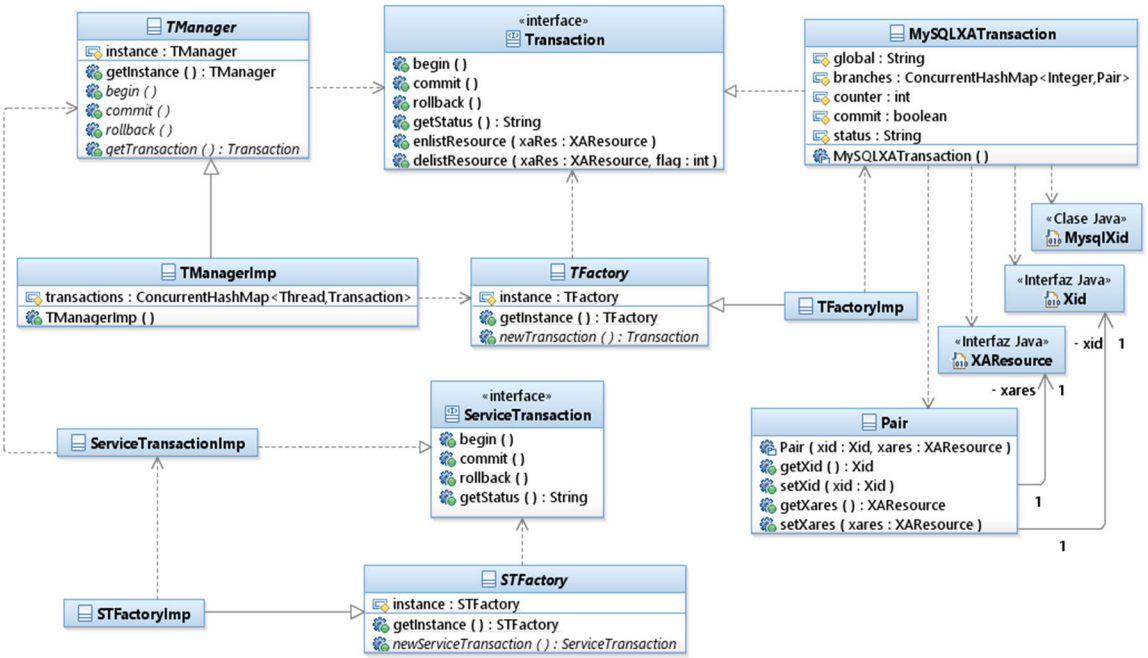


FIGURE 12. Basic global transaction manager and transactions with no remote services.

In *checked transactions,* it is guaranteed that whenever a transaction ends, there can be no thread active within the transaction which has not completed its processing. *Unchecked transactions* do not enforce this behaviour [4].

Applications that do not create new threads and only use synchronous invocations within transactions implicitly exhibit checked behaviour. However, when asynchronous invocation is allowed, explicit synchronization is required between threads and transactions in order to guarantee checked behaviour [4].

8) ACID VS EXTENDED TRANSACTION

ACID transactions are those started and finished in a short amount of time. How long is ''short''? The time that allows resources to remain engaged in a transaction for its duration. For example, to buy a trip that includes acquiring airplane tickets and making a hotel reservation, selecting both choices and buying both at the same time is a good example of ACID transaction. Thus, ACID transactions are those that can be synchronously resolved.

However, there are other transactions that cannot be resolved in such a short amount of time. For example, in the former example, if the customer can make the hotel reservation and then the application waits for two days looking for a good offer in flight tickets, cancelling the hotel reservation if no flight for a certain amount of money appears, this cannot be managed as an ACID transaction. Here is where *extended transaction* models such as those for dealing with *long-lived transaction* appear [4].

For the sake of conciseness, this paper only focuses on ACID transactions.

9) TRANSACTION MODELS

According to [21] there are three transaction models: local, programmatic, and declarative. These models do not explicitly define transactions, but the way transactions are used.

In the *local model* the programmer manages connections instead of transactions. Because services create connections, DAOs must access the same connection in order to participate in the same transaction. This causes two problems: c*onnection passing* from the service to the DAOs and from the coupling business tier to the resource tier. In addition, only one RDBMS is normally used. Note that the transaction management depicted in figures 3 and 4 is based on the local model with connection passing and coupling. The transactional management depicted in figures 5 and 6 solves the coupling by introducing the interface `Transaction`. The transaction management depicted in figures 7, 8 and 9 solves the connection passing by using the singleton `TransactionManager`. The term *connection-based* would be more precise because local transactions can be performed without using an explicit connection as JPA does.

In the *programmatic model,* transactions are managed by a framework that hides RM connections. Thus, the programmer manages transactions rather than connections. In order to implement this model, JTA can be used. In this case, the JTA `UserTransaction` interface is used to manage transactions. The main problem associated to this model in JEE is the *transaction context problem*: transaction contexts cannot be passed between different services implemented as stateless session beans. Reference [23] calls this model the *explicit* or *bean-managed* model. The transaction management depicted

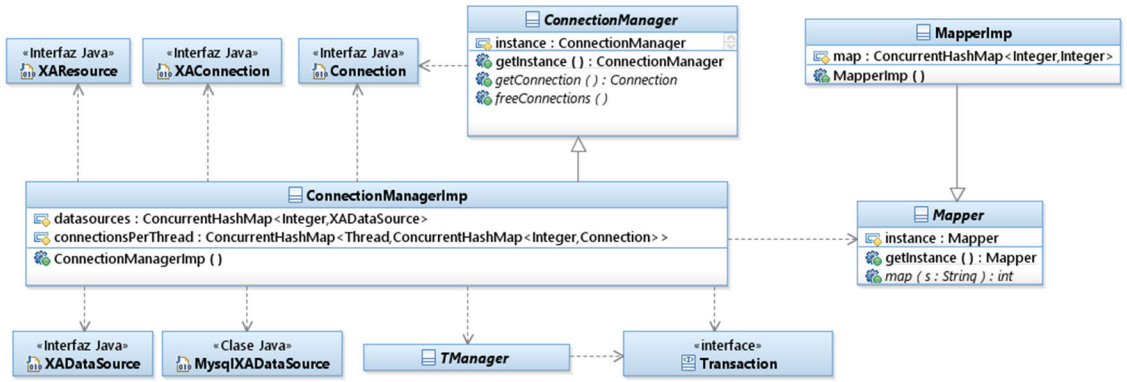


FIGURE 13. **Singleton** `ConnectionManager` + `ConnectionManagerImp` **for the management of JDBC connections and auxiliary singleton** `Mapper` + `MapperImp`.

in figures 10 and 11 conforms to the programmatic model, without hiding the transaction manager behind JTA user transactions (or any similar interface).

In the *declarative model,* transactions are managed by containers, i.e., there are *Container Managed Transactions* (CMT). Thus, the container (or underlying framework) manages the start and end of the transaction, including different services in the same or different application servers. Transaction attributes (i.e., `Required`, `Mandatory`, etc.) tell the container how it should manage the transactions. Reference [23] calls this model the *implicit* model. This model has no significant problems beyond the complexity of implementing it in transactional frameworks and application servers.

The local and programmatic models use unchained transactions, while the declarative model uses chained transactions configured by the transaction attribute.

10) RM TRANSACTIONS VS SERVICE TRANSACTIONS

In the literature about global transactions there is an implicit assumption: although transactions have to be implemented by transactional resource managers, in practice, these transactions have to be wrapped in some way to be used by the business tier irrespectively of the underlying resource managers.

This paper explicitly distinguishes between *RM transactions (or simply transactions)*, which are those performed by transactional RMs, and *service transactions*, which are those demarcated by application services omitting connections to RMs. Note that it is possible to have applications that perform RM transactions without service transactions (i.e., the local transaction model [21]), while it is not possible to have service transactions without underlying RM transactions that implement them (at least one RM transaction for each service transaction). Service transactions can be programmatically or declaratively demarcated. In this paper, only programmatic service transactions are considered and no declarative demarcation is considered because it is more pedagogical to first learn to use programmatic demarcation and then learn to use declarative demarcation.

Business transactions are not the same as service transactions. Thus, the business transaction related to performing a purchase (selection, addition to the basket and payment) involves three service transactions (which indeed involve a minimum of three RM transactions). Do system transactions correspond to RM transactions? It is unclear, because system transactions are those performed by RMs and transaction monitors. Those performed by RMs are indeed system transactions. Maybe those performed by transaction monitors could be matched to service transactions. In this case, both RM and service transaction could be considered to be system transactions.

Reference [56] also defines the term *service transaction*, but with a more generic semantics in the context of the definition of a basic system metamodel for mobile enterprise architectures: complete set of operations between two resources, comprising sending a stimulus and the reaction to it by the receiving resource. Reference [57] defines a pattern named *atomic service transaction* which deals with the propagation of transactions with rollback capabilities across messaging-based services, which have a more specific semantics. Reference [58] considers a sample tagged with the title *Service Transaction Behaviour*. This is the example that is most related to the term defined in this paper, but [58] does not define it as something relevant. It is only the title of an article.

B. XA TRANSACTIONS

The understanding of global service transactions is not evident for undergraduate students that have had their first contact with service transactions. The understanding and implementation of a naïve transaction manager such as that depicted in figures 10 and 11 requires students to make a significant effort. In this context, presenting global transactions is a challenge (not yet assumed by the present author in his classes). However, beginning the teaching of global transactions with standards such as *OSI Distributed Transaction Processing* [43], [44], *X/Open XA Specification* [20] or *OMG Transaction Service Specification* [59] can be very hard for the students, although the main concepts of X/Open XA should be presented to them.

These are relevant standards for advanced developers or for those that build application servers or frameworks for

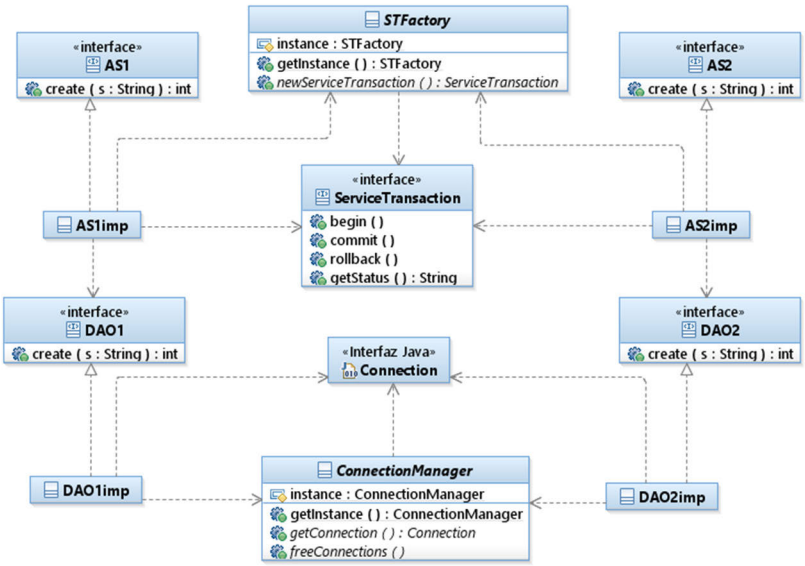


FIGURE 14. Use of the proposed framework by application services and DAOs.

enterprise application development. However, these standards are very technical for undergraduate students. Indeed, the mere understanding of all the different classifications of transactions made in Section V.A can be too much for the students.

Transactions can be global because: (i) at least one AP needs to deal with data in two (or more) RMs within the same transaction; or (ii) because two (or more) remote APs need to deal with data (in the same RM or not) within the same transaction. It is not unusual, at this level, for students to have no idea about distributed programming. Therefore, the concept of an AP remotely invoking another AP is not evident for students. In this context, introducing a global transaction within remote invocations is something that raises the level too much for most students. Therefore, introducing several RMs in a transaction managed by a single AP (or several local APs) is something more feasible.

The classical concepts bound to the two-phase protocol defined in Section II.H (and not previously presented) should be presented to the students. The definitions of Section II.H provide references that may be the basic bibliography for this issue. In addition, the concepts of failure handling and process structuring (included the tree-of-processes model of a transaction execution) [22] should be presented to the students. It is important to remark to them that:

- APs and RMs interact through *native interfaces* (such as JDBC).
- APs and TMs interact through the *TX interface*.
- TMs and RMs interact through the *XA interface*.

However, these concepts cannot be fully understood if they are not used in implementations programmed by the students. The need for the execution of a code leads us to introduce specific programming frameworks such as *Jakarta Transactions* (JTA), which is used for managing global transactions in the Java application. The JTA concepts that must be presented in order to exemplify the use of XA transactions are, at the very least, the interfaces `javax.transaction.xa.Xid` and `javax.transaction.xa.XAResource` as well as the JDBC interfaces `javax.sql.XAConnection` and `javax.sql.XADataSource`. Reference [60] provides a simple example of a use of Java `DataSources` for MySQL RDBMS. `DataSources` are necessary for using XA transactions in Java. Reference [61] provides a good example of the use of XA transactions with MySQL as underlying RM (although the `globalId` of `xid1` and `xid2` should be the same). The following example is an adaptation of [61]:

```
public class Main {
public static void main(String[] args) {
//Creation of MysqlXADataSource from configuration
//files
Properties props= new Properties();
FileInputStream fis = null;
MysqlXADataSource mysqlXAds1= null;
MysqlXADataSource mysqlXAds2= null;
try {
fis = new FileInputStream("conf/db1.properties");
props.load(fis);
mysqlXAds1= new MysqlXADataSource();
mysqlXAds1.setURL(props.getProperty("url"));
mysqlXAds1.setUser(props.getProperty("user"));
mysqlXAds1.setPassword(props.getProperty("password
"));
fis = new FileInputStream("conf/db2.properties");
props.load(fis);
mysqlXAds2= new MysqlXADataSource();
mysqlXAds2.setURL(props.getProperty("url"));
mysqlXAds2.setUser(props.getProperty("user"));
mysqlXAds2.setPassword(props.getProperty("
```

```
  password
  "));
  } catch (IOException e) {e.printStackTrace(); }
  Statement stmt1 = null;
  Statement stmt2= null;
  Connection conn1= null;
  Connection conn2= null;
  //casting MysqlXADataSource to XADataSource
  XADataSource ds1= (XADataSource) mysqlXAds1;
  XADataSource ds2= (XADataSource) mysqlXAds2;
  try {
  //request transactional resource
  XAConnection xaCon1 = ds1.getXAConnection();
  XAConnection xaCon2 = ds2.getXAConnection();
  //get XAResource
  XAResource xares1= xaCon1.getXAResource();
  XAResource xares2= xaCon2.getXAResource();
  //XA.Resource.start()
  String global= Thread.currentThreadtoString();
  Integer branch = 0;
  byte[] globalId = global.getBytes();
  byte[] branchId = new byte [1];
  branch ++;
  branchId[0]= branch.byteValue();
  Xid xid1= new MysqlXid(globalId, branchId, 100);
  branch ++;
  branchId[0]= branch.byteValue();
  Xid xid2= new MysqlXid(globalId, branchId, 100);
  xares1.start(xid1, XAResource.TMNOFLAGS);
  xares2.start(xid2, XAResource.TMNOFLAGS);
  //get connection
  conn1= xaCon1.getConnection();
  conn2= xaCon2.getConnection();
  conn1.setAutoCommit(false);
  conn2.setAutoCommit(false);
  //some work
  stmt1= conn1.createStatement();
  stmt1.executeUpdate("INSERT INTO table1 (value)
   VALUES ('Hello');");
  stmt2= conn2.createStatement();
  stmt2.executeUpdate("INSERT INTO table2 (value)
  VALUES ('world');");
  stmt1.close();
  stmt2.close();
  //dissociating transaction from RM
  xares1.end(xid1, XAResource.TMSUCCESS);
  xares2.end(xid2, XAResource.TMSUCCESS);
  //XAResource.prepare()
  int res1 = xares1.prepare(xid1);
  int res2 = xares2.prepare(xid2);
  //XAResource.commit
  if (XAResource.XA_OK == res1 && XAResource.XA_OK ==
res2)
  {
    xares1.commit(xid1, false);
    xares2.commit(xid2, false); }
  else
     { xares1.rollback(xid1);
    xares2.rollback(xid2); }
  } catch (Exception e)
   {
    e.printStackTrace();
    System.out.println("error");
  }
  finally
  {
  try {
  if(stmt1 != null) stmt1.close();
  if(stmt2 != null) stmt2.close();
  if(conn1 != null) conn1.close();
  if(conn2 != null) conn2.close();
  catch (SQLException e) {e.printStackTrace();
                                  }
          }
        }
    }
```

Mentioning JTA is necessary, but mentioning the *Java Transaction Service* (JTS), which is the Java implementation of the OMG TS, is optional. There are four important concepts that arise from JTA that should be presented to the students [4]:

- The functionality of XA specification is mapped to JTA through the `javax.transaction.xa.ID` and the `javax.transaction.xa.XAResource` interfaces.
- The functionality of the Open Group TX specification is mapped to JTA through the `jakarta.transaction.UserTransaction`, `jakarta.transaction.TransactionManager`, and `jakarta.transaction.Transaction` interfaces. `UserTransaction` provides applications with the capability to control transaction boundaries programmatically. An application server uses the `TransactionManager` interface to demarcate transaction boundaries on behalf of an application. Client applications in a non-managed, non-JEE environment may choose to manage transactions via the `TransactionManager` interface or the `UserTransaction` interface. The `UserTransaction` interface allows applications to invoke operations on the transactions associated with the target object.
- For each thread in a process, the system maintains a mapping to the current transaction associated with that thread ([4], pg. 51).
- The philosophy of use of JTA, i.e., the general procedure an application server follows in a transaction management scenario such as [4] describes.

A detailed description of JTA is not necessary, because the use of `UserTransaction`, `Transaction` and `TransactionManager` is hidden by the declarative management of transactions made by JEE. Only in specific courses about JTA should these interfaces be described. Ref-

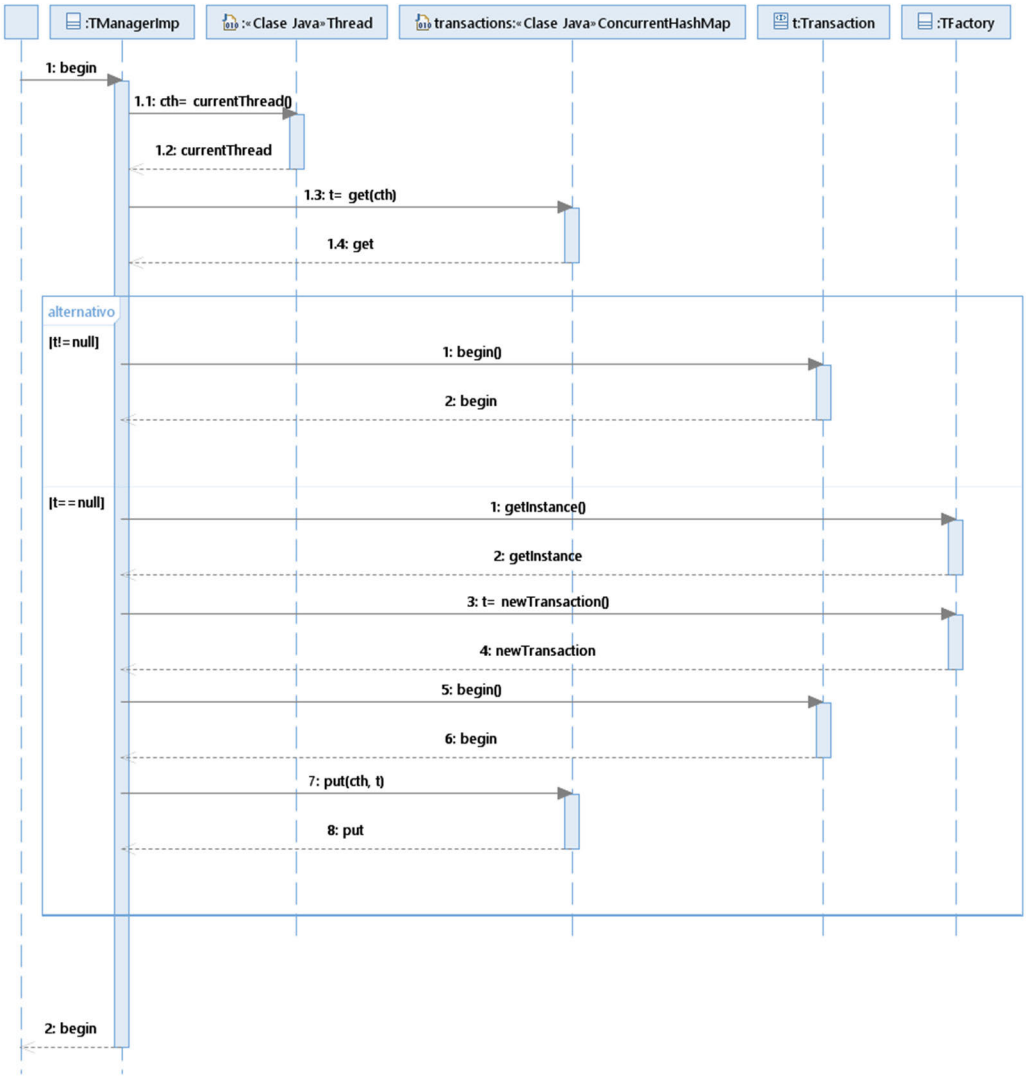


FIGURE 15. Starting a transaction in the proposed transaction manager of Fig. 12.

erence [4] provides a good description of when to use each interface. There are transactions engines such as *ArjunaCore* included in the *Narayana* transaction suite [62], which can be used as a standalone transaction engine. However, the implementations of these engines are very complex because they have to deal with all the capabilities of JTA. Therefore, it is better to introduce students to simpler transaction managers such as those defined in this paper.

C. A TRANSACTION MANAGER FOR GLOBAL SERVICE TRANSACTIONS WITH NO REMOTE SERVICES

If I had to implement a commercial Java application that manages persistent data distributed in several RMs, I would choose JPA as a persistence mechanism and some implementation of JTA to manage XA transactions. However, as in the case of the previous section, this type of implementation hides significant details related with the integration tier that students should be aware of. This section describes a simple transaction manager for global service transactions with no remote services.

The core elements of this implementation are described in figures 12 and 13. Fig. 14 describes the use of these classes by applications services and DAOs.

Fig. 12 describes the new transaction manager for global transactions with no remote services. In this solution, simple transaction composability is enabled. The basic role that it plays (to keep a transaction bound to the thread that created it) remains unchanged with regards to the transaction manager of Fig. 10. Thus, only one transaction can be active per execution thread. The transaction manager is responsible for transaction demarcation (on behalf of application services) now. Fig. 15 describes the starting of a transaction using this transaction manager. The transaction manager's commit and rollback operations perform these actions on the transaction bound to the calling thread. `getTransaction()` returns the transaction bound to the calling thread.

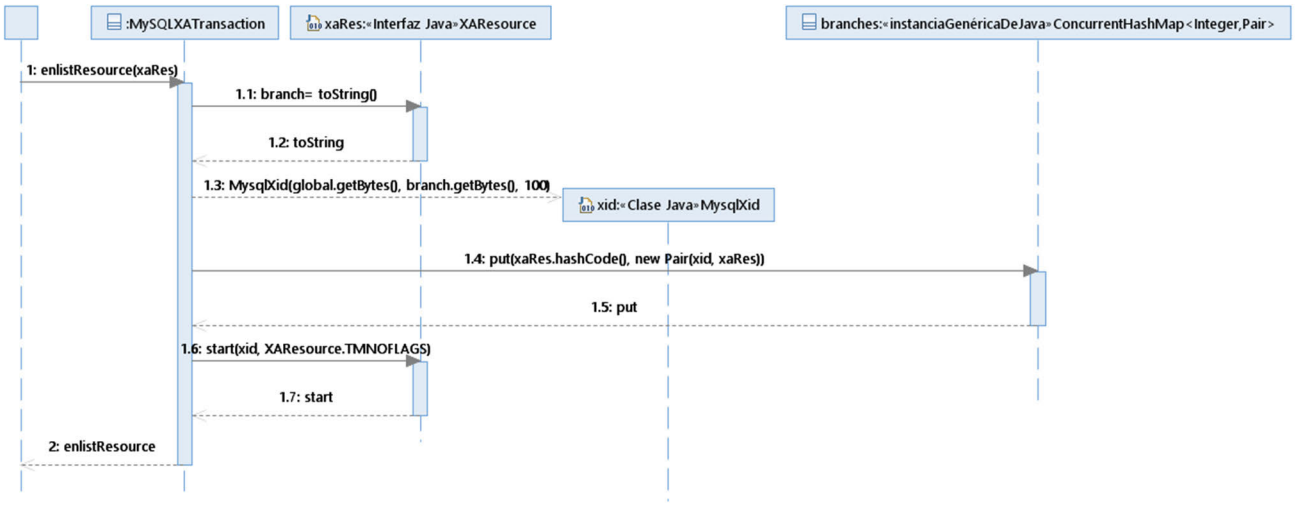


FIGURE 16. `Transaction` **enlistment in the class that implements the** `Transaction` **interface of Fig. 12.**

The `Transaction` interface described in Fig. 12 is, to some extent, equivalent to the `jakarta.transaction.Transaction` JTA interface. The `MySQLXATransaction` class that implements it takes care of the global ID (`global`), the branches (`branches`), the number of begins and commit/rollback actions performed (`counter`), whether no one has made rollback (`commit`) and the status (`status`).

The branches are `Pairs` of (`Xid, XAResource`) objects indexed by the `XAResource` hash code. This information is needed because in the two-phase commit, `XAResources` need the branch id (`Xid`) in order to interact with the RMs. Fig. 16 shows the enlistment of a resource in a transaction and Fig. 17 shows the commit of the transaction according to the two-phase commit protocol. The `ServiceTransaction` interface described in Fig. 12 is, to some extent, equivalent to the `jakarta.transaction.UserTransaction` JTA interface, and it is a simple adapter that hides the existence of the transaction manager from application services. Thus, for example, the code of `ServiceTransactionImp::begin()` is simply `TManager.getInstance().begin()`.

Managing connections (i.e., JDBC `Connections`) in a multithreaded environment is a well-known issue in concurrency management [53]. Reference [21] defines *connection passing* as the technique that establishes a database connection at the higher-level method and passes the connection into the DAOs. Reference [63] (pgs. 272-273) has an interesting discussion about resource handling in JEE containers and in Hibernate. Reference [4] (pg. 182) discusses the management of connections in the context of global transactions. This issue is very important in the frameworks that implement JTA [29], [30], [31]. For example, the configuration file of Bitronix for `DataSources`, has the parameter `shareTransactionConnections`, which enables a thread-associated cache, i.e., if the same thread requests several connections from a `DataSource`, the same connection is always retrieved [30].

With these ideas in mind, the management of connections made in Fig. 13 is fairly simple. If n RDBMS are going to be used, n configuration files have to be provided for the configuration of each `XADataSource`. Therefore, in our approach, each RDBMS is bound to a specific `XADataSource`. Each RDBMS/`XADataSource` has a unique ID, provided by the developer. Thus, if there are three RDMBSs there will be three IDs: 1, 2 and 3 (the order is of no importance). An additional configuration file that binds DAO classes with RDMBS/`XADataSource` IDs is provided. The singleton `ConnectionManager/ConnectionManagerImp` holds a `ConcurrentHashMap` that indexes each `XADataSource` by its ID (`dataSources`). These data sources are configured in the constructor of the class `ConnectionManagerImp`. This singleton also holds the JDBC `Connections` established in a transaction. Because in our solution there can only be an active transaction per thread, the `ConnectionManagerImp` indexes connections by execution thread (note that in Fig. 17 commit frees the connections bound to a transaction). These `Connections` have to be bound to their RDBMS/`XADatasource ID`. Therefore, bound to a thread, they are held in a `ConcurrentHashMap` that indexes them by this ID (`connectionsPerThread`). Thus, this solution only allows one active connection per RDBMS/`XADataSource` in a thread.

Each time a `Connection` is required by a DAO, a `Mapper` class maps the class of the DAO with the RDMBs/`XADatasource` ID, and the connection manager checks if there is a connection bound to this ID in the thread. If there are none, it gets the `XADatasource` bound to the ID and generates a `XAConnection`, which is used for the generation of an `XAResource` and a `Connection`. This mapping is driven by a configuration file that maps the

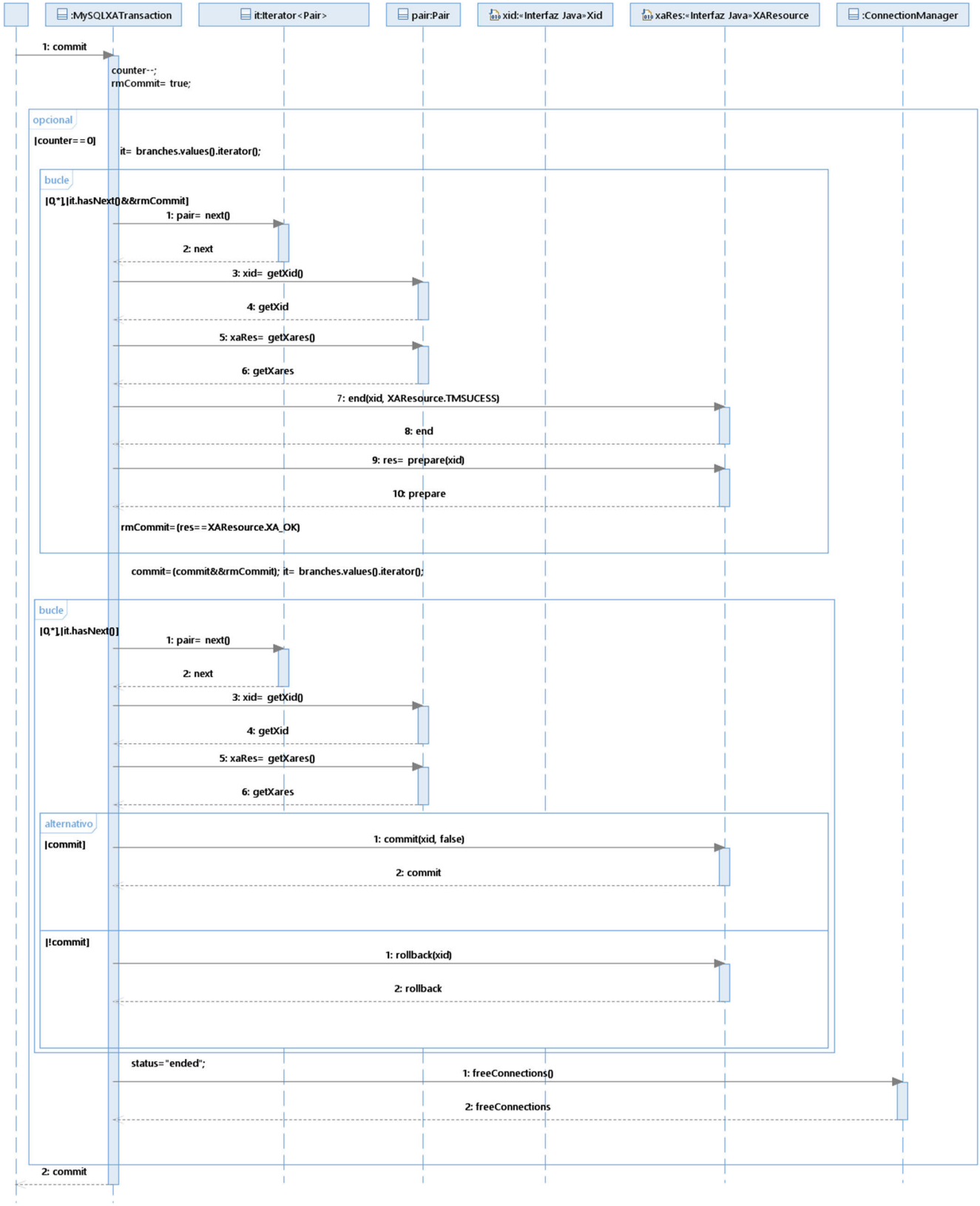
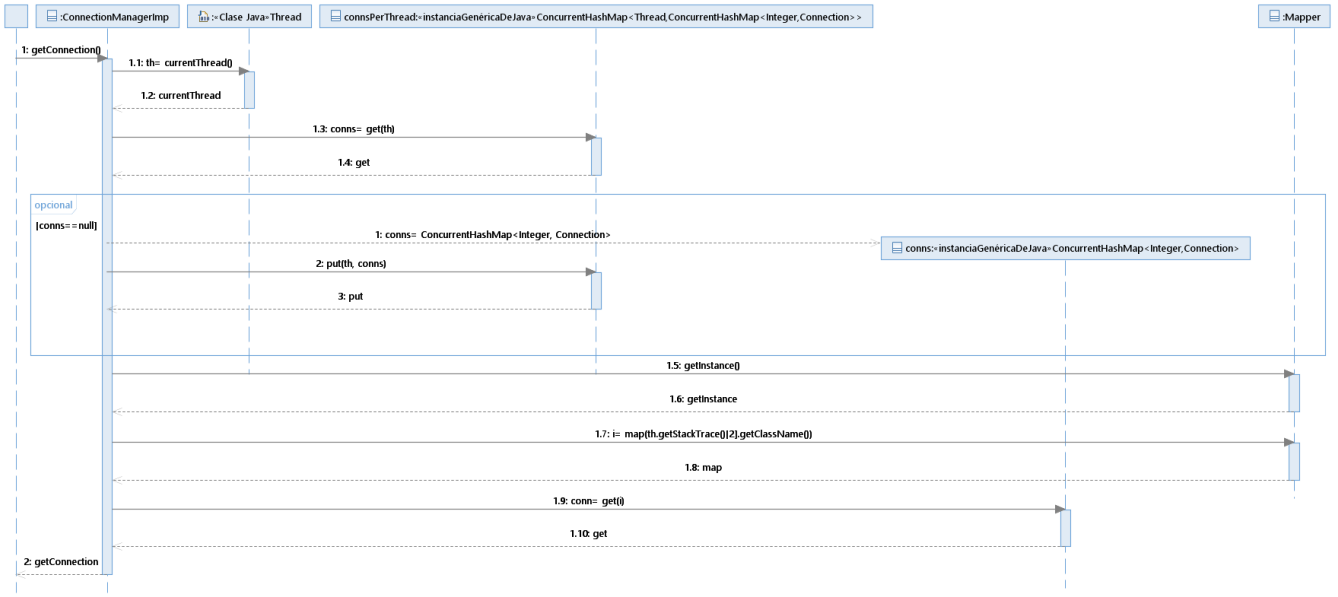
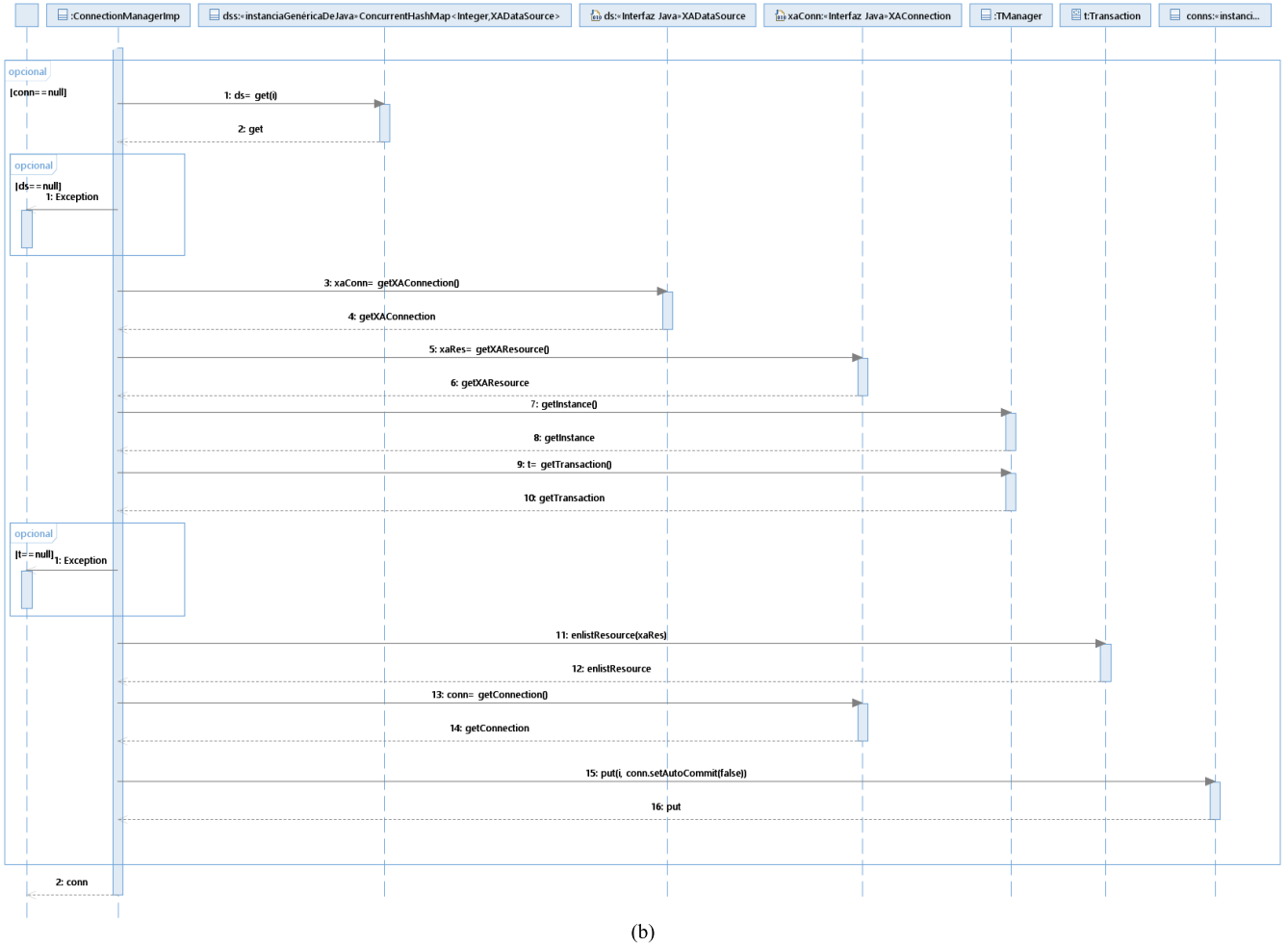


FIGURE 17. Transaction commit in the class that implements the `Transaction` interface of Fig 12. Note the two-phase commit protocol for all the `XAResources` enlisted in the transaction.



(a)



(b)

FIGURE 18. **a.** `Connection` **management in the** `ConnectionManagerImp` **class of Figure 13 (first part). b.** `Connection` **management in the** `ConnectionManagerImp` **class of Figure 13 (second part).**

DAO classes with the datasource's IDs. The `XAResource` is enlisted in the transaction bound to the thread and the `Connection` is stored in the connections bound to the thread (indexed by the RMDBs/`XADatasource ID`). Fig. 18 describes this process. Due to the size of the diagram, the figure has been divided into two parts (a) and (b), where interactions in part (a) precede those in part (b). In the transaction management solution depicted in Fig. 10, as there is only one RDBMS per transaction, and, therefore, no enlistment has to be made, the transactions simply wrap `Connections` and store them by thread in the transaction manager.

This framework does not permit the invocation of remote application services but is a fairly simple method to manage global service transactions in a multithreaded application.

VI. ELECTIVE. SERVICE TRANSACTIONS FOR SEVERAL RMs AND REMOTE SERVICES. PERSISTENCE APIs.

This section focuses on several advanced topics regarding the use of service transactions with remote services and persistence frameworks. I include web services instead of RPC-like remote invocations because web services are more widespread at present.

Service-Oriented Architecture (SOA) represents an architectural model that conceives services as the primary means through which solution logic is represented [57]. A *service* is defined as a unit of solution logic [57]. Services can be implemented as components, web services and REST services [57]. A *component* is a software program designed to be a part of a distributed system [57]. A *web service* is a body of solution logic that provides a physically decoupled technical contract consisting of a *Web Services Description Language* (WSDL) definition and one or more XML Schema definitions and also, possibly, WS-Policy expressions [57]. *REST services* (or *RESTful services*) are lightweight programs that are designed with an emphasis on simplicity, scalability, and usability [57]. *Representational State Transfer* (REST) provides a means of constructing distributed systems based on the notion of resources [64]. A *resource* is a piece of information that can be referred to using a URI [64]. Web services are also called *SOAP services* because they use a *Simple Object Access Protocol* (SOAP) to exchange messages [64]. The differences between SOAP and REST services consist in three elements [64]:

- Message format: SOAP services use XML inside a SOAP Envelope and REST services can use any format, such as XML.
- Interface definition: SOAP services use WSDL and REST services do not conform to a formally adopted interface definition language, although several attempts have been made in this respect.
- Transport: SOAP can use different protocols (HTTP, JMS, FTP, etc.) and REST services use HTTP.

Some consider SOAP and REST services to be different, but inasmuch as enterprise buses make automatic conversions among them [65], we do not distinguish them and simply call them web services. This is a mere simplification for making the writing of this paper easier and does not try to determine whether or not SOAP and REST services should be considered equivalent.

Regarding the use of persistence frameworks, no specific information about this is provided in this paper because there are a good number of references that can be used to present the basic concepts identified in the KU. Reference [24] is a good reference for JPA. A simple example of how to ensure the persistence of entities related by a 1 to N relationship (e.g., a department with several employees) using JPA local transactions could be presented to students. Attention must be paid to the concept of *persistence unit* and to the `persistence.xml` file.

A. SERVICE-ORIENTED ARCHITECTURE AND MICROSERVICES ARCHITECTURE

SOA can be considered a type of multitier architecture where applications services are exposed using the *web service broker* pattern [2], [65]. Indeed, multitier architecture is service-oriented regardless of how the application services are exposed: (i) directly to the *commands* [52] of a web tier controller which is in the same physical tier as the business tier; (ii) wrapped in *session façades* [2], which allow RPC access to them; (iii) wrapped in web service brokers, which allow access to them via SOAP or REST services [57]; or (iv) wrapped in *service activators* [2], which allow asynchronous message-based access to them.

In order for a transaction to span a distributed number of services, the transaction context has to flow between these services. The transaction context contains the transaction identifier, a coordinator location or endpoint address for participants to be registered, and implementation specific information [4]. In SOAP web services, standards such as *WS-Transactions* for SOAP [66] propagate transactional context from the caller to the callee [4]. WS-Transactions considers two types of transaction models: *atomic*, which mimics the two-phase commit for web services, and *long-running*, where participants in a transaction may not participate in a synchronous way. *WS-Atomic* [67] and *WS- Business Activity* [68] are OASIS standards for dealing with these types of transaction models in SOAP web services, respectively. Both standards are built on the generic standard for activity management in the web services *WS-Coordination* [69].

Regarding the inclusion of global transactions in REST web services, there is some controversy as to whether or not they should be considered [70]. However, [71], [72] are good references for this issue.

Microservices architecture [73] can be considered an evolution of SOA. Although microservice authors are prone to call any architecture different from microservices *monolithic* [8], [73] due to the microservices' deployment architecture, from the point of view of software architecture, the main difference between microservices and SOA is the fact that each microservice uses its own database. This characteristic makes microservices unsuitable for those

applications in which ACID transactions, especially consistency, are needed [8]. However, microservices are even promoted in the banking industry [74]. For example, let us suppose a microservice that deals with department deregistering (DD), where the business logic says that no department can be deregistered with active employees (let us suppose a one-to-many relationship between department and employees). Let us suppose a microservice that deals with employee registering (ER), and in which the business logic says that the employee must be assigned to an active department. Let us suppose this interaction:

1. DD reads department 7 and checks that it is active, and it is.
2. ER reads department 7 using an additional service and checks that it is active, and it is.
3. DD checks that department 7 has no active employees, and it does not.
4. ER inserts employee 25 assigned to department 7 and commits.
5. DD marks department 7 as inactive and commits.

The result is employee 25 being assigned to inactive department 7. In an SOA architecture, where tables department and employee can be managed by the same service, and supposing that data persistence is managed using JPA, simple optimistic locking solves this issue:

1. DD reads department 7 and checks that it is active, and it is.
2. ER reads department 7, locks it using JPA optimistic force increment locking, checks that it is active, and it is.
3. DD checks that department 7 has no active employees, and it does not.
4. ER inserts employee 25 assigned to department 7 and commits (which adds one to the department version number [24]).
5. DD marks department 7 as inactive, commits and an optimistic locking exception rises, because the version number for department 7 in memory is different from that in the database, avoiding the inconsistency.

If steps 4 and 5 are interchanged, then DD could be able to mark department 7 as inactive, incrementing its version number after committing, and ER would detect it and would make rollback.

An approach for handling consistency across microservices is *eventual consistency*. This model doesn't enforce distributed ACID transactions across microservices. Instead, it proposes to use some mechanisms for ensuring that the system would be eventually consistent at some point in the future [34], [75]. Other choices for dealing with the consistency of data among microservices are avoiding transactions across microservices (if possible) [75] or the two-phase commit protocol that is not recommended [73], [76], [77]. Therefore, the *saga* pattern [78] which uses the eventual consistence model, is the most widely used mechanisms for handling consistency across microservices [33], [73], [76], [77], [79]. The Narayana blog [80] contains valuable information about global transactions in microservices as well as *Long Running Actions between microservices.* Atomikos [81], [82] is another valuable source of resources about distributed transactions in microservices.

B. A TRANSACTION MANAGER FOR WEB SERVICES

There are two ways to organize several web services (micro or not) when they collaborate in order to implement a function [83]: orchestration and choreography. In *orchestration*, there is a hierarchical organization, where one service, the coordinator, is responsible for managing the rest of services. In *choreography*, there is no hierarchy and services interact with each other as peers. In the context of the saga pattern for microservices, [8] and [73] agree that orchestration has more advantages than choreography.

Irrespective of service organization, it is not extremely complex to build a naïve transaction manager that can cope with global transactions for web services, because the concepts of transaction coordinator, participant and transaction context are still valid [4]. In this paper, I restrict myself to atomic transactions because the management of long-running transactions lies outside the scope of this paper. In order not to extend this paper any further, I do not depict the UML diagrams that characterize the design. Instead, I provide some key elements about this design:

- The transaction manager plays two roles: local transaction manager and remote service that can commit/rollback transactions. Therefore, transaction managers use the interposition technique acting as subordinate coordinators of the coordinator that originated the transaction.
- Transactions must keep the remote transaction managers of the remote services invoked within its scope.
- Business delegates or invocation proxies can register the remote transaction manager in the ongoing transaction and transmit the transaction context (basically the global id) to the invoked services.
- Begin increments the transaction counter in the local machine.
- Invoked web services brokers must begin a transaction in their machines with the global id received.
- Invoked services provide their response, while the connections that the DAOs use in the context of a global transaction remain opened waiting for the remote invocation of the commit/rollback. This approach is valid because there is a top service (the director of the orchestration or the first pair that started the interaction in a choreography) that will end the transactions immediately (i.e., in a few seconds or less) because the transaction is not long-running. The commit/rollback invocation is made using the global id, and the thread is not useful any more for finding transactions or connections. Therefore, the connection manager needs to keep track of the connections bound to the global id.

- If in the same machine a service is invoked twice with the same global id, the transaction manager puts the existing transaction in the current execution thread to make it accessible to the application services that run in that thread. The connection manager is invoked by DAOs running in that thread, which get connections from the connection manager. The connection manager can choose between reuse connections indexed by the global id, reuse the connections bound to the thread (and the global id), or create new connections.
- Commit/rollback decrements the transaction counter in the local machine.
- When the counter is 0, commit/rollback is done in the local machine and to the remote transaction managers within the scope of the transaction. Note that because invoked web services increment the local transaction counter, only the commit/rollback performed on the top-level transaction, which has no previous invocation, can set the counter to 0.
- The transaction global id becomes customary for indexing transactions and for indexing the connections created within the transaction in the connection manager. Moreover, the transaction must keep information about the local threads in which it is involved in order to delete it from the threads in which it has been involved (the same machine can receive more than one call in the context of the same transaction). In any case, as in in EJB, transactions are flat: each transaction is decoupled from and independent of other transactions in the system, and another transaction cannot start in the same thread until the current transaction ends [84].

I am aware that this description is too high-level, but the description of this transaction manager in detail would expand the paper to a considerable length.

VII. CONCLUSION AND FUTURE WORK

There is a huge amount of literature about transactional management. The section about related works considers about thirty references, and only a couple of prototypical references have been chosen in most categories. The seven categories analysed in that section have a different impact in this paper. Curricula recommendations have to be carefully analysed in order to determine whether or not the academy has recognized the concept and importance of service transactions, and in view of the analysis performed in this paper, this is not the case. Information systems and software engineering literature are essential for the development of enterprise applications. Information system literature conceives transactions as database transactions. On the contrary, software engineering literature (which in this paper does not include those references which provide a wide view of the discipline) considers service transactions but is mainly focussed on local transactions, ignoring global transactions. Global transaction literature focuses on the infrastructure for supporting global transactions but its examples are in some cases influenced by the standards it conforms to (e.g., JPA) and, in all the cases, it lacks the detail of the examples described in this paper, as is customary in order to make students understand the details involved in global transaction management. The literature about platforms and frameworks for enterprise application development focuses on the technical details for managing service transactions using these frameworks, but: (i) without knowledge about global transactions these texts are very difficult to understand; and (ii) it does not provide any insight about how service transactions are implemented. The same problem affects the documentation of frameworks for global transaction management, but to a higher degree, due to its higher complexity. Finally, the relevance of the literature about cloud computing mainly comes from the microservices used in this architecture for enterprise application development, but this literature has the same problem as the software engineering literature.

After this analysis, it is easier to see the four main contributions of this paper to the literature about transaction management in enterprise application. First, it explicitly identifies a use of transactions that have been widely used in the industry for years but which the academy has not recognized. This is what we have called the software engineering point of view of transactions, and the concept of service transaction constitutes the flagship element of this point of view. Of course, there were service transactions before this paper defined them, but they were not explicitly considered because they were to some extent overshadowed in curricula recommendations by database transactions. Second, it defines a knowledge unit focused on service transactions that describes the main concepts on this topic not covered in the existing curricula recommendations. Third, it gathers and refines into one paper valuable knowledge about service transactions spread in several references, which makes this paper a simple and powerful tool for teaching. And fourth, it defines several simple frameworks for teaching service transactions to undergraduate students. There are excellent open frameworks for dealing with service transactions, but their huge complexity makes them unsuitable for a pedagogical issue and, up to now, the academy had not tackled the issue of providing a simplified version of them for teaching. Note that the knowledge gathered in this paper as well as the frameworks defined in it describe the main contents for the knowledge unit proposed in this paper.

Comparing time and resource consumption of the frameworks proposed with regard to the professional frameworks is very complex. Frameworks proposed in sections IV and V have no counterparts in the industry, because they are simple frameworks used for a basic management of transactions and the industry focusses on solutions closer to the needs presented in Section VI. Is the framework drafted in Section VI comparable to professional frameworks? No, it is not remotely comparable because it does not take into account issues such as implementing JTA or its use within application servers. It is a draft for understanding transactional management in enterprise applications, but it does not try to compete against professional frameworks.

However, this does not detract from the frameworks defined in this paper. The framework presented in Section IV has been in use for, at least, ten academic courses about software engineering taught by the author and has also been used in the development of the Virtual Campus of the Universidad Complutense de Madrid, which has been running flawlessly for more than ten years. The framework presented in Section V is a simple extension of the concepts used in the previous framework for dealing with several resource managers and, along with the framework presented in Section VI, it has passed extensive tests performed by the people that designed the transactional framework used in the Virtual Campus of the Universidad Complutense de Madrid.

Future work is split into two big domains: the practical use of transaction management and its pedagogical presentation to the students. The practical use of transaction processing has to deal with the management of data in new deployment environments such as cloud computing, as well as data resource managers beyond those used in enterprise application development up to now. In most cases, the industry takes the baton of responsibility in this area because it needs to implement transactions in those new environments.

Regarding pedagogy, the academy needs to focus on teaching the common elements of transactional management, and not on teaching specific frameworks. Furthermore, the main abstract concepts that underlie these frameworks have to be presented to the students if a competent use of them is intended. The work done in this paper goes in this direction, presenting simplified frameworks that take into account the key elements of the industry solutions but omitting the huge number of details which exponentially increase the complexity of understanding professional frameworks. Therefore, a paper for delving into the complexity of teaching extended transaction models would be highly desirable, and the same goes for papers for dealing with the new challenges that the industry is undertaking at present and will undertake in the future regarding transactional management in new computing environments.

REFERENCES

[1] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture*. Crawfordsville, IN, USA: Addison-Wesley, 2003.

[2] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[3] Wikipedia. *Database Transaction*. Accessed: Jul. 8, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Database_transaction

[4] M. Little, J. Maron, and G. Pavlik, *Java Transaction Processing Design and Implementation*. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.

[5] *Jakarta EE*. Accessed: Jul. 8, 2022. [Online]. Available: https://jakarta.ee/

[6] *Microsoft.NET Framework*. Accessed: Jul. 8, 2022. [Online]. Available: https://docs.microsoft.com/es-es/dotnet/framework/

[7] *Spring Framework*. Accessed: Jul. 8, 2022. [Online]. Available: https://spring.io/

[8] I. Sommerville, *Engineering Software Products: An Introduction to Modern Software Engineering*. Hoboken, NJ, USA: Pearson, 2020.

[9] F. Pérez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and J. Vuckovic, "Highly available long running transactions and activities for J2EE applications," in *Proc. ICDCS*, 2006, p. 2.

[10] ACM/IEEE. (2020). *Computing Curricula 2020: Paradigms for Global Computing Education*. [Online]. Available: https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf

[11] ACM/IEEE. (2013). *Curriculum Guidelines for Undergraduate Programs in Computer Science*. [Online]. Available: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

[12] ACM/IEEE. (2020). *A Competency Model for Undergraduate Programs in Information Systems*. [Online]. Available: https://www.acm.org/binaries/content/assets/education/curricula-recommendations/is-2010-acm-final.pdf

[13] ACM/IEEE. (2014). *Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. [Online]. Available: https://www.acm.org/binaries/content/assets/education/se2014.pdf

[14] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 7th ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2022.

[15] C. J. Date, *An Introduction to Database Systems*, 8th ed. Crawfordsville, IN, USA: Addison-Wesley, 2004.

[16] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Burlington, MA, USA: Morgan Kaufmann, 1993.

[17] P. M. Lewis, A. Bernstein, and M. Kifer, *Databases and Transaction Processing: An Application-Oriented Approach*. Crawfordsville, IN, USA: Addison-Wesley, 2002.

[18] B. Weikum and G. Vossen, *Transactional Information Systems*. Burlington, MA, USA: Morgan Kaufmann, 2002.

[19] P. Bourque and R. E. Fairley. (2014). *IEEE Guide to the Software Engineering Body of Knowledge, SWEBOK V3.0*. [Online]. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering

[20] The Open Group. (1991). *Distributed Transaction Processing: The XA Specification*. [Online]. Available: https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[21] M. Richards, *Java Transaction Design Strategies*. San Francisco, CA, USA: C4Media, 2006.

[22] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*. Burlington, MA, USA: Morgan Kaufmann, 2009.

[23] A. L. Rubinger and B. Burke, *Enterprise JavaBeans 3.1*, 6th Ed. Sebastopol, CA, USA: O'Reilly Media, 2010.

[24] L. Jungmann, M. Keith, M. Schincariol, and M. Nardone, *Pro Jakarta Persistence in Jakarta EE 10*. New York, NY, USA: Apress, 2022.

[25] J. Cosmina, R. Harrop, C. Schaefer, and C. Ho, *Pro Spring 5: An In-Depth Guide to the Spring Framework and its Tools*. New York, NY, USA: Apress, 2017.

[26] M. J. Price, *C# 10 and .NET 6—Modern Cross-Platform Development*, 6th ed. Birmingham, U.K.: Packt Publishing, 2021.

[27] D. Geary and C. Hostmann, *Core JavaServer Faces*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2010.

[28] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *J. Object Technol. SIGS Publications*, vol. 1, no. 3, pp. 26–49, 1988.

[29] *Atomikos Resources*. Accessed: Jul. 8, 2022. [Online]. Available: https://www.atomikos.com/Main/AtomikosResources

[30] *Bitronix JTA Transaction Manager*. Accessed: Jul. 8, 2022. [Online]. Available: https://github.com/bitronix/btm

[31] *Narayana Documentation*. Accessed: Jul. 8, 2022. [Online]. Available: https://www.narayana.io/documentation/index.html

[32] J. Atelsek. (2020). Why 76% of companies are adopting multicloud and hybrid cloud approaches. S&P Global and Market Intelligence. [Online]. Available: https://www.oracle.com/es/a/ocom/docs/cloud/oracle-451-research-advisory-blog-adopting.pdf

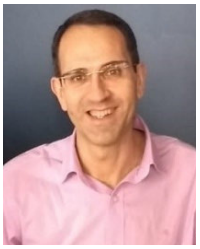
[33] A. Fachat, J. Laredo, and H. Verhoeven. (2022). Extend Saga to the cloud for distributed transactions. IBM. [Online]. Available: https://www.ibm.com/cloud/architecture/architecture/practices/extend-saga-to-the-cloud/

[34] (2022). *Compensating Transaction Pattern. MS Azure Cloud Design Patterns*. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

[35] P. Fan, J. Liu, W. Yin, H. Wang, X. Chen, and H. Sun, "2PC*: A distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform," *J. Cloud Comput., Adv., Syst. Appl.*, vol. 9, no. 1, pp. 1–22, Dec. 2020.

[36] S. M. Aghamirmohammadali, B. Momeni, S. Salimi, and M. Kharrazi, "Blue-pill oxpecker: A VMI platform for transactional modification," *IEEE Trans. Cloud Comput.*, early access, Mar. 22, 2021, doi: 10.1109/TCC.2021.3067829.

[37] A. Tripathi and G. Rajappan, "Scalable transaction management for partially replicated data in cloud computing environments," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 260–267.
[38] C. Wang and X. Qian, "RDMA-enabled concurrency control protocols for transactions in the cloud era," *IEEE Trans. Cloud Comput.*, early access, Sep. 29, 2021, doi: 10.1109/TCC.2021.3116516.
[39] G. Koloniari and E. Pitoura, "Transaction management for cloud-based graph databases," in *Proc. ALGOCLOUD*, in Lecture Notes in Computer Science, vol. 9511, 2016, pp. 99–113.
[40] D. Choi and S. Song, "Concurrency control method to provide transactional processing for cloud data management system," *Int. J. Contents*, vol. 12, no. 1, pp. 60–64, Mar. 2016.
[41] A. Waqas, A. W. Mahessar, N. Mahmood, Z. Bhatti, M. Karbasi, and A. Shah, "Transaction management techniques and practices in current cloud computing environments: A survey," *Int. J. Database Manag. Syst.*, vol. 7, no. 1, pp. 41–59, Feb. 2015.
[42] Wikipedia. *Application Server*. Accessed: Jul. 8, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Application_server
[43] *X/Open Preliminary Specification P209: Distributed Transaction Processing—The TX (Transaction Demarcation) Specification*, X/Open Company, Mountain View, CA, USA, 1992.
[44] G. Chen, "Distributed transaction processing standards and their applications," *CITR Tech. J.*, vol. 1, pp. 41–52, Sep. 1995.
[45] Oracle. *Developing Applications with Oracle XA*. [Online]. Available: https://docs.oracle.com/database/121/ADFNS/adfns_xa.htm#ADFNS761
[46] IBM. (2022). *X/Open Distributed Transaction Processing Mode*. [Online]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=managers-designing-xa-compliant-transaction
[47] Object Management Group. (2003). *Transaction Service Specification. Version 1.4*. [Online]. Available: https://www.omg.org/spec/TRANS/1.4/
[48] Micro Focus. (2020). *Orbix 6.3.12. Corba OTS Guide: Java*. [Online]. Available: https://www.microfocus.com/documentation/orbix/orbix6312/pguide_java.pdf
[49] A. Navarro, J. Cristóbal, C. Fernández-Chamizo, and A. Fernández-Valmayor, "Architecture of a multiplatform virtual campus," *Softw., Pract. Exp.*, vol. 42, no. 10, pp. 1229–1246, 2012.
[50] M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, 3rd ed. Crawfordsville, IN, USA: Addison-Wesley, 2003.
[51] Oracle. *The Java Tutorials. JDBC Introduction*. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html
[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. New York, NY, USA: Pearson, 2015.
[53] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Crawfordsville, IN, USA: Addison-Wesley, 2006.
[54] Jakarta. (2020). *Jakarta Transactions 2.0*. [Online]. Available: https://jakarta.ee/specifications/transactions/2.0/jakarta-transactions-spec-2.0.html
[55] (2012). *EJB 3.0—Nested Transaction ! = Requires New? Starckoverflow*. [Online]. Available: https://stackoverflow.com/questions/10817838/ejb-3-0-nested-transaction-requires-new
[56] J. Delgado, "A service-based framework to model mobile enterprise architectures," in *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*. Hershey, PA, USA: IGI Global. 2011.
[57] T. Erl, *SOA Design Patterns*. Upper Saddle River, NJ, USA: Prentice-Hall, 2008.
[58] Microsoft. (2021). *Microsoft.NET. Service Transaction Behaviour*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/service-transaction-behavior
[59] Object Management Group. (2003). *Transaction Service Specification*. [Online]. Available: https://www.omg.org/spec/TRANS/1.4/PDF
[60] P. Kumar. (2014). *Java DataSource, JDBC DataSource Example*. [Online]. Available: https://www.journaldev.com/2509/java-datasource-jdbc-datasource-example
[61] ProgrammerSought. *Distributed Transactions Based on XA in MySQL*. Accessed: Jul. 8, 2022. [Online]. Available: https://www.programmersought.com/article/96704095327/
[62] *ArjunaCore*. Accessed: Jul. 8, 2022. [Online]. Available: https://www.narayana.io/arjuna-core/index.html
[63] R. Johnson and J. Hoeller, *J2EE Development Without EJB*. Hoboken, NJ, USA: Wiley, 2004.
[64] M. D. Hansen, *SOA Using Java Web Services*. Upper Saddle River, NJ, USA: Prentice-Hall, 2007.
[65] A. Navarro and A. da Silva, "A metamodel-based definition of a conversion mechanism between SOAP and RESTful Web services," *Comput. Standards Interfaces*, vol. 48, pp. 49–70, Nov. 2016.
[66] OASIS. (2009). *Web Services Transaction (WS-TX) TC*. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx#technical
[67] OASIS. (2009). *Web Services Atomic Transaction (WS-AtomicTransaction)*. [Online]. Available: http://docs.oasis-open.org/ws-tx/wsat/2006/06
[68] OASIS. (2009). *Web Services Business Activity (WS-BusinessActivity)*. [Online]. Available: http://docs.oasis-open.org/ws-tx/wsba/2006/06
[69] OASIS. (2009). *Web Services Coordination (WS-Coordination)*. [Online]. Available: http://docs.oasis-open.org/ws-tx/wscoor/2006/06
[70] M. Little. (2009). REST and transactions? InfoQ. [Online]. Available: https://www.infoq.com/news/2009/06/rest-ts/
[71] G. Pardon and C. Pautasso, "Towards distributed atomic transactions over RESTful services," in *REST: From Research to Practice*. New York, NY, USA: Springer, 2011.
[72] M. Little. (2013). *REST-Atomic Transactions 2.0. Draft 8*. [Online]. Available: https://www.narayana.io/docs/specs/restat-v2-draft-8-2013-jul-29.pdf
[73] C. Richardson, *Microservices Patterns*. Shelter Island, NY, USA: Manning, 2019.
[74] K. Bhole and R. Nareddy. (2020). Opening banking through architecture re-engineering. A microservices-based roadmap. Deloitte. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/financial-services/us-enabling-platform-banking-pov.pdf
[75] S. Petunin. (2021). A guide to transactions across microservices. Baeldung. [Online]. Available: https://www.baeldung.com/transactions-across-microservices
[76] R. Ganji. (2021). How to get closer to consistency in microservice architecture. DZone. [Online]. Available: https://dzone.com/articles/transaction-management-in-microservice-architectur
[77] K. Xiang. *Patterns for Distributed Transactions Within a Microservices Architecture*. [Online]. Available: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture#
[78] H. Garcia-Molina and K. Salem, "SAGAS," in *Proc. SIGMOD*, 1987, pp. 249–259.
[79] O. Başkök. (2019). *SAGA Pattern Briefly*. [Online]. Available: https://medium.com/trendyol-tech/saga-pattern-briefly-5b6cf22dfabc
[80] *Narayana Blog Team*. Accessed: Jul. 8, 2022. [Online]. Available: https://jbossts.blogspot.com/
[81] G. Pardon. (2016). *ACID Transactions Across REST Microservices*. [Online]. Available: https://www.atomikos.com/Blog/ACIDTransactionsAcrossMicroservices
[82] G. Pardon. (2017). *Transactional REST Microservices With Atomikos*. [Online]. Available: https://www.atomikos.com/Blog/TransactionalRESTMicroservicesWithAtomikos
[83] T. Erl, *Service-Oriented Architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, 2005.
[84] (2010). *Sun Java System Application Server Platform Edition 9 Developer's Guide. Handling Transactions With Enterprise Beans*. [Online]. Available: https://docs.oracle.com/cd/E19501-01/819-3659/beaje/index.html



ANTONIO NAVARRO received the Ph.D. degree in mathematics computer science from Universidad Complutense de Madrid, Spain, in 2002. He is an Associate Professor at the Departamento de Ingeniería del Software e Inteligencia Artificial, Universidad Complutense de Madrid. His research interests include software engineering, software architectures, software design patterns, software modeling, and model-driven architecture. He is the author and the coauthor of several papers related to these research topics.