

Diss. ETH Nr. 13993

Composite Systems: Decentralized Nested Transactions

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Guy Pardon
Dipl. Ingenieur Vrije Universiteit Brussel
citizen of Belgium
born May 30, 1970

accepted on the recommendation of
Prof. Dr. G. Alonso, examiner
Prof. Dr. C. Beerli, co-examiner

2000

Contents

I. Concepts and theory	13
1. Introduction	15
1.1. Contribution of the dissertation	15
1.2. Layout	16
2. Motivation	17
2.1. Emerging architectures in information systems	17
2.2. Summary of desirable properties	18
2.3. Flat transaction technology	21
2.3.1. Flat transactions	21
2.3.2. Flat transactions and distribution	22
2.3.3. Robustness	26
2.3.4. Parallelism	26
2.4. Nested transaction models: state-of-the-art	26
2.4.1. General characteristics of nested transactions	26
2.4.2. Closed nested transactions	27
2.4.3. Open nested transactions	27
2.4.4. Multilevel transactions	28
2.5. Existing industrial efforts for Internet transaction processing	29
2.5.1. Enterprise Java Beans (TM)	29
2.5.2. OTS: CORBA Object Transaction Service	30
2.5.3. Transaction Internet Protocol (TIP)	30
3. Theory of composite systems	31
3.1. Introduction	31
3.2. Basic assumptions	32
3.3. Defining composite systems	34
3.3.1. Local definitions	34
3.3.2. Composite definitions	37
3.4. Correctness without failures	40
3.4.1. Structure of a composite execution	40
3.4.2. Correctness of failure-free composite systems	46

3.5. Correctness in the presence of failures and recovery	48
3.5.1. Atomicity as seen locally in a schedule S	49
3.5.2. Atomicity in a composite execution	53
3.5.3. Correctness in the presence of failures	54
4. Protocols for correct composite systems	57
4.1. Non-compensatable invocations	57
4.1.1. Invocation states	57
4.1.2. Recoverability	58
4.1.3. Ensuring that a local computation exists	58
4.1.4. Ensuring that a global computation exists	59
4.1.5. Nested semantics: abort propagation to descendants	61
4.1.6. Lock Inheritance	61
4.1.7. Orphans	62
4.2. Compensatable invocations	64
4.2.1. Invocation states	64
4.2.2. Lock inheritance	67
4.3. Abort policy and completion of a composite execution	67
4.3.1. Correctness of abort	69
4.4. Protocols for correct scheduling	70
4.5. Enforcing SI and SR	72
4.5.1. Full lock inheritance	72
4.5.2. No intra-root parallellism	73
4.5.3. No lock inheritance	73
4.6. The case of recursive calls	73
4.7. Global knowledge required for the different solutions	74
4.7.1. Information needed for FS	74
4.7.2. Lock inheritance	74
4.7.3. Recursion detection	74
4.7.4. Orphan detection	75
4.7.5. Two-phase commit	75
4.8. Implications of <i>two-phase commit</i> on the system assumptions	76
4.9. A discussion of existing transaction monitor environments	76
4.9.1. Blocking among siblings	76
4.9.2. No blocking among siblings	77
4.9.3. Encina	77
4.10. Asynchronous transaction management	77
4.11. Summary	77
 II. Implementations of composite systems	 79

5. Composite systems in practice: the CheeTah system	81
5.1. The framework approach to software development	81
5.2. Introducing <i>CheeTah</i>	82
5.3. Architecture of <i>CheeTah</i>	82
5.4. Programming <i>CheeTah</i> applications	85
5.5. Scheduling and concurrency control	86
5.5.1. Call-level locking: compensation	86
5.5.2. Operation-level locking: no compensation	87
5.6. Implementation of atomicity	87
5.7. Dealing with undo operations	88
5.8. Optimizations of Logging and Locking	89
5.9. Summary	89
6. Implementation aspects of CheeTah	91
6.1. Lock management	91
6.2. Log management and recovery	91
6.3. Application-level classes and interfaces	93
6.3.1. Interaction with the transaction engine: <i>CompositeUserTransaction</i>	95
6.3.2. Implementing a simple component	95
6.3.3. Implementing parallel calls	95
6.4. Propagation of 2PC	95
6.5. The resource (database) interfaces	97
6.5.1. Application level connections	97
6.5.2. System level interfaces	97
6.6. The core interfaces	97
6.7. Summary	100
7. Performance of CheeTah	103
7.1. Performance Analysis	103
7.1.1. Measurements and Results	105
7.1.2. Comparison with Existing Systems	108
8. Conclusions	111
8.1. Transactional theory	111
8.2. Programming distributed transactional applications	112
8.3. Performance	112
8.4. Two-tier, three-tier and multi-tier systems	112
8.5. Possible extensions	113
8.5.1. Theory issues	113
8.5.2. Practical issues	114
A. Appendix: Selected examples of CheeTah application code	123
A.1. Programming a service component	123
A.2. Configuring the server	125

A.3. Providing intra-service parallelism 126

Abstract

This thesis deals with how to define and build correct distributed nested transactions in a completely decentralized way. In particular, it studies nested transactions in a distributed system of arbitrarily interconnected independent components, with only local knowledge in each component and local rules to enforce correctness. Such systems are called composite systems, and the corresponding transactions are composite transactions. Invocations of remote servers are automatically treated as subtransactions of the issuing transaction, improving both fault-tolerance and response times. Open transaction models are supported, in order to release resources as soon as possible. Correctness of composite transactions is defined, and some correct and minimal protocols are proposed. The theoretical framework also applies to existing distributed transaction models and identifies their problems. By proposing a more powerful transaction model, namely the more general composite transactions, these problems can be solved. On the practical side, this work presents the *CheeTah* prototype, a black-box framework written entirely in *Java*, for light-weight composite transaction management. It implements all the protocols that were presented in the theoretical discussion, and shows that they are feasible in practice.

Kurzfassung

In dieser Doktorarbeit wird die Definition und Implementation von verteilten, verschachtelten Transaktionen im dezentralen Fall untersucht. Von besonderem Interesse war dabei die Frage, wie die Korrektheit verschachtelter Transaktionen für beliebig verbundene, unabhängige Komponenten mit ausschliesslich lokalem Wissen und lokalen Regeln garantiert werden kann. Ein solches System wird in seiner Gesamtheit als ein komposites System bezeichnet, die ablaufenden Transaktionen entsprechend komposite Transaktionen. Der Zugriff auf Server im Netzwerk wird automatisch als Unter-Transaktion der ausführenden Transaktion behandelt, wodurch sowohl Fehler Toleranz als auch Antwortzeit verbessert wird. Um Systemkapazitäten so bald als möglich wieder frei zugeben, wird ein offenes Transaktionsmodell unterstützt. Die Korrektheit der kompositen Transaktionen ist definiert, und es werden eine Reihe von korrekten und doch einfachen Protokollen diskutiert. Das theoretische Modell kann auch auf andere, schon existierende Transaktionsmodelle angewandt werden, wobei eine Reihe von Schwächen in diesen aufgezeigt werden. Diese Probleme koennen durch unser generalisiertes, komposites Transaktionsmodell gelöst werden. Im praktischen Teil dieser Arbeit wird der *CheeTah* Prototyp vorgestellt, welcher eine sehr kleine, komposite Transaktionsverwaltung als in Java geschriebene Bibliothek zur Verfügung stellt. Dieses implementiert alle im theoretischen Teil vorgestellten Protokolle, und beweist ihre Alltagstauglichkeit.

Acknowledgements

It can only be because of pure luck that one ends up in what feels like probably one of the best *and* one of the nicest groups in the world. I feel like I was very fortunate indeed, to be able to perform this work under the supervision of Gustavo Alonso, and I enjoyed every moment of it. Of course, this was not only due to his inspiring way of managing the *IKS* group, but also thanks to the excellent company of all members of his team. I wish to express special gratitude to Bettina Kemme and Andrei Popovici who shared the office with me for a long time, and have repeatedly shown to be very fine colleagues and friends. Also, I thank Etzard Stolte for his generosity, both personally and professionally. I should say that his years of experience outside academia have been an enriching source of information, part of which he has been so kind to share with his *IKS* colleagues.

Naturally, I should extend this expression of gratitude to the *DBS* group of Prof. Schek's, where it all started in the first place. Finally, it has been an honour to have Prof. C. Beeri as my co-examiner, and I am very grateful to him for having taken the time to thoroughly read my work and present useful feedback.

Concerning the non-professional part of things, I should thank all my friends and family both for their moral support and for making life as pleasant.

Part I.
Concepts and theory

1. Introduction

Transactions greatly facilitate the task of dealing with failures, recovery, and concurrency control. They also allow to encapsulate operations and associate concrete semantics to invocations. Tools that provide transactional primitives for the design and development of information systems have a long history behind them, starting with the first TP-monitors [BN97] of almost three decades ago. Today, transactional technology is well understood and widely used. A plethora of products support transactional primitives and are used in a large variety of applications. TP-Monitors, in particular, in any of their versions (TP-heavy or TP-light), are at the core of many running systems and have even provided language extensions to allow programmers to use transactions directly in their code (e.g., Transactional-C in *Encina* [Enc]).

In spite of this success, there is a growing number of applications for which existing tools are not suitable. The main problem is that current products use a centralized component for scheduling transactions [BK99]. To centralize operations in this way might be exceedingly difficult if the components reside in different organizations or if the components interact over the Internet. It may also be quite difficult in large web-farms or in clusters expanding several LANs. Unfortunately, none of the existing alternatives quite solves the problem. For instance, the TIP protocol [LEK99] provides a limited form of atomicity but no concurrency control. Similarly, persistent queues [MD94] provide atomic asynchronous interaction but concurrency control cannot be easily enforced. In practice, it is very difficult if not impossible to find a tool or product that supports transactional interaction without a centralized monitor and without enforcing a static configuration of the components.

The main objective of this thesis is to show how to build completely autonomous components that, without any centralized coordination, can interact transactionally. Components act as application servers that invoke each other's services to implement increasingly complex application logic. The components can be combined in any configuration and can be dynamically added or removed without compromising correctness. Such systems are called *composite systems*.

1.1. Contribution of the dissertation

By studying composite systems, this report makes a number of contributions. Firstly, it provides a detailed analysis of the impact of composite system requirements on concurrency control and recovery theory. More precisely, it offers insights on the influence

of issues like autonomy, distribution, decentralization and locality on transactional theory and scheduling protocols. The classical work of [BBG89] is cast in the context of these restrictions, allowing a precise definition of correctness in concurrent distributed architectures. In addition, the dissertation clarifies the influence of the requirements on practical protocols, and explores the amount of information that must be exchanged between participants in composite transactions. Problems in existing solutions and architectures are identified, and solutions offered. The dissertation also presents a pure *Java* implementation of a reusable framework for composite systems. Its performance and scalability, and how it could be extended in the context of *Enterprise Java Beans (TM)* [EJB, DP00] technology and component-based development is also illustrated.

1.2. Layout

The dissertation is organized in two main parts: a first part covering all theoretical aspects and a second for the implementation issues.

The first part consists of an extended motivation in chapter 2, a theoretical foundation in chapter 3 and a discussion of several solutions in chapter 4. It is the main part of this thesis.

The second part contains the practical proof-of-concept, and demonstrates that composite systems are feasible in practice. We will discuss a prototype implementation called *CheeTah*, that implements all the solutions that are presented in chapter 4. Chapter 5 focuses on the way these solutions are incorporated, chapter 6 discusses some of the interfaces in the implementation, and chapter 7 discusses performance and scalability. Finally, chapter 8 presents the conclusions that can be drawn from this work, as well as possible extensions.

2. Motivation

2.1. Emerging architectures in information systems

In the era of the *Internet*, application areas such as *business-to-business* and *business-to-consumer electronic commerce* are important for information systems, as well as for economics [MB97]. Essential topics in this context are, among others, information retrieval (search engines of all kinds), information theory (cryptography [Sch94] and payment protocols [MB97]), and semistructured data (XML [ABS00]). All these technologies try to facilitate the way in which distributed systems can co-operate across networks in general and the Internet in particular. In this work, we deal with the same kind of environments, but the accent is on yet another aspect. We will be dealing with *transactions*, or ways to allow multiple users to manipulate data concurrently. As we will soon show, existing solutions in this area are far from ideal and based on assumptions that might no longer be valid.

In this dissertation, we are interested in distributed and dynamic environments where a set of different, autonomous information systems interact transactionally. We call such systems composite systems. Figure 2.1 is an example of a composite system (for simplicity, only part of the system is represented). The figure illustrates the hierarchy of invocation calls between different services across a variety of components. In this case, a complex electronic commerce architecture is shown, where different and independent servers (from different organizations) invoke each other's services to accomplish an e-commerce transaction. For instance, buying a complex product (as in the top left corner of the example) involves retrieving the necessary parts to assemble it, as well as planning the assembly and arranging the shipment procedure. This decomposition into basic steps is done at entry point A in the figure. All of these activities are again services offered at other servers in the distributed system. For instance, checking the stock for availability of each part is done at the *Inventory control system*. There, lack of availability is translated into yet another invocation of a third party server, namely the *ACME supply e-commerce interface*. On the other hand, customers are allowed to trace the status of their order, as illustrated in the lower right part of the figure. Through the *Manufacturing control system*, yet another server at another location, queries concerning the order status can be issued, and again translated into delegated calls somewhere else. Thus, in principle each component is implemented as an independent entity residing in a different location (over a LAN or a WAN as, e.g., the electronic commerce *interface*). These components invoke the services provided by other components (Figure 2.2.a) forming an arbitrary

nested client-server hierarchy in which increasing levels of abstraction and functionality can be introduced (Figure 2.2.b).

In our model, the most important aspects of each component are the application logic layer (a *server*) and a resource manager (usually a database), that is accessed by the former.

Our goal here is to design and implement an inherently decentralized and advanced transactional mechanism. This mechanism should allow to combine such components in any possible configuration, so that transactions can be executed across the resulting system guaranteeing correct (transactionally correct) results, even if the configuration is dynamically altered. It is crucial for these components to remain independent of each other, that is, there should not be a centralized component controlling their behavior. Additionally, nested transactions should be supported, because of the inherent distributed nature, leaving room for alternatives on failure of a particular remote call. For instance, if the *ACME e-commerce* interface is down, then another supplier can be tried.

To see why these characteristics are important, it suffices to look at the *Internet*: servers may be unreachable, new servers appear without notice, and the nature of the *Internet* itself already excludes the possibility of relying on a fixed configuration between the different systems that co-operate. For these same reasons, such a system should also be able to cope with failing calls in a flexible and elegant way: if a server cannot be reached at a given moment, it should be possible to try an alternative service. One of the golden rules in *electronic commerce* is to maximize customer satisfaction by minimizing the number of service denials. This suggests that remote failures be dealt with in the service itself, without making them visible to the customer invoking the service. Finally, there is no reason why different remote service invocations within the same task should not be executed in parallel, thereby minimizing response time. However, as will become clear in the next sections, hardly any of these desirable properties is feasible with existing solutions. We will see why, and what can be done about it.

2.2. Summary of desirable properties

In order to offer a powerful and flexible architecture for distributed systems, we will show that the following characteristics have to be provided:

1. Autonomous and completely decentralized components: reliance on a fixed part of the system is not feasible since it constitutes a single point of failure and does not scale well as the system grows.
2. Nested characteristics [Mos85]: these improve parallelism (response times) and fault tolerance.
3. Local transaction management: no matter how components invoke each other, and how complex the execution of a transaction, the system ensures correctness in all cases by using only local knowledge at each component.

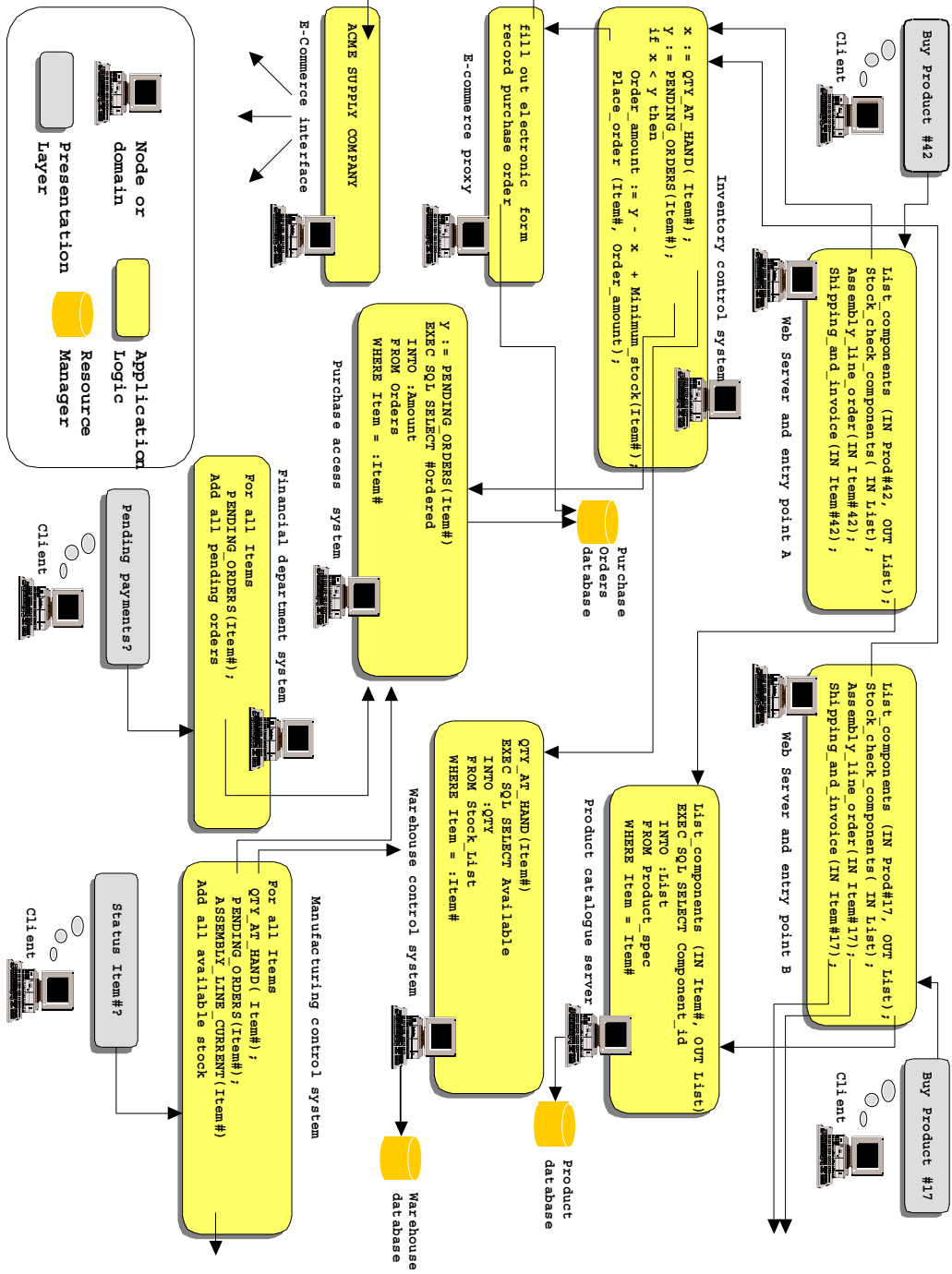


Figure 2.1.: Example of a composite system

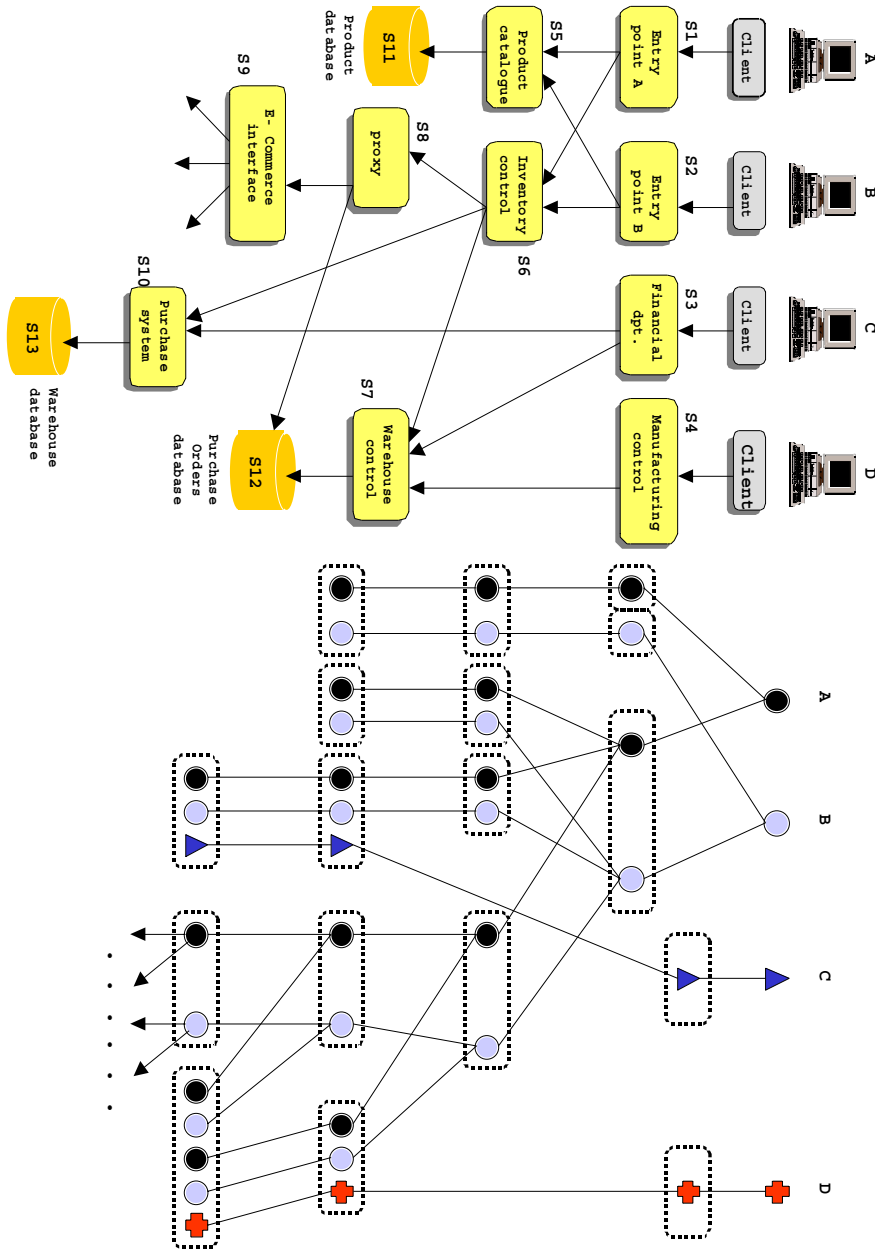


Figure 2.2.: Invocation hierarchy (a, left-hand side) and transactional structure (b, right-hand side) in the composite system shown in Figure 2.1

As we will argue, the current state of the art does not address these requirements in a practical and satisfactory way. Therefore, the later chapters of this work are dedicated to developing both the theoretical and practical aspects of a possible solution. What is needed is a general framework, capable of reasoning in an inherently distributed, decentralized way (unlike anything existing to date). In such a model, lack of global knowledge and lack of central roles must be fundamental assumptions. Once this foundation is in place, it should be possible to design protocols that allow distributed systems to interact transactionally in an ad-hoc way, fully decentralized and with nested characteristics.

In the rest of this chapter, we will give an overview of various existing approaches, all of which are based on a centralized architecture [Gor00], and most of which support only flat transactions. Only one commercial nested transaction system exists [Enc] (at least to the knowledge of the author), and it is definitely centralized, and the nested characteristics cannot be tuned in a flexible way (client applications cannot request certain quality of service based on their needs).

2.3. Flat transaction technology

2.3.1. Flat transactions

The transactional model for (distributed) computing has been around for many years and it is considered a well-established and mature technology [Ber90, GR93, BHG87, BN97, BG82, ACM94].

The basis of the classical transaction theory are the four ACID properties – atomicity, consistency, isolation and durability – that define a computation as being transactional. Within the scope of this work, the influence of *distribution and autonomy* on *isolation and atomicity* will be of particular interest.

Although there is nothing fundamentally wrong with ACID-ity, most of the transactional technology in use today [Mal94, BN97, GR93, Gor00, Ber90] has been developed before the *Internet* grew into what it is today. Consequently, in today’s transaction systems, distribution appears as an ‘add-on’ and not as an inherent feature. The term *flat transactions* refers to the fact that conventional transactions have no internal structure (no logical subparts) and that atomicity is implemented by aborting everything as soon as one operation fails. While this is fine for centralized systems, this model is not suitable for distributed architectures. In a distributed system, almost by definition, tasks have an internal structure where each remote call is a separate logic component. For instance, in case of remote failure, another node (or network route) could be used to solve the task. Thus, it seems inadequate to abort everything in a distributed computation simply because one server appears to be down at a given moment. Yet this is what flat transaction technology implies, thereby losing a big opportunity to improve services by exploiting properties of distributed architectures.

2.3.2. Flat transactions and distribution

The earliest applications of flat transactions were in the field of databases [Gra81]. Distributed transactions, along with distributed databases, were not seriously considered until the eighties and early nineties [YHMA92, Ber90, OV91]. Not suprisingly, this more or less coincides with the earlier stages of the global network. At that time, a lot of attention was devoted to all kinds of concurrency control techniques (such as *strict two-phase locking* and timestamps [BHG87], to name a few) and how they could be reconciled with a distributed transactions system, i.e., in which a distributed or *global* transaction may have *local transactions* on multiple sites (although it was common to assume that each distributed transaction would have at most one local transaction on a given site).

One of the important conclusions of these research activities was that it suffices to have *strict two-phase locking* and *two-phase commit* [BHG87, GR93] on each node of a distributed database system to provide correct isolation and atomicity. Because *strict two-phase locking* already was, and still is, the basic technique that virtually every database system used for enforcing isolation, there has been no fundamental change in technology. Even today the flat transaction is pervasive, and systems have been enriched with the *two-phase commit* protocol to make them work in distributed environments [Ber90]. Thus, much of the work on distributed transactions and advanced transaction models in this context [Elm90] – focusing on techniques other than locking and how to avoid *two-phase commit* – turned out to be practically irrelevant.

A lot of work already exists concerning distributed commitment. An important theoretical fact is the impossibility of non-blocking consensus in asynchronous systems with communication or node failures [FLP85]. The prevailing protocol with acceptable message overhead has proven to be *two-phase commit* [Gra78, LS76, Mul93, LAA94]. Other protocols exist, such as *three-phase commit* [Ske81], which tries to avoid blocking. However, it is more expensive in terms of messages exchanged and blocking is only avoided when no communication failures occur (which makes it impractical and expensive).

In the *two-phase commit* protocol, distributed consensus is reached by two message rounds, under the supervision of a coordinator. In the first round, the coordinator asks each of the participating nodes whether it can agree with an eventual commit outcome. If the participant detects no local problem on behalf of the transaction, it votes yes, thereby giving up any local right to abort unilaterally (leaving the participant in the so-called *in-doubt* state). Otherwise, the vote will be no. The coordinator collects all votes, and only if all participants vote yes it will decide on global commit and, as the second round, send commit messages to everyone. If at least one participant did not reply with a yes vote (either because of timeout or because a no was received), then the coordinator decides on global abort and notifies, as the second round, any in-doubt participants. Of course, all this has to be done with the proper amount of logging (to survive crashes). When and how this logging is done, and how to use it during recovery is the main difference between the many variants that have been proposed [ML83, AHCL97, BHG87].

Delicate problems arise in case nodes fail, especially if the coordinator fails while some nodes are in-doubt. This is the so-called blocking time window of the protocol,

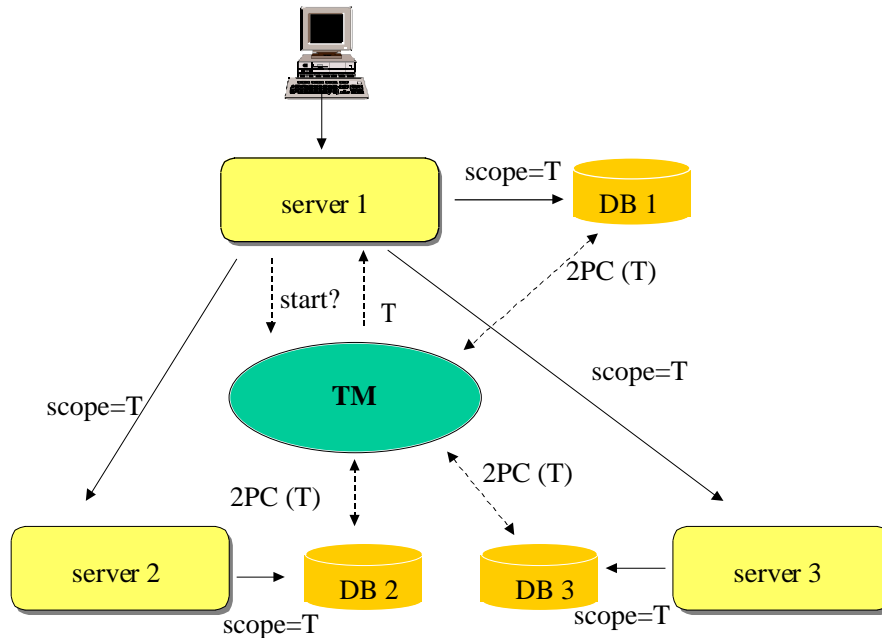


Figure 2.3.: Today's transaction processing architecture

which is preferably kept as small as possible. Nevertheless, this window exists, and because of the impossibility result of [FLP85] it is something that one has to live with.

Within this context, the resulting architecture for distributed transactions is shown in Figure 2.3. A number of *RDBMS* (DB_1, DB_2, DB_3 in the example), also called *resource managers* in the literature, are subject to the coordination of a central *transaction monitor* (*TM* in the illustration). This *TM* is responsible for creating the *transaction identifier* and coordinating the *two-phase commit* protocol. A *RDBMS* is usually never invoked directly: instead, a server process is invoked (such as $server_1, server_2, server_3$ in the example), and this process accesses the data. The reason for this is load balancing: by adding more server processes on more CPUs, the system can better distribute the load. Server processes can invoke each other (re-use each other's logic) as well as directly access their local *RDBMS* on behalf of some transaction with *transaction identifier* T . The identifier T is part of all communications within the system: both inter-server calls and *RDBMS* calls are aware of which T is being executed. For isolation, each *RDBMS* inspects incoming requests and sets locks based on the *transaction identifier*. This is where *two-phase locking* is enforced. Because there is only one *transaction identifier* for every transaction, different intra-transaction accesses to the same data will be allowed by every *RDBMS* involved. For atomicity, the *transaction monitor* runs a *two-phase*

commit protocol between all resources involved, which again uses the *transaction identifier* to keep track of which operations to commit. Note that the *transaction monitor* is the only entity that knows the resources that are part of one given transaction. Such an architecture does not favour decentralization. One of the goals of this dissertation is to eliminate the central role the *transaction monitor* plays.

In the above example, there was only one *transaction monitor* process involved. As long as this assumption holds, and each server knows what other servers do (this point will be clarified in a later chapter when we discuss recursive client-server relationships), no serious anomalies arise if a distributed flat transaction is used: in an ideal situation, with no failures and no concurrency, every transaction can be executed and will be correct.

However, when large-scale distribution is considered, it is not realistic to assume a central coordinating entity that manages transactions: if multiple information systems interact transactionally, more than one *transaction monitor* will be involved. An example is shown in Figure 2.4. Because each *transaction monitor* works with its own policies for determining a *transaction identifier*, a global distributed transaction will have multiple and possibly different identities in each system. In Figure 2.4, three organizations interact, each of them with their own *transaction monitor* (TM_A, TM_B, TM_C in the example). Due to the different policies for identifiers, a *mapping* has to be performed when invocations cross organizational (and therefore *transaction monitor*) boundaries. In practice, [GR93] mentions two possibilities: the *push* model and the *pull* model, depending on where the mapping is maintained (on the caller side or on the callee side). In the particular case of the *Internet*, there have been some recent efforts to define the *transactional internet protocol (TIP)* [LEK99] standard for doing this type of mapping. Nevertheless, irrespective of where or how it is done, information is lost in the process. For instance, Figure 2.4 clearly shows that if a client invocation reaches *server_C* through the domains of two different transaction managers (TM_A, TM_B) then the two invocations of the same global transaction will be known to TM_C as two *different* local transactions T_2 and T_4 . If both calls need to access the same data, the resource manager will block one of them, thereby deadlocking the execution. The other option is that all transaction managers in the system use the same identifier for the work of the same client, but as noted in [GR93, Gro91], this is not usually the case.

This subtle but very limiting feature of current technology is due to the fact that existing systems are not required to recognize different parts of the same distributed transaction as one unit. Consequently, *strict two-phase locking* will treat them as different transactions and block one call accordingly. In distributed transaction processing, this problem is also known as the *diamond problem* [EJB]. One might argue that diamond problems are probably very rare, since they only happen on common data access through different invocation hierarchies. However, by definition, these accesses are done on behalf of the *same* client and therefore much more likely to happen in practice, simply because the different calls share the same context.

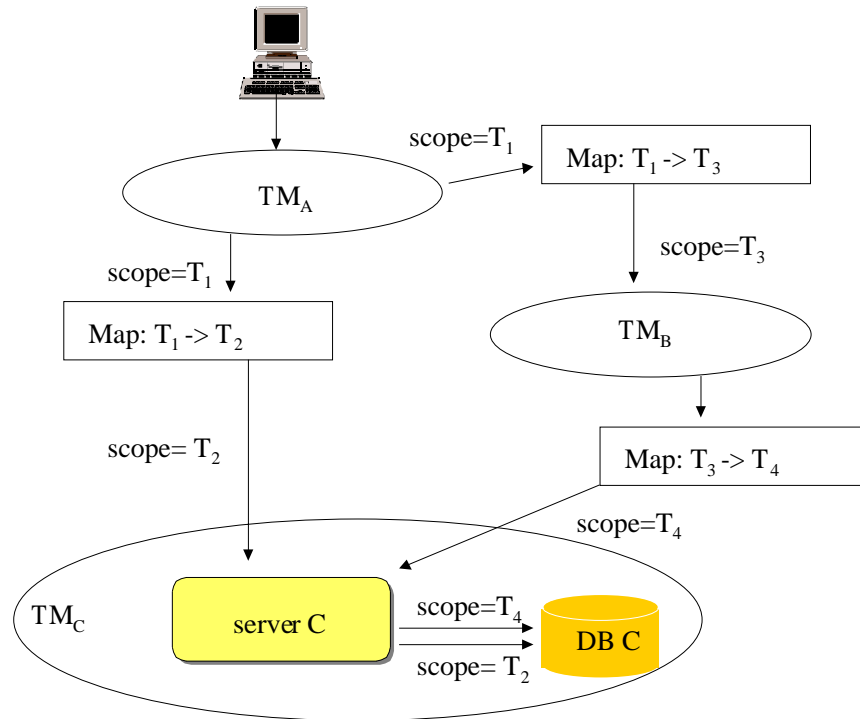


Figure 2.4.: Multiple TM processes in one transaction (a.k.a. the diamond problem)

2.3.3. Robustness

Another problem with flat transactions and distribution concerns intra-transaction robustness against remote failures. Indeed, aborting everything as soon as one operation fails may make sense for centralized databases, but on *Internet* systems failures do not merely depend on local effects, but also on the availability of remote sites. This suggests that a more robust model be used such as, for instance, a nested transaction model. In nested transactions a remote failure does not restrict locally started transactions from completing because the failure can be detected at execution time and one of a number of provided alternatives can be tried. Examples of existing systems exploiting this feature are [ELWR82, SAS99]. To the knowledge of the author, however, there is only one commercial *transaction monitor* that provides nested transactions (*Encina* [Enc], the commercial extension of a research project [EMS91]). Otherwise, nested transactions remain a theoretical curiosity [GR93].

2.3.4. Parallelism

The third and last problem with existing transactions is their restriction to serial execution within one transaction: multithreaded transactional applications are not allowed. This seems to be overly restrictive, especially when considering remote transactional calls on the *Internet*: if two entirely different sites are to be called, then there is no obvious reason why this should not be done in parallel. Although it would probably be possible to incorporate this feature into the flat model, it appears as a natural extension as soon as one moves into nested transaction models, as discussed in the rest of this chapter.

2.4. Nested transaction models: state-of-the-art

So far discussion has concentrated on the classical *flat* transaction model. The reason for this is that virtually no existing product or system will use anything else. And yet more advanced and more elegant models exist. One should keep in mind, however, that most of these concepts have never been implemented [Moh98]. Relevant to the subject are the different paradigms of *nested transactions* [Mos85]. There are many variants around, and a brief review of each of them now follows.

2.4.1. General characteristics of nested transactions

The term *nested* refers to the fact that a transaction can be (recursively) decomposed into *subtransactions*, parts that form a logically related subtask. In this way, a *parent* transaction can have multiple *children*, each child being a subtransaction. A key point is that a successful subtransaction only becomes permanent (i.e., committed) if all its ancestors succeeded as well, whereas the inverse does not hold: if a child fails, the parent is free to try an alternative task, thereby rescuing the global work.

The advantages of doing this are twofold: firstly, a failure of a subtransaction clearly delimits the scope of the failed part, allowing a clear definition of alternative policies. Secondly, subtransactions define clear boundaries of isolation among parts of the *same* overall task. This can be exploited by the system to allow parallelism inside one and the same global transaction.

2.4.2. Closed nested transactions

In the closed nested paradigm [Mos85] locks are acquired as in *two-phase locking*, but extra policies determine the behaviour of subtransactions with respect to each other. More precisely, as soon as a subtransaction finishes its locks are passed on to its parent. A child of that very same parent will *inherit* these locks whenever it needs access to the same data. Without this feature, children of the same parent could block each other, resulting in very impractical systems. It is this very characteristic that so far has made closed nested transactions unfit for practical use: no existing *RDBMS* supports lock inheritance. Note that [Mos85] does not discuss any implementation but merely describes the main concepts such as lock inheritance and introduces the nested transaction model. Indeed, implementing lock inheritance is difficult and expensive, even in a centralized system [GR93, Lis88, DGV94]. The fact that distribution comes into play makes it even more complicated. Practical evidence for this fact can be seen in *Encina* [Enc] [EMS91], the only existing product using nested transactions: upon configuring a server, one has to choose a mapping mode that determines how different subtransactions are mapped to the underlying database transactions, thereby determining whether lock inheritance can be simulated or not. Indeed, there are essentially two policies:

1. Two subtransactions of a common parent transaction are mapped to *different* database transactions.
2. Two subtransactions of a common parent transaction are mapped to *the same* underlying database transaction.

In the first case conflicting subtransactions will block each other, which is the equivalent of no lock inheritance. In the second case, there is no isolation among parallel subtransactions. Furthermore, this mapping is implemented as a setup choice and cannot be changed dynamically based on the client's needs.

As to adoption in recent standards, we should mention that closed nested transactions are optional in *OTS: CORBA Object Transaction Service* [Spe97a], and have not yet been adopted by *Enterprise Java Beans (TM)* [EJB, DP00]. However, apart from products derived from *Encina* [Enc], no commercially available implementations exist.

2.4.3. Open nested transactions

Open nested transactions differ from the closed variant in that the locks of a subtransaction are released as soon as that subtransaction is finished (either released entirely

or replaced by a semantic lock – depending on the variant). If locks are released entirely [GR93], then there is hardly any guarantee about isolation nor about atomicity of the global transaction. When openness is introduced, practical systems (based on a commercial *RDBMS*) will have to use *compensating* tasks [KLS90]. These are tasks that reverse the effects of a given task, after that task has released its locks and has committed. This is necessary because in current databases the only way to instruct a *RDBMS* to release locks is by committing the transaction. In order to make a compensation correct (so that it really reverses all updates correctly), certain restrictions must be imposed. Therefore, in most cases, some kind of higher-level semantic lock has to be maintained until compensation may no longer happen (when the transaction has been terminated at all sites). As a simple example, consider the following: a bank has an open nested system in charge of executing transfers between different accounts. Suppose that a general policy rule states that no account should be allowed to have a negative balance. Transferring money from one account (A) of bank $Bank_A$ to a different and empty account (B , in another bank $Bank_B$) consists of two steps:

1. The amount Am to be transferred is added to account B . This is implemented as an open subtransaction of the transfer operation, and is immediately committed in $Bank_B$. In this way, the new balance is exposed to concurrent activities.
2. Next, the same amount is taken from account A . However, due to a communication failure, this step fails.
3. To cancel the entire operation, the amount Am is withdrawn again from account B .

In isolated circumstances this system works fine, but not if different and concurrent activities are going on. Indeed, it is easy to see that if the owner of account B withdraws money between steps 1 and 3, the final balance of his account might be negative. Therefore, in this case, a lock should prevent any such withdrawals as long as step 3 may still be necessary.

To date, no existing implementation of open nested transactions is readily available. Most work seems to have been done in prototypes as part of research activities [Wei91, ELWR82, MRW⁺93] and in simulations [Kum96]. Furthermore, the composite system requirement of node autonomy and independence introduces new issues and restrictions, which to the knowledge of the author have not been addressed in this kind of research. For instance, as we will show later, autonomy implies the freedom to abort, and this severely restricts the applicability of open models.

2.4.4. Multilevel transactions

This is a variant of open nested transactions, where the transaction structures are perfectly balanced trees, all of the same depth [Wei91, SWS91, GS90]. This allows the execution to be decomposed into *layers* or levels. The prime emphasis was not so much

on distribution and autonomy, but on elegance and composition of primitive operations into more complex ones.

As noted in [GR93], the principles of multilevel transactions can be stated in three rules:

1. abstraction hierarchy: a hierarchy of objects exists, along with their operations.
2. layered abstraction: objects of layer N are completely implemented by using operations of layer $N - 1$.
3. discipline: there are no shortcuts from layer N to layers lower than $N - 1$.

Just like in open nested transactions, multilevel transactions rely on the existence of a compensation for each operation on any layer. Moreover, the compensations on layer $N - 1$ are scheduled by layer N or higher, which introduces a recovery dependency across layers.

These facts underline the dependence on a central system, or on a clearly structured and ‘trusted’ (in the sense of a reliable client layer) federation rather than autonomous and arbitrary distribution. For instance, [GR93, BSW88] mention the possible advantage of multilevel concepts for performing page-level operations and other low-level optimizations in a *RDBMS*. Although originally proposed as a model for federated databases as well [GS90, Sch96], the layered approach and the recovery dependency make this paradigm less favorable for the more general case of composite systems.

2.5. Existing industrial efforts for Internet transaction processing

Before we move on, a brief overview of existing industrial activities with respect to transaction processing on widely distributed systems is in order. Doing so will place this work into the proper perspective and make the practical contribution clearer to the reader.

2.5.1. Enterprise Java Beans (TM)

This is the Java vision for distributed transaction processing applications. *Enterprise Java Beans (TM)* [EJB, DP00] is a standard, meaning that it consists of specifications rather than implementations. The main objective is to provide a portable way of writing transactional applications. By taking all server-specific issues out of the application (such as transaction management, pooling of resources, swapping of inactive components), it is possible to create portable applications (so-called Beans, the Java terminology for a software component). The key idea is that all these components have to adhere to a standardized way of interacting with the server environment. In practice, this means that a component has a set of predefined methods that are called by the server in case of important events. For instance, before swapping out an inactive component, this

component is notified by calling its method *ejbPassivate()*, whose implementation should discard any volatile data and synchronize the component's database state. The whole concept of this technology is thus oriented towards component-based server applications, and the contract between a component and the server can be very complex. As such, it is orthogonal to our objectives here: although EJB mainly targets applications with transactional aspects, the issue of transaction management itself is left open. Some of the problems with distributed transactions are recognized, but no attempts are made to solve them. Finally, nested transactions are not currently supported.

2.5.2. OTS: CORBA Object Transaction Service

As part of the global CORBA standard, the OTS [Spe97a, Gor00, BK99] specification deals with transactions in CORBA environments. The objective is to standardize the way in which distributed objects can interact with a transaction manager, and how different transaction managers can communicate with each other. However, it is not specified how transaction management can be done. Rather, the interfaces between application and transaction manager, between transaction manager and resources (databases) and between different transaction managers are the main scope of this standard. Nested transactions are optional, and the interfaces exist. However, the internal aspect of how this transaction management should (could) be done is left open. We are aware of only one ORB that incorporates nested transactions: Orbix' OTM [OTM, Gor00], whose functionality is based on *Encina* [Enc].

2.5.3. Transaction Internet Protocol (TIP)

The *transactional internet protocol (TIP)* [LEK99] is another industrial standardization effort dealing with standardizing *two-phase commit* over TCP/IP networks. As such it specifies how different *transaction monitor* instances could co-ordinate a transaction's *two-phase commit* outcome by using a character stream connection. The effort is implicitly oriented towards flat transactions – which is reflected in the specification's protocols and interfaces – and, in most cases, towards point-to-point interactions rather than multiple accesses through different network paths. As such, it is not sufficient for the type of composite systems we have in mind.

3. Theory of composite systems

This chapter presents the theoretical basis for composite systems. We will first establish the necessary definitions, and then introduce a correctness criterion for systems without failures (where all transactions have terminated). This is an ideal situation, and serves as the basis for the theory that incorporates failures and active transactions, which is presented in the last sections of this chapter.

3.1. Introduction

In order to discuss correct composite transactions we will now present a theoretical framework for reasoning about composite transactional executions. Most of the discussion in this chapter is based on the theory in [BBG89], where the theoretical correctness of nested transactions was greatly clarified.¹ This chapter adds to this previous work issues related to distribution and recovery, whereas the next chapter discusses implications for practical protocols. The resulting theoretical framework could be called a theory of *Transactional Remote Method Invocations (TRMI)*. Most of the classical literature on distributed transaction management and its restrictions can be explained within this framework [ABFS97, AFPS99a, AFPS99b]. Initial efforts in this area have been towards special architectural configurations such as a stack [ABFS97] or a fork and a join case [AFPS99b, Fes00]. This series of investigations would be incomplete without a theory of fully distributed composite systems in arbitrary configurations. This is exactly what is accomplished in this work.

From a historical perspective, most of this theory was developed in order to answer an interesting basic question: “what is the minimum information that needs to be passed among schedulers in order to have correct distributed nested transactions using only local knowledge?”. A basic assumption is that schedulers should be as independent as possible (autonomy) and that the resulting protocols should only pass a minimum of information, favoring interoperability between heterogeneous systems by requiring only a minimal standard context to be interchanged. Additionally, each scheduler should be able to make its own decisions about whether or not to allow an operation, based on local rules and locally available knowledge only.

The resulting theory not only allows us to reason about correctness of composite transactions in general, it also allows the identification of possible pitfalls in existing

¹Other approaches for correctness were proposed in [LMWF94, KS88], but they are less intuitive and very formal, hence difficult to understand and use.

systems, as well as a clear definition of useful and complete interfaces for distributed nested transactions.

3.2. Basic assumptions

The development of this theory and the resulting protocols were guided by the following assumptions. The justification for these assumptions is our target application domain of Internet services.

1. A transaction executes within a thread [OW99] in a component (for simplicity, we assume that a thread is associated with at most one transaction at a time).
2. Remote calls originating from a given thread become subtransactions of the issuing transaction.
3. Autonomy: each component in the system should be autonomous. This implies the liberty to release resources by aborting a local transaction on local timeout. Also, servers can disappear or be unreachable, and this should only have a minimal effect on the rest of the system.
4. Decentralization: central components should be avoided.
5. Locality: each component should function (make decisions) about which transactional operations are allowed by using locally available knowledge. In practice, this means that conflict information is not known for outgoing remote calls, and locking is based only on local information.
6. Minimal context propagation: transactional context needs to be passed between servers, but it should be kept to a minimum in order to make the servers as independent as possible. As seen in the previous chapter, passing no significant context leads to diamond problem cases. On the other hand, passing all possible context information leads to unnecessary overhead and can severely limit the applicability of the system (more context means less independence among servers). Therefore the goal is to exchange only as much context as needed.
7. Arbitrary invocation structures: transaction trees can become arbitrarily complex, as components invoke each other throughout the system. There is no single entity in the system that has global knowledge about the invocation structure, and each component only knows about the transactions in which it is involved.

Basically we still pursue ACID properties but with a nested transaction model for remote calls. In order to achieve overall atomicity a *two-phase commit* protocol will be used.

In the following, we will generally distinguish between specifications and invocations. A specification is statically defined. An invocation is the actual execution of a specification. In classical computing terminology one could think of the specification as the

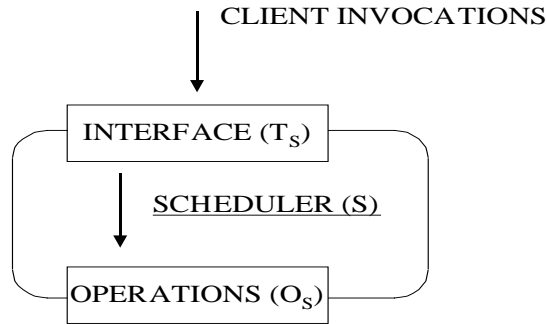


Figure 3.1.: Our model of a scheduler

program, the invocation being the process. In *Java* terminology a specification is the source code for a remote object and an invocation is the execution of an incoming RMI call [Sri97] on the specified object.

The operations executed within the scope of an invocation are assumed to be subject to a *scheduler*, responsible for synchronizing accesses to data items or operations. Our interpretation of a scheduler is shown in Figure 3.1. Clients can interact with a scheduler by invoking the *interface* layer, resulting in a set of *invocations* T_S . The interface is how the services of a composite server are exported to the outside world. These services are implemented in terms of *operations* (O_S) hidden to the outside (encapsulation). These operations can be local reads and writes, as well as *remote calls* to the interface layers of other schedulers. For the purposes of composite theory, we assume that a scheduler not only acts on the operations layer (O_S) but also on the interface layer (T_S) whenever this is necessary. Thus our scheduler models the combined effect of the local database scheduler and composite system-specific advanced scheduling. Because the scheduler is practically the only relevant part with respect to this theory, we will usually omit all other elements such as database or network protocol. Consequently, a server in a composite system will simply be represented by a rounded box similar to Figure 3.1. As will become clear, our notion of a scheduler includes much more than the classical notion in [BHG87].

A scheduler can order *local* operations if they conflict, or if an order is required by the specification. Additionally, a scheduler can also order *remote calls*, if the specification requires such an order. In certain cases, it may be necessary to have a schedule acting at the *invocation* level as well, thereby delaying entire invocations until others have finished.

In principle, a given scheduler is assumed to be aware of *every* incoming invocation or local operation that it executes. This is because when two *different* schedulers manage the same local data, and hence are unaware of each other's operations, anomalies are bound to arise since certain pairs of conflicting operations might go undetected.

3.3. Defining composite systems

We will model a composite system as a set of recoverable schedulers, each server being a scheduler, and the invocations between them will be called *transactional remote method invocations*. This terminology is not entirely arbitrary given the popularity of technologies such as *Java* and *CORBA* [Spe97b] that essentially provide remote method technology as the basis of distributed computing. The major contribution of this work is the nested and decentralized transactional aspect of such invocations, and how to support it in practice. An asset is that the underlying theory and conclusions are not restricted to these environments only: they can be applied to any existing distributed transaction platform.

3.3.1. Local definitions

Invocations will be termed TRMIs. Each TRMI can have a return value, as a function of the return values of its operations. Operations can be reads, writes, or remote calls.

Definition 1 (Transactional Remote Method Invocation (TRMI))

A transactional remote method invocation t is a partial order, $<_t$, of operations with either an abort, a , or a commit operation, c , as the last operation. An operation can be a read, a write, or a remote call. Reads and writes are local operations. The partial order $<_t$ expresses a temporal order between operations: if two operations are ordered $o_1 <_t o_2$ then o_1 returns before o_2 starts. A TRMI can specify a return value, as a function of the return values of its operations. In case of abort, the return value indicates this fact. The set of executed operations of t will be denoted O_t . \square

A TRMI is the execution of an *incoming* call. The term *invocation* as short for TRMI will frequently be used. In our examples, we will generally omit the commit or abort for simplicity. In the first part of this chapter, focus will be on committed TRMIs only. The second part deals with aborts, and unless otherwise mentioned, every example is assumed to be committed. Note that, although we use a nested model, commit denotes the *root commit*, and not the *relative commitment of a child with respect to its parent*, as in the nested model proposed in [Mos85]. Relatively committed children will be called *locally finished*, *preliminary committed* or in the *waiting state* in the rest of this work. We do not assume that locally finished subtransactions are durable before commitment of the root, which simplifies logging and recovery and keeps the overhead to a minimum. Indeed, simple distributed atomic commitment and logging will suffice, whereas otherwise one would need more complicated schemes as shown in [Mos87].

Definition 2 (Transactional Remote Method Specification (TRMS)) A Transactional Remote Method Specification (TRMS) is a set of TRMI. \square

A TRMS represents a service specification in a composite system; each of its executions will be a particular TRMI and hence we define a TRMS as a set of possible executions. Although the definition is an exhaustive one, in practice a TRMS is a program and can be written in any regular programming language. In *Java* for instance, one would typically use *JDBC API* [JDB, WFC⁺99] and *Java Transaction API* [JTA] interfaces for specifying database access and transaction boundaries respectively. In general, a TRMS can have many valid executions, a particular TRMI being only one of them.

Definition 3 (Local projection of a TRMI) The local projection t_S of a TRMI t is obtained by leaving out any remote calls, the return, as well as any orders involving such remote calls. \square

A local projection consists of only reads and writes and their relative orders. Since a $t \in T_S$ for some S , the local projection t_S is relative to S .

Definition 4 (Schedule) A schedule at S is a seven-tuple $(T_S, O_S, LO_S, <, <_{t_S}, CON_S, COMMUTE_S)$ where:

1. T_S is a set of TRMI, with O_S denoting the set of all operations in T_S , and $LO_S \subset O_S$ the local operations. $<_{t_S}$ is the set of all $<_t$ -pairs in T_S .
2. CON_S is a conflict predicate over $LO_S \times LO_S$.
3. $COMMUTE_S$ is a commutativity predicate over $T_S \times T_S \cup LO_S \times LO_S$.
4. $<$ is the output order, a partial order over O_S , such that:

- $\forall t \in T_S, \forall o, o' \in O_t : (o <_{t_S} o') \Rightarrow (o < o')$.
- $\forall o, o' \in O_S : CON_S(o, o') \Rightarrow (o < o') \vee (o' < o)$.

$<$ will usually be interpreted in a transitive way. \square

Abusing notation, we will often interchange the *scheduler* S and the *schedule* at S , as well as represent $<_{t_S}$ as $<_t$ to simplify the context whenever possible. According to the second point, CON_S represents the local knowledge about conflicts between operations. On the other hand, local operations can also commute, as expressed by the third point. However, two *invocations* can commute as well. Hence, $COMMUTE_S$ is defined not only over $LO_S \times LO_S$ but also over $T_S \times T_S$. In principle, conflict and commutativity information are complementary. It is customary to represent this information in the form of a conflict table, as done here with CON_S . However, as shown later, in composite executions the local conflict information is not sufficient for correctness. Rather, in our proof techniques

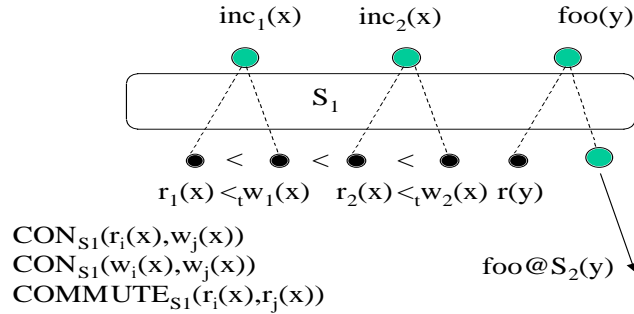
$$\text{COMMUTE}_{S_1}(\text{inc}_1(x), \text{inc}_2(x))$$


Figure 3.2.: Schedule: an example

conflicts will be propagated among schedules, *for discussing correctness*. This is done from children up to their parents, until some parent nodes are known to commute by COMMUTE_S . It is convenient to represent this information as a commutativity table, which is COMMUTE_S here. Due to the restrictions of local knowledge, CON_S and COMMUTE_S are *not* complementary. We will come back to this as we go along in our theory. The scheduler imposes a partial order on operations, and this order is called $<$. For now, the only requirement we impose on this order is that operations are ordered if they are ordered by $<_t$ in their TRMI, or if they happen to conflict in S . This condition reflects the classical notion of a scheduler [BHG87]. Note that no order is imposed on remote calls (CON_S is purely local), except for any given order $<_t$. This reflects the principle of local knowledge and autonomy: a scheduler should not be expected to know about remote conflicts, nor should it try to prevent them from occurring.

An example of a schedule is given by S_1 in Figure 3.2.

There, $T_{S_1} = \{\text{inc}_1(x), \text{inc}_2(x), \text{foo}(y)\}$, $O_{S_1} = \{r_1(x), w_1(x), r_2(x), w_2(x), r(y), \text{foo}@S_2(y)\}$. Finally, $LO_{S_1} = O_{S_1} \setminus \{\text{foo}@S_2(y)\}$, and $\text{CON}_{S_1}, \text{COMMUTE}_{S_1}$ are as shown. Especially note that the two increment *invocations* are known to commute in the example, illustrating that COMMUTE_S also ranges over $T_S \times T_S$.

In general, one TRMI can lead to another being invoked within the scope of its execution. This means that an operation $o \in O_S$ can actually be an invocation $o \in T_{S^*}$, $S^* \neq S$. This is how the tree structure of composite invocations arises. The topmost TRMI (the one without parent) will be called the *root*. Slightly abusing classical tree terminology, descendants of the same root that are not themselves related by a descendant relationship are called *siblings* in the rest of this work.

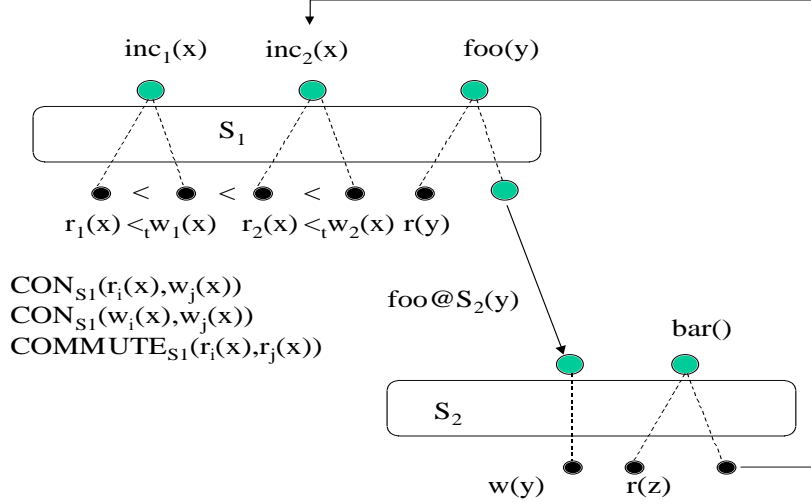


Figure 3.3.: Composite system: an example

3.3.2. Composite definitions

In the following, $<$ as generated by some schedule S will be called $<_S$ to avoid confusion. We can informally define a composite system as a collection of schedules, where each schedule can invoke another for the execution of one of its operations. This way, a remote call is an invocation at some other schedule. The only restriction that is imposed is that a descendant cannot execute on the same schedule as one of its ancestors, a case which will be often referred to as *recursion*.

An example of a composite system is shown in Figure 3.3. Only the most relevant information is shown, in order not to overload the picture. Note that although recursion within one root is not allowed, recursive client-server relationships *are allowed*: S_1 acts as as client to S_2 and vice versa – just not within one composite transaction.

Definition 5 (Composite execution (CE)) A composite execution *consists of a set of nodes, each node organized into a tree structure, with a partial order $<_t$ on the children of each node t , and a partial observed order $<_e$. Formally, composite execution is a seven-tuple $(T, L, N, CON, COMMUTE, <_e, <_t)$ such that:*

1. T is the set of non-leaf nodes,
2. L is the set of leaves,
3. N is the set of all nodes, $N = L \cup T$,
4. $COMMUTE$ is a commutativity relation over N ,
5. CON consists of triples of the form $CON(o_1, o_2, p)$, where $o_1, o_2 \in L$ and $p \in T$,

6. $<_e$ is a partial observed order defined on $N \times N$,
7. $<_t$ is a partial order defined on the children of some node $t \in T$.

Additionally, *COMMUTE* and *CON* are complementary on leaves: $\forall(o_1, o_2) \in L \times L : \text{CON}(o_1, o_2, p) \text{ (for some } p \in T) \Leftrightarrow \neg \text{COMMUTE}(o_1, o_2)$. \square

The root of each tree is called the *root transaction* or root for short. A root and its tree are referred to as a composite transaction (CT).

The reason why *CON* is a triple relation will be clarified in the next section. Here, it suffices to say that conflicts are conditional, and can disappear under certain conditions, namely if certain parents are *reduced*, as explained in the next sections. There, we will simplify a composite execution by constructing an equivalent execution in which less nodes appear. Equivalent means that all remaining nodes have the same return values. Nodes will be abstracted out in a bottom-up way. In this, $\text{CON}(o, o', t)$ means that o, o' conflict, and this conflict should be remembered *as long as t , some ancestor, has operations left in the composite execution*. In our notation, $\text{CON}(o, o', \phi)$ means that no ancestor applies, $\text{CON}(o, o')$ for short will denote either $\text{CON}(o, o', \phi)$ or $\text{CON}(o, o', t)$.

In general, $<_e$ is a partial order defined on pairs of nodes. We will assume that $t_1 <_e t_2$ denotes a composite execution where $<_e$ applies to *all descendants of t_1, t_2* as well. In particular, it means that all descendants of t_1 are $<_e$ ordered before all descendants of t_2 , and $<_e$ is total among the children of each node separately, in a recursive way. In fact, this means the *serial execution of t_1, t_2* .

Definition 6 (Serial composite execution)

A composite execution is serial if $<_e$ is total over the roots. \square

This is expressed in a similar way as in [BBG89].

Definition 7 (Equivalence of composite executions)

Two composite executions are equivalent if they have the same set of committed roots, and the same return values for each committed root, as well as for all possible future invocations. \square

Composite executions arise as the combined effect of different schedules that interact in a distributed system. Therefore, it needs to be specified how a composite system and a composite execution are related. This is done in the following definition.

Definition 8 (Composite execution for a composite system) A composite execution for a composite system CS is a seven-tuple $(T, L, N, \text{CON}, \text{COMMUTE}, <_e, <_t)$ such that, for all S_i in CS:

1. $T = \bigcup T_{S_i}$
2. $L = \bigcup LO_{S_i}$
3. $N = L \cup T$

4. *COMMUTE* is as follows:

- $t_1, t_2 \in T$: $COMMUTE(t_1, t_2)$ iff $COMMUTE_{S_i}(t_1, t_2)$ for some S_i
- $o_1, o_2 \in L$: $COMMUTE(o_1, o_2)$ iff $\exists S_i : CON_{S_i}(o_1, o_2)$.

5. *CON* consists of triples, where $CON(o_1, o_2, \phi)$ if $CON_{S_i}(o_1, o_2)$ on some S_i

6. for $o, o' \in L$: $o <_e o'$ iff $o <_{S_i} o'$

7. $<_t = \bigcup <_{tS_i}$

The tree structure arises from the fact that operations can be invocations at another schedule. \square

A composite execution reflects all the relevant information of a composite system: what operations and invocations were done in what order, and what semantics are known. Although irrelevant for each scheduler individually, it is of significant importance for defining correctness. Important to note is that *conflicting leaves* are always related by $<_e$, because by definition of a schedule they are ordered by $<$ and by point 6 above they are in $<_e$.

At this point, it is possible to elaborate on the CON_S and $COMMUTE_S$ relationships. They reflect the locality restriction of each S , and therefore only range over certain nodes in the corresponding composite execution. More precisely, we can formulate the following definitions.

Definition 9 (Commutativity of invocations in S)

Two invocations $r, s \in T_S$ commute according to S, $COMMUTE_S(r, s)$, if $r <_e s$ and $s <_e r$ have the same return values, both for r, s and for any other or future $t \in T_{S^*}$, where S^* may be different from S . \square

Because S does *not* know about any remote semantics, a practical implication of this definition is the following: $t_1, t_2 \in T_S$: $COMMUTE_S(t_1, t_2)$ can only be determined at S in the following cases:

1. both are purely local, in which case S knows their complete semantics
2. one is purely local and the other one executes only remote calls, in which case S is sure that they act on different data (except with recursion, which we will disallow).

Definition 10 (Conflicting operations in S: CON_S)

Two local operations $o_1, o_2 \in LO_S$ conflict in S, $CON_S(o_1, o_2)$, if they access the same data item and at least one of them is a write. \square

Again, this definition is local with respect to S , since it only considers *local* operations and no remote calls.

Definition 11 (Commutativity of local operations in S)

Two local operations $o_1, o_2 \in LO_S$ commute in S, $COMMUTE_S(o_1, o_2)$, iff they do not conflict according to CON_S . \square

Note that in the corresponding overall composite executions, *CON* and *COMMUTE* are *complementary on leaves*, as required by the definition of a composite execution.

3.4. Correctness without failures

First we will concentrate on the ideal and static case where no failures occur, and where all executions have terminated. So, we will look at a completed composite execution that did not suffer from any kind of failure. Although not very practical as a model, this will be used as a basis for the dynamic theory that incorporates failures.

3.4.1. Structure of a composite execution

The interactions in a composite system can be quite complex, and for the purpose of discussing correctness we will need to take these effects into account. The following definitions specify how to proceed.

Definition 12 (Propagation of $<_e$) *The observed order in a composite execution CE is propagated as follows: for any invocation t and an operation $o \notin O_t$ such that $\exists o' \in O_t \wedge CON(o, o')$: $t <_e o$ if $o' <_e o$, and $o <_e t$ if $o <_e o'$. \square*

This is an extension of the classical serialization order [BHG87]. An observed order can arise between an operation and a local transaction, something not classically done in case of serialization orders [BHG87]. Although we defined the observed order on a composite execution, we will regularly use the same concept for individual schedules. This should not be a problem, since a single schedule can be seen as a special case of a composite execution. An example of propagated observed orders can be given based on Figure 3.4. Starting at the bottom scheduler S_4 , we see that T_{11} and T_{21} execute a conflicting pair of operations a and b . The observed order for these operations is therefore the same as their order shown, according to rule 6 of Definition 8. So we have $a <_e b$. According to propagation, we have $a <_e T_{21}$ and $T_{11} <_e b$, respectively.

An important notion in concurrent environments is the concept of isolated set of operations:

Definition 13 (Isolated set) *In a composite execution CE , a set of operations O_t on behalf of some invocation t is isolated in CE if every other invocation's operation is $<_e$ ordered either before or after all operations of O_t . \square*

For the sake of correctness, we will try to simplify a composite execution as explained in this section, by starting with the existing execution and transforming it gradually by introducing abstractions. The highest abstraction we can reach is a composite execution including only the roots. The main way of simplifying a composite execution is by constructing a computation and reducing it.

Definition 14 (Computation) *A computation (of a TRMS) in a composite execution CE is a TRMI t with only leaf operations, such that its set of operations O_t is an isolated set in CE . Furthermore, $<_e$ should not contradict any $<_t$ specified in the TRMI. \square*

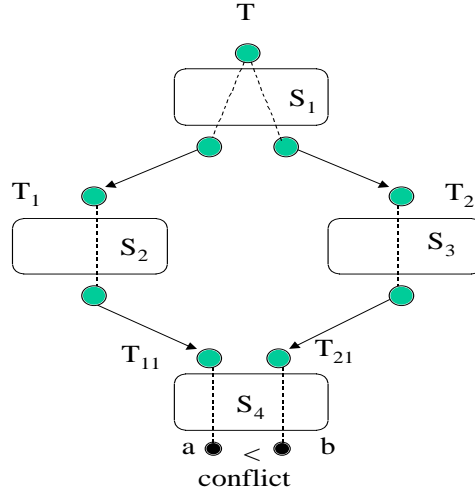


Figure 3.4.: Observed order: an example

Following [BBG89], a general way to find a computation in a given composite execution is by reversing the order $<_e$ for commuting pairs of operations. Note that commuting pairs are given by *COMMUTE* in Definition 8. A computation is a TRMI that happens to be ‘correct’ since no interfering concurrent operations exist. It may be impossible to construct a computation because, for instance, a pair of $<_e$ -related operations does not commute, and $<_e$ contradicts $<_t$.

If a computation exists, then the invocation that it represents has been executed correctly (without interference of concurrency). A computation indeed implies that there exists a composite execution, indistinguishable from the original, in which the TRMI is executed in isolation and therefore is correct. Note that a computation is only defined on *leaves*. This means that we will not work with arbitrary nodes, but only with those whose operations are leaves (either because they are local operations or because they have been reduced).

Definition 15 (Reduction) *A reduction is an abstraction step in which a computation (in a composite execution CE) is replaced by one single node n , symbolizing the abstract execution of the corresponding TRMI. This way, a simpler execution CE^* is constructed. The new execution CE^* should contain the following $<_e$ and *CON*, *COMMUTE* relations involving the reduced node n :*

1. all $CON(o, o', n)$ are discarded,
2. if $\exists o \notin O_n, o' \in O_n, CON(o, o')$:
 - if $CON(o, o', \phi)$: if for p , parent of o , $COMMUTE(p, n)$ then $CON(o, n, p)$, and $CON(o, n, \phi)$ otherwise.

- if $CON(o, o', p) \wedge n \neq p$ then $CON(o, n, p)$
3. if $\exists o$, a leaf operation, and $\neg CON(n, o)$ (according to the previous rules) then $COMMUTE(n, o)$.
 4. any $<_e$ orders that follow from Definition 12.

These rules are to be applied in this order. Furthermore, n replaces O_n in the set L of CE (it becomes a leaf), and any CON involving any $o' \in O_n$ are forgotten. \square

The reduction process constructs a simpler composite execution while keeping the relevant information concerning interference by means of CON and $<_e$. It also implies that conflicting leaves are always related by $<_e$, by points 2,4 and Definition 12. In addition, for *leaves*, the CON and $COMMUTE$ relationships are complementary by the third rule, and the fact that they are complementary before reducing. Note that an open nested model is supported, because conflicts can disappear if the reduced node and an existing node are known to commute at some scheduler S , as in point 1. On the other hand, new conflicts may have to be introduced in the reduction step, according to point 2, because of the incompleteness of CON_S and $COMMUTE_S$ on each schedule. This results in conflicts whenever two nodes in the reduced execution have had an interaction, and no commutativity is explicitly known at the current stage of reduction. Furthermore, if the reduced node and any existing *leaf operation* (i.e., all local operations and already reduced nodes) do not conflict, then they must commute according to point 3. This is because if they did not have *any* conflicting interaction anywhere, then surely their relative order of execution does not matter. This allows to gradually expand the set of $COMMUTE$ in Definition 8. So, this step allows us to find more computations because it expands the set of commutes.

The motivation for using ‘conditional’ conflicts, that can disappear if a parent is reduced, is as follows: the process of transforming a composite execution into equivalent ones, via computations and reductions, is based on abstraction, and switching around *leaves*, i.e., parts of the execution that have been shown to have executed *as in isolation*, and are therefore correct in themselves. In order to be able to construct computations that preserve all return values of their operations, we can only switch commuting pairs of operations. As such, a *conflicting* interaction on the same data on a lower level for some parents t, u means that these t, u , *after having been proven ‘correct’ in themselves*, should not be switched. Otherwise, the same return values can no longer be guaranteed. However, *if t, u can eventually be shown ‘correct’ by reduction*, and they are *known to commute*, then the lower-level interaction can be discarded and the conflict should no longer be propagated.

Definition 16 (Transformation)

Given a composite execution CE , a transformation of CE is another composite execution, obtained from CE by repeatedly reversing commuting pairs of $<_e$ ordered leaves, and reducing computations that arise this way. \square

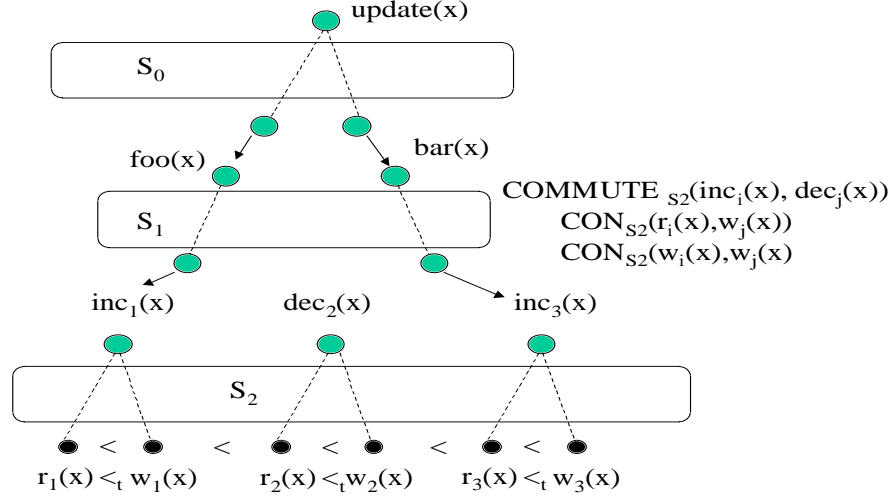
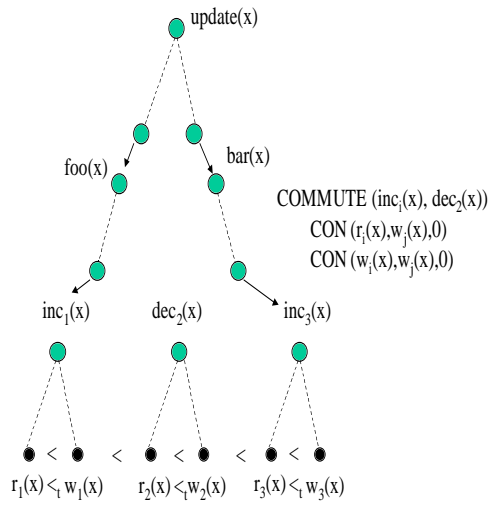


Figure 3.5.: Reduction: example case

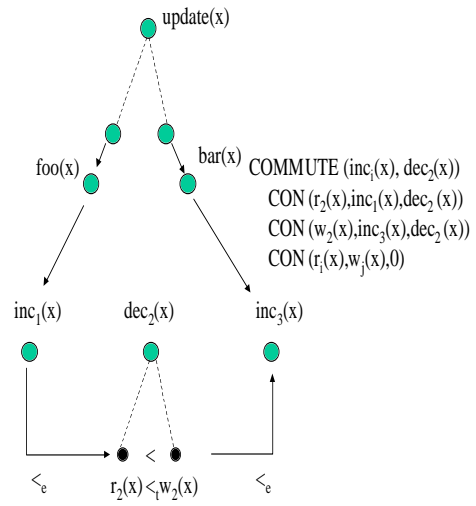
An example of such a transformation process for Figure 3.5 is shown in Figure 3.6. In step 1, the composite execution is shown. Note the conflicts. Step 2 reduces both *inc*, since they have a computation as shown. Important are the new conflicts $CON(r_2(x), inc_1(x), dec_2(x))$ and $CON(w_2(x), inc_3(x), dec_2(x))$, as well as the associated $<_e$ orders. The meaning is the following: both *inc* have the equivalent isolated execution shown in this step, and can therefore be considered correct. However, there has been a low-level interaction $<_e$ whose order matters as indicated by the conflict. The implication is that for constructing higher-level computations, we can not switch such operations, since they do not commute. However, as soon as $dec_2(x)$ can be shown correct as well, meaning that reduction is possible, this $<_e$ becomes irrelevant: the interaction between *inc* and *dec* commutes. After reduction of *foo* and *bar*, we have $CON(r_2(x), foo(x), dec_2(x))$ and $CON(w_2(x), bar(x), dec_2(x))$ in a similar way, shown in step 3. Here it becomes clear why we need conflicts as triples: with the original *COMMUTE* information, stating that *inc* and *dec* commute, we can not capture the fact that the interactions between *foo*, *bar* and the operations of *dec* become irrelevant, since the order of reduction has been such that the *inc* nodes are no longer present. By including this information in a conflict triple for each conflict that is introduced, we can still take this fact into account, and in step 4 it is shown that *after successful reduction of dec*, there exists commutativity between *foo*, *dec* and *bar*, *dec*.

To conclude here, we mention the following Lemma:

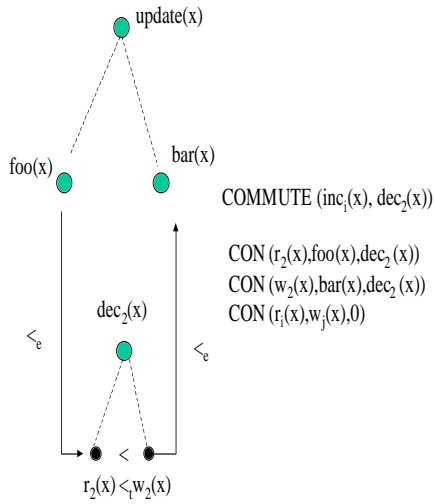
Lemma 1 (Isolation and $<_e$) Consider an invocation t and its set of reduced operations O_t . If it is impossible to isolate all operations for t , then this implies the existence of a $<_e$ sequence $o <_e o_1 <_e \dots <_e o_n <_e o'$, with $o, o' \in O_t$, $o_i \notin O_t$, and each $<_e$ pair



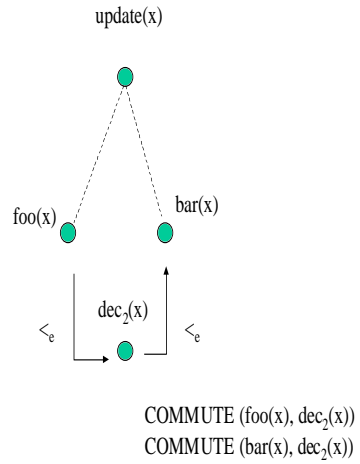
Step 1: composite execution



Step 2: reduction of inc



Step 3: reduction of foo, bar



Step 4: reduction of dec

Figure 3.6.: Reduction example

represents a conflict. □

Proof. Since by assumption t can not be isolated, there must be at least two operations $o, o' \in O_t$ that are separated by some other invocation's operations o_i . We will proceed with the proof in two steps:

1. Suppose no sequence exists where operations were *not all consecutively related* by $<_e$. Then we can only have $o <_e o_1 <_e \dots <_e o_k$ and $o_l <_e \dots <_e o_n <_e o'$, where the sets $\{o_1, \dots, o_k\}$ and $\{o_l, \dots, o_n\}$ are disjunct. In that case, given the fact that all conflicting operations are ordered by $<_e$, hence no $<_e$ between two operations implies no conflict, we can construct the following, equivalent execution: $o_l <_e \dots <_e o_n o o' o_1 <_e \dots <_e o_k$. Indeed, $o_l \dots o_n$ can be switched to *before* o , since none of them conflicts with o (no $<_e$ exists with o). The same holds for o' and $o_1 \dots o_k$, and we have successfully moved o, o' together, which contradicts our initial assumptions. Consequently, there exists a sequence such that all operations are consecutively ordered by $<_e$.
2. Let us now consider the *shortest* such sequence, seq . Here, we will show that in seq , all $<_e$ pairs must also be conflicting pairs. Indeed, suppose these were not all conflicting pairs. Then there would be at least one pair $o_i <_e o_j$ that does *not* represent a conflict, with seq of the form $o <_e o_1 <_e \dots <_e o_i <_e o_j <_e \dots <_e o_n <_e o'$. Consequently, o_j does *not* conflict with any of the set $\{o, o_1, \dots, o_i\}$ either, because if it did then there would also exist a $<_e$ between them and then seq would not be the shortest sequence in the first place. As a result, we can again isolate o, o' by constructing $o_j <_e \dots <_e o_n o o' o_1 <_e \dots <_e o_i$. Again, this contradicts the initial assumptions of this Lemma. □

Lemma 2 (Acyclic $<_e$) *If only computations are reduced, then no conflicting $<_e$ cycles exist among leaves.* □

Proof. By induction, on the number of reduced computations N . Suppose for some N , this is true. For $N = 0$ this is the case, because by definition $<_S$ is acyclic among each O_S , the initial set of leaves is $\bigcup LO_S$, and $<_S$ does not span different S . We will show that for $N + 1$ the same must hold. Let n be the node reduced in reduction $N + 1$, and suppose there *does exist a cycle C after reduction*, namely $a <_e n <_e b \dots <_e a$. Because n was reduced in reduction $N + 1$, there must exist a computation of n after N reductions only. Since the operations of n are the only ones that were replaced in the composite execution from reduction N to reduction $N + 1$, a must have been present *before* reduction of n . The existence of C implies that $\exists o, o' \in O_n$ such that two conflicting pairs $a <_e o$ and $o' <_e b$ exist *before reduction of n* . But this means that there was a *conflicting series* $o' <_e b <_e \dots <_e a <_e o$ separating o, o' and contradicts that n had a computation. □

Although $<_e$ was never shown to be a real order in the sense that it is acyclic, according to this Lemma any cycle that arises during the reduction process can be immediately eliminated by reversing the $<_e$ order of at least one pair of commuting operations. This is possible, because the cycle can not involve only conflicting pairs. This way, the ‘well-formedness’ of a CE can always be guaranteed.

3.4.2. Correctness of failure-free composite systems

The definition of correctness is similar to the one in [BBG89]:

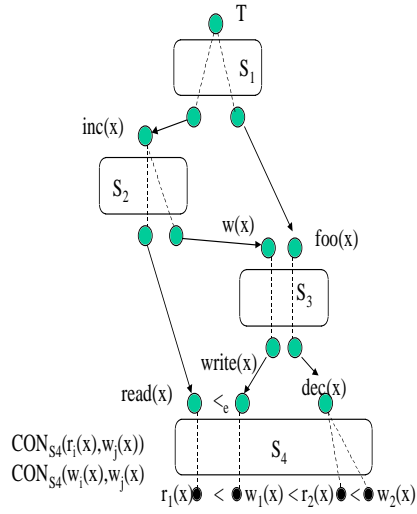
Definition 17 (Correctness of composite executions (CTRMI)) *A composite execution CE is correct (CTRMI) if it is equivalent to a serial composite execution CE^* containing only the committed roots.* \square

This equivalent serial CE^* , if any, can be found as a transformation of the original execution CE . Note that this implies correctness of all intermediate nodes as well, since each of them had to be reduced in the process. What is the influence of our basic assumptions on the reduction process for correctness? First of all, the semantics are more restrictive than in a centralized case, where exact knowledge about the entire execution (more precisely about *commutativity*) is given. This is illustrated in Figure 3.7, showing the reduction of a composite execution until an increment $inc(x)$ and a decrement $dec(x)$ are found to conflict. In classical theory with higher-level semantics, this would not be the case. However, in the example shown, there is no scheduler that knows this information. The point of Figure 3.7 is that, during the construction of computations, there is less freedom to switch pairs of operations (because less pairs will be known to commute), for which reason more executions will be labelled *incorrect* whereas maybe they could be called correct from an absolute viewpoint. However, for proving our protocols on arbitrary executions we can only use the knowledge that is available at some schedulers, and this is less than in the absolute, centralized case. Thus, composite theory allows only a subset of the executions that would classically be allowed (i.e., in case one had ‘absolute’ knowledge about everything in the system).

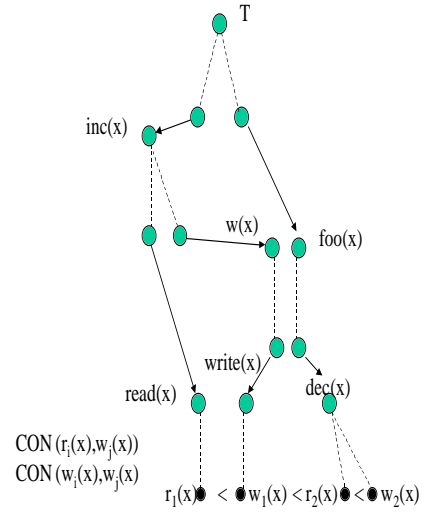
Another interesting aspect of composite systems is that, because arbitrary invocation structures can arise, including recursive structures, it may not be clear how to systematically proceed with reduction and whether this process ends at all. The following theorem solves that question.

Theorem 1 (Finite Reduction Property) *Let CE be a terminated composite execution for which $N \geq 0$ calls have been reduced so far. Then there exists at least one TRMI t , for which all operations are leaves.* \square

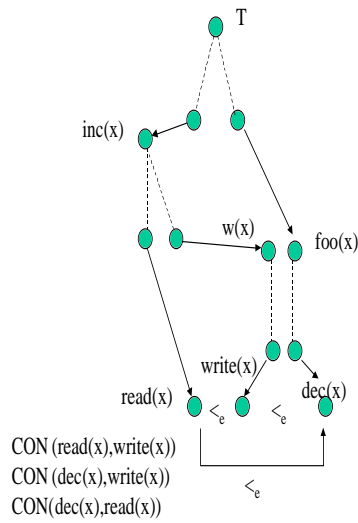
Proof. Let us consider the case $N = 0$ first. This is the original execution for which no reduction effort has taken place yet. We can view any root execution’s structure as a *tree* where each operation represents a node. The depth d of any such tree is finite, because otherwise the execution itself would never finish, which by assumption it does. Hence, starting at any root, and following a path of depth d down into its execution



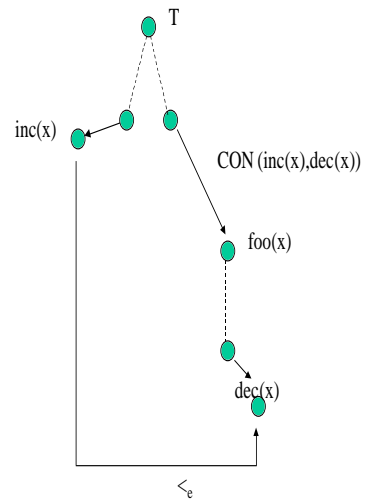
Step 1: composite system



Step 2: composite execution



Step 3: reduction



Step 4: no commute

Figure 3.7.: Restricted commutativity knowledge

tree, we will eventually reach a leaf operation, i.e., an operation which is not again a remote invocation. The *parent* of this operation is the t we are looking for. Indeed, that parent does not have any other operations that are not reduced yet, because we have taken the deepest path. For $N > 0$ the nature of the execution does not change: it is still a tree and we can use the same strategy to find t . However, the leaf operation can now also be a reduced call. \square

Corollary 1 *For any terminating composite execution, reduction is a finite process and it is always possible to find the next candidate for constructing a computation. Furthermore, if reduction to the roots fails, this can only be because it is impossible to construct a computation for some t .* \square

Proof. The general technique of reduction can thus proceed as follows: first, a candidate for a computation is to be found, among the invocations with only local operations, because a computation is defined for leaves. A candidate can always be found, by Theorem 1. If it is not possible to construct a computation, then the composite execution is incorrect. If, on the other hand, such a computation is found, then reduction is applied, and $CON, COMMUTE, <_e, L$ are adjusted to include any new pairs that might arise. The next candidate for trying to construct a computation can be found by Theorem 1, because $N = 1$. By repeatedly applying the same reasoning, one sees that Corollary 1 must hold. It should be pointed out here that a reduction up to the roots implies correctness, since there can not be a conflicting $<_e$ cycle among those roots. Therefore, in case of a cycle, reversing one commuting $<_e$ pair breaks it, and a simple topological sort along $<_e$ ensures a serial and equivalent root execution. This one $<_e$ pair of commuting operations must exist because it has been shown that a cycle can not involve only conflicting pairs. \square

3.5. Correctness in the presence of failures and recovery

The presence of failures complicates the theory in the sense that not every TRMI is completed as it should. We first formalize this as follows:

Definition 18 (Remote method invocation (RMI)) *A remote method invocation (RMI) is a partial execution of a TRMI, i.e., not all operations in the TRMI have been executed.* \square

Note that, as long as some transactions are active in the system, we will always have to deal with RMI rather than with TRMI. Therefore, this extension of the theory not only applies to failure and recovery, but also to so-called *dynamic scheduling*. This means that we will no longer have to restrict ourselves to terminated and static composite executions.

Definition 19 (Effect-free in CE) *An execution of a partially $<_e$ -ordered set of leaves $O \subset L$ on behalf of an invocation i is effect-free in CE if omitting O will not influence the return values of any remaining or future invocation t in T , which is not an ancestor nor a descendant of i . \square*

By omitting O from CE , we mean leaving out all operations of O , as well as omitting any $<_e, CON, COMMUTE$ relationships with operations/invocations in the rest of CE . Note that if t is an ancestor or a descendant of i , then influence on the return values is not important, since t will either have an alternative for the failed call, or will be aborted as well. For instance, consider a parent t that executes a remote call t_1 . Correct abort of t_1 implies that its effects disappear and leave the rest of the execution with the same return values *except for t* , which of course has a missing remote call. However, t is aborted as well, and hence this poses no problem in the eventual resulting execution. Alternatively, if t_1 aborted before executing all its operations and returning, then t may have seen the abort of its child as reflected by its return value and may have executed an alternative, thereby eliminating the need to abort itself.

The relevance of this definition is in abort processing: if one models the operations that are executed during abort as regular operations (for reasons of theoretical completeness), then correct abort requires that the ‘normal’ and abort operations together appear as effect-free. Similar approaches were taken in, for instance, [VHYBS98, Has96, AVA⁺94].

Definition 20 (Partial computation)

An RMI t is a partial computation (of its specification) in a composite execution CE , if all local operations executed thus far on behalf of t appear isolated, and in the order according to $<_t$ for t . \square

This definition is needed because the original computation is defined for complete executions only. As we will see, reasoning about correct abort requires the notion of a partial computation. Like normal computations, a partial computation is constructed by reversing the $<_e$ orders of commuting pairs of operations. A partial computation can also be reduced, and the partial reduction will be symbolized as t_S . It represents an isolated set of *local* operations of t . As for a normal reduction, we can also include $<_e, CON, COMMUTE$ pairs for a partial reduction.

3.5.1. Atomicity as seen locally in a schedule S

For simplicity, we will assume *no concurrent invocations and serial executions only* in this section. The main objective here is to introduce the concepts of *expansion* and *compensation*. In order to ensure atomicity, a scheduler will have to take action by executing additional operations or invocations. For instance, if a data item x has been written on behalf of an invocation r that can not finish, then abort of r implies that the writing of x is not exposed to any other invocation, as required by ACID-ity. Since here we assume no concurrency, this means the following: if an invocation can not finish,

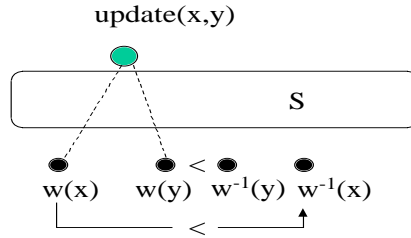


Figure 3.8.: Expansion

then its effects must not be visible to any *later* invocation. The scheduler ensures this by *expansion* [AVA⁺94], as follows:

Definition 21 (Expansion) *Let r be an active invocation in some S , and $O_r \subset LO_S$ the partially $<$ -ordered set of local operations of r that have been executed so far. The expansion of r in S is constructed by adding a partially $<$ ordered set O_r^{-1} , containing the inverse write w^{-1} of each write $w \in O_r$. Additionally, the inverse writes are $<$ ordered in the reverse order as their respective writes in O_r , and $w < w^{-1}$ for each pair of write and corresponding inverse write.* \square

An example is shown by Figure 3.8. Because we assume no concurrency here, applying expansion if an invocation needs to abort – and doing the same for all remote invocations that were triggered – implies that the result is effect-free in the composite execution CE, because there is no interference from concurrent invocations, nor from parallelism. Expansion is used by the scheduler for invocations that may have executed only partially, and need to be aborted. In addition, it may happen that an invocation that has executed *all* its *local* operations needs to be aborted, e.g., because one of its ancestors aborts. In that case, the scheduler can either use expansion as above, or use an alternative strategy: compensation. In this case, the programmer of the service has to provide a way of making the original invocation effect-free by supplying a compensation. The scheduler can invoke this compensation and thereby also proceed with the abort.

Definition 22 (Compensation) *A compensation in S , of an RMI $r \in T_S$ is another RMI $r' \in T_S$, with only local operations, such that replacing r by its local projection r_S , directly followed by r' is effect-free in the resulting overall execution CE.* \square

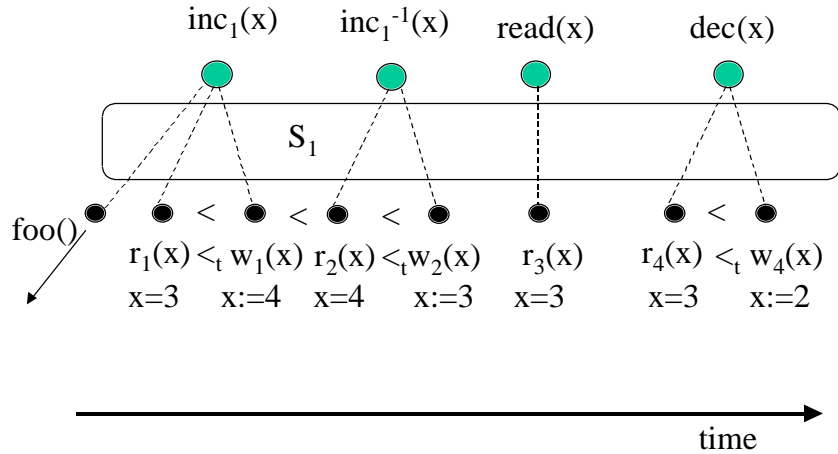


Figure 3.9.: Programmer-defined compensation

A compensation is somewhat abstract, because it considers only the local projection. A programmer can provide a compensation by ‘forgetting’ the remote calls of r and dealing with its local effects only. Of course, the execution CE is no longer equivalent to the original, but it only serves to define compensation, not to define correctness.

The example in Figures 3.9 and 3.10 clarifies this: there, $inc_1(x)$ has as its compensation $inc_1^{-1}(x)$, which is programmer-defined, and essentially decrements x again. Together, they are effect-free in Figure 3.10 (none of the later invocations sees their effect). The justification for leaving out remote calls is that in correct abort, these must eventually disappear anyway, as explained in the next sections.

The main practical uses of programmer-specified compensation are the wrapping of legacy systems, like databases without a *two-phase commit* interface, and early reuse of resources by preliminary releasing database level locks and database connection resources prior to *two-phase commit*, for instance, in open nested transaction models. However, programmer-specified compensation introduces the need for scheduling at the incoming invocation level. This is because a lower level scheduler, such as a database lock manager, only uses read and write semantics, which are typically too restrictive to be able to cope with this kind of compensation. Semantically rich locking has been used by many other projects before, such as in the open nested models mentioned before, and in other

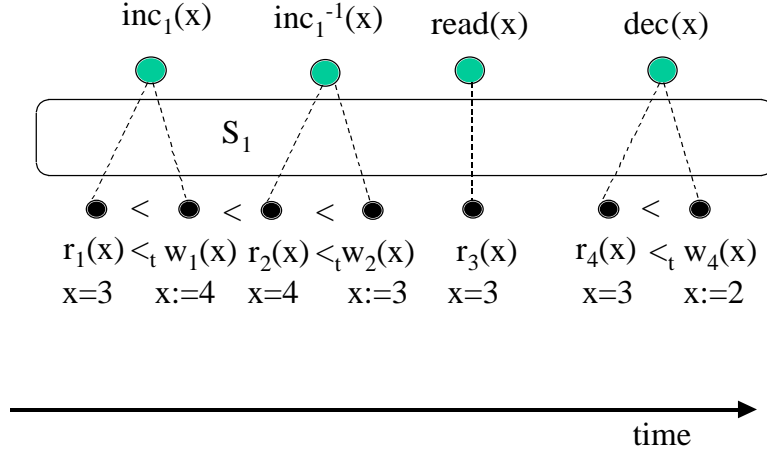


Figure 3.10.: Programmer-defined compensation: local projection

research, mostly in flat-transaction object-based systems [Wei88, Wei89].

A scheduler is responsible for ensuring that either expansion or compensation is always possible for any of its active invocations.

As a consequence of failures, what we need to consider here are executions that contain compensations and expansions as well as normal RMIs. For that purpose, we redefine our original notion of a schedule to incorporate these. For consistency of the theoretical model, we will represent compensation and expansion related operations as well as normal operations. More precisely: whenever an RMI r has a compensation in a composite execution, we will represent the compensation as another RMI r' (hence, we model compensation and recovery at the *interface layer* rather than at the operation layer). Whenever there is an expansion, we will represent the inverse writes explicitly in the schedule (and, hence, in the composite execution). The set of local operations O_S will be completed with the set of inverse operations, needed for expansion. Finally, CON_S and $COMMUTE_S$ relationships will be assumed to range over these additional operations and invocations as well; inverse writes are essentially write operations in themselves, and compensation is in essence again an invocation.

Definition 23 (Schedule) *A schedule S is a seven-tuple $(T_S, O_S, LO_S, <, <_t, CON_S, COMMUTE_S)$ where:*

1. T_S is the set of RMI (completed and active), as well as any compensation (in case of failed invocations)
2. O_S is the set of operations, LO_S its subset of local operations (reads and writes) as well as the operations needed for expansion (inverse writes).
3. $COMMUTE_S$ is a commutativity predicate over $T_S \times T_S \cup LO_S \times LO_S$
4. CON_S is a conflict predicate, defined over $LO_S \times LO_S$
5. $<$ is the output order, a partial order over O_S , such that:
 - $\forall t \in T_S, \forall o, o' \in O_t : (o <_t o') \Rightarrow (o < o')$.
 - $\forall o, o' \in O_S : CON_S(o, o') \Rightarrow (o < o') \vee (o' < o)$.

□

3.5.2. Atomicity in a composite execution

The presence of parallelism and concurrency no longer implies that expansion and compensation as applied in the previous case are still correct. Indeed, interference of concurrent invocations can imply that normal invocations and their expansion or compensation no longer appear as effect-free.

Definition 24 (Correct expansion) *A partially executed RMI $r \in T_S$, which so far has executed the partially $<_e$ -ordered set of operations O_r , is correctly expanded in an execution CE if:*

- expansion, O_r^{-1} , appears in CE ,
- it is possible (by switching commuting pairs of leaf operations) to re-arrange the execution in CE such that O_r and O_r^{-1} appear isolated, i.e., with no other operation in between, making them effect-free in CE .

□

Definition 25 (Correct compensation) *An RMI $r \in T_S$ that has executed all its local operations is correctly compensated in a composite execution CE if:*

- a compensation, r^* , appears in CE ,
- both r_S and r^* are reduced,
- it is possible to transform CE , by switching commuting $<_e$ pairs of leaves, such that r_S and r^* appear together with no other leaf in between (resulting in an effect-free pair).

□

As with expansion, we only consider the local parts of r , and not the remote calls that may have been made.

3.5.3. Correctness in the presence of failures

Since the previous definitions only consider the local parts and discard any remote calls, we need a few more definitions to specify what correctness of abort means.

Definition 26 (Cancellation) *A cancellation of a node $r \in T_S$ in a composite execution CE consists of r and its correct expansion or correct compensation.* \square

A cancellation is the counterpart of a computation, but for aborted invocations.

A simple example of what can go wrong and prevent an RMI from being correctly aborted, is the following: $dec_1(x), write_2(x, 0), inc_1(x)$ where the indices identify a particular RMI. Indeed, one RMI (labeled 1) decrements x , which is later compensated for by an increment. However, in the meantime x has been updated to 0. The result is not the same as if 1 had never executed at all, hence abort is incorrect.

Definition 27 (Annihilation) *Annihilation is an abstraction step in which a node and its cancellation are left out of the execution.* \square

Annihilation is the equivalent of reduction, but for aborted invocations. By definition, the execution of a cancellation can not be observed by any invocation (except for descendants or ancestors as argued already), so it can be left out without affecting the rest of the execution. Similar techniques were used in, for instance, [Has96, AVA⁺94].

Definition 28 (Correct abort)

An aborted invocation t is correctly aborted in a composite execution CE if there exists a transformation CE^ in which t and all its descendants have been annihilated.* \square

Definition 29 (Correctness of a TRMI execution with failures (CRMI))

An execution of a set of concurrent TRMIs is correct (CRMI) if it is equivalent to a serial execution involving only the committed roots. \square

The equivalent serial execution can be found by repeatedly switching commuting leaves, constructing cancellations and annihilations for aborted nodes, and computations and reductions for committed ones. Finally, extending a partial $<_e$ order to a total one may be necessary, as required by the definition of *serial*.

In practice, this correctness has to be ensured by the schedulers in the system. Because the future is not predictable, any RMI may still have to be aborted as long as it is not committed. More precisely: there will always be RMIs as well as committed or aborted TRMIs. Therefore, we can speak of RMI executions rather than TRMI executions in practical cases. Consequently, to determine the correctness in such an environment, we have to take into account the fact that these RMIs could still have to be aborted. For discussing correctness in cases with active invocations, we will work with the so-called completed composite system and complete composite execution.

Definition 30 (Completed composite execution)

For a composite execution CE with active nodes, the completed composite execution is obtained by, at each schedule S , adding the expansion after all active nodes and where appropriate, adding compensation for the remaining active calls. \square

For some nodes, where compensation is available, expansion may not be appropriate. In general, there are different ways to complete an execution. In the next chapter, we will clarify how protocols should do this. Consequently, for any composite execution, there will be a completed composite execution as determined by the protocols in question.

A real system will then use a correctness criterion such as the next one:

Definition 31 (Prefix Correctness with failures (PCRMI)) *A composite execution of a set of concurrent RMIs is prefix correct (PCRMI) if its completed composite execution is CRMI.* \square

The schedulers in a composite system are responsible for ensuring that this property holds at all times.

4. Protocols for correct composite systems

In this chapter we can finally present the implications of distribution, autonomy and the rest of the basic assumptions on practical protocols for composite systems. We will distinguish two main types of invocations: those that have a compensation and those that have not. As will be seen, scheduling requirements are slightly different for each of these categories. A fundamental initial assumption is that an update-in-place strategy applies to each scheduler, as is the case in most commercial database systems.

4.1. Non-compensatable invocations

In order to simplify the rest of the discussion we will start with a definition of the different transactional invocation states.

4.1.1. Invocation states

Definition 32 (Invocation states) *A non-compensatable invocation is always in exactly one of the following states:*

- *active: the invocation is still submitting local operations to its schedule*
- *waiting: the invocation is no longer submitting local operations to its schedule, and waiting for either remote calls to return, for root termination notification or for timeout abort*
- *aborting: the invocation is being aborted: no new local operations are being executed and the scheduler is applying the expansion*
- *aborted: abort has been completed, and no more operations are necessary on account of this invocation*
- *committed: commit notification has arrived, no more operations are necessary on account of this invocation*

□

The states along with their transitions are shown in Figure 4.1.

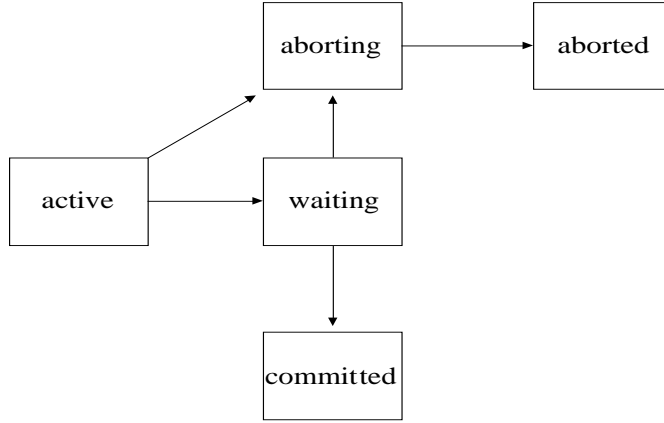


Figure 4.1.: States and transitions for non-compensatable invocations

4.1.2. Recoverability

As soon as a scheduler S executes a local operation o of some invocation $t \in T_S$, the possibility arises that t will have to be aborted by applying its expansion containing o^{-1} . For that expansion to be correct in the corresponding execution, it is absolutely necessary that there exists no independent o' such that $CON_S(o, o') \wedge CON_S(o', o^{-1})$ and $o <_e \dots <_e o' <_e \dots <_e o^{-1}$.

This brings us to the definition of recoverability at the operation level:

Definition 33 (Operation-level recoverable (REC-O)) *A schedule S is operation-level recoverable with respect to an aborted invocation $t \in T_S$ if $\forall o \in (O_t \cap LO_S)$ with inverse $o^{-1} : \nexists o' \notin (O_t \cup O_t^{-1})$ such that $o <_e \dots <_e o' <_e \dots <_e o^{-1}$ and $CON_S(o, o') \wedge CON_S(o', o^{-1})$. \square*

In practice, this implies that S should block any such conflicting o' as soon as it has executed an undoable operation o on behalf of an invocation that may still have to be aborted. This means that REC-O is to be ensured as long as an invocation is in any other state but *committed* or *aborted*. REC-O is inherently coupled with the desire to avoid cascading aborts.

4.1.3. Ensuring that a local computation exists

Whereas recoverability in itself is crucial, another aspect concerns the guarantee that a (partial or complete) computation exists for each invocation $t \in T_S$. This is again primarily the responsibility of S . According to Lemma 1, this means we have to make

sure that there are no conflicting sequences of the form $o_1 <_e \dots <_e o_2 <_e \dots <_e o_3$ where $o_1, o_3 \in O_t$ and $o_2 \notin O_t$. In order to address this issue, we need the following definition.

Definition 34 (Forward strict w.r.t. states M (FS- M)) *A schedule S is FS with respect to a set of states M if for any invocation $t \in T_S$ with local operation $o_1 \in O_t$ and t in a state $\in M$, it delays any later operation $o_2 \notin (O_t \cup O_t^{-1})$ if $CON_S(o_1, o_2)$, until t is no longer in any state of M . \square*

For *aborting* transactions a similar need exists: the expansion has to be isolated from interference of concurrency. Therefore, we can formulate a related property for aborting correctly:

Definition 35 (Abort strict schedule (AS)) *A schedule S is AS if for any aborting invocation $t \in T_S$ with local undo operation $o_1 \in O_t^{-1}$, it delays any later operation $o_2 \notin O_t^{-1}$ if $CON_S(o_1, o_2)$, until t is aborted. \square*

As can easily be seen, AS+FS-(active,aborting,waiting) of S for an aborted transaction t implies that S is REC-O with respect to that t . These two properties together correspond to the classical *strict two-phase locking* property, the most widely used scheduling protocol in practice.

For reasons that will become clear later in this chapter, we have separated active and aborting invocations and defined requirements for each separately in the form of FS and AS.

4.1.4. Ensuring that a global computation exists

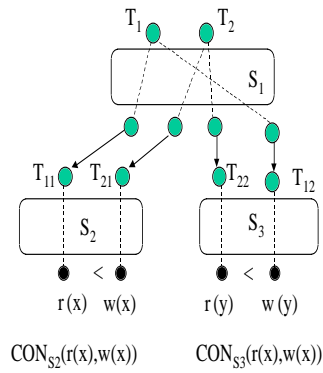
So far we have discussed how to ensure that a local computation exists and how recoverability can be guaranteed. For committed invocations, it is also important to have a computation in the global composite execution. A classical problem for any distributed transaction environment is shown in Figure 4.2; it is caused by the fact that different schedules each decide on different observed orders for local invocations of the same root. A straightforward way of preventing such problems is, again, by enforcing FS-(active,aborting,waiting) on each schedule. According to the state diagram, this means that conflicting operations in S are blocked until after the root termination has been dealt with.

Observe that a schedule that enforces FS-(active,aborting,waiting) and AS for aborting transaction always satisfies REC-O.

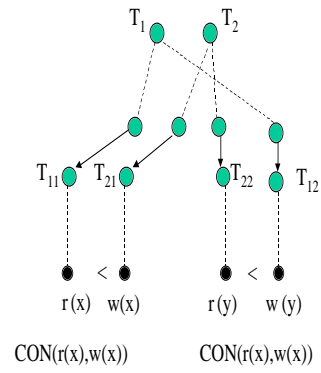
Lemma 3 (Effect of FS-(active,waiting,aborting) on $<_e$)

*For any $<_e$ ordered pair o_1, o_2 of **conflicting** and local operations of different invocations that are subject to FS-(active,waiting,aborting), we have the following: if $o_1 <_e o_2$, then the root of o_1 must have finished before o_2 was started. \square*

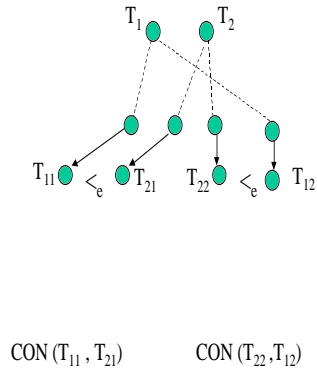
The proof of this Lemma is trivial and follows directly from the definition of FS.



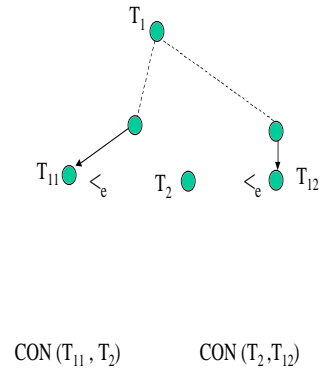
Step 1: composite system



Step 2: composite execution



Step 3: reduction



Step 4: no further reduction possible

Figure 4.2.: Example with only local strictness

Lemma 4 (Effect of FS-(active) on $<_e$)

For FS-(active) schedulers, any $<_e$ ordered pair o_1, o_2 of **conflicting and local** operations implies the following: if $o_1 <_e o_2$, and $o_1 \in O_r, o_2 \in O_s$ for two different invocations $r \in T_S$ and $s \in T_S$, then r must have previously entered the waiting state or the aborting state before o_2 was started. \square

Proof. Trivial by definition of FS and the state transition diagram. \square

Lemma 5 (Preservation of $<_t$ for local operations under FS-(active))

For FS-(active) schedulers, no $<_t$ order between two local operations as specified in a TRMI can be contradicted by $<_e$. \square

Proof. Suppose it were. Then there must exist $o_1, o_2 \in O_t$, local operations for some TRMI t , and $o_1 <_t o_2$ whereas a conflicting sequence exists such that $o_2 <_e \dots o \dots <_e o_1$. o can not be of another invocation at S , since this is ruled out by Lemma 4: it would mean that t executes a local operation after leaving the active state, which is a contradiction. Consequently, any such sequence with some o can involve only (local) operations of t itself, meaning that S generates an output order that does not respect $<_t$, a contradiction with the definition of a schedule. \square

After the previous two lemmas, it should be clear that FS-(active) ensures that the local projection of an invocation can be isolated and hence has a local computation. Also, AS ensures that the expansion can be isolated in itself. FS-(active,aborting,waiting) and AS together ensure REC-O.

4.1.5. Nested semantics: abort propagation to descendants

In any case, correct abort for nested transactions requires the introduction of the following rule:

Definition 36 (Descendant abort rule (DA))

The abort of any invocation t also implies the abort of all its descendants. \square

Without this rule, correct abort can not be enforced. Note that completed composite executions automatically satisfy DA, since no descendant can have committed if one of its ancestors is still active.

4.1.6. Lock Inheritance

In the case of FS-(active,waiting,aborting) scheduling, the problem of the diamond case is not avoided: siblings that access the same data through a conflicting low-level interaction will not be able to commit, since FS will make the later sibling wait for the earlier to be committed before executing a conflicting operation. Of course, commit only happens after the root can commit, which in turn happens only after all siblings are no longer active. Hence, a deadlock arises in diamond cases. This hints that FS-(waiting)

is too restrictive to allow lock inheritance, hence to avoid diamond problems. A similar argument can be made for REC-O: it is not difficult to see that avoiding certain diamond cases could require a violation of REC-O. Consequently, cascading aborts become necessary among siblings. Fortunately, this is not a serious problem: the annoying aspect of cascading aborts is that it normally affects multiple users' transactions. Siblings, however, are for the same root and hence also for the same user.

Definition 37 (Lock inheritance) *A schedule S provides lock inheritance if it does not block operations on account of siblings during the waiting state.* \square

Consequently, a scheduler that always applies REC-O, even to siblings in their waiting state, does not provide lock inheritance and, therefore, will isolate all siblings from each other. FS-(active,aborting) and AS are still needed, because each invocation in itself has to be correct. Lock inheritance specifies what happens (or rather, what does not happen) between *different invocations of the same root* after one of them reaches the *waiting* state. Right now, it suffices to say that there are important implications with respect to abort: siblings cannot abort independently of each other because they may share conflicts and may have seen each other's effects. The issue of recoverability among siblings to which lock inheritance is applied will be discussed later.

Lock inheritance not only has consequences with respect to abort but also with respect to correctness of executions in the absence of failures: indeed, FS-(active,aborting) in itself is not enough. Something more is needed, as captured in the following:

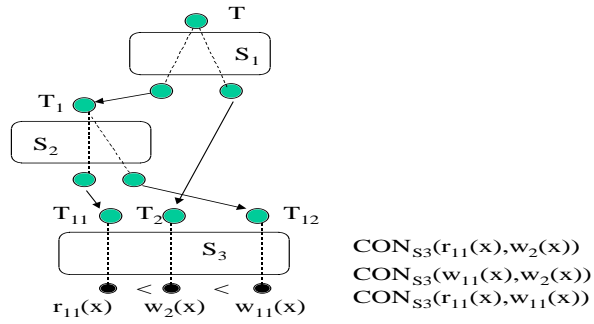
Definition 38 (Sibling isolation (SI)) *A protocol family has the property SI if in every possible composite execution CE that it produces, the following holds: given a committed invocation t with only leaves left in the reduction process (hence candidate for construction of a computation), and $o_1, o_2 \in O_t \Rightarrow \nexists o_3 \in O_{t'}$, where t' is a committed sibling of t and $o_1 <_e o_3 <_e o_2 \wedge CON(o_1, o_3), CON(o_2, o_3)$.* \square

SI is needed for an invocation to have a computation free from interference with its siblings, so that it can be reduced. Note that FS-(active,aborting) is *not* enough to ensure this, as shown by Figures 4.3.a to 4.3.c. Indeed, in the example all local invocations can happen in a schedule where invocations are subject to FS-(active,aborting), yet T_1 has no computation due to remote interference among different siblings. The problem lies in the fact that the example does not enforce SI.

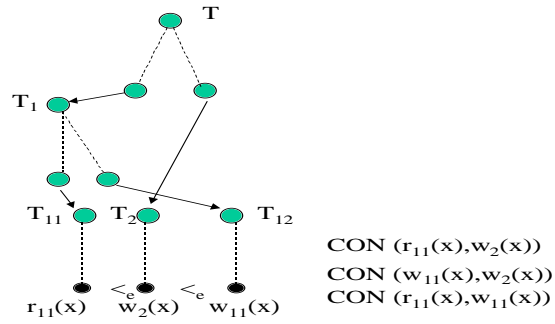
4.1.7. Orphans

Orphans [IMP94] are pending siblings whose acknowledgement was lost and hence no longer form part of the computation. They are a problem specific to nested transaction systems (in flat transactions, the loss of an acknowledgement would mean rollback of the entire root due to a failed call on the client side).

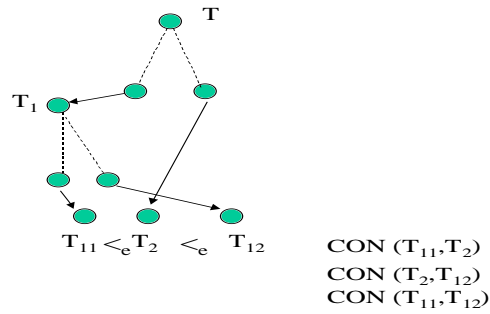
A very important implication for systems with lock inheritance is that, in general, orphans can not be allowed to exist after commitment of the root. Indeed, otherwise the eventual abort of an orphan (which will happen after a timeout on its server) can



(a) Example execution for sibling isolation problem



(b) Corresponding composite execution



(c) No computation for T_1 possible

Figure 4.3.: FS-(active, aborting) is not enough for sibling isolation

not be guaranteed to be correct since its abort can not be applied arbitrarily but rather has to be done together with the other siblings' aborts. However, the other siblings can no longer abort after commitment of the root.

4.2. Compensatable invocations

For compensatable invocations, we can repeat the previous arguments, but with slight variations related to the compensatable aspect.

4.2.1. Invocation states

Definition 39 (Invocation states) *A compensatable invocation is always in exactly one of the following states:*

- *active: the invocation is still submitting local operations to its schedule*
- *waiting: the invocation is no longer submitting local operations to its schedule and waiting for either remote calls to return, for root termination notification or for timeout abort*
- *aborting: the invocation could not execute all its local operations and is being aborted: no new local operations are being executed and the scheduler is applying the expansion*
- *compensating: the invocation has reached the waiting state already, but a later event triggered abort by compensation*
- *aborted: abort has been completed, and no more operations are necessary on account of this invocation*
- *committed: commit notification has arrived, no more operations are necessary on account of this invocation*

□

A state diagram for a compensatable invocation is shown in Figure 4.4. Especially note that a waiting invocation can no longer be aborted by expansion in this case; compensation allows increased concurrency and will be preferable as soon as a non-active transaction has to abort.

Indeed, REC-O can be replaced by something less restrictive. For an example, see Figure 4.5. There we have an increment labeled inc_1 and its compensation labeled inc_1^{-1} . By ‘uncoupling’ the normal increment invocation and its correction in case of abort (i.e., its compensation) we no longer depend on REC-O for recoverability: between inc_1 and inc_1^{-1} it becomes possible to allow other invocations provided that they commute with either inc_1 or inc_1^{-1} . For non-compensatable invocations, this is precluded by the inherent dependence of inverse writes on their original data values.

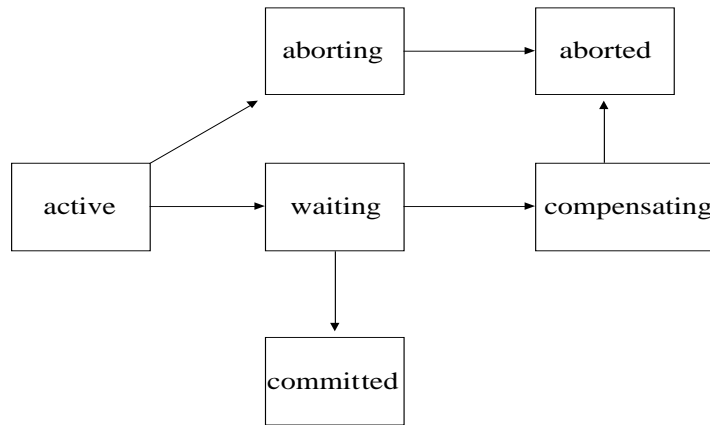


Figure 4.4.: States and transitions for compensatable invocations

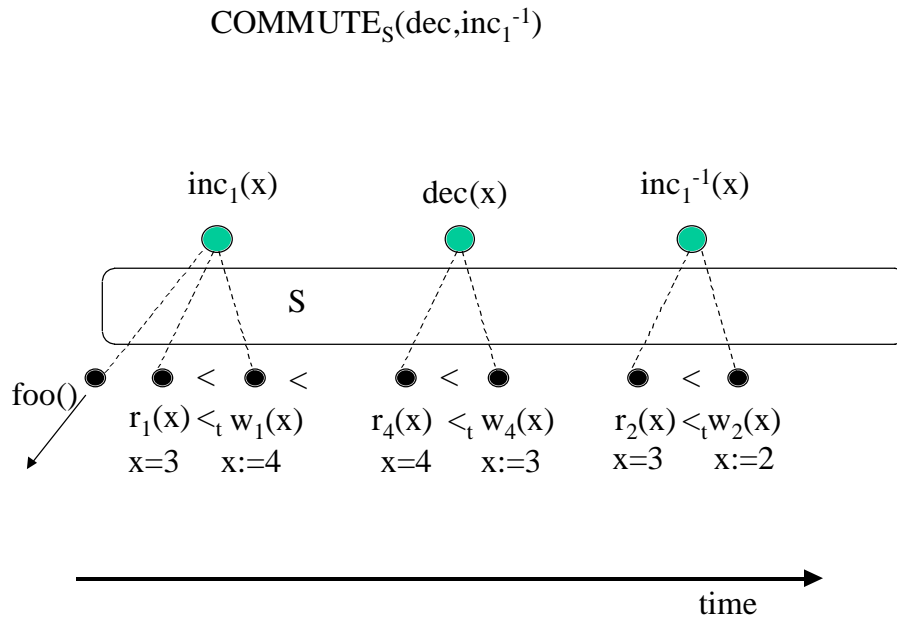


Figure 4.5.: Scheduling with COMMUTE and compensation knowledge

To see why this is an increase in concurrency, it suffices to note that this way of scheduling allows some other invocations *as soon as the invocation is locally finished*, instead of having to wait until the *root* – i.e., the global composite transaction – is finished (as is the case for non-compensatables). However, this is only possible if two conditions hold:

1. compensation is available, and
2. a later invocation is known to commute.

The big difference between compensatable invocations and others is the way schedulers deal with the waiting phase. Assuming that a schedule enforces FS and AS only during the active and aborting states, we can say the following for a compensated execution $t \in T_S$:

- both the local projection of the invocation as well as its compensation can be isolated in the resulting composite execution (Lemmas 4 and 5) resulting in t_S and t^{-1}
- $<_e$ ordered in between, there can be another invocation u that was started during the waiting state of t (because REC-O is abandoned in the waiting state)
- u has a local projection that can be isolated (Lemmas 4 and 5).

For now, assume that such u has also reached its waiting state (i.e., all local operations of u are done). Then u has a local computation at S , called u_S . Consequently, the case we are describing here is of $t_S <_e \dots <_e u_S <_e \dots <_e t^{-1}$, all local effects at S .

Note that since compensation is treated as another invocation, S enforces FS-(active,aborting) also for t^{-1} . Also let the reader be reminded that $COMMUTE_S$ also contains the semantics for compensation. Additionally, if we assume from now on that *commuting pairs of invocations cannot involve non-compensatable invocations nor their expansions*, we can formulate the following recoverability criterion for scheduling when commutativity and compensation are known and an invocation has reached its waiting state.

Definition 40 (REC-I) *A schedule S is interface-level recoverable (REC-I) with respect to a compensatable invocation $t \in T_S$ if S delays, for any active, waiting or compensating $t \in T_S$, any later invocation or compensation $u \in T_S$ if:*
 $\neg COMMUTE_S(t, u) \wedge \neg COMMUTE_S(u, t^{-1})$. □

For compensatable transactions we will allow certain compensatable invocations or compensations to be scheduled between a compensatable invocation and its compensation. However, doing this without restriction leads to cases similar to Figure 4.2, so additional care is needed. This is easily taken care of by adopting the next policy:

Definition 41 (Forward strict invocation scheduling (FSI)) *A schedule S is FSI if S delays, for any compensatable $t_1 \in T_S$ in state active, waiting or compensating, any later invocation or compensation $t_2 \in T_S$ if:*
 $\neg COMMUTE_S(t_1, t_2)$. □

4.2.2. Lock inheritance

Lock inheritance can be defined for compensatable cases in a very similar way as for the non-compensatable case:

Definition 42 (Lock inheritance) *A schedule S provides lock inheritance after a compensatable invocation t if, with respect to at least some set of siblings, it does not apply FSI during the waiting state of t .* \square

4.3. Abort policy and completion of a composite execution

The presence of siblings requires great care in processing aborts; in general siblings can not be aborted independently due to lock inheritance and the corresponding lack of REC-O or REC-I. Intuitively, abort of siblings should be done in the reverse of the $<_e$ order between the sibling invocations themselves. For the sake of generality we will define a property that schedulers must ensure in order to guarantee that this can be done.

Definition 43 (Sibling recoverability (SR)) *Consider an aborted invocation at some S : $a \in T_S$ and $a_S <_e \dots <_e a^{-1}$, where a_S is the local computation of a at S followed by its compensation or expansion a^{-1} . S ensures sibling recoverability if the following two conditions are always true:*

- *if \exists a sibling b : $a_S <_e \dots <_e b_S <_e \dots <_e a^{-1}$ then b_S can be annihilated before annihilating a_S*
- *if \exists a sibling c : $a_S <_e \dots <_e c^{-1} <_e \dots <_e a^{-1}$ then c_S can be annihilated before annihilating a_S*

\square

SR is not trivially ensured, as can be seen in Figure 4.6. Assuming that only siblings are shown, it is clear that there is no invocation whose annihilation can happen first.

If a scheduler ensures SR then there is a clear order that can be derived for the annihilation of siblings. In order to find that order, we need a graph structure.

Definition 44 (Abort Precedence Graph (APG)) *The APG for an invocation $t \in T_S$ is defined as follows:*

- *the nodes of APG are all siblings of t at S*
- *an edge is added from node b to node a if $a_S <_e \dots <_e b_S <_e \dots <_e a^{-1}$*
- *an edge is added from node c to node a if $a_S <_e \dots <_e c^{-1} <_e \dots <_e a^{-1}$*

\square

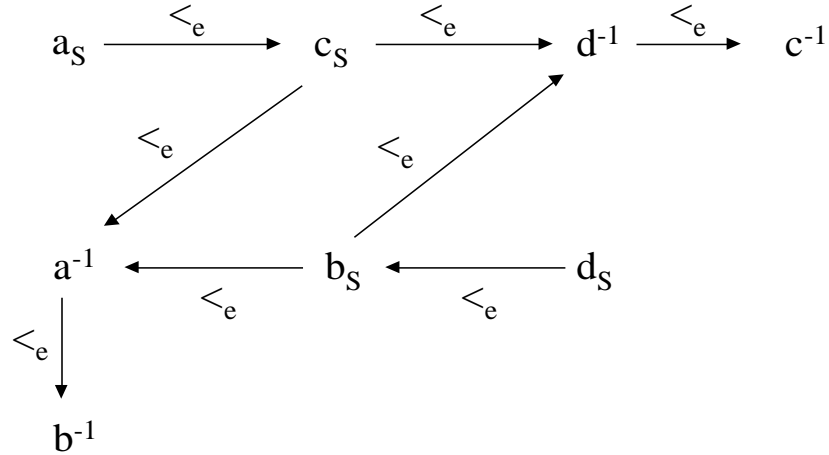


Figure 4.6.: Example of siblings where SR does not hold

For a schedule that enforces SR, the APG is acyclic. This fact allows the following statement to be made for aborting siblings: in a composite execution, correct abort of siblings requires the following:

Definition 45 (Sibling Abort Rule (SAR))

1. each S ensures SR
2. abort of one TRMI t implies the abort of **all siblings of t , executed on the same scheduler**, in the order given by a topological sort of the APG for t
3. DA is applied for aborting descendants.

□

Definition 46 (Abort policy in a composite execution (ABORT)) *Abort is performed as follows:*

1. any active invocation is aborted by adding its expansion
2. a waiting invocation can be aborted by applying expansion for non-compensatable invocations and compensation for compensatable invocations, consistent with SAR and SR.

□

At this point it becomes possible to specify the way to perform the completion of a composite execution containing any number of active invocations. We will implicitly assume that the aborts are done according to SAR, and that at least FS-(active,aborting)+AS holds for each invocation. In that case completion has to be done according to the following rule:

Definition 47 (Completion rule for a composite execution (COMP)) *Any composite execution CE with RMIs is completed by applying ABORT to each active or waiting invocation.*

□

4.3.1. Correctness of abort

Assume any composite execution subject to FS-(active,aborting)+AS for each invocation, and where FSI for compensatables or FS-(waiting) for non-compensatables holds among non-siblings. Additionally, assume that all schedulers ensure SR. The purpose here is to show that abort as specified by ABORT is correct in any such composite execution. The reasoning is incremental: we will start with a given composite execution CE and gradually annihilate all aborted calls. On account of the FS-(active) and AS property, all local projections in CE can be isolated, and we will assume that this has already been done.

Lemma 6 (Correct expansion of active invocations) *For any composite execution CE where each invocation happens under at least FS-(active,aborting) and AS, adding the expansion of some active invocation $t \in T_S$ ensures annihilation of t .*

□

Proof. Suppose that t were not correctly annihilated. This implies that there does not exist a cancellation of t . For reasons similar to Lemma 1 this implies that there exists a conflicting operation $o' : o' \notin O_t \wedge o' \notin O_t^{-1}$ such that $o <_e \dots <_e o' <_e \dots <_e o^{-1}$, each ordered pair also being a conflict pair and $o \in O_t, o^{-1} \in O_t^{-1}$. This clearly violates FS-(active,aborting) with AS and the fact that t was still active, a contradiction. By adding the expansion (which by definition applies inverse writes in the reverse order of the original writes), the effects of all original writes are annihilated and the result is effect-free. Therefore expansion is correct.

□

As a consequence of this lemma all active invocations can be annihilated, meaning that the only remaining invocations are those that have already executed all *local* operations at their schedule. This has the important advantage that commutativity can be applied to pairs of such invocations. Let CE^* be the resulting transformed execution.

Lemma 7 (Correct abort) *Consider CE^* : performing abort according to ABORT ensures all aborted nodes can be annihilated.* \square

Proof. We will prove that aborted invocations can be annihilated on each schedule where they are present. Because of the previous steps, active expanded invocations have been annihilated already. Suppose ABORT results in an aborted invocation t_S (represented by its local projection here, because of the locality of abort) and its compensation or expansion t^{-1} . Assume that t is found by the APG construction as ABORT requires. If there are any siblings of t on S , then by ABORT and the APG for t , t is the first one to annihilate. There are two cases to consider: either there exists a $<_e$ series between t_S and t^{-1} , or not. If not, annihilation is straightforward and we can continue with the next invocation to annihilate. The interesting case is where there does exist such a series: $t_S <_e u_S <_e \dots <_e t^{-1}$. Then by ABORT we know that u can not be a sibling: all such siblings have been annihilated already. Hence u concerns a non-sibling and is either an invocation, an expansion, or a compensation. If t is not compensatable, then *REC* – *O* forbids this from happening at all, and annihilation of t is trivial. If t is compensatable then it must have been in the waiting state when u was started, because compensation only happens afterwards. Hence, limitations imposed by FSI apply and u can not be an expansion. Neither can it be an invocation or compensation that does not commute with t_S . Consequently, u_S is for an invocation u that commutes with t . But if they commute they must both be purely local and commutativity also applies to t_S, u_S because they involve the same set of operations. Therefore, any such u_S must commute with t_S and can be moved *in front of* t_S , and t can be annihilated. Note that for u_S , the only $<_e$ order that was changed was the one with respect to t , which is now annihilated. All the other $<_e$ orders concerning u are still as in the original composite execution. Repeatedly applying the technique described here will eventually annihilate all aborted invocations on each schedule. \square

The conclusion of this exercise is that any composite execution satisfying the necessary requirements can be completed by COMP and be transformed to an equivalent execution where only committed nodes remain.

4.4. Protocols for correct scheduling

So far a number of properties have been introduced such as SI and SR. Exactly how these properties can be enforced on a composite execution has not been specified yet; in general there will be more than one way of doing so. By leaving such implementation issues out of the discussion, we can formulate a set of protocols that provide correct composite executions. This set is specified by LISP.

Definition 48 (Lock Inheritance Scheduling Protocols (LISP))

A scheduling protocol is in LISP if the following holds:

1. FS-(active,aborting) and AS for each invocation
2. among non-siblings, FS-(waiting) holds for non-compensatables and FSI for compensatables
3. abort is done as in ABORT

□

Theorem 2 (Correctness of LISP+SI) *The following holds: $LISP+SI \Rightarrow PCRMI$. More precisely, if all local schedules in a composite system follow LISP and SI holds in every composite execution, then the resulting composite executions are always correct.* □

Proof. According to ABORT of LISP, all ongoing RMIs are aborted. According to Lemmas 6 and 7, all aborted transactions are correctly aborted. The only remaining nodes in the resulting equivalent composite execution are invocations that committed. According to Lemma 10, each of these nodes can be reduced up to the roots. Consequently, an equivalent composite execution exists, containing only the committed roots. There can be no conflict cycle along $<_e$ for this equivalent composite execution, as shown in Lemma 2. Therefore, a cycle in $<_e$ involves at least one commuting pair of root invocations. In such a case we can reverse the $<_e$ order for that pair and we obtain an acyclic $<_e$. Topologically sorting along $<_e$ gives an equivalent composite execution containing only the committed roots. □

Lemma 8 (Effect of FS-(active) on observed order) *Let $t, u \in T_S$ on some schedule S , and let u be started after t has left the active state. Then it can never happen that $u_S <_e s_S <_e \dots <_e t_S$.* □

Proof. Any $<_e$ between local projections must be due to local operations at S . For FS-(active) schedulers, $<_e$ between u_S and s_S means that u must have left its active state before s executed the operation that caused the order to arise. Hence, u must have left the active state before s left its active state. Applying this to each pair shows that each such $<_e$ reflects the order in which invocations left their active states. The proposed series then contradicts that u was started after t left its active state, which concludes the proof. □

Lemma 9 (Root termination and observed order) *Let $t \in T_S, t' \in T_S$ be two committed invocations on account of different roots. If $t <_e t'$ then the root of t must have committed before the root of t' could commit.* □

Proof. Trivial by FS-(active,waiting) for non-compensatable t and FS-(active)+FSI for compensatable t . \square

Lemma 10 (Reducibility of nodes)

Let n be an invocation that has executed at least all of its local operations. n can be reduced if n and its descendants are subject to LISP. \square

Proof. Reduction can only fail if n , or some descendant t , does not have a computation, by Theorem 1. There are two steps in constructing a computation at some stage in this process:

1. Isolating all operations of each node in n 's tree, which is possible. Indeed, suppose not. Then there exists at least one TRMI t such that o_1, o_2 are operations of t and a series s of conflicting $<_e$ pairs $o_1 <_e o_3 <_e \dots <_e o_2$, with o_3 belonging to $t' \neq t$ (cf. Lemma 1). First, suppose that t and t' are from different roots R and R' . According to Lemma 9, this means that the root of t has committed before itself, a contradiction. The remaining case is when t, t' are of the **same** root R , in which case SI ensures that s can not occur.
2. After isolation, no TRMS specified order can be contradicted by $<_e$. To see why, suppose such a case does exist. Hence, for some isolated TRMI t , we have $o_1 <_t o_2$ in its TRMS but a series $s: o_2 <_e o_3 <_e o_1$ (without loss of generality we assume only three operations here). Because t was already isolated in the previous step, **all** operations are part of t . If $o_1 <_t o_2$, this means that o_1 and all its descendants left the active state before o_2 started. Consequently, on all sites where o_1, o_2 interact, o_1 or one of its descendants left its active state before the interacting invocation was started. Hence, by lemma 8, this problem can not happen.

\square

4.5. Enforcing SI and SR

So far, details of how to enforce SI or SR properties have been left open. In general, there are several ways of ensuring that SI holds for siblings.

4.5.1. Full lock inheritance

The most expensive one is by implementing true lock inheritance, which amounts to blocking a second and conflicting sibling until the scheduler is notified that the first sibling has terminated and will no longer submit new operations. It is not difficult to see that this requires, besides the already expensive *two-phase commit* message rounds, another message round for each finished ancestor of a given local transaction.

4.5.2. No intra-root parallelism

A less expensive and more pragmatic way is the following: for cases where lock inheritance is desirable, all activity inside the same root transaction has to happen serially.

Lemma 11 (SI for serial intra-root executions) *If intra-root executions are executed serially, then SI as required in LISP is ensured.* \square

Proof. Suppose it were not so. Then there exists a committed t whose operations have all been reduced but that does not have a computation due to a problem of the form $o_1 <_e o_3 <_e o_2$ such that $CON(o_1, o_3) \wedge CON(o_2, o_3)$, for $o_1, o_2 \in O_t$ and a committed sibling's operation $o_3 \in O'_t$. Due to FS-(active), there must have been a descendant of o_1 that left its active state before a descendant of o_3 left its active state. Hence, t was active at some schedule *before* t' was active there. However, for $o_3 <_e o_2$ we can say that there was a place where t' must have been active before t . Since t' is not a descendant of t (because t is left with only leaves, and t' is not one of them), this means that a non-descendant of t was active at the same time that t was active, contradicting the intra-root serial execution. \square

With this solution, enforcing SR is equally easy: if expansion and compensation are applied (at each schedule) in the reversed serial order of their original invocations, then SR must hold. This can be seen as follows: because there is no intra-root parallelism, all siblings on the same S will arrive serially. Let c be the *last* of all sibling invocations at S . Due to FS-(active), there can be no sibling b such that $c <_e b$. After adding c^{-1} , there is no expansion or compensation b^{-1} present yet, and SR holds for c . Annihilation of c is straightforward. Let a be the one-but-last sibling at S . After c^{-1} , a^{-1} is applied. Due to the serial execution, there could only have been one sibling $<_e$ ordered after a (namely c), and that one is already annihilated. Also, there could only have been one expansion/compensation $<_e$ ordered before a^{-1} (again for c), and that has been annihilated already as well. The same argument for all remaining siblings shows that SR must hold.

4.5.3. No lock inheritance

If no lock inheritance is applied, then SI holds trivially. Indeed, any $o_1 <_e o_3 <_e o_2$ implies that the root of o_1 must have finished before the root of o_3 , and the root of o_3 before the root of o_2 . But for siblings, all these roots are one and the same, a contradiction.

SR holds as well, simply because there will be no sibling or sibling expansion/compensation $<_e$ ordered between an invocation and its compensation or expansion.

4.6. The case of recursive calls

The case of recursive TRMI calls is interesting enough to deserve its own discussion. As mentioned earlier, recursion happens if inside a composite transaction a child executes

on the same scheduler as one of its parents. A practical and safe approach seems to be to forbid recursion in every case (except when recursion happens within one scheduler and thus within the same local transaction).

One reason is shown in Figure 4.7: if a client (S_1) receives a recursive call (T_{11}) via a remote server (S_2), then given the encapsulation properties of that remote service we can not be sure what that recursive call will do. More precisely: the *programmer* of the client service can not have been sure of what the remote service (T_1 , the one making the recursive invocation) will do. Hence, it is very well possible that some things are done twice for the same root: once in the client, and once more by the remote service through a recursive call. This is shown by the double occurrence of operation *op* (which could be, for instance, the insertion of a due payment in an invoice table) on S_1 . This results in a violation of the consistency of the local database at the client (because the application no longer does what it is believed to do – namely inserting a payment just once), and it is nobody’s fault since none of the servers can know what the other really is doing.

4.7. Global knowledge required for the different solutions

For the suggested set of protocols there are a number of requirements concerning global knowledge about the composite transaction being executed. In here, we do not want to commit ourselves to implementation issues, so the discussion is limited to mentioning the matters that have to be addressed by typical implementations.

4.7.1. Information needed for FS

The information needed for FS is essentially the notification of root termination in order to be able to release the locks on commitment.

4.7.2. Lock inheritance

For lock inheritance one needs to be able to recognize siblings in one way or the other. What is needed is a decision function that yields true if two invocations are siblings and false if they are not. The implementation of this function will normally require a certain amount of global information.

4.7.3. Recursion detection

For detecting recursion it is also necessary to pass on some global information. Concretely, some entity in the system must be able to see if a descendant and an ancestor share a common schedule.

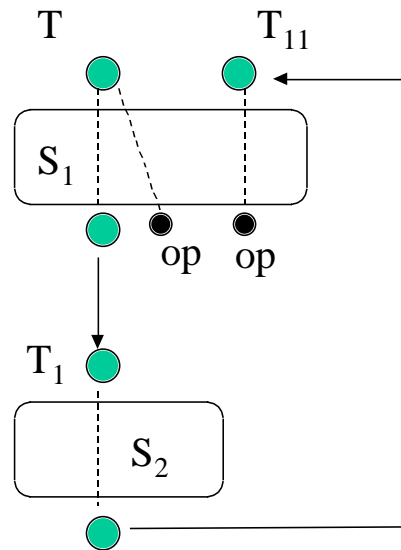


Figure 4.7.: A recursion example

4.7.4. Orphan detection

Orphans should be avoided, and as a rule this can not be achieved without a certain degree of information exchange between schedules. To be more specific, client schedules must agree with server schedules on the outcome of invocations between them.

4.7.5. Two-phase commit

Of course, *two-phase commit* also requires some information exchange; this instance is discussed in more detail in the next section.

4.8. Implications of two-phase commit on the system assumptions

One of our basic assumptions is that we have a *two-phase commit* termination protocol at the end of each root transaction. Another basic assumption is maximum site autonomy. Given the fact that *two-phase commit* can block in certain cases [BHG87], this may seem to lead to a contradicting set of assumptions. Indeed, if a server votes to prepare a transaction, then it loses the right to unilaterally abort on timeout. Consequently, this leads to a situation where resources are blocked and unavailable for other client and the server loses its autonomy to do something about that. If the coordinator then fails or becomes unreachable this situation can become problematic. However, this is a restriction one has to accept if ACID properties are desired: it has been shown [FLP85] that no non-blocking termination protocol exists that supports both site and communication failures. Since the Internet has both these failures as inherent possibilities, there is nothing that we can do about this limitation. The only practical way (which is used in current systems and therefore accepted policy) is to have a server violate the original *two-phase commit* protocol by still allowing a unilateral decision in case it is blocked [GR93]. This is called a *heuristic decision*, and can lead to conflicting situations where human intervention is required. These problem cases can be detected when the blocking situation is eventually resolved, because the server taking such unilateral decision is required to remember it on stable storage. If afterwards an incompatible decision is received from the coordinator, an exception can be raised to attract human intervention.

4.9. A discussion of existing transaction monitor environments

Now, we will discuss the two main sibling isolation strategies applied to existing transactional systems, and show the limitations of these systems. The first two paragraphs will discuss existing technology in general, and the next paragraphs are devoted to a few specific examples.

4.9.1. Blocking among siblings

Virtually all existing databases are (or can be made) strict (except in the case of different database isolation levels as explained further on). Therefore, existing (flat) transaction models that use this strategy will in most cases be correct. However, a root that submits multiple (and conflicting) calls to the same scheduler will deadlock itself, even if executed serially. This implies that some correct specifications can *never* be executed, not even if they are serial and tried in absence of any concurrency. In *Enterprise Java Beans (TM)* [EJB, DP00], this problem has been recognized, and is called the *diamond problem*. Since there is no easy way for a client to change the isolation mode of siblings in existing systems, there is no easy way around this problem.

4.9.2. No blocking among siblings

In normal flat transaction systems, the SAR abort rules are automatically satisfied, because abort is always global and a before-image restoration performs correct annihilation. However, to our knowledge no commercial product checks for recursive calls.

Most environments that forbid intra-transaction parallelism (such as EJB) implicitly comply to SI, due to the prevailing strict scheduler model.

4.9.3. Encina

In *Encina* [Enc], nested transactions are supported, but the isolation among siblings is determined at the server side, as a configuration parameter of the server. Hence, a client has no way of influencing this and can not unset isolation if required. Also, to the author's knowledge, there is no detection of orphans if siblings are not isolated.

4.10. Asynchronous transaction management

Although we assume that each TRMI terminates only after all its subtransactions have finished, we can still use the same theory for asynchronous transaction management (persistent queues [BHM90]). In reality, these are merely a variant of composite transactions, where the root terminates before it receives confirmation of its remote calls. For correctness, the remote calls of the root can be put in a queue and completed asynchronously by a queue manager component (note that this implicitly assumes that these tasks can be retried if they should fail). However, this completion (although done later) will have to happen synchronously (and more precisely serially on behalf of each root), especially in the case where 'siblings' are not isolated (otherwise the same anomalies may arise as shown in our theory). Should 'siblings' be isolated from each other, then asynchronicity would be possible. Note that the main purpose of asynchronous transaction management is to decrease response times at the root, so the requirement for doing the queued parts in a synchronous mode is not a drawback for this technology. Existing research on queues has addressed the problem of consistency across sites in a satisfying way [WC93], but the issue of isolation was not properly solved. Because the rest of this work is not concerned with asynchronous transaction management, we will not go into further detail on this subject. It is worth noting, however, that [DRD99] advocates the need for more research in this direction. Composite system theory certainly holds promises in this respect.

4.11. Summary

In this chapter, we have introduced and discussed a set of protocols that take care of composite transaction scheduling, and proven correctness of those. These protocols are minimal in the sense that removing any of the required features will violate correctness. With these protocols, it is not difficult to build correct transactional middleware based

on a solid theoretical foundation. The presentation has been general enough to allow a number of varying implementations to be designed in a straightforward way. Other topics that have been addressed are the nature of the global information that needs to be exchanged among schedules in a composite system, and a discussion of existing solutions with respect to composite correctness.

Part II.

Implementations of composite systems

5. Composite systems in practice: the CheeTah system

Reading our theoretical work will undoubtedly prompt the question of how to translate these ideas into practical systems for effective use in real-life situations. In this chapter, we introduce our practical approach towards composite systems as incorporated in the *Java* based framework called *CheeTah*. An early description of this system appeared in [PA00].

5.1. The framework approach to software development

Frameworks [Joh] are reusable parts of applications. Object-oriented technology has made it possible to reuse code by subclassing it. However, complex applications do not let themselves be cast into simple object hierarchies alone: there needs to be a way to interconnect different parts of an application. Thus, in order to reuse application-level logic, one needs to identify relevant abstractions in the application domain and group them together via a reusable ‘application kernel’, the *framework* [FSJ99, Joh]. This relieves programmers from re-implementing all these parts again and again: since the necessary abstractions are already in place as well as their ways to interact, the programmer can concentrate on case-specific logic. In general, there are two different kinds of frameworks: *white-box frameworks* and *black-box frameworks*. The former kind is more primitive, in that a case-specific extension requires knowledge of the internals of the rest of the framework. The latter is more general but also more difficult to construct, because it assumes that the framework that is provided can be reused without changing or understanding the existing implementation of the core classes.

Examples of successful framework domains are, among others, user interfaces [SWI] and data management [EJB].

It is not the purpose of this work to explain nor to investigate frameworks in themselves, and interested readers who want to know more are referred to the literature. Related to frameworks, but on a finer granularity, are so-called *design patterns* [GHJV95, Gra98]. These represent recurring problems in object-oriented design and corresponding solutions that are generally accepted as good. Frameworks usually contain a variety of these patterns as parts of their design.

5.2. Introducing CheeTah

The *CheeTah* system is our interpretation of a composite system building block. It is a black-box framework, written entirely in *Java*, and meant to make transactional interaction easy, decentralized and light-weight.

The resulting architecture is as follows. In a composite system (Figure 5.1) each server (from S_1 to S_{13}) is an independent component performing its own scheduling and transaction management. These servers are built using *Java* and inheriting from the classes *Cheetah* provides. The interface to each server defines the services it implements. An invocation of one of these services (through RMI [Sri97]) results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information required to build a global composite transaction is implicitly added by the system to each call. However, it is important to emphasize that each transaction is independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way components can be dynamically added and removed from the system without compromising correctness. All a new server needs to know is the interfaces and the addresses of the servers it will invoke. If a naming and directory service is used, then the addresses need not be given explicitly. Regardless of the configuration, *CheeTah* guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

5.3. Architecture of CheeTah

The conceptual architecture of a *CheeTah* server is shown in Figure 5.2. The structure depends on the particular application, the one being shown here is the one that was also used in the performance measurements (described in one of the next chapters).

The server simulates a purchase point where a number of different items can be bought. The interface to the service is a method called *Buy*, that takes as argument the id of the item to be bought (*itemid*). Invocations are done through RMI, the stubs of which are generated by the framework. The method itself is implemented as a *Java* component that makes calls to a local database (*JDBC API* [JDB, WFC⁺99]) and invokes the services of other servers (by RMI calls). This component is all that needs to be implemented by the designer. The provided lock table indicates that two invocations to the *Buy* service conflict if they have the same *itemid*. Each server uses a local database (currently *Oracle (TM)*) for storing its own local data as well as for keeping a log and undo table. Connections to the local database are pooled for optimal performance. The server also uses the local filesystem to store additional information (the global log used to track *two-phase commit* status). Each server has conceptually three transaction manager modules (*incoming TM*, *outgoing TM* and *internal TM*). The incoming TM takes

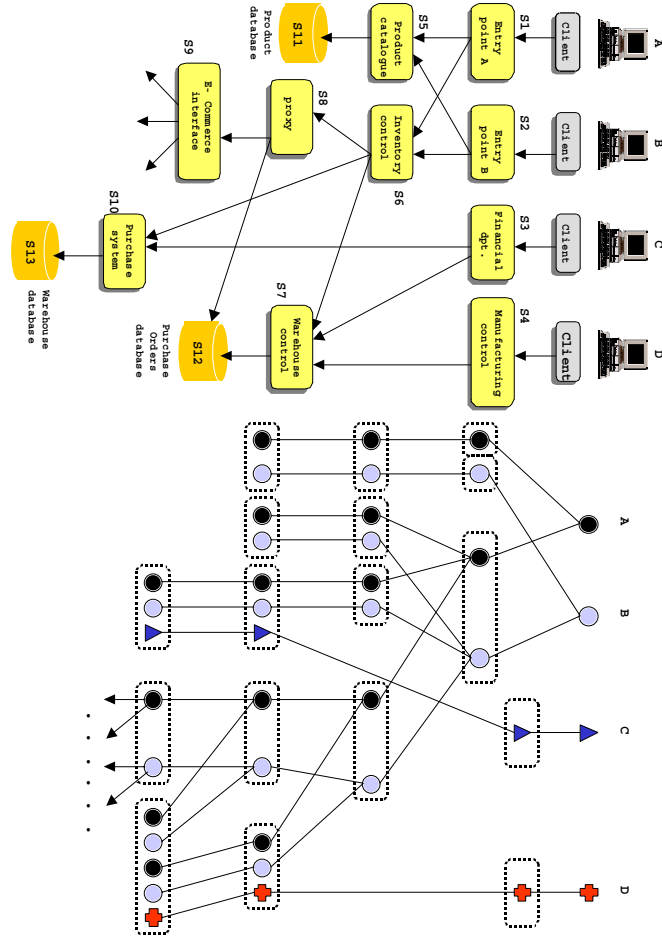


Figure 5.1.: Invocation hierarchy (a, left-hand side) and transactional structure (b, right-hand side) in the composite system shown in Figure 2.1

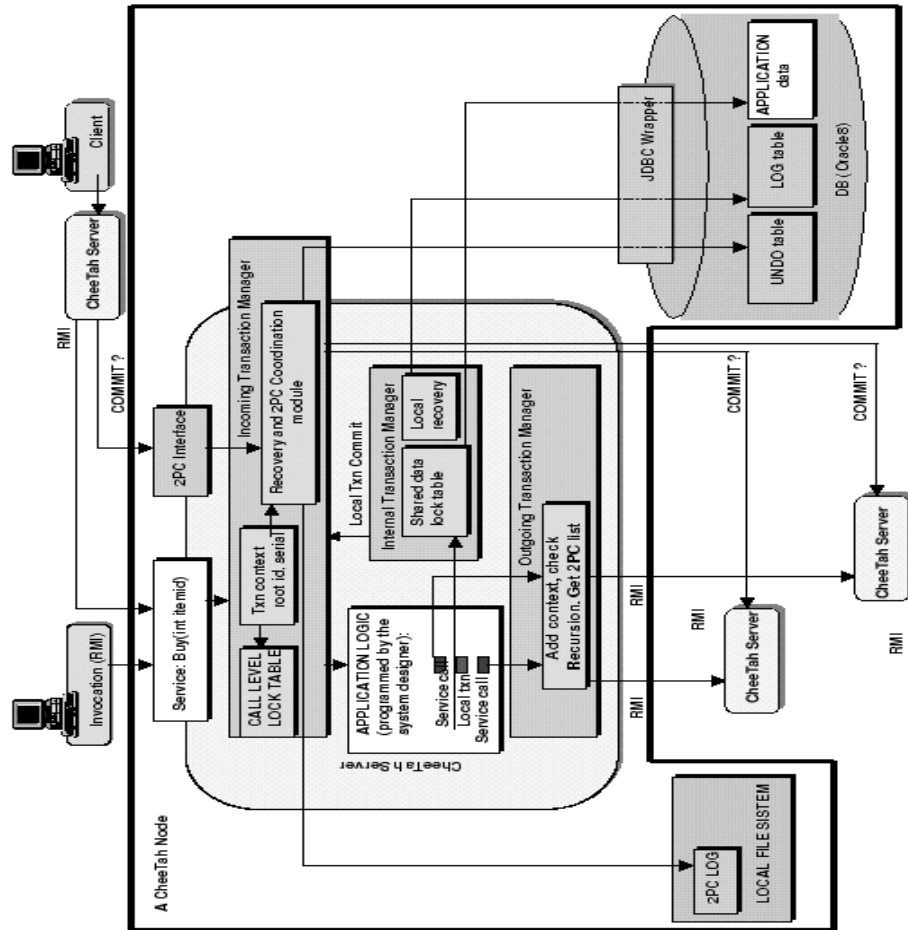


Figure 5.2.: Component Overview

care of invocations to this server (extracting the root context and starting a local transaction for the thread that deals with the incoming call). If invocation-level scheduling is used (an optional feature, chosen if the component has a compensating method for each of its normal methods), then it is this part that is responsible for ensuring compensatable properties. This layer is also responsible for producing log entries related to *two-phase commit*. The internal TM takes care of internal consistency (access to shared variables and internal data, as well as access to the local database). It produces entries in the database log and undo tables: in case of programmer-defined compensation, compensation-related context needs to be stored for recovery after a crash; this is done in these tables. The internal TM is also in charge of performing local compensation whenever necessary. The outgoing TM mainly adds the root context for remote calls and retrieves a list of invoked servers after a remote call returns, both for checking recursion and for determining which servers have to be included in the *two-phase commit*.

5.4. Programming CheeTah applications

Programming an application comes down to three steps:

1. Programming the application logic component. This is a portable chunk of application logic, that can in principle be run on any *CheeTah* server. This is similar to *Enterprise Java Beans (TM)* [EJB, DP00] components but much simpler because *CheeTah* does not provide issues like *passivation* and *activation* and does not currently cache database tuples in main memory (as, for instance, in *entity beans* in the *Enterprise Java Beans (TM)* [EJB, DP00] architecture). Transactional logic follows the *Java Transaction API* [JTA] interfaces, local data access is done with *JDBC API* [JDB, WFC+99], and remote calls are done via RMI. Thus, no significant effort is required to learn how to program *CheeTah*, rather it suffices to know existing and very popular standards.
2. Setting up a connection factory: since the framework itself is independent of which database is underneath, it uses a connection factory for establishing the necessary (pooled) connections. The connection factory is a more or less standard *JDBC API* [JDB, WFC+99] setup of a data source, by providing the user name, password, server name and TCP port number.
3. Linking (1) and (2) into a server: by providing an XML configuration file, one can indicate that the previous steps should be combined into a running server. Some additional settings (such as timeouts for active transactions' aborts) also belong here.

The framework will generate the necessary stubs (RMI), based on the component's methods, and generate a client class archive that can be published. A client that downloads this archive can invoke the local composite server as part of a composite transaction. Examples of programming a server can be found in appendix.

5.5. Scheduling and concurrency control

CheeTah supports all the protocols that were proposed in the theoretical part of this work. More precisely, the application developer has a choice between compensation or not. We will discuss each one of them now, and the actual implementation of locking is discussed in the next chapter.

5.5.1. Call-level locking: compensation

This option is interesting in certain cases, such as:

- Application-level compensation allows more concurrency. If application-level compensation is used, then the system commits the local transaction in the database as soon as the invocation finishes (instead of waiting for the *two-phase commit*). For certain hot-spot applications this might be an interesting option, because this allows other invocations to proceed without having to wait for the root to terminate (because it is an open model).
- In certain cases, where the resource manager (database) does not provide a *two-phase commit* interface, it might be the only option to incorporate it into a composite system.

In this mode, invocation locks are maintained by *CheeTah*. In practice, this means that two invocations will conflict *unless the application developer explicitly states that they commute*, which is done in the server configuration file. This forces the application developer to think carefully about allowing concurrency. Otherwise correctness might be violated, for instance, if a compensation can not be applied due to the occurrence of interfering invocations. As an example: consider $inc(x)$ with compensation $inc(x)^{-1}$, where x is a number, and suppose there is an integrity constraint (inside the *RDBMS*) that does not allow x to be negative. Initially, let $x = 0$ and imagine the occurrence of $inc(x) <_e dec(x)$ before the root of $inc(x)$ is finished. If now that root has to abort, compensation of $inc(x)$ will be needed. However, this will not work anymore, since $inc(x) <_e dec(x) <_e inc(x)^{-1}$ will leave the database in a state that violates the integrity constraint (namely with $x = -1$), and for this reason the compensating invocation will fail. This shows that care has to be taken when specifying commutativity; knowledge about integrity constraints and triggers is needed. Fortunately, as pointed out in [Ber90], typical transactional applications only account for a small amount of different services, so specifying commutativity is only necessary between a limited number of invocation pairs.

Compared to established open models [Wei91], our methodology has the advantage that commutativity as well as compensation are purely *local*. This is a very important feature, making life easier for compensation-related semantics.

There are, however, a few drawbacks to this mode: first, independent access to the *RDBMS* is not allowed since the invocation-level locks have to ensure correct compensation, and independent access does not set these locks. For this reason, independent

local data access is not supported in this case. Another disadvantage concerns scalability. Indeed, whatever invocations arrive in the system, they all have to go through an invocation-level lock manager. Thus, if a cluster of servers is maintained on a replicated database (possibly using group communication protocols for efficient scale-up, such as in [Kem00]), then still every invocation has to go through a lock manager, which then becomes a bottleneck. It is precisely for this reason that we have chosen not to implement full lock inheritance in the classical way: if possible, invocation locks should be avoided and the scalability of the *RDBMS* underneath should be maximally exploited. With full lock inheritance, requiring notification of parent termination and re-evaluation of lock requests, this can only be done at the invocation level (because no *RDBMS* supports this).

In a nutshell, the applicability of compensation should be carefully considered, and the nature of the application should be kept in mind. If, later, scalable replication is to be used, it may be better to avoid this configuration.

5.5.2. Operation-level locking: no compensation

Currently, each *CheeTah* server has to enforce either compensation for all invocations, or for none of them. Non-compensatable mode is enforced by merely using a local transaction per invocation and having the *RDBMS* do all the locking. Programming is easier since no compensation needs to be defined. Also, commutativity is determined internally by the database system and independent local data access is possible (because the database's lock manager will see all accesses, independent of whether they go through a *CheeTah* server or not).

5.6. Implementation of atomicity

A global transaction is committed using a cascaded variant of 2PC: each server assumes the role of coordinator for all servers it invokes. To speed the process up, different servers are contacted in parallel: each communication round in the two-phase commit protocol is implemented by one separate thread per server involved. The two-phase commit protocol uses the root identifier as the label to indicate to each server which subtransactions are to be committed. Just like all other communication in *CheeTah*, 2PC happens through RMI. This solves problems with firewalls, because RMI calls can automatically be tunnelled through http. A negative acknowledgement (a NO vote) is implemented as a `RemoteException` being thrown. A read-only optimization, where a commit/abort notification is not sent to read-only nodes, is currently not implemented.

In addition, and also for reasons of efficiency, it is not always feasible to wait until the root decides to abort or commit. For instance, servers could be disconnected from the rest of the system or network partitions may occur. In those cases, and given that the system is built upon independent components, each server has the right to compensate local transactions on its own – as long as it has not heard from any global outcome. After the compensation all local locks and the call level lock can be released. The

“right” to undo a local transaction has to be constrained, otherwise a server could undo its local transaction during the time the global commit message travels through the system. Thus, when a server receives a prepare message and agrees to it, it loses the right to perform a server-side undo.

This approach is complicated by the fact that RMI does not provide *exactly once* semantics. More precisely, the failure of a remote call does not necessarily mean that it has not been executed. It could have been executed, leaving behind a locally committed transaction (t_1) and the corresponding call level locks set. The invoker, however, sees the call fail and may think the transaction has actually aborted. In that case, the server will eventually time out and undo the transaction locally, releasing the call level locks. This might result in incorrect executions if – on that server – later (successful) calls exist for the same root transaction. Let t_2 denote one such sibling subtransaction executed right after t_1 . Locally undoing t_1 with u_1 will only be correct if the sequence $t_1 t_2 u_1$ is equivalent to the sequence $t_1 u_1 t_2$ or $t_2 t_1 u_1$. In the theory chapters we have described this as the orphan problem. To avoid these and similar problems, we have taken an expeditive approach. When a server propagates a prepare request, it adds to the message the number of invocations it has made to a given server on behalf of the root transaction to be prepared. The server that receives the request checks this figure against its own. If they match, then the commit protocol proceeds. Otherwise the transaction will be aborted. Since in the latter case there are discrepancies about what has been done at each node, aborting seems to be the safest option.

To keep track of all the information needed to perform these operations, CheeTah relies on logging. For compensatable scheduling, each server keeps a log-table and an undo-table inside its local database. As soon as a transaction commits locally, the log-table reflects the fact that the transaction made local changes (needed after recovery). The undo-table contains the needed parameters for executing a compensation if necessary. All these data are written in the same transaction as the user’s logic, thereby reducing the number of database transactions to a minimum (one, in this case). For both the compensatable and the non-compensatable modes of operation an external file system log is used in order to track the *two-phase commit* status. Implementation specific details are given in the next chapter.

5.7. Dealing with undo operations

When run in compensatable mode programmer-defined compensation (alias undo) plays a significant role in CheeTah. Thus, a major question is whether we can always undo an operation. In this regard, it is important to emphasize that we rely on the service designer to provide the undo operation. Since in CheeTah the programmer only needs to worry about the data integrity of the local server, writing undo operations is relatively straightforward.

From the concurrency control point of view, executing an undo poses no problem because there is a lock on the corresponding service. If an undo operation needs to be executed, it will always be serialized immediately after the operation it is supposed to

undo. However, writing undo operations can be made quite complex by the underlying database system. The typical problems involve dealing with constraints and triggers. In general, as long as there are no non controllable side effects (triggers or constraints that the system – or its programmer – does not know about), CheeTah can handle these cases just like any existing system handles them. That is by blocking concurrent updates to the same items; so-called LSREC behaviour.

From our experience gained by working with CheeTah, the knowledge necessary to write undo operations can be compared to what a typical Oracle user has to know about isolation levels to ensure data consistency in the local database.

5.8. Optimizations of Logging and Locking

Any information that the undo operation may need, has to be persistently stored (logged). With CheeTah doing this is quite easy. The programmer only needs to insert any information needed for the undo into a *collection object*. This information can be the value of certain variables, the tables used for the undo, transaction id's, and so forth. When the transaction commits, this collection object is streamed out to the log table. In case an abort occurs, the system restores the transaction's undo stack. The undo operation can then read this object and proceed. This is significantly easier than what has to be done in existing TP-Monitors.

When creating the log entry, we use a very important optimization. If this information were saved as an insert into a logging table and eventually discarded by a corresponding delete, performance would be very poor. Rather, CheeTah uses a pool of undo entries in a fixed-size table (a parameter that can be changed if needed). This table is indexed on a numeric *index* entry. The server component keeps in RAM a list of available entries in the logging table and allocates them to a transaction when needed. Storing undo data is done by *updating* the log table, rather than *inserting* into it. Without this technique, we would never have been able to reach the current performance.

5.9. Summary

In this chapter we have introduced our *CheeTah* (black-box) framework, implemented in pure Java and supporting both compensation and non-compensatable invocations as defined in the theory part. Correctness of the resulting composite executions follows directly from the proofs in our theory. We have also shown that it is relatively easy to program compensation and define semantical information that is based on local effects only. This very much distinguishes the open model here from the open model classically used. In the next chapter we will proceed with a discussion of the design of the system, in terms of the most important modules and interfaces.

6. Implementation aspects of CheeTah

This chapter describes the most interesting features in the design of the *CheeTah* system such as lock management, log management, application-level interaction, propagation of 2PC and the network layer and system issues such as core interfaces. It is not meant as a full documentation of the system but rather as a brief summary of interesting design aspects. The illustrations in this chapter are expressed in UML [Gro].

6.1. Lock management

The system comes with a powerful package for lock management, which can be extended and adapted to specific environments. The main classes and interfaces are shown in Figure 6.1. A lock is set on behalf of a transaction, identified by a *GenericTID*. A lock itself is represented by a *LockDescriptor*, which has a name (the item to lock), exists on behalf of a transaction, and (if granted) has a time of granting. This is very abstract and can be used (subclassed) for virtually any situation where locks are needed. For instance, in case of composite transactions a serial mode indicator is needed for lock inheritance issues.

The most general interpretation of a lock manager is shown by *GenericLockManager*. It allows to set a lock (by supplying a descriptor and a timeout), provides unlocking both for a descriptor or for all locks of a given transaction, and user-definable semantics. This is possible with the *Semantics* interface, which makes the decision about whether two lock descriptors are compatible an externally decidable issue: the lock manager implementation calls the *compatible* method to check if two locks conflict or not. By providing a specific implementation of *Semantics*, an application can influence the behaviour of the lock manager to suit its needs.

6.2. Log management and recovery

Because only *two-phase commit* status tracking is needed, logging is significantly simpler than in heavy-weight systems [MHL⁺92]. The basis for logging is shown in Figure 6.2. A *LogManager* is an interface with two main methods: one for flushing *LogDescriptor* objects and one for retrieving the current history of relevant and previously flushed descriptors. Flushing is done during normal operation whereas the history is most relevant during recovery.

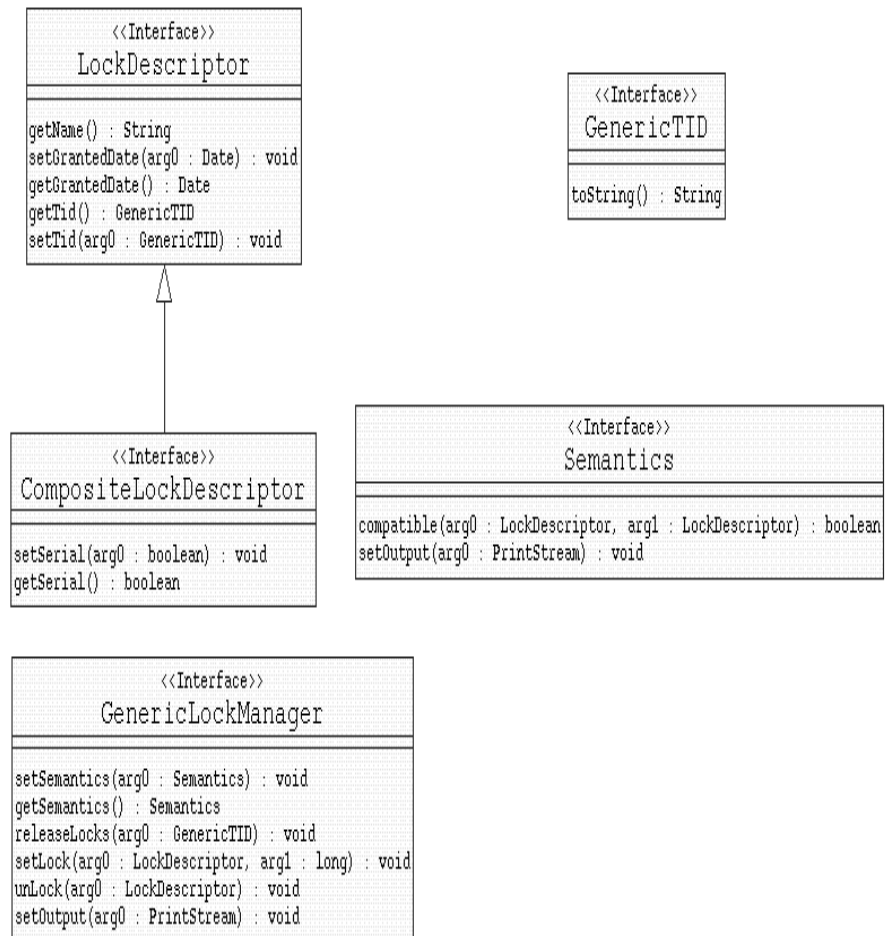


Figure 6.1.: Locking package

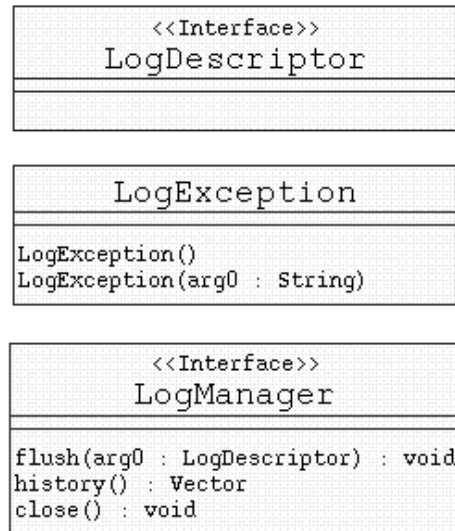


Figure 6.2.: Logging package

The *LogManager* is just the place where logging is done, it does not have to know about policies and interpretation of *LogDescriptor* objects. A *LogManager* can be provided by any kind of permanent storage, be it a *RDBMS* or file or anything else.

The concrete policy for recovery and interpretation of *LogDescriptor* objects is done by a *RecoveryAgent*, shown in Figure 6.3. It listens on state changes (of transactions in the system) and logs the appropriate events in a *LogManager*. The current version of the system uses a *FileLogMgr* that uses the file system as permanent storage for the log.

6.3. Application-level classes and interfaces

For the application developer the following topics are of relevance: how to interact with the transaction engine, how to implement a reusable application-level component (with business logic comparable to *Enterprise Java Beans (TM)* [EJB, DP00]), and how to make parallel remote calls in a service. Each one of these will be discussed separately.

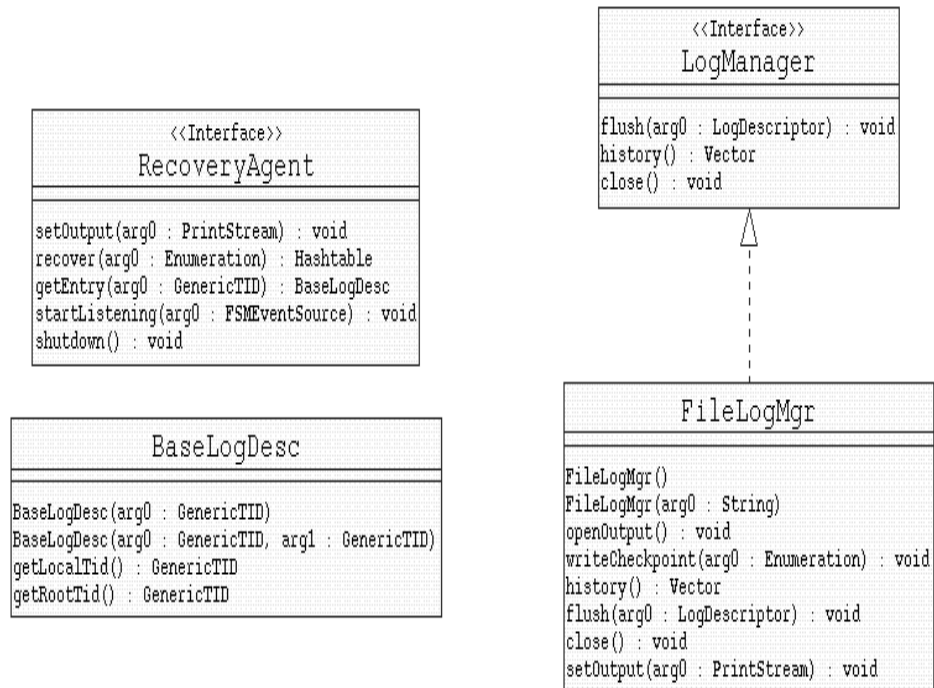


Figure 6.3.: Recovery package

6.3.1. Interaction with the transaction engine: CompositeUserTransaction

In order to allow the application to refer to the current (local) transaction, *CompositeUserTransaction* (Figure 6.4), an extension of the *Java Transaction API [JTA] UserTransaction* is provided, giving access to not just the standard functions in *Java Transaction API [JTA]* but also to advanced topics found in *CheeTah* systems. Its functionality allows setting a lock and supplying the proper compensation context (the collection object mentioned in the previous chapter for compensation services), setting and retrieving isolation modes for siblings and registering a synchronization callback allowing the application to be notified of the outcome of the current transaction.

6.3.2. Implementing a simple component

To implement a simple reusable component for *CheeTah*, it suffices to implement the interface *SimpleService* as shown in Figure 6.4. It forces the application to implement the two methods *setUserTransaction* and *setDataSource* so that the transaction engine can set the proper context for invocations of this service. The former sets the transaction handle for the application (as specified above) and the latter allows the application component to be unaware of which database is used, since it receives a general handle by the system with this method. This handle provides the necessary *JDBC API [JDB, WFC+99]* connections, needed for application-level interaction with the local data.

6.3.3. Implementing parallel calls

The *CheeTah* system comes with *SubTxThread*, an extension of the standard *Thread* class in *Java*, specially designed for executing parallel subtransactional calls. If the application wants to have parallelism it should implement the parallel task in a method called *exec()*, by implementing the interface *SubTxCode*. With this, any number of *SubTxThread* instances can be created and started (just like normal threads). For detecting termination of all parallel subcalls, a *Waiter* object can be used. It is considered bad practice *not* to wait for a call, since the computational model we use assumes that all subcalls have terminated when an invocation returns.

6.4. Propagation of 2PC

The *two-phase commit* is a cascaded implementation, where each client becomes a coordinator for the servers that it has invoked. To make the system sufficiently modular to be able to exchange the communication layer (currently RMI), a clear abstraction was necessary. This is shown in Figure 6.5: the transaction manager invokes an instance of the *Cascader* class, with a list of server references (of generic type *Object*). This *Cascader* creates a thread for each server reference, and delegates in each thread the calls to an instance of *Outgoing2PC* (which represents the network interface).

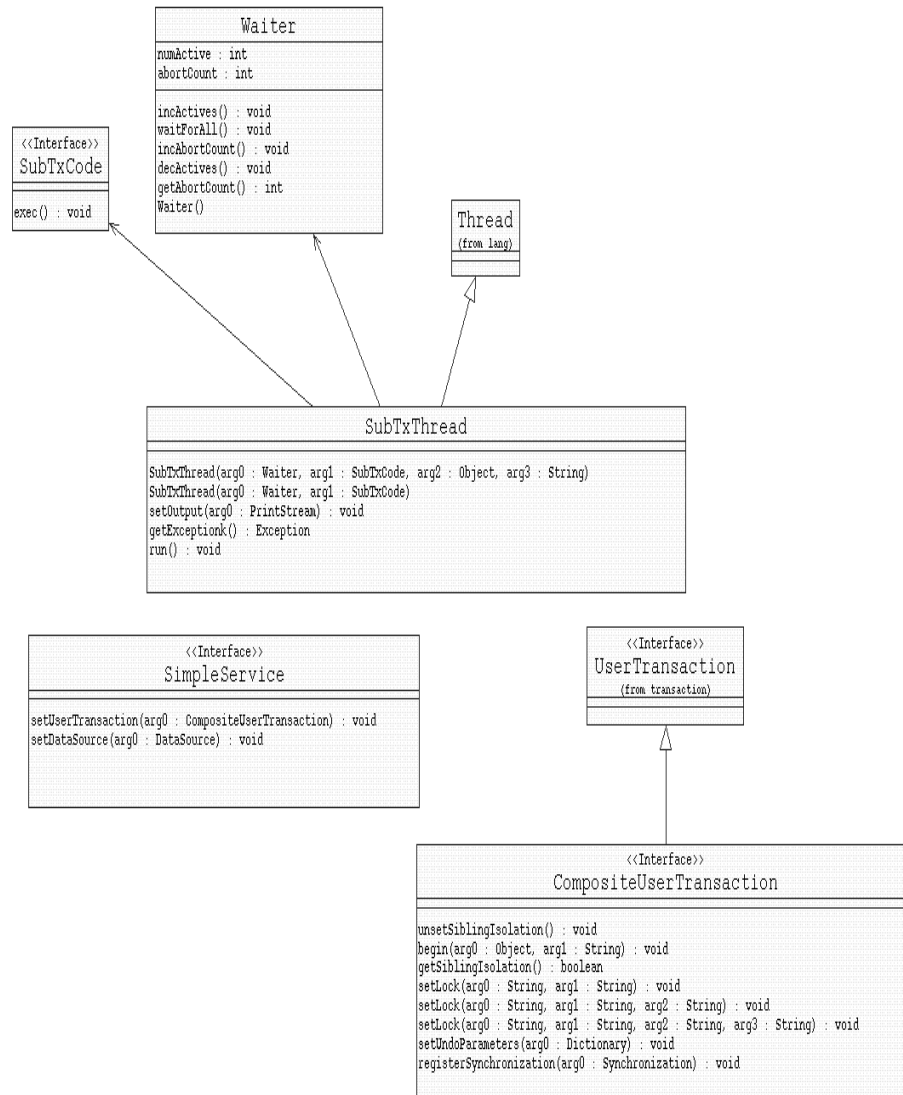


Figure 6.4.: Application package

On the other side of the communication there is the server listening for incoming *two-phase commit* calls. This is specified by both interfaces *IndoubtListener* and *Incoming2PC*. The *Outgoing2PC* implementation for a particular communication layer is responsible for mapping the abstract server references of *Object* type to the appropriate instances of these two interfaces. For instance, in the current RMI version a server reference is actually a *String* (the name on which the remote server is listening for RMI calls) and the *Incoming2PC* instance is obtained by looking up a reference to this name in RMI.

6.5. The resource (database) interfaces

6.5.1. Application level connections

Figure 6.6 shows how the server interacts with the local database. For the application logic a connection pool is maintained. In order to abstract the underlying database this pools creates a number of connections from a *ConnectionFactory* instance provided by the programmer. The working principle of connection pooling is according to the *Java Transaction API* [JTA] standard, where closing a connection in the application logic triggers notification of the pool and allows the connection to be reused from that moment on. The connection pool is not visible to the application logic because it is embedded in the *DataSource* instance that is supplied to the *SimpleService* instance.

6.5.2. System level interfaces

For the system interaction with the database, the interface *RecoverableResource* is provided, analogous to the *Java Transaction API* [JTA] *XAResource* version of the XA [Gro91] interface, but more general in the sense that also non-XA databases can be used. It has calls that can be invoked for important events during the lifetime of a particular transaction and that might need special actions from the data source. On failure of any of these methods, a special exception is given (*ResourceException*).

6.6. The core interfaces

The core interfaces are shown in Figure 6.7. Every sibling is represented by an instance of *CompositeTransaction* which contains the relevant information on a per-invocation basis. The interface *CompensatableTx* extends the base interface with undo-specific data, such as the index in the database undo table, the undo parameters (to be saved in that table) and an indication whether early commit has happened or not.

All local siblings of a given root are contained in a *RootSet*, which is responsible for enforcing the correct order in case of compensation, for maintaining the list of invoked servers as part of remote calls (needed for *two-phase commit*, which is also dealt with by the same instance), and for dealing with timed out transactions.

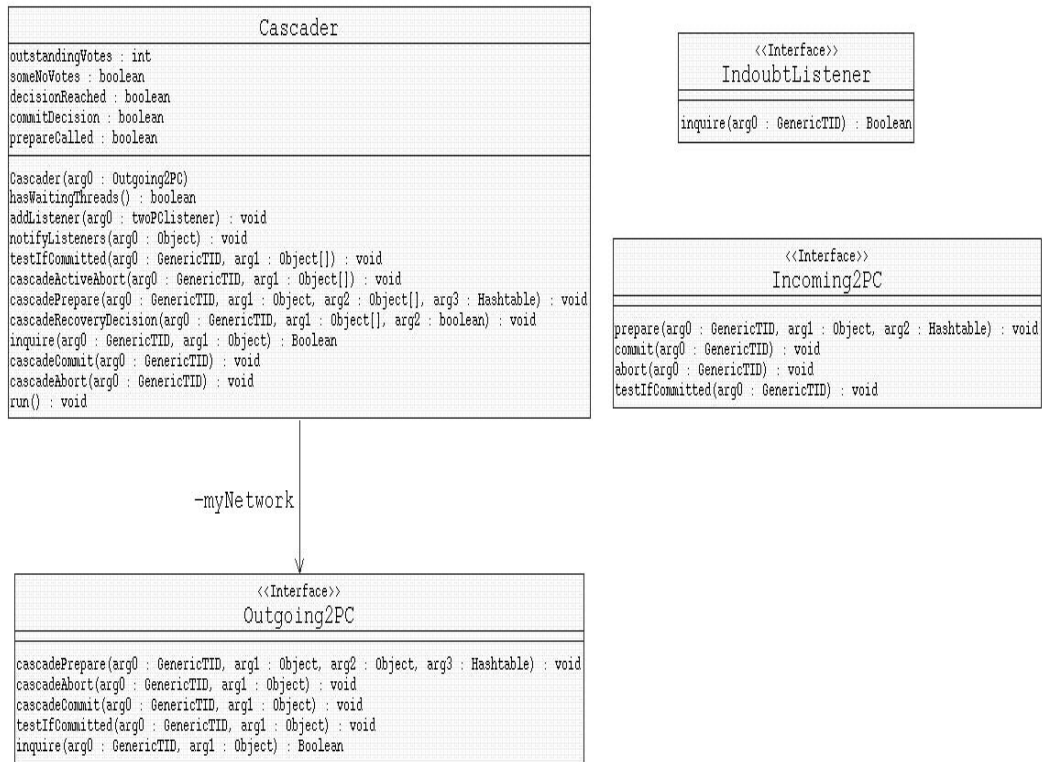


Figure 6.5.: Propagation package

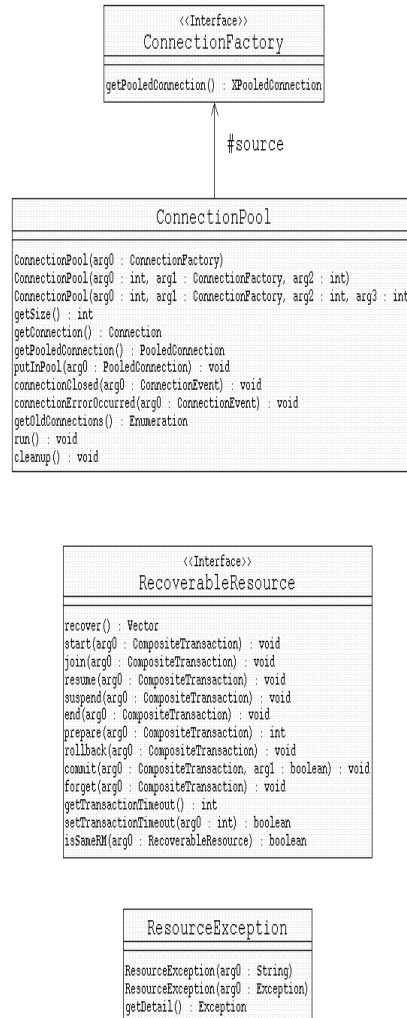


Figure 6.6.: Resource interfaces

The *RootSet* itself is contained by a *CompositeTransactionManager*, responsible for creating *RootSet* instances when necessary and delegating most of the other calls to it.

6.7. Summary

Space limitations prevent documenting the entire system (150 classes) in this text. After this chapter the reader should have an impression of the major design issues in the *CheeTah* system, such as locking, logging, *two-phase commit* propagation, core interfaces, and application-level interfaces. It should also be clear that it is not very difficult to exchange the semantics for locking, the network layer, the log facility, and the recovery strategy. Finally, it is worth mentioning that the current size of the java class archive is only about 500 KBytes (although there are many classes, most of them are relatively small). This fulfills the promise of a light-weight system.

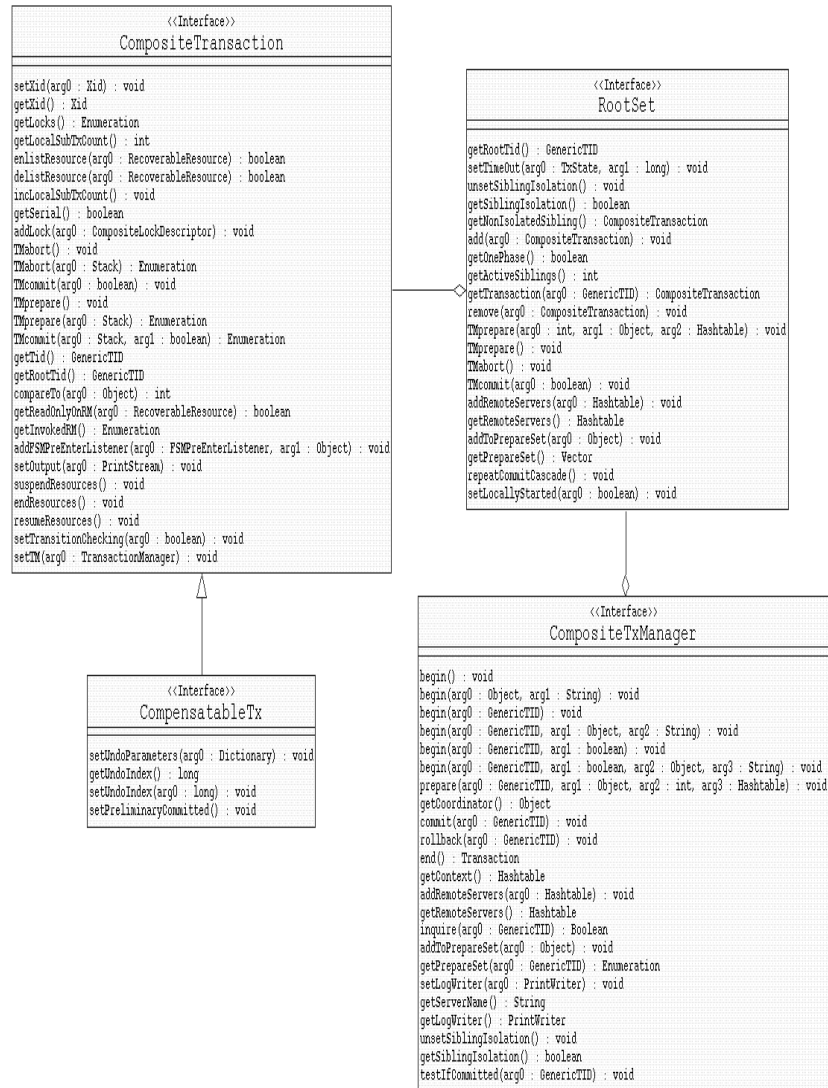


Figure 6.7.: Core package

7. Performance of CheeTah

7.1. Performance Analysis

The tests performed are based on the component shown in Figure 5.2. For the sake of simplicity, the server implements one single service invoked with different items as argument. The service is implemented as a Java program that performs a number of internal operations, including a short local transaction (updating the record with key *itemid*) and one service invocation *for each server on the next level*. Both the local transaction and the Java program itself have been kept as short as possible to make sure the measurements reflect the overhead caused by CheeTah and not that of the database or the JVM.

The experiments conducted were based on a total of 10 different system configurations (Figure 7.1, cases (a) to (d)), ranging from the simple wrapper mode ($1x1$) to complex invocation hierarchies ($3x3$ or $4x2$) including as well relevant structures like federations ($2x5$). The goal of the experiments was to analyze the performance of CheeTah and to better understand the impact of system depth and system width on the overall performance. For each root transaction, the depth of the system indicates the height of the transaction (how many levels until the leaves are reached). The width of the system indicates how many direct subtransactions each (sub)transaction has. Thus, for instance, in the $3x3$ configuration each subtransaction has three children (the root has three children and each one of these subtransactions has another three children). To indicate how many subtransactions are executed on behalf of a given root transaction, we use C , the *cardinality*. Thus, the $3x3$ configuration has a cardinality of 13 (the root, plus three children, plus three children for each child of the root). Since invocation of child subtransactions is done serially, the width of the system significantly affects the time it takes to execute a transaction.

In all but two experiments transactions conflict whenever they access the same item. Only for the configuration $4x2$ we tested conflict reduction through semantics on the higher level. We considered three cases: no semantics used, half of the servers can use semantics to make conflicts disappear, and all of the servers can make conflicts disappear.

Table 1 summarizes the common settings that apply to our tests. For each one of the tested configurations we measured throughput, response time and abort rate at the root level and also overall throughput (transactions per minute at all servers). The measurements are based on executions of 10.000 root transactions. For throughput, we measured the time the server needed per 100 transactions, yielding about 100 measure-

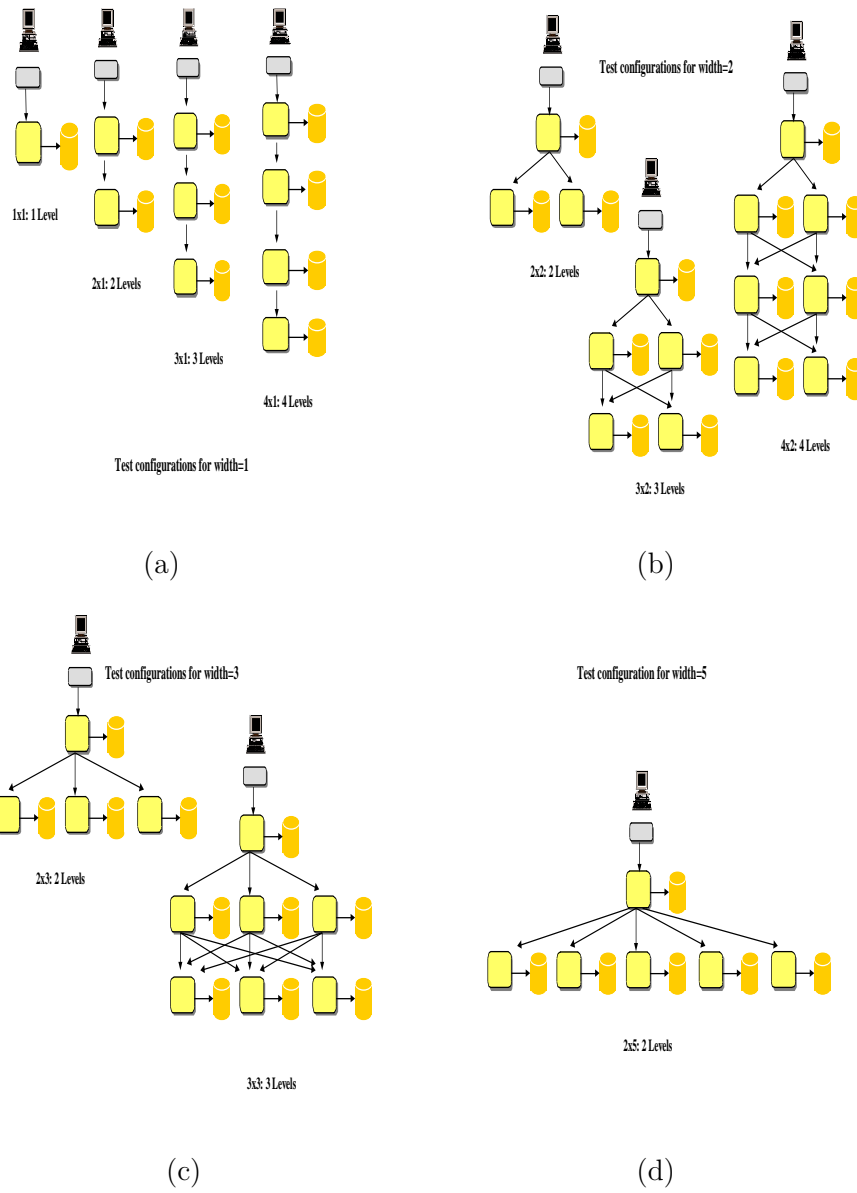


Figure 7.1.: The different configurations used for the tests: (a) Width=1 (b) Width=2 (c) Width=3 (d) Width=5

ments per experiment in case of our 10.000 roots. The average throughput was obtained by averaging these 100 throughput results for each experiment. The standard deviation represents the confidence (stability) that can be attached to the results. For response times, the average of each of the 10.000 roots' individual response times is given, as well as the standard deviation which reflects the confidence interval.

Node CPU	Sun Ultra 5, 269MHz UltraSparc Ili CPU
Node RAM	192 MB
Node OS	Solaris 2.6
Node interconnection	100Mbps Ethernet
Java platform	Sun JDK 1.2
Java VM heapsize	16 MB
RDBMS	Oracle 8.0.3
DB Buffer size	64 MB
Database size (per node)	10K tuples
Access path (per node)	unique index on primary key (itemid)
Access mode	80-20 (80% of transactions is on 20% of records)
JDBC Connection pool size (per component)	7
Number of concurrent top-level clients (roots)	25
Root inter-arrival time (per client)	0 (worst possible load)
Total number of <i>root</i> transactions per test run	10,000

Table 7.1.: General system parameters

It is important to emphasize that the tests were all performed on a *worst case scenario* basis. Transactions are made as complex as the system configuration, no semantic information is used to reduce conflict rates (except in two experiments where this technique was tested) and subtransactions are invoked serially. In addition, the load in the system is kept artificially high (as soon as one transaction finishes another one is submitted). The idea behind this approach is that if CheeTah can be made to work in such adverse circumstances, it will certainly work in more realistic, less demanding environments. In practice not all transactions will use all servers in the composite system (e.g., Figure 2.1), subtransactions can be executed in parallel and using semantic information will help to reduce abort rates. Any of these optimizations will improve the results obtained.

7.1.1. Measurements and Results

Table 2 contains the results of the experiments for the configurations in Figure 7.1 (the standard deviation for the overall throughput has been omitted for reasons of space; it is similar to that of the throughput at the root).

As expected, the throughput at the root decreases and the response time increases with system complexity (i.e., cardinality). This is easier to see in Figures 7.2.a and 7.2.b, which show the throughput and response time as a function of the cardinality. In terms of response time, the behaviour is obvious. A bigger cardinality implies more complex transactions that obviously take longer to complete. However, the linear relation observed in Figure 7.2.a (almost matching $RT = C \cdot RT_{1x1}$) demonstrates that the increase

Config	C	tpm (root) Avg	tpm Stdev	tpm (overall) Avg	RT Avg (ms)	RT Stdev (ms)	Abort Rate (%) Avg
1x1	1	1400	150	1400	520	500	0
2x1	2	1000	100	1991	1060	530	1
3x1	3	870	160	2647	1300	380	1.7
4x1	4	720	170	2872	1700	510	5.7
2x2	3	800	60	2402	1650	700	1.6
3x2	7	400	80	2872	3200	1100	7.1
4x2	14	175	25	3096	6500	1100	31
2x3	4	720	70	2882	1800	600	2.1
3x3	13	200	20	2924	5900	1000	22
2x5	6	410	20	2485	3000	800	5.7

Table 7.2.: Results for different configurations

in response time is directly related to the complexity of the transaction. Therefore, CheeTah does not add additional overhead as the system becomes more complex. This is surprising since one would expect that longer transactions would block more resources and, therefore, would add more overhead. In practice CheeTah behaved very well. Observing each individual server it turned out that the open nested policy allowed servers to free resources quite quickly. The policy of aborting transactions as soon as they run into a conflict also helped in that these transactions could be quickly restarted and, with high probability, succeeded the second time. By aborting them early the overall delay introduced was minimal even in those cases with high abort rates ($3x3$ and $4x2$).

The throughput results are also interesting. As the lower curve in Figure 7.2.b shows, the throughput at the root quickly decreases with cardinality. This is an artifact of the experimental setting (we maintained a fixed number of root transactions in the system at all times; the more complex the transaction, the longer it takes to complete and, therefore, the less transactions entering the system). We observed that, even for the highest cardinalities, the servers had enough spare capacity to run additional transactions. This is clear from the results for overall throughput. As the upper curve in Figure 7.2.b shows, the overall throughput increases as we go from cardinality 1 to 3 and then remains stable (the bump in the curve is a result of the different configurations; the low point being cardinality 6, in the $2x5$ case). Again, this proves that system complexity does not affect performance. In reality, as the system becomes more complex, there is more processing capacity. If transactions follow different paths from the root to the leaves, then the system will be able to process many more transactions. In fact, in our experiments, we only saturated the server in the $1x1$ case.

These results show that CheeTah is very close to having optimal performance. For the range of configurations tested the response time directly depends on transactional complexity and the throughput remains stable. Both parameters are unaffected by the complexity of the composite system.

For high cardinality the number of aborted transactions is very high (31 % for the $4x2$ and 22 % for the $3x3$ configuration). These results, however, are due to the worst case

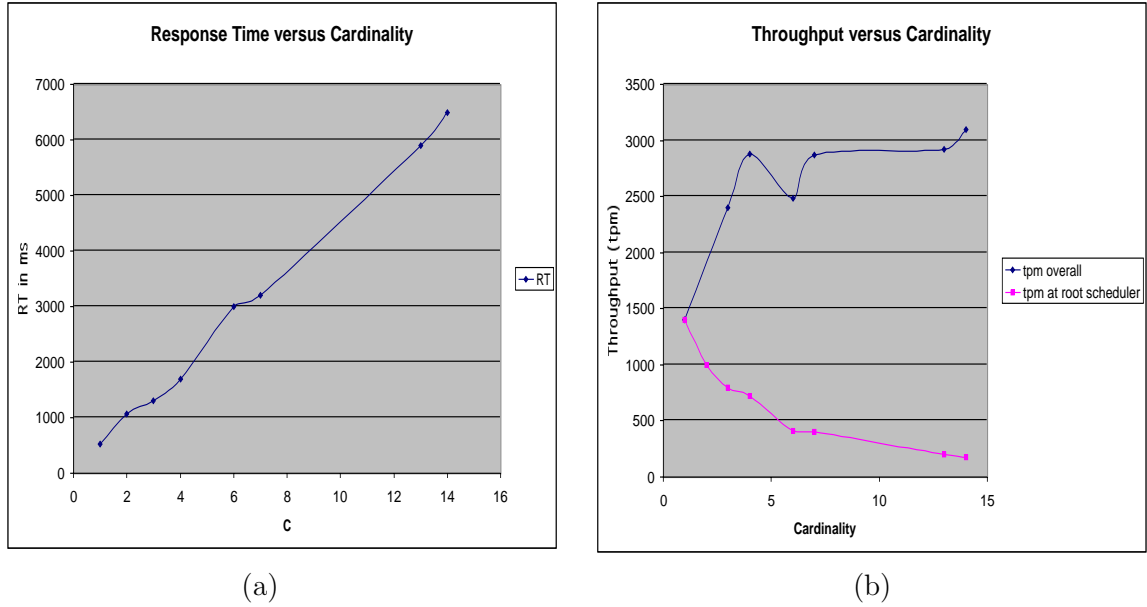


Figure 7.2.: (a) Response Time vs Cardinality and (b) Throughput vs Cardinality

scenario we used for tests and not to any characteristic intrinsic to CheeTah. Any system with this kind of load would have similar abort rates. In more realistic applications, these numbers will go down significantly. To test this hypothesis, we ran a set of experiments using semantic information to reduce conflict rates. These experiments reflect the fact that in composite systems conflicts are rare at the higher levels and, if they occur at all, tend to happen at the lower levels, when accessing the databases. The experiments are performed for the 4×2 configuration assuming all transactions accessing the same item conflict (as in Table 7.1.1), eliminating half of these conflicts and then eliminating these conflicts entirely. The results are shown in Table 7.1.1.

Configuration (Levels x Width)	C	tpm Avg	tpm (overall) Avg	tpm Stdev	Resp.Time Avg (ms)	Resp.Time Stdev (ms)	Abort Rate (%) Avg
4x2 (all conflict)	14	175	3096	25	6500	1100	31
4x2 (half conflict)	14	210	3300	30	6000	1200	17
4x2 (no conflicts)	14	200	3100	20	6700	1200	1.25

Table 7.3.: Results with high-level semantics advantage

These results show that once transactions follow different paths and subtransactions are executed in parallel the inter-arrival rate will be more evenly distributed and the percentage of aborts will drop significantly. Interestingly, throughput and response times do not seem to improve by exploiting semantics. This is due to the fact that transactions that abort are aborted very quickly (because of conflicts at the higher levels in the

system) and then restarted, thereby incurring a minimal penalty in terms of response time. Similarly, and as pointed out above, since the system is far from its saturation point, abort rates have no influence on the throughput.

7.1.2. Comparison with Existing Systems

The previous results show that CheeTah has close to an ideal performance behaviour as the system complexity increases. In order to evaluate the basic overhead of CheeTah we compared similar systems implemented in Oracle8 and Encina. To obtain clear results we compared the *1x1* case.

Encina provides us with a yardstick to test CheeTah against tools used in 3 tier architectures. The results (Figure 7.3) show that CheeTah favourably compares with Encina. In the *1x1* configuration CheeTah achieved more than three times the throughput reachable with Encina. The reason is that CheeTah performs all the transaction management inside the server and, unlike Encina, no context changes are necessary to access the different modules of the TP-monitor. We corroborated these results by also comparing performance in the *2x2* case. For root transactions CheeTah again outperformed Encina, a fact that becomes very clear when the overall throughput in the system is considered. Again, this is due to the fact that Encina, like all existing commercial products, uses a centralized component for transaction management. Thus, distribution does not bring much in terms of overall performance since the centralized component is the bottleneck and it cannot be distributed. This is where CheeTah excels: each component has its own transaction manager and, therefore, the more components, the more distributed the load is on the transaction manager functionality. These results clearly speak in favour of the language framework approach followed in CheeTah.

The comparison with Oracle gives us a way to test the performance of CheeTah against 2 tier architectures where the services are embedded into a database. For the test, the equivalent to a CheeTah server was implemented inside Oracle using the JVM provided. The clients connected to this internal server via a pure JDBC interface and connection pooling (of the same size as in CheeTah) was used. The results are shown in Figure 7.4. If we consider peak rates instead of average numbers (to eliminate the overhead due to the current behaviour of the JDBC interface to Oracle), the performance of CheeTah is comparable to that of Oracle.

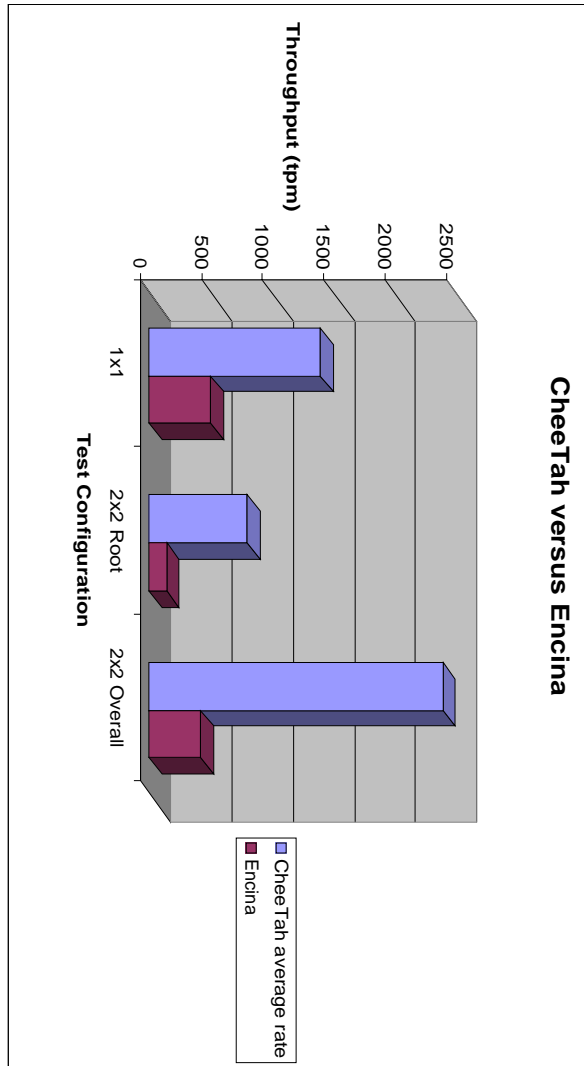


Figure 7.3.: CheeTah vs Encina

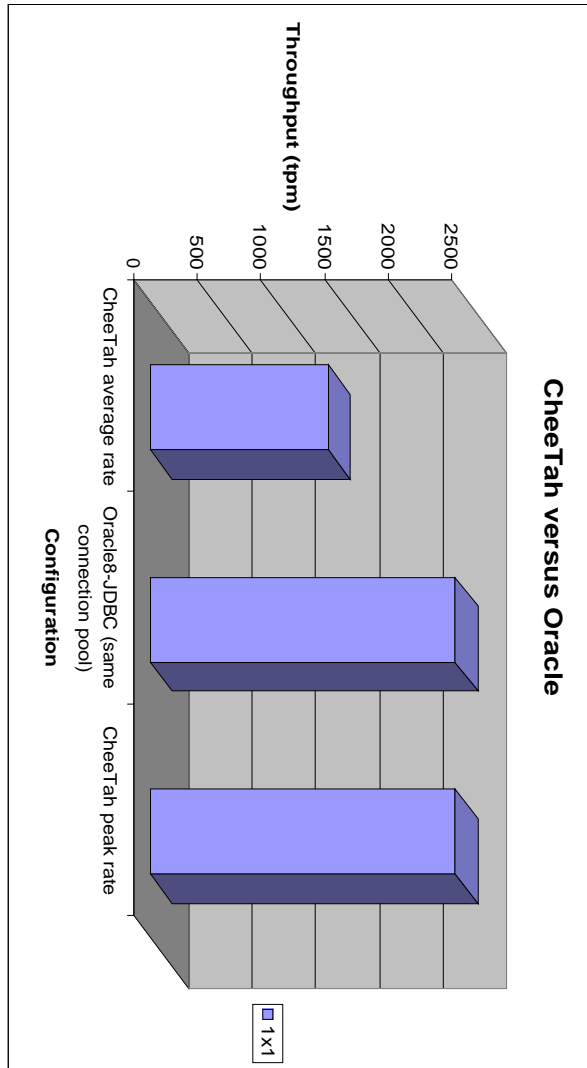


Figure 7.4.: CheeTah versus Oracle

8. Conclusions

8.1. Transactional theory

With respect to transactional theory this text presents a solid and clear extension to both open and closed nested transaction models for arbitrary configurations of independent schedulers. The prevailing distributed transaction model of flat transactions and at most one site access per global transaction is replaced with a much more powerful model. This model increases fault-tolerance, flexibility, response time and independence among servers.

It is important to note that for distributed transactions open models can only be applied when restricted to purely local transactions. No semantics are known for remote calls, and therefore conflicts can not ‘disappear’ across different schedules as is possible in, for instance, multi-level transactions[Wei91].

Concerning the minimal information needed between schedulers in such systems, we have shown that one can not do without any information about the root. The property SREC requires at least the possibility to determine if two invocations are of the same root, and notification upon termination of that root. Because a nested transaction model allows certain invocations to fail and be aborted independently by timeout in case of communication failures, there must also be a way for the committed root to indicate which sets of siblings should be included in the commit set. In the protocols proposed, this is done by passing the identifier of any servers invoked, as well as the number of subtransactions detected by the committing root. If the server disagrees on that number then commit will fail. This solution covers both the lock inheritance as the blocking siblings solution. Another approach could be to pass the identifiers of each sibling up to the root and have the commit notification carry this explicit list of identifiers. This course, however, requires more information to be exchanged, and was therefore avoided. In case of lock inheritance, isolation problems of concurrent siblings have to be avoided. The approach offered here is to have no concurrent siblings in that case, implying serial execution of the whole root. This requires the inclusion of a tag that indicates which execution mode the root has chosen, so that each scheduler can detect whether it is expected to allow conflicts among siblings or not. To detect recursion, a source of possible problems, one additionally needs the list of servers directly or indirectly invoked during each remote call. Finally, the fact that each server should be autonomous and be able to abort a timed out invocation implies that global atomicity can only be reached by a distributed consensus, such as *two-phase commit*.

8.2. Programming distributed transactional applications

The prototype provides a very simple and yet powerful way of programming distributed transactional applications. It is based on application-level components that are independent of the actual location, and therefore portable. In principle an application component can be implemented by anyone, and installed on any *CheeTah* location. Furthermore, because of its compatibility with existing programming standards and database interfaces, no significant effort is required to learn how to program *CheeTah* systems. The overall system is very light-weight, making the traditional installation and setup of existing systems irrelevant. Due to the small code archive size of less than one megabyte, a minimal infrastructure is sufficient for installing and running *CheeTah*. These properties make a strong argument for using *CheeTah* systems for pluggable *Internet* servers.

8.3. Performance

The performance of the proposed *Java* implementation is comparable to what existing, platform-specific binaries can achieve in classical transaction environments. Additionally, the scalability characteristics of both the *Java* platform and *CheeTah* make it a very powerful alternative if a light-weight implementation is preferable.

8.4. Two-tier, three-tier and multi-tier systems

During several decades of distributed computing history, a number of different paradigms for distributed software architectures have been proposed. Among the earliest was the so-called *two-tier* architecture between a client and a server, as shown in Figure 8.1. Classical client-server architectures are typical examples of this configuration. In this architecture, presentation is at the client, data is at the server, and application logic is divided between both, leading to different degrees of either heavy clients or heavy servers. An example of an industrial heavy server advocate is *Oracle (TM)*.

A number of problems with this architecture, such as a combinatorial explosion of connectivity complexity and scalability limitations, have led to the *three-tier* model, as in Figure 8.2. There, the application logic and connection logic, as well as transactional co-ordination are put in an intermediate layer. For connecting N clients to M servers at most $N + M$ different protocols are necessary, against $N \times M$ for two-tier systems. Thus, all interaction happens through the middle layer, therefore called *middleware*. More clients can be handled by adding more application servers. This is the prevalent architecture today, and it is centralized in nature.

The next logical step for decentralization is *n-tier or multi-tier*, shown in Figure 8.3. Now, services are built along different components in the system thereby departing from centralization. In this architecture there are clients, servers and any number of *application servers* that co-operate to build incremental services. This is exactly

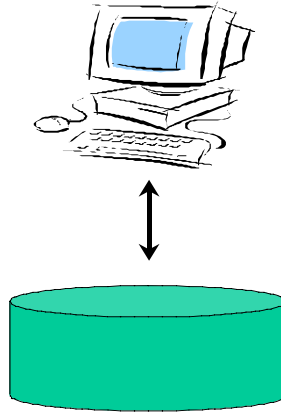


Figure 8.1.: Two-tier architecture

the architecture of a composite system and this work offers the technological basis for transaction management in these environments.

8.5. Possible extensions

This work can be seen as a basis for a number of possible extensions, both theoretical and practical.

8.5.1. Theory issues

As briefly mentioned, asynchronous transaction management and persistent queues fall under the kind of models we describe in chapter 3. Therefore, correctness as discussed in this work also applies to these environments. The protocols in chapter 4, however, are oriented towards synchronous environments. A possible extension could be to investigate the influence of asynchronous models on the protocols.

Another possible extension applies to *group communication* protocols [Kem00, MMSA⁺96], to enforce the same serialization orders among different sites. The problem here, however, is that most group communication systems are centralized themselves, which defeats the purpose of composite systems in the first place.

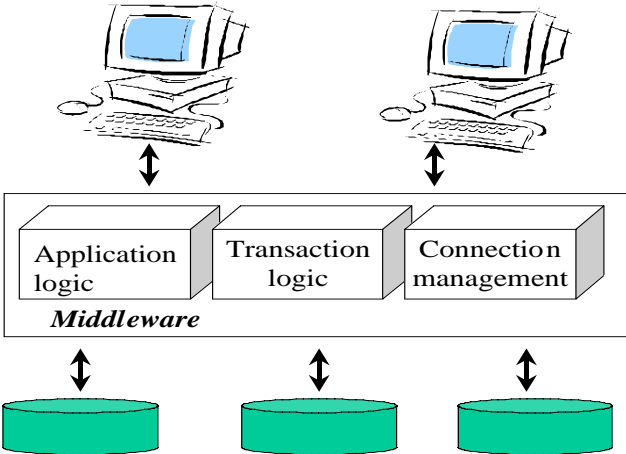


Figure 8.2.: Three-tier architecture

8.5.2. Practical issues

With the resulting *CheeTah* prototype, initial and promising results have been shown for composite systems. For real applicability, the issues of security and load balancing are essential [Gor00, Buy99]. Furthermore, the incorporation of current standards such as *Enterprise Java Beans (TM)* [EJB, DP00] may be a practical advantage. These issues are pending and interesting practical extensions.

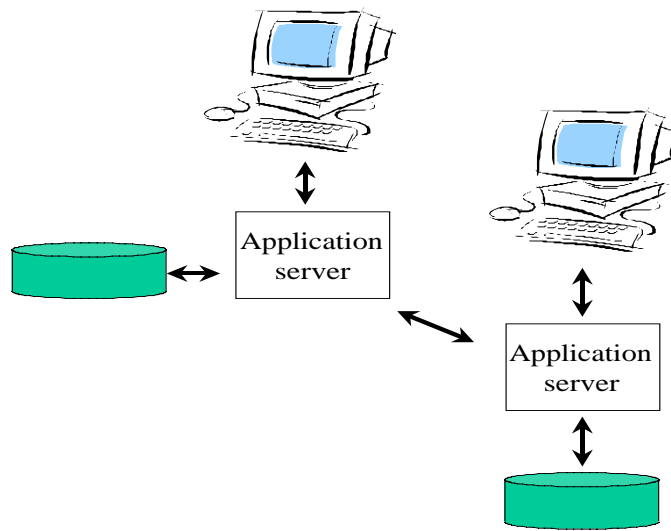


Figure 8.3.: N-tier architecture

Bibliography

- [ABFS97] Gustavo Alonso, Stephen Blott, Armin Fessler, and Hans-Joerg Schek. Correctness and Parallelism in Composite Systems. In *PODS97*, 1997.
- [ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.
- [ACM94] J.M. Andrade, M.T. Carges, and M. R. MacBlane. The TUXEDO system: an open on-line transaction processing environment. *Data Engineering Bulletin*, 1994.
- [AFPS99a] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Correctness in general configurations of transactional components. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99)*, Philadelphia, PA, May 31 - June 2 1999.
- [AFPS99b] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Transactions in Stack, Fork and Join Composite Systems. In *Int. Conference on Database Theory*, 1999.
- [AHCL97] Y.J. Al-Houmaily, P.K. Chrysanthis, and S.P. Levitan. An Argument in favor of the Presumed Commit Protocol. *Proceedings of 13th Int. Conf. on Data Engineering*, 1997.
- [AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. Abbadi, H. Schek, and G. Weikum. Unifying Concurrency Control and Recovery of Transactions, 1994.
- [BBG89] Catriel Beerli, Philip A Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.
- [Ber90] Bernstein. Transaction Processing Monitors. *Communications of the ACM*, 33(11), 1990.
- [BG82] Philip Bernstein and Nathan Goodman. A Sophisticate's Introduction to Distributed Database Concurrency Control. In *Procs. 8th Int. Conference on Very Large Data Bases; Mexico City*, 1982.

- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
- [BHM90] P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Procs. of ACM SIGMOD*, 1990.
- [BK99] K. Boucher and F. Katz. *Essential Guide to Object Monitors*. John Wiley, 1999.
- [BN97] Bernstein and Newcomer. *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufman, 1997.
- [BSW88] Catriel Beeri, Hans-Joerg Schek, and Gerhard Weikum. Multi-Level Transaction Management, Theoretical Art or Practical Need? In *International Conference on Extending Database Technology*, 1988.
- [Buy99] Rajkumar Buyya. *High Performance Cluster Computing*, chapter 36. Prentice Hall, 1999.
- [D.G94] M. Rusinkiewicz D. Georgakopoulos. Using Tickets to Enforce the Serializability of Multidatabase Transactions. *IEEE Transactions on Knowledge and Data Engineering*, pages 393–481, February 1994.
- [DGV94] Laurent Dayns, Olivier Gruber, and Patrick Valduriez. On the Cost of Lock Inheritance in Lock Managers supporting Nested Transactions. In *Proceedings of 10th BDA*, 1994.
- [DP00] S. Denninger and I. Peters. *Enterprise JavaBeans*. Addison Wesley, 2000.
- [DRD99] Lyman Do, Prabhu Ram, and Pamela Drew. The Need for Distributed Asynchronous Transaction Management. In *Procs. of ACM SIGMOD*, 1999.
- [EJB] Enterprise JavaBeans Technology. <http://java.sun.com/products/ejb/index.html>.
- [Elm90] Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufman, 1990.
- [ELWR82] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. *Proceedings of VLDB*, 1982.
- [EMS91] J.L. Eppinger, L.B. Mummert, and A.Z. Spector. *Camelot and Avalon: A Distributed Transaction Family*. Morgan Kaufman, 1991.
- [Enc] Transarc Corporation. *Encina Transactional C Programmer's Guide and Reference*.

- [Fes00] Armin Fessler. *A Generalized Transaction Theory for Database and Non-Database Tasks*. PhD thesis, Swiss Federal Institute of Technology Zürich, 2000.
- [FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association of Computer Machinery*, 32(2):374–382, April 1985.
- [FSJ99] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks*. Wiley, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [Gor00] Ian Gorton. *Enterprise Transaction Processing Systems*. Addison Wesley, 2000.
- [GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.
- [Gra78] Jim Gray. Notes on Database Operating Systems. *Operating systems: an advanced course. Lecture notes in computer science*, 60:393–481, 1978.
- [Gra81] J. Gray. The Transaction Concept: Virtues and Limitations. In *Procs. 7th Int. Conference on Very Large Data Bases; Cannes, France*, 1981.
- [Gra98] Mark Grand. *Patterns in Java*. John Wiley, 1998.
- [Gro] The Object Management Group. UML Resource Page. <http://www.omg.org/technology/uml/>.
- [Gro91] The Open Group. Distributed Transaction Processing: The XA Specification. Technical report, X-OPEN, 1991.
- [GS90] G.Weikum and H.J. Schek. *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann, 1990.
- [Has96] H. Hasse. *A Unified Theory for Correct Parallelization and Fault-Tolerant Execution of Database Transactions*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1996. In German.
- [IMP94] V. Issarny, G. Muller, and I. Puaut. Efficient Treatment of Failures in RPC Systems. In *Procs. 13th Int. Symposium on Reliable Dist. Systems*, 1994.
- [JDB] Java JDBC API. <http://java.sun.com/products/jdbc/index.html>.
- [Joh] Ralph Johnson. Frameworks Home Page. <http://st-www.cs.uiuc.edu/users/johnson/frameworks.html>.

-
- [JTA] Java Transaction API. <http://java.sun.com/products/jta/index.html>.
- [Kem00] Bettina Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology Zürich, 2000.
- [KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Procs. 16th Int. Conference on Very Large Data Bases; Brisbane, Australia*, 1990.
- [KS88] H.F. Korth and G.D. Speegle. Formal Model of Correctness Without Serializability. In *Proceedings of ACM SIGMOD*, 1988.
- [Kum96] Vijay Kumar, editor. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, 1996.
- [LAA94] M. Liu, D. Agrawal, and A. Abbadi. The Performance of Two-Phase Commit Protocols in the Presence of Site Failures, 1994.
- [LEK99] J. Lyon, K. Evans, and J. Klein. Transaction Internet Protocol (TIP). Technical report, Tandem and Microsoft, February 1999.
- [Lis88] Barbara Liskov. Distributed Programming in ARGUS. *Communications of the ACM*, 31(3), 1988.
- [LMWF94] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufman, 1994.
- [LS76] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Computer Science Laboratory, Xerox, Palo Alto, 1976.
- [Mal94] Susan Malaika. A Tale of a Transaction Monitor. *Data Engineering Bulletin*, 1994.
- [MB97] McKnight and Bailey. *Internet Economics*. MIT Press, 1997.
- [MD94] C. Mohan and D. Dievendorff. Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queueing. *Data Engineering Bulletin*, 1994.
- [MHL⁺92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks using Writeahead Logging, 1992.
- [ML83] C. Mohan and B. Lindsay. Efficient commit protocols for the Tree of Processes Model of distributed transactions. *Proceedings of 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, 1983.
-

- [MMSA⁺96] L.E. Moser, P.M. Melliar-Smith, D.A. Agrawal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, April 1996.
- [Moh98] C. Mohan. Transaction Processing and Distributed Computing in the Internet Age. <http://www-rodin.inria.fr/mohan/abstracts.html>, July 1998.
- [Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
- [Mos87] J.B. Moss. Log-based Recovery for Nested Transactions. In *Procs. 13th Int. Conference on Very Large Data Bases; Brighton*, 1987.
- [MRW⁺93] Peter Muth, Thomas C. Rakow, Gerhard Weikum, Peter Broessler, and Christof Hasse. Semantic Concurrency Control in Object-Oriented Database Systems. In *9th IEEE Conference on Data Engineering*, 1993.
- [Mul93] Sape Mullender. *Distributed Systems*. Addison Wesley, 1993.
- [OTM] Orbix OTM. Orbix OTM documentation. <http://www.iona.com>.
- [OV91] M. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [OW99] Scott Oaks and Henry Wong. *Java Threads*. O' Reilly, 1999.
- [PA00] Guy Pardon and Gustavo Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *Proc. of Very Large Databases*, 2000.
- [SAS99] H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 316–326, Philadelphia, Pennsylvania, USA, May/June 1999. ACM Press.
- [Sch94] Bruce Schneier. *Applied Cryptography*. John Wiley, 1994.
- [Sch96] Werner Schaad. *Transaktionsverwaltung in Heterogenen, Foederierten Datenbanksystemen*. PhD thesis, ETH Zuerich, 1996.
- [Ske81] D. Skeen. Nonblocking Commit Protocols. *Proceedings of ACM SIGMOD*, 1981.
- [Spe97a] CORBA Specification. Object Transaction Service. Technical report, OMG, 1997.
- [Spe97b] CORBA Specification. The CORBA Specifications. Technical report, OMG, 1997.

- [Sri97] Prashant Sridharan. *Advanced Java Networking*. Prentice Hall, 1997.
- [SW00] Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federations. *Proceedings of Cooperative Information Systems (CoopIS)*, 2000.
- [SWI] Java SWING. <http://java.sun.com/products/jfc/index.html>.
- [SWS91] H.J. Schek, Gerhard Weikum, and Werner Schaad. A Multi-Level Transaction Approach to Federated DBMS Transaction Management. In *First Intl Workshop on Interoperability in Multidatabase Systems*, 1991.
- [VHYBS98] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, 190(2):363–396, 20 January 1998.
- [WC93] J. Widom and S. Ceri. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In *Procs. 19th Int. Conference on Very Large Data Bases; Dublin, Ireland*, 1993.
- [Wei88] W. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12), 1988.
- [Wei89] W. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11, 1989.
- [Wei91] Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM TODS*, 1991.
- [WFC⁺99] White, Fisher, Cattell, Hamilton, and Hapner. *JDBC API Tutorial and Reference, Second Ed.* Addison Wesley, 1999.
- [YHMA92] Y.Breitbart, H.Garcia-Molina, and A.Silberschatz. Overview of multi-database transaction management. In *VLDB Journal*, February 1992.

A. Appendix: Selected examples of CheeTah application code

A.1. Programming a service component

The following shows an example of a *CheeTah* application component. This corresponds to the programming effort that is required to program a transactional application in *CheeTah*. The component is implemented as a *Java* class that adheres to a predefined interface called *SimpleService*. This interface enforces two methods that are used for callbacks by the transaction manager part of the framework: *setUserTransaction* and *setDataSource*. The former is used for supplying a handle to the transactional properties of an invocation, and the latter serves to provide a way for the application to get connections to the underlying *RDBMS*, without actually having to know the details of each connection (setup, username and password, etc.). By convention, a method name prefixed with ‘do.’ is a transactional method. In the example, this is the case for the method called *do_buy*. The implementation of this method executes some standard *JDBC API* [JDB, WFC⁺99] calls on a connection obtained by the data source supplied by the server, through the *setDataSource* method. Also, it executes two remote calls, one on a *Credit_Client* stub, another on an *AddDelivery_Client* stub. These stub classes are obtained from the remote server’s web page, and are responsible for constructing the *Transactional Remote Method Invocations (TRMI)*. In this example, no remote exceptions are caught independently, meaning that remote failures in this case will also lead to a failure of the local invocation. For extra safety, such exceptions lead to the invalidation of the local transaction, by calling *setRollbackOnly* on the transaction handle supplied by *setUserTransaction*. This method has different semantics from the *Java Transaction API* [JTA] equivalent: here, it no longer implies a global abort of a flat transaction, but rather the local abort of the invocation. Because an invocation is a subtransaction of a remote client, it does *not* have the right to force global abort. Only local aborts can be enforced. In order to try alternatives on remote failures, each of the remote calls would have had to be in its own *try ... catch* block, allowing a remote failure to be caught individually. Whatever way *do_buy* is exited, in any case the connection is closed in the *finally* block. This is necessary because the server supplies a connection out of a *pool* of connections, and otherwise the pool would never be reusing connections: the *close* call notifies the pool manager that the connection can be reused.

```
import java.util.*;
```

```
import ch.ethz.inf.lockmanager.*;
import java.rmi.*;
import ch.ethz.inf.cheetah.application.*;
import javax.sql.*;
import java.sql.*;

public class Buy implements SimpleService{
    private DataSource myDS=null;
    private CompositeUserTransaction myUT=null;
    public Buy(){}
    public void setUserTransaction(CompositeUserTransaction ct){
        myUT=ct;
    }
    public void setDataSource(DataSource ds){
        myDS=ds;
    }
    /**
    *Buy a book.
    *@param id The id of the book
    *@param number The number of copies to buy
    *@param CC Credit card no
    *@param holder Holder's last name for CC
    *@param address Address for delivery
    *@param expDate The expiry date for CC
    */
    public void do_buy(long id, int amount, String CC,
        String holder, String address, String expDate)
        throws Exception{

        Connection c=null;
        Statement s=null;
        long price=0;
        RemoteCredit rc=null;
        RemoteAddDelivery rad=null;
        long orderID=System.currentTimeMillis();
        try{
            c=myDS.getConnection();
            s=c.createStatement();
            ResultSet rs=s.executeQuery("select price from "+
                "bookstock where id="+id);
            while (rs.next()){
                price=rs.getLong(1)*amount;
            }
            s.executeUpdate("update bookstock set avail=avail-"+
```


A.3. Providing intra-service parallelism

Below is the implementation of a *Buy* service again, now with a parallel implementation. In particular, two nested classes are defined, *CreditThread* and *DeliveryThread*, both implementing the interface *SubTxCode*. Concretely, this means that both implement a method called *exec*, that performs the actual threaded code. Instances of these classes can be used to create a number of *SubTxThread* objects, subclasses of the *Java Thread* class. The constructor for these takes an instance of *SubTxCode*, as well as a *Waiter* object, to detect termination of all threads. Invoking the *start* method on each instance will cause the subtransaction threads to execute, and their *exec* methods will be called.

```
import java.util.*;
import ch.ethz.inf.lockmanager.*;
import java.rmi.*;
import ch.ethz.inf.cheetah.application.*;
import javax.sql.*;
import java.sql.*;

public class ParBuy implements SimpleService{
    private DataSource myDS=null;
    private CompositeUserTransaction myUT=null;

    public ParBuy(){
    }
    public void setUserTransaction(CompositeUserTransaction ct){
        myUT=ct;
    }
    public void setDataSource(DataSource ds){
        myDS=ds;
    }
    /**
    *Register payment for a credit card purchase
    *@param CC The credit card number
    *@param name Name of CC holder
    *@param expDate Expiry date of card
    *@param amount Number of copies to order
    *@param id ID of book to order
    */
    public void do_buy(long id, int amount, String CC, String holder,
        String address, String expDate) throws Exception{
        Waiter waiter=new Waiter();
        Connection c=null;
        Statement s=null;
        RemoteCredit rc=null;
```

```

RemoteAddDelivery rad=null;
long orderID=System.currentTimeMillis();
long price=0;
try{

    RemoteCredit rp=new Credit_Client("//ik2.inf.ethz.ch/Credit");
    RemoteAddDelivery rds=
        new AddDelivery_Client("//ik3.inf.ethz.ch/AddDelivery");
    CreditThread pt=
        new CreditThread(rp,CC,holder,expDate,price);
    DeliveryThread st=
        new DeliveryThread(rds,"BookShop","BookShop"+orderID,
            (new Long(orderID)).toString(),address);

    SubTxThread sub1=new SubTxThread(waiter,pt);
    SubTxThread sub2=new SubTxThread(waiter,st);
    sub1.start();
    sub2.start();
    waiter.waitForAll();
    if (waiter.getAbortCount(>0)
        throw new Exception("aborted subtx "+waiter.getAbortCount());

    c=myDS.getConnection();
    s=c.createStatement();
    ResultSet rs=s.executeQuery("select price from "+
        "bookstock where id="+id);
    while (rs.next()){
        price=rs.getLong(1)*amount;
    }
    s.executeUpdate("update bookstock set avail=avail-"+
        amount+" where id="+id);
    s.executeUpdate("insert into orders values ("+
        ""+orderID+"', '"+orderID+"', '"+
        address+"', "+id+", "+amount+"");
}
catch(Exception e){
    myUT.setRollbackOnly();
    throw e;
}

}
class CreditThread implements SubTxCode{
    private RemoteCredit myRemoteCredit;

```

```
private String myCC,myName,myDate;
private long myPrice;

public CreditThread(RemoteCredit rc,String cc,String name,
                   String expDate,long price){
    myRemoteCredit=rc;
    myCC=cc;
    myName=name;
    myDate=expDate;
    myPrice=price;
}

public void exec() throws Exception{
    try{
        myRemoteCredit.do_check(myCC,myName,myDate,myPrice);
    }
    catch (Exception e){
        throw e;
    }
}

class DeliveryThread implements SubTxCode{
    private RemoteAddDelivery myRemoteDel;
    private String myClient,myOrder,myDate,myAddress;

    public DeliveryThread(RemoteAddDelivery rad, String client,
                        String orderId, String date, String address){
        myRemoteDel=rad;
        myClient=client;
        myOrder=orderId;
        myDate=date;
        myAddress=address;
    }

    public void exec() throws Exception{
        try{
            myRemoteDel.do_add(myClient,myOrder,myDate,myAddress);
        }
        catch (Exception e){
            throw e;
        }
    }
}
```

```
    }  
  }  
}
```