Dissertation

Middleware Support for Adaptive Dependability through Explicit Runtime Integrity Constraints

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.Prof. Dr.techn. Mehdi Jazayeri und
Dr.techn. Karl M. Göschka

184-1
Institut für Informationssysteme
Arbeitsbereich für Verteilte Systeme
Technische Universität Wien

und

Prof. Dr.Eng. Carlo Ghezzi

Dipartimento di Elettronica e Informazione
Politecnico di Milano

eingereicht an der Technischen Universität Wien

von

Mag. Dipl.-Ing. Lorenz Froihofer
Matr.Nr. 9825055
8654 Fischbach 69a

Wien, September 2007 _____

Kurzfassung

Integrität und Verfügbarkeit sind zwei konkurrierende Attribute von Systemzuverlässigkeit. Während einige Systeme strikte Integrität verlangen ist für andere – abhängig vom Systemkontext – die Verfügbarkeit bedeutender. In diesen Systemen kann die Verfügbarkeit unter temporärer und teilweiser Rücknahme der Integritätsanforderungen erhöht werden. Potenzielle Inkonsistenzen aufgrund von Netzwerkpartitionen werden basierend auf der Validierung von Integritätsbedingungen auf möglicherweise veralteten Kopien von Daten in Kauf genommen. Diese Konsistenzbedrohungen bzw. deren Effekte müssen entsprechend eingeschränkt und etwaige Inkonsistenzen nach der Reparatur von Netzwerkausfällen wieder behoben werden.

Diese Dissertation präsentiert einen Middleware-basierten Ansatz, die Systemzuverlässigkeit von datenorientierten Systemen durch eine Balancierung der Integritäts- und Verfügbarkeitsanforderungen an Fehlersituationen anzupassen. Dieser Ansatz basiert auf zur Laufzeit explizit verwaltbaren Integritätsbedingungen. Während die Unterstützung für die adaptive Zuverlässigkeit in die Middleware integriert wird, werden notwendige Artefakte zur Balancierung von Verfügbarkeit und Konsistenz, z.B. Integritätsbedingungen und entsprechende Metadaten, von der Anwendung bereitgestellt. Zusätzlich werden verschiedenste Plugin-basierte Mechanismen und Callbacks verwendet, um ein flexibles Management von Inkonsistenzen im System zu ermöglichen.

Konfigurierbarkeit und Explizitheit zur Laufzeit fördern die saubere Trennung von verschiedenen Aspekten und führen so zu verbessertem Systemdesign und verbesserter Wartbarkeit. Andererseits wirken sie sich im Allgemeinen negativ auf die Leistung eines Systems aus. Die durchgeführten Studien zeigen, dass sich die Kosten in der Größenordnung von 1-15% Einbuße für einige der fortgeschritteneren Ansätze für explizites Integritätsmanagement rentieren, während andere Ansätze Overheads bis hin zu 40.000% aufweisen.

Integritäts- bzw. Inkonsistenzmanagement basierend auf expliziten Integritätsbedingungen ist ein Hauptteil des Ansatzes für adaptive Zuverlässigkeit. Dies wird ergänzt durch Datenreplikation zur Fehlertoleranz von Rechner- und Netzwerkausfällen. Die Evaluierungen zeigen, dass sich der entwickelte Ansatz vorrangig in Systemen bezahlt macht, wo (i) das Lese-/Schreibverhältnis hoch, (ii) die Anzahl der replizierten Knoten eher klein und/oder (iii) die Geschwindigkeit der Schreibzugriffe nicht der limitierende Faktor ist. Von weiterem Vorteil ist, wenn die Systeme nicht die getätigten Operationen bzw. Zwischenzustände während eines Rechner- oder Netzwerkausfalles kennen müssen, um etwaige Inkonsistenzen zu beheben.

Abstract

Integrity and availability are two competing dependability attributes. While some applications require strict integrity, other applications exist, e.g., safety or mission critical systems, where—depending on the specific situation—availability is more important for dependability than strict integrity. Within our work, we focus on data-centric systems, where availability can be increased by temporarily relaxing data integrity, thereby allowing for certain inconsistencies. Potential inconsistencies are accepted based on constraint validation on replicated copies that are possibly stale in the face of network partitions. Such consistency threats need to be bound and eventually resolved during reconciliation to re-establish a consistent system state.

This dissertation presents a middleware approach to support adaptive dependability by balancing integrity and availability in distributed object systems. The envisaged balancing is based on explicit runtime management of integrity constraints and consistency threats. While the support for adaptive dependability is integrated into the middleware, the application provides the necessary artefacts, e.g., integrity constraints and corresponding metadata, on which trade-off decisions between integrity and availability can be based. Moreover, several plugin-based or even dynamically configurable call-backs allow for runtime interaction between application and middleware in order to perform flexible (in)consistency management within the system.

Runtime configurability and explicitness contribute to separation of concerns and hence to a well-structured system design, robustness, and better maintainability, while generally impairing system performance. Our studies show that the increases in maintainability of some of the more advanced approaches for explicit integrity constraints are certainly worth their performance costs of 1–15% while other approaches might be simply too slow with overheads of up to about 40,000%.

Consistency management based on explicit constraints is the first key part of our adaptive dependability approach. The second major part is an integrated replication service in order to provide fault tolerance for node and link failures. Evaluations of our middleware enhancement show that it is most worth its costs in systems where (i) the read-to-write ratio is high, (ii) the number of replicated nodes in the system is small, and/or (iii) write-performance is not the limiting factor. Moreover, systems can benefit most if they do not require the history of performed operations/applied states in order to reconcile inconsistencies of the system.

Acknowledgements

First of all, I want to thank Karl M. Göschka for his excellent guidance of my dissertation, the several fruitful discussions, the opportunities given during the last three years, and the experience gained in project management.

Furthermore, great thanks to Mehdi Jazayeri for his guidance and interesting discussions within the course of my PhD and for establishing the link to Karl M. Göschka and the DeDiSys (Dependable Distributed Systems) project. Further thanks go to Carlo Ghezzi for his interesting questions and feedback.

I am especially thankful to the master's students who contributed to this dissertation through several discussions and prototype implementations: Markus Horehled, Klaus Fuchshofer, Gerhard Glos, Jörg Artaker, Martin Hinterndorfer, Gerfried Aigner, Markus Baumgartner, Bernhard Rieder, Dominik Ertl, and Xuejun Ji.

Thanks go to my colleagues Johannes Osrael and Karl M. Göschka for the excellent cooperation and working environment. Further thanks go to all the colleagues at the university for the discussions and their feedback—also on early stages of this work. Moreover, I want to thank Hubert Künig for being "my" industry application developer, for the many in-depth discussions, the good cooperation, and his criticism and scepticism that challenged me over and over again.

I am thankful to my girlfriend for her patience and understanding during the last years where my time was often a scarce resource.

Finally, I want to thank the European Community by which this work has been partially funded under the Framework Programme 6 IST project DeDiSys (Dependable Distributed Systems, `http://www.dedisys.org`, contract number 004152).

Contents

List of Figures

Chapter 1

Introduction

Network partitions have been subject to extensive research for many years and interesting results have already been achieved. In 1985, Davidson et al. [DGMS85] contributed with a survey about *consistency in partitioned networks*, where they discussed several optimistic and pessimistic strategies to address the conflicting requirements of availability on the one hand and correctness (integrity, consistency) on the other hand. This interdependence is stated more precisely by the CAP principle [FB99, GL02], specifying that a system can only fully satisfy two of the three properties: **C**onsistency, **A**vailability, and **P**artition-tolerance (strong CAP principle). However, the weak CAP principle already defines that *the stronger guarantees are provided for two of these properties, the weaker guarantees can be provided for the third.* Obviously, there is a trade-off between these requirements. This trade-off potential is addressed, for example, through data partitioning [ÇÖSF01] and optimistic replication techniques [SS05].

Data partitioning techniques address network partitions through (non-redundant) distribution of data items across the system. For example, tickets of a ticket booking system might be distributed across nodes, which subsequently can sell their amount of tickets individually. Of course, redistribution of tickets between the nodes is possible in order to avoid one node running out of tickets. However, according to the CAP principle, this behaviour relaxes the availability requirement as during a network partition, some partitions might run out of tickets while other ones might still have several of them. Optimistic replication techniques on the other hand relax the consistency requirements based on the assumption that (write) conflicts are rare and therefore the benefit gained in higher availability is worth the effort caused through conflict resolution. This dissertation contributes to this issue through explicit

(in)consistency management in order to achieve an adaptive balancing of the two dependability [ALRL04] attributes: integrity, the absence of improper system alterations, and availability, the readiness for correct service.

1.1 Adaptive dependability

Failures are threats to dependability and consequently to availability and integrity. While failures affecting availability might lead to a non-responsive system, integrity violations may lead to system inconsistencies. Strong consistency requirements lead to impaired availability in the face of network partitions, high availability requirements might result in potentially improper system alterations in the face of network partitions, thereby possibly violating integrity and introducing inconsistencies into the system.

Consequently, dependability can be balanced with respect to the two conflicting requirements integrity and availability and thereby be made adaptive to a system's context, environment conditions, and changing users' needs. Such a balancing requires appropriate means to control the amount and severity of inconsistencies introduced in the face of network partitions in order to gain the benefit of higher availability. This requirement is addressed within this dissertation through explicit runtime integrity constraints for data-centric applications: The constraints are defined and implemented according to an application's requirements, while the support for integrity maintenance as well as the balancing support for adaptive dependability is provided by the middleware.

This dissertation focuses on node and link failures, assuming the *crash failure model [Cri91] for nodes*—pause-crash for server nodes—and *links may fail by losing some messages but do not duplicate or corrupt messages.* Link failures may subsequently lead to network partitions, effectively splitting a system into parts that are not able to communicate. However, as node and link failures cannot be differentiated at the time when they occur [FLP85], we initially treat node failures as partitions with a single node. Whether a node or link failed can be detected later when the node is reachable again.

Replication [HHB96], the process of maintaining multiple copies of the same entity (data item, object), is well-known to provide fault tolerance for improved availability in case of node and link failures. The primary partition protocol [RSB93], for example, allows a single partition (the primary partition) to continue "normal" operation while other partitions are blocked or operate in read-only mode. Such an approach prevents replica conflicts as

(write) operations are only allowed in the primary partition. To further increase availability, write access in other partitions would be desirable—at the price of replica inconsistency.

Replica consistency is only one correctness criterion for data integrity. In total, we differentiate three different kinds of consistency with respect to data integrity [SG04]:

- *Concurrency consistency (isolation)*: defines the correctness of data with respect to concurrent, interleaving access to single data items, typically in the context of (even distributed) transactions.
- *Replica consistency*: defines the correct effect of operations on different replicas (copies) of a single "logical" entity with respect to a particular replica consistency model (e.g., 1-copy-serializability [BHG87] or looser consistency models like ϵ-serializability [PL91] and eventual consistency, meaning that every update eventually reaches every replica, used for example in epidemic replication algorithms [DGH+87]). Replica inconsistency is caused by staleness, e.g., if the backup copies differ from the primary copy in a primary-backup replication protocol [BMST93]. Isolation of concurrent access to different replicas of the same logical entity is also achieved by replica control—in cooperation with the (local) isolation mentioned above.
- *Constraint consistency*: defines the correctness of data with respect to data integrity constraints that stem from the application requirements. These constraints have often to be satisfied in the course of business transactions.

These three consistency types can be addressed through a layered approach where replica control operates on top of local isolation and below constraint consistency management, see Figure 1.1. The interrelationship of these types is elaborated in more detail in the dissertation of Robert Smeikal [Sme04]. To summarize, the correctness of the isolation layer is crucial for the overall consistency of the system as the replica control and data integrity layer both depend on it. Replica control provides the mapping of replicated object states to a single object that is checked at the data integrity layer. Therefore, replica consistency directly affects data integrity. On the other hand, correct replica control does not depend on data integrity.

This dissertation focuses on constraint consistency, influenced by replica consistency, as both are affected by node and link failures. Consistency of

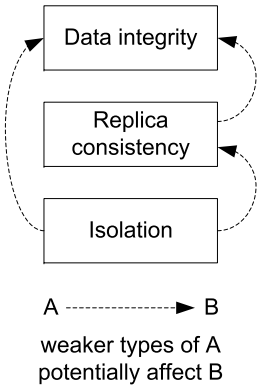


Figure 1.1: Interrelation of consistency types

the constraints themselves, e.g., whether constraints represent conflicting requirements, does not affect our balancing of integrity and availability and is therefore not within the focus of this work. Similarly, concurrency consistency is not affected by node and link failures and hence not in the focus.

1.2 Relationship to the transaction concept

Traditional systems apply ACID[1] transactions [HR83], requiring that all four properties are met. Replication ("R") can synchronously be bound to transactions, e.g., 1-copy-serializability [BHG87]. However, in case of node or link failures, synchronous update propagation would block. Consequently, update propagation can be relaxed to asynchronous behavior, e.g., synchronous per network partition, to avoid blocking. Moreover, if constraints cannot be checked (unreachable objects) or cannot reliably be checked (stale backup copies involved), constraint consistency (the "C") needs to be relaxed, too. Interestingly, Coulouris et al. [CDK05] do not include consistency in their list of transaction properties and rather specify that the "C" is under the responsibility of the application developer.

Atomicity ("A") is not relaxed in principle in our approach, although one business transaction (completed as a single transaction in a healthy system) may result in two or more transactions (one initial transaction during degraded mode and one or more transactions to resolve conflicts and perform compensating actions during reconciliation). These considerations rather correspond to the concepts of atomic transactions [ABH+05a] vs. business

[1] Atomicity, Consistency, Isolation, and Durability

activities [ABH+05b] in the area of Web services (WS). However, in our approach we did not follow these ideas and consequently bound atomicity, isolation, and durability strictly to transactions. Consequently, replication and constraint consistency management operate then on top of such "AID" transactions, see Figure 1.2.

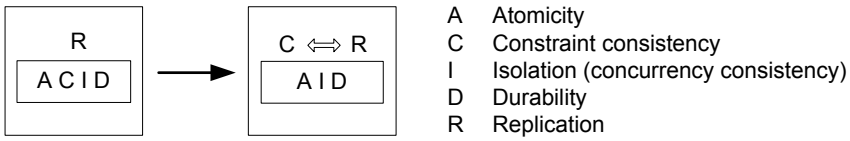


Figure 1.2: Trading transactional properties for adaptive dependability

1.3 Problem illustration & solution overview

The example for our discussion is provided by a distributed flight booking system as illustrated in Figure 1.3. This flight booking system consists of replicated server nodes that store identical data about flights and sold tickets for passengers (persons). This application has a data integrity constraint, the *ticket-constraint*, specifying that the number of sold tickets must be less than or equal to the number of seats of a specific flight. As long as the system operates in the healthy system mode, i.e., no node or link failures are present, this constraint can easily be validated. However, as soon as the system suffers—as illustrated—from a link failure, for example, the balancing of availability and consistency has to be performed.

Suppose that we have a flight with 80 seats of which 70 are already booked based on operations performed in the healthy system. Now the system suffers from a network partition. Due to the high availability requirement for this system, we allow write access in different partitions, temporarily accepting potential inconsistencies. Assume that customers buy seven tickets in partition A, which now has a total of 77 sold tickets. The ticket-constraint is satisfied in this partition. Subsequently, customers buy eight tickets in partition B, leading to 78 sold tickets in partition B. The ticket-constraint is also satisfied in this partition. After the network partitions are reunified, however, the system has to reconcile the updates made in the different partitions, effectively leading to 85 sold tickets in total. Consequently, our ticket-constraint is now violated. To solve this issue, five tickets will be cancelled or rebooked to another flight.

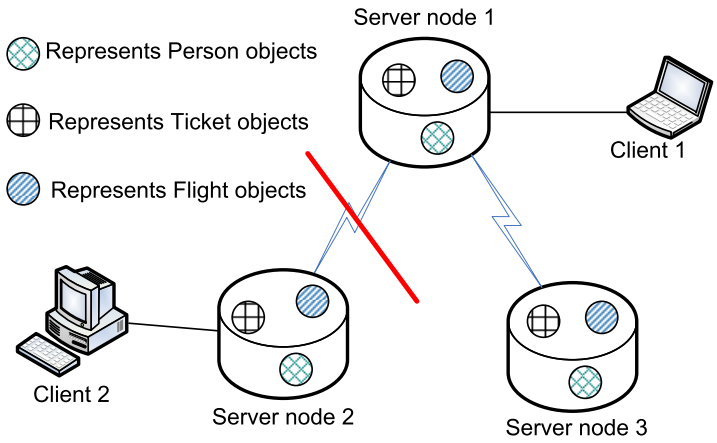


Figure 1.3: Flight booking system

The previous example shows that although a constraint is satisfied in degraded mode while node or link failures are present, it might be the case that it is violated afterwards when the system recovers from a previous failure. Consequently, constraint validation is not reliable during degraded mode if write access in different partitions is possible. We call such situations a *consistency threat*.

Obviously, the effects of a consistency threat depend on the application and have to be reconciled in an application-specific way. However, other tasks such as detection of node or link failures, triggering the validation of constraints at appropriate times, detection of consistency threats or the replication support can be implemented in the middleware. To actually decide, whether a specific consistency threat is acceptable, i.e., it does not lead to catastrophic consequences and its effects can be cleaned up after all nodes are reachable again, we perform an application callback. This *negotiation callback* provides the constraint and the objects affected by the consistency threat to the application-provided *negotiation handler*. The application in turn can examine the situation and decide on whether to accept or reject the consistency threat. If the threat is accepted, the current operation/transaction continues—otherwise it is aborted.

Accepted consistency threats are persistently stored by the middleware and processed again during the reconciliation phase in order to evaluate whether a consistency threat actually introduced a constraint violation. If a constraint is violated, e.g., 85 tickets sold for a plane with 80 seats, we use another callback to the application-provided *reconciliation handler* in order to trigger the clean-up of the inconsistencies.

1.4 Target systems and applications

This dissertation focuses on the class of tightly-coupled data-centric distributed object systems [16]. These systems have their focus on the (business) data, typically stored in database management systems and represented by the business objects (entities) of an application and the relations between them. The Enterprise JavaBeans (EJB) platform, for example, represents such business objects by entity beans. The differentiation criterion between data-centric and other systems is based on the major design focus of the system—and how properties and constraints are formulated and expressed. This, however, does not imply how the system is implemented.

Furthermore, our focus is on distributed object systems where communicating objects reside on different nodes. The main reason for having objects distributed among nodes and not being centralized is strong ownership of these nodes, e.g., the objects might be bound to some hardware facilities or different administrative domains. Application data are encapsulated by objects and their relationships and are modified by (possibly nested) invocations of methods of these objects.

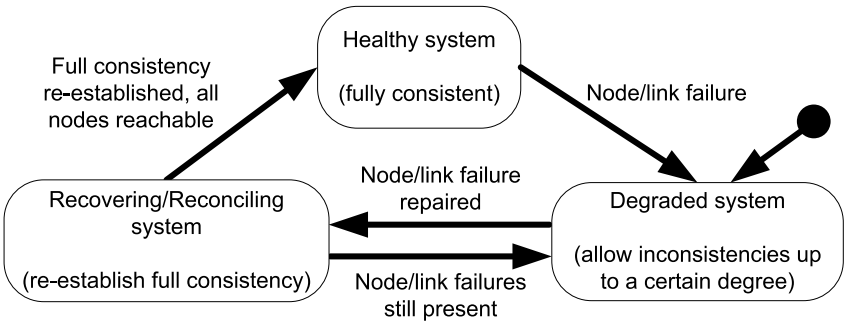


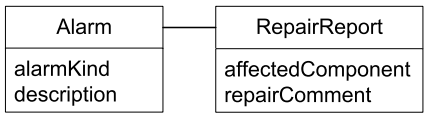
Figure 1.4: Major system states

The system model foresees three major states as locally perceived by each individual node, see Figure 1.4. In a healthy system, no failures or inconsistencies are present. In degraded mode, while node/link failures are present, inconsistencies are potentially introduced. These inconsistencies are cleaned up in the reconciliation phase after node or link failures were repaired. To limit the degree of inconsistency introduced in degraded system periods, integrity constraints have to be explicitly available and manageable during runtime. For this purpose, the middleware also needs metadata about constraints, e.g., whether a constraint must never be violated or might potentially be violated in degraded mode. Given this input, the middleware

afterwards provides the support for runtime-management of constraint consistency in each of the three major system states.

The primary motivation for the performed work stems from a distributed telecommunication management system (DTMS) [SG03]. The DTMS application manages voice communication systems (VCS) installed at different sites. Each site has its own instance of a DTMS, but configuration of the VCS requires DTMS instances of different sites to cooperate. The hardware facilities of the VCS are represented by objects within the DTMS that are bound to the site of the VCS for decentralized management reasons—a failure of a DTMS site should not have effects beyond the specific site. The objects of the DTMS are subject to integrity constraints that possibly span objects of multiple sites, e.g., the configuration parameters for a voice communication channel have to be consistent to enable communication between different sites.

Another application scenario, where a prototype has been implemented by an industry partner based upon our middleware, is a distributed alarm tracking system (ATS) [Ke07]. For our studies within this dissertation, a simplified model of the ATS is given in Figure 1.5, showing two classes: Alarm and RepairReport. Alarms are managed by administrative operators while the repair reports are filled out by technical operators. The alarmKind determines which kinds of components might have to be repaired (affectedComponent). Hence, the system applies certain integrity constraints between an Alarm and a RepairReport. The example provided in Figure 1.5 specifies that an alarm with alarmKind="Signal" can only be removed by repairing a component that is either a "Signal Controller" or a "Signal Cable". Administrative operators and technical operators are working at different locations, potentially accessing different servers. If a network split occurs between these servers, the system should still be available to all of them and allow to make progress—although the corresponding Alarm and RepairReport objects might become inconsistent.




```
context RepairReport inv ComponentKindReferenceConsistency:
    alarm.alarmKind.equals("Signal") implies
        (affectedComponent.equals("Signal Controller") or
        affectedComponent.equals("Signal Cable"))
```

Figure 1.5: Simplified ATS model with constraint

A further application scenario is the flight booking application illustrated in Section 1.3. A research prototype of this scenario was implemented based upon our middleware enhancement in order to study different concepts.

Additional application scenarios are available in [Fe07] along with a corresponding evaluation.

1.5 Explicit integrity constraints

Constraint validation is one of the most essential tasks of a system to ensure integrity. The integrity constraints are defined according to an application's requirements and explicitly stated (and probably negotiated with stakeholders) during the requirements analysis phase, e.g., by stating "The system must not sell more tickets than available seats for a flight.". Consequently, the constraints represent a subset of an application's requirements that should be ensured by the implementation. Being typically stated only informally in the requirements analysis phase, e.g., written down in natural language, the constraints are often more formally described and attached to the application model in the design phase. The Unified Modeling Language (UML), for example, provides the Object Constraint Language (OCL) to explicitly specify constraints—in addition to the possibility to already express some constraints, e.g., cardinality or XOR, in the graphical notation of UML. Figure 1.6 provides an example for the ticket-constraint.

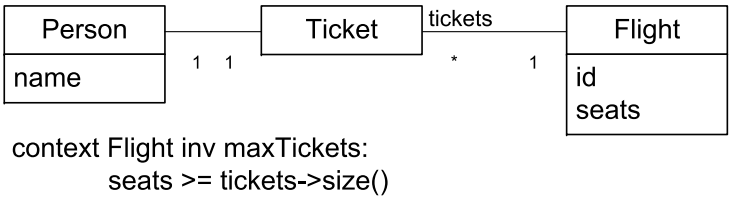


Figure 1.6: Design-phase (OCL) constraint example

In contrast to analysis and design, the constraints are most often no longer explicitly available in a system's implementation, i.e., the constraint validation code is often tangled with code for the business logic. Listing 1.1 provides such an example. For some applications, this might be a reasonable or satisfying approach. However, just following the simple approach to use if statements to validate constraints, for example, often turns out not to be the best solution due to several reasons: A single constraint might have to be checked in different places in the program, which might lead to

inconsistent implementations of the same constraint throughout the implementation code. Obviously, ensuring that specific constraints were correctly implemented in a system might become a tedious task and might lead to inconsistencies between system requirements, design, and implementation. Furthermore, constraints might also describe contracts between different system entities. Unfortunately, an implicit constraint implementation does not support this "design by contract" principle [Mey92]. Some systems might even be more demanding by requiring explicit runtime handling of integrity constraints, e.g., to support object versioning [GCG02] or adaptive dependability by (temporarily) relaxing integrity requirements as addressed in this dissertation.

Listing 1.1: Constraint implementation example

```
public class Flight {
  ...
  public Ticket[] sellTickets(Person p, int ticketCount) {
    if (getSeats() >= getSoldTickets().count()+ticketCount) {
      // Sell ticket
    }
    else {
      // Do not sell ticket
    }
  }
}
```

Meyer [Mey92] already proposes the "design by contract" principle where constraints are made explicit through so called *contracts*. Contracts can be expressed as preconditions and postconditions for methods and invariant constraints for classes. This already improves the traceability of constraints within the implementation code. However, the constraints are intertwined with business functionality during compile time and no longer explicitly available (and manageable) during runtime.

In distributed object systems, constraint or contract enforcement becomes more complex as node and link failures affect the constraint validation process. For example, if nodes are unreachable, some constraints might not be checkable due to unavailable objects. Systems requiring strict consistency will have to (at least partially) block in such situations. However, for mission or safety-critical applications, availability might be more important than strict consistency as blocking the system due to partial failures might lead to disadvantageous if not catastrophic consequences. On the other hand, introducing inconsistencies to the system must be performed in a controllable way. Relaxing consistency to improve availability can benefit from integrity con-

straints that are explicitly available and manageable during runtime. This approach allows for constraint validation at any time or adaptation to changing consistency requirements.

Consequently, this dissertation addresses data integrity constraints as first class runtime citizens within a distributed object system in order to balance dependability [ALRL04] with respect to node and link failures. Similar to [VS02], we encapsulate the constraint checking code in classes of an object-oriented programming language—Java within the scope of this dissertation—where one class represents exactly one integrity constraint. Each class provides a validate(...) method which is called to validate the constraint. Listing 1.2 provides a corresponding explicit constraint example for the ticket-constraint. As the constraint is now separated from the business logic, appropriate middleware support ensures that the constraint validation is triggered appropriately. If the constraint is violated, the middleware will abort the current operation/transaction. Consequently, the middleware provides the glue logic between the business code and the constraint implementation.

Listing 1.2: Explicit constraint classes

```
public class Flight {
  ...
  public Ticket [] sellTickets (Person p, int ticketCount ) {
    // Sell ticket(s)
  }
  ...
}

public class TicketConstraint {
  public boolean validate (ConstrValidationContext ctx) {
    Flight flight = ((Flight)ctx.getContextObject());
    return flight.getSoldTickets().count() <= flight.getSeats();
  }
}
```

1.6 Target constraints and trigger points

Generally, data integrity constraints are predicates on data, evaluating to true, if a constraint is satisfied, or false, if it is violated. In our case, constraints are defined upon a class model, e.g., by using the Object Constraint Language (OCL) of the Unified Modeling Language (UML) for UML class diagrams. We follow the well-established approach to differentiate between

preconditions, postconditions and invariant constraints [Mey92]. *Preconditions* are bound to and checked before a specific method invocation, *postconditions* are bound to and checked after a method invocation, and *invariant constraints* are bound to a certain class—the *context class*. For invariant constraints, we further differentiate between hard (checked at the end of an operation during a transaction) and soft constraints (checked at the end of a transaction) [JQ92] for transactional applications. Invariant constraints are defined solely on the state of objects (static constraints) and hence can be validated at any time. Dynamic constraints defined on state transitions, sequences of states or state transitions or temporal predicates are not in the primary focus of our work.

While pre- and postconditions are explicitly bound to methods and hence have to be triggered before or after method invocations, invariants are bound to a certain class and the triggering methods for validation of invariants have to be specified. Triggering constraint validation of invariant constraints upon each call to a method of the context class or only upon each call to a public method of the context class are two possible options. However, invariant constraints have at least to be checked whenever a method that potentially might lead to a constraint violation is called. We call such a method an *affected method* of the constraint.

Although an invariant constraint is defined for a certain context class, affected methods might belong to other classes as well. For our ATS example in Figure 1.5, the constraint ComponentKindReferenceConsistency has to be checked whenever the alarmKind of an Alarm or the componentKind of a RepairReport is changed. Consequently, Alarm.setAlarmKind(...) and RepairReport.setComponentKind(...) are affected methods of the constraint ComponentKindReferenceConsistency while the constraint itself is an *affected constraint* of these methods with two *affected objects*, an Alarm object and a RepairReport object. The validation of a constraint requires access to all affected objects. Obviously, the affected methods of a constraint cannot generically be determined without further knowledge of the constraint. Moreover, checking the ComponentKindReferenceConsistency constraint if only the description of an alarm is changed (caused by following the "trigger constraint at all public method invocations" paradigm) unnecessarily impairs performance. Due to these reasons, we only trigger constraint checking for affected methods specified by the application developer.

1.7 Structure, contributions, publications

This dissertation contributes to the areas of dependability research, runtime monitoring, and runtime program analysis with respect to adaptive dependability and integrity/(in)consistency management in distributed systems. More specifically, it primarily contributes through the following topics:

- Chapter 2 elaborates the trade-off between performance impairments and maintainability issues resulting from explicitness of integrity constraints through an overview and evaluation of constraint validation approaches for the Java programming language [2]. Starting from a simple handcrafted approach to constraint validation, more sophisticated approaches, such as the ones inspired by design by contract [Mey92], aspect-oriented programming (AOP) [KIL+96], or the requirement for explicit runtime integrity constraints are evaluated.
- Chapter 3 contributes with the theoretical foundations for adaptive dependability by balancing integrity and availability based on explicit runtime constraints [4, 5]. Given the fact that constraint validation cannot be performed reliably when node and link failures are present in distributed systems with high availability requirements, the concept of a consistency threat is elaborated in order to manage potential inconsistencies introduced to the system during degraded system periods. Additionally, the potential inconsistencies are re-evaluated and—if necessary—reconciled after corresponding node and link failures were repaired.
- Based on the theoretical foundations, Chapter 4 describes how adaptive dependability through balancing of availability and integrity can be supported and implemented in middleware [3, 6, 1]. More specifically, it provides details about a proof-of-concept prototype implementation integrated into the JBoss Application Server (JBoss AS) for Enterprise JavaBeans (EJB) applications. This prototype implementation is thereafter evaluated in Chapter 5.

Chapter 6 discusses related work to the topics covered in this dissertation and Chapter 7 finally concludes this dissertation and provides ideas for future work.

The dissertation of Johannes Osrael on “Replication Techniques for Balancing Data Integrity with Availability” [Osr07] provides complementary in-

sights and results to this dissertation. An overview of the content of [Osr07] and some corresponding publications is found in Section 4.3 on page 54.

The publications authored and co-authored by the writer of this dissertation provided in the references above and within the dissertation in numeric references are used in parts of this dissertation without always being referenced explicitly.

Chapter 2

Constraint Validation Approaches

Several constraint validation techniques for the Java programming language have been developed in the past. Most of these techniques are inspired by the way constraints are integrated in the Eiffel programming language—by building upon the design by contract principle. While this principle has a strong focus on the detection of contract violations between producers and users of a certain piece of code, e.g., between the writer and the caller of a method, other approaches focus on how system integrity can be achieved via validation of integrity constraints. Besides this slightly different focus, both approaches aim at improved system dependability through runtime constraint validation. Each of these constraint validation techniques has its advantages and disadvantages. Increasing the explicitness and runtime configurability of constraint handling and enforcement generally comes at the price of impaired performance. Motivated by these considerations and our requirement for explicit runtime handling of constraints, the following sections contribute with an overview and evaluation of different constraint validation approaches in Java with respect to implementation, maintainability, and performance.

2.1 Constraint implementation strategies

Starting from the simple approach of handcrafted constraints, we will continue to more flexible approaches such as constraint code generation and explicit runtime constraints. While this section will shortly describe how each approach is performed, sections 2.2 and 2.3 will discuss their advan-

tages and disadvantages with respect to implementation, maintainability, and performance. To unify the trigger points for invariant constraints in order to make our performance evaluations comparable, we check invariant constraints before and after the invocation of public methods of our reference application—complying with design by contract in the way that if and only if an invariant holds before a public method invocation, it must also hold after the method invocation.

2.1.1 Handcrafted constraints

The most simple approach to implement constraint checking in Java is to tangle the constraint checking code with other code, e.g., for the business logic. This approach does not require any formal specification of constraints as the transformation from the integrity requirement to the constraint implementation has to be performed by the programmer. Generally, this results in one or more if-statements to check a certain constraint and act according to whether these conditions are true or false. A source code example for this approach is provided in Listing 2.1.

Listing 2.1: Simple constraint implementation

```
class A {
  void someMethod() {
    if ( ( (...) OR (...) ) AND (...) ) {
      //business logic code
    } else if (! ...) {
      throw SomeException();
    } else {
      if (...) {
        //exception handling code
        printErrorMessage(...);
      } else {
        //business logic continued
      }
    }
    //further business logic
  }
}
```

2.1.2 Code instrumentation

Code instrumentation refers to injecting automatically generated code into a piece of original code, e.g., to add some additional functionality to the existing code. With respect to Java, we differentiate between source code and

byte code instrumentation, depending on whether the Java sources are instrumented with Java code before compilation (i.e., pre-compiler approaches) or byte code is instrumented with byte code after the Java sources are already compiled. Within this section, we focus on source code instrumentation and will discuss differences to byte code instrumentation at the end of Section 2.2.3.

However, common to both approaches is the requirement that the injected code is generated according to constraints specified in a constraint language known by a certain tool. Generally, tools use either UML class models paired with OCL constraints [VS02, Wie00], constraints defined within Java comments, or methods with names according to naming conventions. The languages for constraint specification in the latter cases range from Java code [KHB99] over OCL and OCL-like expressions [Kra98] to tool-specific constraint languages [LBR99]. For source code instrumentation, the two primary approaches are in-place code injection and wrapper-based validation:

In-place validation code. This approach injects code for constraint checks directly at the place where the validation should be performed, e.g., within the performed method. If a constraint affects several methods of possibly different objects, constraint checking code for the same constraint is injected at any place where constraint checking is required. This approach is illustrated in Listing 2.2. An example for this approach is the iContract tool [Kra98].

Listing 2.2: In-place code injection

```
public int countChar (char c) {
  //--> code for validation of invariant
  //     constraints and preconditions
  //BEGIN original code
  int result = 0;
  char[] chars = toCharArray();
  for (int i=0; i<chars.length; i++) {
    if (chars[i] == c) result++;
  }
  //END original code
  //--> code for validation of invariant
  //     constraints and postconditions
  return result;
}
```

Wrapper-based constraint validation. In this case, methods restricted by constraints are wrapped and the constraint validation code is contained in the wrapper method. Generally, the original method is renamed and only

called via the wrapper method. For example, the original method countChar would be renamed to countChar_wrapped and the wrapper method would be named countChar, see Listing 2.3. Consequently, calls to countChar execute the wrapper method including the constraint validation code. A typical example for this approach is the Dresden OCL toolkit [Wie00].

Listing 2.3: Wrapper-based validation

```
public int countChar (char c) {
  //--> code for validation of invariant
  //    constraints and preconditions
  //--> Call the original method
  int result = countChar_wrapped(c);
  //--> code for validation of invariant
  //    constraints and postconditions
  return result;
}

public int countChar_wrapped(char c) {
  int result = 0;
  char[] chars = toCharArray();
  for (int i=0; i<length(); i++) {
    if (chars[i] == c) result++;
  }
  return result;
}
```

2.1.3 Compiler-based approaches

Compiler-based approaches build upon a specific Java compiler that is enhanced with functionality to read constraint specifications and integrate the corresponding constraint validation mechanisms into the Java byte code. As also possible for pre-compiler approaches, the constraint specifications can be provided in special Java comments or in special statements according to custom extensions of the Java programming language. In contrast to code instrumentation approaches, the transformation from source code to constraint checking byte-code is performed in a single step. An example for a compiler-based approach is the Java Modeling Language (JML) [LBR99]. Listing 2.4 provides an example for an input to a compiler-based approach that does not use a custom extension of the Java programming language.

Listing 2.4: Compiler-based constraint checks

```
/**
 * @pre o != null;
 * @post size() == size()@pre + 1;
 */
public void add(Object o) {...}
```

2.1.4 Explicit constraint classes

Encoding constraint validation code in explicit Java classes is an approach that completely separates validation code from, e.g., the code for the business logic. For example, the constraint validation code may be contained in a validate(...) method that is executed with appropriate arguments whenever a certain constraint has to be checked. This approach requires appropriate trigger mechanisms to ensure that the validate(...) method is called whenever necessary. Trigger mechanisms include explicit code statements made by the programmer, in-place code generation of the calls, wrapper-based approaches, and interceptor mechanisms discussed in Section 2.1.5. Explicit constraint classes are used, for example, in [VS02] as well as in this dissertation. Listing 2.5 illustrates this approach by repeating the TicketConstraint from Listing 1.2.

Listing 2.5: Explicit constraint classes

```
public class TicketConstraint {
  public boolean validate (ConstrValidationContext ctx) {
    //--> Get the instance of the constraint's
    //    context class
    Flight flight = ((Flight)ctx.getContextObject());
    //--> Check the constraint and set the
    //    return value "result" depending on
    //    whether the constraint is satisfied.
    return flight.getSoldTickets().count() <= flight.getSeats();
  }
}
```

Constraint repository. Encapsulating the constraint checking code in separate classes allows for more flexible handling of integrity constraints. For example, all constraints of an application can be registered within a constraint repository. At any point in time, this repository can be queried for constraints based on different criteria such as the class of the invoked object or the signature of invoked methods. Consequently, preconditions, postconditions, and invariant constraints affected by method invocations can be queried from the constraint repository. Moreover, using such a constraint

repository allows to add, remove, enable, and disable integrity constraints even during runtime.

2.1.5 Interceptor mechanisms

Interceptor mechanisms provide the possibility to intercept different events such as the call to a method. Subsequently, the interceptor can perform some actions and continue or possibly abort the current action, e.g., the method call. Hence, interceptor mechanisms are an appropriate mechanism to implement so called "cross-cutting concerns" such as logging or—in our case—constraint validation. For constraint validation, one can either implement the constraint checking code within the interceptor [WM05] or use a generic interceptor that redirects calls, e.g., to explicit constraint validation classes (Section 2.1.4). The second approach can be achieved by combining the interception mechanism with a constraint repository. Within the following paragraphs, we describe the major interceptor mechanisms available for the Java programming language:

Aspect-oriented programming. Aspect-oriented programming (AOP) [KIL+96] is closely related to code instrumentation as AOP is often achieved through (byte) code instrumentation. It follows the programming paradigm of interception and weaving of so called aspects into program execution. A typical example for the weaving of an aspect is to introduce logging functionality to an existing program. Today, several tools supporting the AOP programming paradigm already exist (http://aosd.net). Within this dissertation, we concentrate on AspectJ and JBoss AOP as two well-known tools with a significant user base.

Proxy implementations. The Java programming language provides the concept of a proxy implementation for interfaces (java.lang.reflect.Proxy). If a method is invoked on the proxy, the registered invocation handler is notified with details about which method was invoked on which object with which arguments.

CORBA and EJB interceptors CORBA and Enterprise JavaBeans also provide the possibility to intercept method invocations as both technologies separate interface from implementation. Hence, the interceptor mechanisms of CORBA and EJB can be used as trigger mechanisms for constraint val-

idation. However, within this evaluation we concentrate on plain Java applications not building upon higher level specifications, while EJB will be addressed within Chapter 4 and 5 discussing the adaptive middleware support.

2.1.6 Summary

Table 2.1 summarizes this section with an overview of the most influential tools supporting constraint validation. For each tool, we provide how constraints are specified and which mechanism is used for integration of constraint checks.

Table 2.1: Constraint validation tools

Name and reference	**Constraint specification**	**Integration of constraint checks**
Dresden OCL toolkit [Wie00]	OCL constraints defined for a UML class model	Wrapper-based source code instrumentation
Handshake [DH98]	Custom language in a separate file	Runtime byte code instrumentation on class load time
iContract [Kra98]	OCL in custom tags of Java comments	Source code instrumentation
Jass [BFMW01]	Custom language in special Java comments	Source code instrumentation
jContractor [KHB99]	Java methods that follow a defined naming	Byte code instrumentation by class loader
JML [LBR99]	JML constraints in Java comments or separate file	Custom compiler
JMSAssert	Custom language in custom tags of Java comments	Pre-processor for standard Java compiler, paired with custom library
Kopi Java compiler [LKP02]	Extension of the Java language with certain keywords	Custom compiler
USE [RG00]	OCL expressions	Runtime interpretation of OCL constraints

2.2 Implementation & maintainability issues

This section provides a description of implemented constraint validation approaches and discusses several issues with respect to implementation of constraint checks and maintainability of the resulting code.

2.2.1 Implemented approaches

For evaluation and comparison of the different constraint validation approaches, we implemented the following particular alternatives:

- *No checks*: is an implementation of the application without any constraint checks.
- *Handcrafted constraints*: is the case where the constraint checks are manually integrated into the application according to Section 2.1.1.
- *Dresden OCL Toolkit*: is a wrapper-based source code instrumentation approach with tool-generated constraints.
- *JML*: implements a compiler-based approach with manually specified constraints.
- *AspectJ-Interceptor*: is an AOP approach where the constraint validation code is implemented directly in the AspectJ aspect specifications.
- *AspectJ-Repository*: uses explicit constraint classes and a constraint repository to allow flexible runtime handling of constraints.
- *JBoss-Repository*: implements the same as AspectJ-Repository but uses the JBoss AOP toolkit as interception mechanism.
- *Java-Proxy-Repository*: uses the Proxy mechanism of Java as interception mechanism and also makes use of the constraint repository to look up affected constraints.

The selection of the different approaches is primarily motivated by our requirement for explicit runtime constraints to balance the dependability properties availability and consistency. Consequently, we evaluate the benefits and costs of different explicit runtime constraint checking approaches as well as other constraint checking approaches with better performance or tool support.

As expected, our performance studies showed a major overhead in the repository-based approaches, caused by searching for affected constraints. Hence, each of the repository-based approaches is also evaluated with an optimized repository that performs caching of query results. In this case, a subsequent query for constraints based on the same criteria reduces to a lookup in a hash table with a key that combines our search criteria: class, method and constraint type.

2.2.2 Handcrafted constraints

With handcrafted constraint checks, the programmer retains full control over where, when and what is to be checked and how to react on violations. However, the programmer is also responsible for accurately documenting these checks and to update them if the integrity requirements of an application change. Unfortunately, this tends to lead to inconsistencies between application requirements, documentation, and implementation of constraints. Moreover, the same constraint might be implemented differently (and inconsistently) at several places within an application.

In some situations, where tool support is not sufficient because it lacks expressiveness for a specific constraint, or some exceptional error handling has to be hand coded, this approach is the usual choice. Unfortunately, there are major drawbacks. The programmer is responsible for accurately documenting and updating the constraint checks, which tends to be a time consuming and error prone task and therefore impairs maintainability. Moreover the business logic becomes tangled with tricky constraint checking statements, making the program flow more complex. This may also lead to non-uniform error handling: Some errors are caught and reported normally, while other errors receive special treatment, provoking error retention, error masking and cryptic error messages.

2.2.3 Code instrumentation

The main advantage of code instrumentation approaches is that they allow a separation of business logic code from constraint checking code at design *and* implementation level. The main disadvantage is that the original code is changed through code injection. Several works [Ver01, Wie00] already investigated constraint implementation by code instrumentation and found that code instrumentation approaches generally suffer from different problems we can summarize as follows:

- *Return statement*: The code for checking postconditions and invariant constraints must be executed before the `return` statement. Hence, the result must be available before the method actually returns. This poses problems when the result to be computed is declared within the return statement itself, e.g., `return 2*x`.

- *Code duplication*: Due to control flow issues, there may be several return points for a method. Therefore, it is necessary to insert the code for checking postconditions and invariants at several points in the same method. Moreover, it could be necessary that constraints have to be checked in more than one method. This leads to even more duplication of constraint checking code.

- *Reachable point*: For methods of return type void it has to be decided whether it is sufficient to only insert the code for checking postconditions at the end of the method. This depends on whether the end is a reachable point of code—or more specifically, executed for every method invocation. Hence, complex control flow analysis is required.

- *Super statement*: If a subclass calls the constructor of a superclass (by invoking `super()`) in Java, the compiler allows no other statements in advance. Therefore, special measures must be taken to implement checks for preconditions and class invariants. Moreover, wrapper-based approaches may lead to infinite loops. For example, if a method `m_wrapped()` of a subclass calls `super.m()`, `m()` of the superclass subsequently will—due to polymorphism—call `m_wrapped()` of the subclass again, thereby introducing an infinite loop. Consequently, appropriate measures, such as adding the fully qualified class name to the name of the wrapped function, have to be taken.

- *Pollution of application code*: The instrumented code is cluttered with constraint checking statements.

- *Shift of line numbers*: Line numbers shown in compiler messages or stack traces of exceptions point to the modified source code instead of the original code. Hence, the mapping of the line numbers from modified code to original code has to be performed manually by the developer.

- *Black-box instrumentation*: The developer loses control over code changes, as they happen in a black box fashion. This may further lead to unexpected behaviour and performance losses during runtime of the program.

- *Debugging*: Debugging becomes more difficult as standard debuggers will only allow the debugging of the tangled instrumented source code. The shift of line numbers issue described above makes debugging even more complex.
- *Naming conflicts*: Conflicts in the names of, for example, methods or variables must be prevented between the original code and the generated code, since the generated code may define variables or helper methods.

Source code vs. byte code instrumentation. The comparison of source code instrumentation and byte code instrumentation shows that the instrumentation method has a major impact on the generated code fragments:

- *Source code pre-processing*: translates the constraint definitions into standard Java source code which is directly inserted into the original source code. This leads to highly tangled code, which is hard to change and maintain without original sources and constraint definitions.
- *Byte code post-processing*: translates the constraint definitions to Java byte code and injects the generated code into the existing byte code after compilation—either before runtime or dynamically during runtime through the usage of a custom class loader. In any case, this preserves the original source code. Hence, the developer does not recognize any changes to the code, but also has no control over the possibly dynamically instrumented code. This leads to difficulties in debugging, possibly unexpected behaviour during runtime, and a performance loss because of the constraint checks and—if performed at runtime—the dynamic code instrumentation through the class loader. However, some issues can be solved by byte code instrumentation compared to source-code instrumentation. For example, on byte code level, the Java Virtual Machine allows arbitrary statements in advance of calling the superclass constructor. This solves the problem of code insertion before a `super()` statement. Another example is that in byte code the computation of variables is separated from control flow statements (branching statements). Branching statements denote the only exit points of methods. Consequently, the issues of inline `return` statements and the determination of insertion points for postconditions are easier to address.

2.2.4 Compiler-based approaches

Custom compilers are often used if the constraint definitions are not provided within Java comments or Java annotations, i.e., the constraint-enhanced programs use an extended grammar of the standard Java language. In this case, the program code is no longer compilable without the custom compiler—or at least a compiler pre-processor—introducing a dependency on the vendor of the (pre-)compiler. Using a pre-processor falls into the category of code-instrumentation approaches described in Section 2.2.3 and hence will not be addressed here. Compiler-based approaches perform a direct transformation from source code to Java byte code and integrate the constraint checks during this transformation. As the example of JML shows, compiler-based approaches are also used without extensions to the Java language. While this allows to compile the code with a standard Java compiler, it still does not remove the dependency on the custom compiler for constraint validation.

Generally, the principle that constraint checks are generated out of separate constraint statements is similar to code instrumentation above. Hence, some of the issues described in Section 2.2.3, e.g., black-box instrumentation, debugging, and naming conflicts, also apply to this approach.

2.2.5 Constraints encoded in interceptors

Within our studies, we used constraints encoded as aspects in AspectJ as representative for constraints encoded in interceptors. No tool support was available for this approach. Hence, we had to manually code the constraints as AspectJ aspects. This already provides a clear separation between constraint validation code and code for the business logic. While the code for the business logic remains in *.java files, the code for constraint checking is contained in separate *.aj files, defining the constraint checks as AspectJ aspects.

One disadvantage of this approach is the strong coupling of the aspects to the base code. Pointcut definitions specify the interception points for constraint validation. If these definitions exactly match specific method signatures, they are very susceptible to changes in the underlying base code in which case the pointcut definitions have to be changed as well. Refactoring support of Integrated Development Environments (IDEs) could provide support here to improve productivity and reduce errors. General pointcut definitions that match multiple method signatures, e.g., by using wildcards, may on the one hand still be matching after some parts of the original method signature in the

underlying code have been changed, but on the other hand may be triggered even when not intended. Such errors are difficult to detect and fix, especially in the context of constraint checking as a failing constraint indicates that the reason is a problem in the base code, rather than a mismatch in the pointcut definitions.

2.2.6 Explicit constraint classes

Encapsulating the constraint checking code in explicit constraint classes allows for explicit runtime handling of integrity constraints. The degree of flexibility, however, heavily depends on the triggering mechanism for constraint validation. While manual integration or code instrumentation are feasible mechanisms to trigger constraint validation, we focused on the combination of a constraint repository paired with a generic interceptor mechanism. This combination allows for a maximum of flexibility, e.g., to add, remove, enable, or disable integrity constraints during runtime of a system—which would require code modification and recompilation in the other constraint validation approaches. However, this flexibility comes at the price of decreased performance compared to other approaches that manually integrate constraints.

2.3 Performance studies

Our application scenario for the performance evaluation is the management of projects and employees within a company. Employees participate in projects and perform a certain amount of work on a daily basis. Within this model, several restrictions apply, e.g., an employee can only handle a certain amount of workload. The application contains a mixture of preconditions, postconditions and invariant constraints—78 constraints in total. This application scenario was only used for the investigation of the different constraint validation approaches and not with respect to whether the balancing of integrity and availability could be applied.

2.3.1 Comparison conditions

In order to allow for comparison of the different approaches, the validation of integrity constraints was performed in a uniform way. More specifically, we applied the following principles:

Constraint scope: Preconditions, postconditions and invariants only constrain public methods. Public constructors are constrained by invariants, private constructors remain unchecked.

Constraint checking: Constraints are checked before (preconditions) or after (postconditions) the actual code of the guarded method is executed. This also holds true for nested method calls. Invariants are immediately checked after public constructor calls and before and after public methods.

Constraint inheritance: In order to address object substitution and behavioral subtyping, constraints of extended superclasses or implemented interfaces are also taken into account. Preconditions of superclasses and interfaces are concatenated with the logical OR operator. Postconditions and invariants are concatenated with the logical AND operator [DL96].

Error handling: To exclude runtime differences due to different treatment of constraint violations, the measured application scenario does not violate any integrity constraints. However, in other scenarios we ensured that all the approaches actually check the same number of constraints and also correctly detect constraint violations.

2.3.2 Results

To measure the performance of the individual approaches, we implemented some use cases within our application scenario and let them run a number of times. To reduce the effects of environmental noise and just-in-time (JIT) compilation, we performed 2500 runs of the same example scenario before we measured another 2500 runs of the example scenario with each constraint validation approach. Each run triggers 4875 checks of invariants, 1097 checks of postconditions, and 433 checks of preconditions. The constraint repository based approaches intercept 1605 methods and trigger 7677 search operations within the repository for each run.

2500 runs of the scenario without constraint checks take 125ms to execute on an AMD Athlon XP 2600+ with 512MB of RAM running under Windows XP. The handcrafted constraints approach is the fastest version, but already runs 35 times slower than the same scenario without constraint checks. However, as this is the fastest approach, it is the baseline for comparison of the other approaches with respect to *additional* overheads. These overheads are calculated according to (2.1).

$$Overhead = \frac{Runtime\ for\ approach}{Runtime\ for\ baseline} \tag{2.1}$$

Figure 2.1 provides an overview of the fastest constraint validation approaches where the handcrafted constraints approach provides the baseline. This figure shows that constraints integrated as aspects in AspectJ are almost as fast as handcrafted inline constraints. The overhead introduced is only 1.06 times the runtime of the handcrafted constraints approach. The second fastest approach uses JBoss AOP for invocation interception and an optimized constraint repository containing explicit constraint classes. This introduces a runtime overhead of 7.99. Using a Java proxy with an optimized repository runs 9.54 times slower than handcrafted constraints and AspectJ with an optimized repository shows an overhead factor of 10.86.

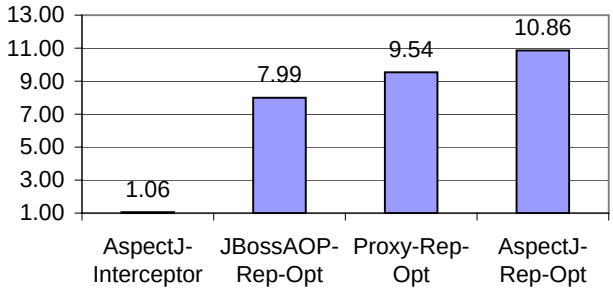


Figure 2.1: Fastest approaches

Figure 2.2 provides a comparison of the slowest constraint validation approaches where handcrafted constraint checks again provide the baseline for comparison. The approach to use the Java proxy mechanism and a non-optimized constraint repository requires 48.03 times the runtime of the handcrafted approach—nearly 4.5 times slower than AspectJ with the optimized constraint repository, which was the slowest approach in Figure 2.1. After the proxy mechanism follows JML requiring 61.37 times the runtime of handcrafted checks. AspectJ with a non-optimized repository shows an overhead factor of 70.71 and JBoss AOP with a non-optimized repository already runs 103.17 times slower than handcrafted constraint checks. Finally, the Dresden OCL toolkit with tool-generated constraints shows a runtime overhead of 405.71 times the runtime of the handcrafted approach.

Interestingly, the order of the different interceptor mechanisms with respect to performance changes for using an optimized and a non-optimized constraint repository. While JBoss AOP is the fastest mechanism with the optimized repository, followed by the Java proxy and AspectJ, the Java proxy

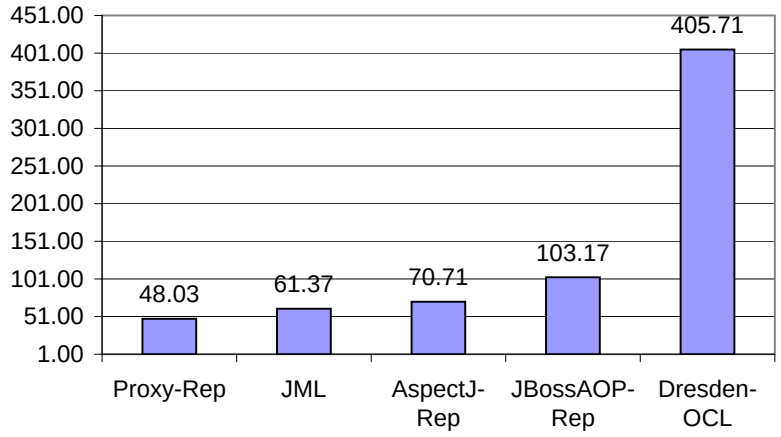


Figure 2.2: Slowest approaches

approach is the fastest mechanism for the non-optimized repository, followed by AspectJ and JBoss AOP. This is an unexpected result, which we will investigate in the following paragraphs.

Search overhead is not the only overhead introduced by constraint validation through generic interceptors combined with a constraint repository. In order to evaluate the additional overheads in detail, we separate the total runtime into five major slices (Figure 2.3):

R1: Application without constraint checks	R2: Invocation interceptions	R3: Parameter extractions	R4: Constraint searches	R5: Constraint checks

Figure 2.3: Runtime slices

R1 is the net application runtime without constraint checks

R2 is the overhead introduced through invocation interception by the different interceptor mechanisms (Java proxy, AspectJ, and JBoss AOP).

R3 provides the overhead to extract search parameters based on the information that an interceptor mechanism provides. This includes getting invoked method, method arguments and/or class of the invoked object.

R4 is the runtime overhead required for searching constraints within the constraint repository.

R5 is the overhead introduced by the constraint checks themselves

Figures 2.1 and 2.2 showed a comparison of all approaches considering the total overhead for constraint checking. Further on, we investigate the different overheads of the respective runtime slices in order to provide a more in-depth comparison of the constraint repository approaches. As the following configurations do not contain constraint checking code, the application without constraint checks (R1) provides the baseline for these comparisons. For JML and the Dresden OCL toolkit, we only considered the total overhead as the methodology is different and cannot reasonably be fitted to the five runtime slices provided above.

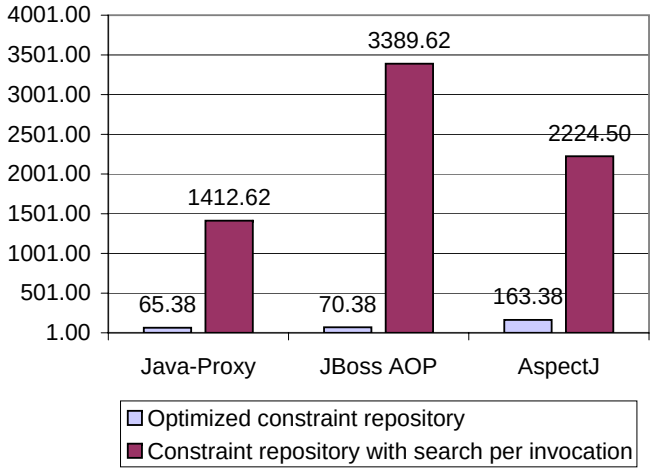


Figure 2.4: Search overhead: $\frac{R1+R2+R3+R4}{R1}$

Figure 2.4 shows the search overhead of the optimized and non-optimized constraint repository compared to the application without constraint checks. These versions include the overheads R2, R3, and R4, but do not check constraints (R5). The difference between the optimized and the non-optimized repository is that the runtime overhead R4 is reduced through caching of previous queries. The performance improvements of the optimized constraint repository reduced the overall runtime by a factor between 13.62 (AspectJ) and 48.16 (JBoss AOP). While we configured all of the interceptor mechanisms not to intercept calls performed for searching constraints within the repository, only the Java proxy approach does not modify the Java byte code. Consequently, we attribute the different runtime behaviour of JBoss AOP and AspectJ with an optimized and a non-optimized repository to the byte code modifications of the AOP tools. Compared to the Java proxy we see an additional runtime overhead introduced by the AOP approaches between 1.07 (JBoss AOP with optimized repository) and 2.50 (AspectJ with optimized repository).

We further evaluated the performance of constraint lookups based on combinations of different numbers of classes (25, 50, 100), methods per class (10, 25, 50), and constraints (at least one per method as constraint lookup does not depend on the number of constraints). The used repository performs caching of results and the provided lookup times assume that the repository is already fully initialized, e.g. after an initializing run. Consequently, the lookup operation reduces to a single lookup in a hash table. Our evaluation showed that the lookup time is in the range of 0.25–0.52 microseconds and the time does not depend on the number of entries in the hash table [Glo07]. These times could also be confirmed for the sample application that was run a number of times with and without lookups to the repository. Depending on the interceptor mechanism, a single constraint lookup took between 0.18 (Java proxy) and 0.43 (AspectJ) microseconds. These times were acquired according to (2.2).

$$Lookup\ time = \frac{Total\ time\ with\ lookups\ -\ Total\ time\ without\ lookups}{Number\ of\ lookups} \tag{2.2}$$

Figure 2.5 illustrates the interception overhead introduced by the different mechanisms. In this case, the intercepted method invocations were immediately forwarded by the interceptors to the called method of the object instance. Hence, the overhead of R1+R2 was compared to R1 (the plain application). This comparison shows that AspectJ provides the fastest interception mechanism, requiring 2.38 times the runtime of the plain application. JBoss AOP shows an overhead factor of 9.25 and the Java proxy requires 28.13 times the runtime of the plain application. As the Java proxy is part of the Java reflection mechanism and does nothing more than invoking the intercepted method via java.lang.reflect.Method.invoke(...), we primarily attribute this major performance impact to the Java reflection mechanism.

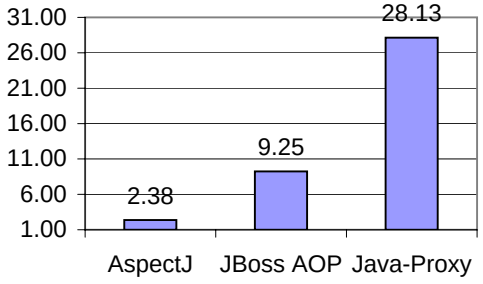


Figure 2.5: Interception overhead: $\frac{R1+R2}{R1}$

The performance advantage of AspectJ gained through quick interception, however, is lost during parameter extraction. While JBoss AOP and the Java proxy mechanism already provide access to the called Method via a java.lang.reflect.Method object, this reference to the method has to be obtained via costly calls to Object.getClass().getMethod(...) in AspectJ. Hence, the overhead of R1+R2+R3 compared to R1 provides a different order between the interception mechanisms, ranging from an overhead factor of 19.50 for JBoss AOP over 36.62 for the Java proxy to 98.26 for AspectJ, see Figure 2.6.

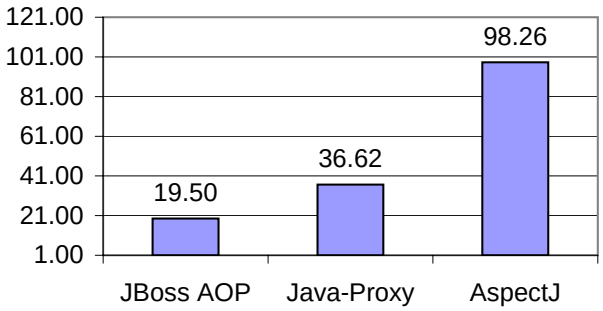


Figure 2.6: Overhead of invocation interception and parameter extraction for searching the constraint repository: $\frac{R1+R2+R3}{R1}$

2.4 Conclusion

Integrity management in software systems has already been addressed by several researchers and a range of possible solutions exists. However, the selection of an appropriate solution for a specific system will also include implementation, maintainability, and performance considerations. Within the previous sections we described several constraint validation approaches and contributed by discussing advantages and disadvantages of the different approaches including performance issues. Our results show that the benefits of some of the more advanced approaches are certainly worth their costs by introducing a runtime overhead of only two to ten times the runtime of the fastest approach while other approaches introduce runtime overheads of more than 100, which might be simply too slow in certain applications.

To sum up, handcrafted constraints showed to be the fastest approach. However, to only separate constraints from, e.g., the code for the business logic, using constraints encoded as aspects in AspectJ is a good choice, requiring

only 1.06 times the runtime of handcrafted constraints. If flexible runtime handling of constraints is required, e.g., if it should be possible to add, remove, enable, and/or disable constraints during runtime, an optimized constraint repository paired with the JBoss AOP toolkit as interceptor mechanism should be envisaged. With respect to performance, the automatically generated constraint checks by JML and especially the Dresden OCL toolkit were part of the slower approaches. However, JML provides several tools to thoroughly support the design by contract principle. If one is only interested in stating constraints, a thorough support of design by contract, and it is acceptable that java.lang.Errors are thrown in case of contract violations, JML is certainly a good choice. The Java annotation mechanism introduced in version 5.0 would be an alternative to the definition of constraints/contracts in comments, allowing also runtime access to the constraints. This approach, however, has not yet been exploited.

Chapter 3

Balancing Integrity and Availability

In a distributed system, where objects are located at different nodes, constraint validation is affected by node and link failures as some affected objects might not be available. If the objects are replicated, we might be able to validate the constraints (partially based on backup copies). However, if updates on replicas are allowed in different partitions, we cannot be sure whether the validation is reliable, because backup replicas of affected objects might be stale due to an update in another partition. Consequently, if we require high availability for our system and want to continue operation even in the face of such situations, we will potentially introduce inconsistencies into the system.

To use constraints as a flexible means to limit the degree of inconsistency potentially introduced during degraded system periods, we classify constraints into tradeable and non-tradeable. Non-tradeable constraints are critical for correct operation of the system and must never be violated. Tradeable constraints must be satisfied in a healthy system—during degraded mode, however, they might temporarily be relaxed in order to increase availability. The decision on *whether a constraint is tradeable has to be provided by the application developer according to an application's requirements.*

This classification between tradeable and non-tradeable constraints is mainly useful for invariant constraints, because constraint validation can be performed at any time and hence be decoupled from business activities. Therefore, invariants can be used for re-establishing constraint consistency during system reconciliation. Pre- and postconditions can—in principle—be traded as well. If necessary at all, the effects of such trading have to be compensated by invariant constraints as pre- and postconditions cannot be re-evaluated in the reconciliation phase.

3.1 The notion of a consistency threat

In a distributed system, the validation of integrity constraints is more complex as constraint validation itself becomes subject to node and link failures. Consequently, there are three different categories of constraint checks:

- *Full Constraint Check (FCC)*: Constraint checking is possible without restrictions. All affected objects are up-to-date.
- *Limited Constraint Check (LCC)*: Constraint checking is possible, but some affected objects are possibly stale. For example, in the case of a primary-backup protocol only a backup replica is reachable that might have missed updates performed on the primary copy since the partitioning occurred. Consequently, the validation result might be something different than if we would validate based on the primary copies.
- *No Constraint Check (NCC)*: Constraint validation is impossible due to the unavailability of at least one affected object (no replicas accessible).

A *consistency threat* occurs whenever we can only perform an LCC or cannot validate a constraint at all (NCC). Of course, in a system that does not use replication to provide fault tolerance, LCCs are not possible due to the lack of redundancy.

Figure 3.1 provides an example for a consistency threat assuming a primary-backup replication protocol where write-access is only allowed on the primary copy in any case. Constraint C1 affects objects O1 and O2. The system suffers from a link failure splitting the four computers in two partitions, each containing two computers. In partition A, we can validate C1 based upon the primary copy of O1 and the backup copy of O2. As the backup copy of O2 is possibly stale, we can only perform an LCC for C1 which results in a consistency threat. The situation in partition B is similar for O2 with a backup of O1. Differently, a constraint validated on O1 and O3 in partition A could be validated without restrictions (FCC).

Interestingly, (constraint) inconsistencies can be introduced in such a system, although replica write-write conflicts are not possible, e.g., write access is only allowed on the primary copy. These inconsistencies result from the fact that replication usually considers single objects while constraints might have several affected objects of different partitions and consequently might require read access to possibly stale copies for validation.

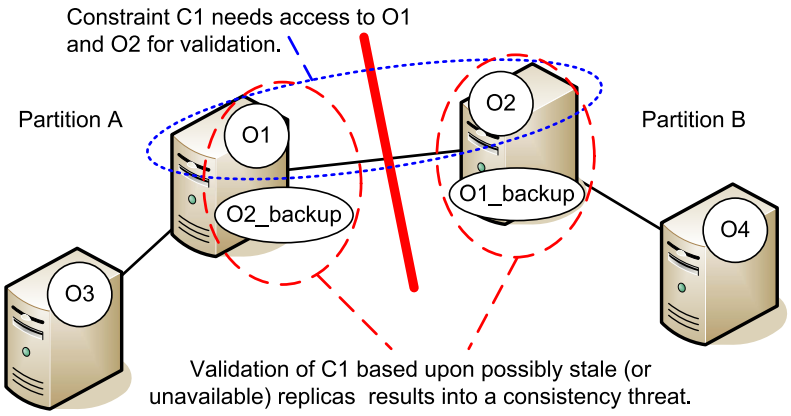


Figure 3.1: Consistency threats

A more specific consistency threat example can be provided for the ATS application scenario (Figure 1.5 on page 8). Imagine that the technical operator sets the componentKind of a RepairReport while the system operates in degraded mode, suffering from a network partition between (the primary copies of) an Alarm and a RepairReport object. The administrative operator might have changed the corresponding Alarm in the meantime in the other partition. Consequently, the constraint validation performed because of the changed RepairReport is not fully reliable, resulting into a consistency threat.

Combining the previously defined constraint checks with the general definition of a constraint to be either satisfied or violated provides three additional constraint validation results (*satisfaction degrees*), identifying a consistency threat: *possibly_satisfied* and *possibly_violated* in case of an LCC—*uncheckable* in the case of NCC. However, differentiation between these three results is only useful if combined with further application-specific knowledge. For the ticket-constraint example in Figure 1.6 on page 9, we would accept possibly_satisfied, meaning that we potentially sell more tickets than available seats while possibly_violated indicates that we would already sell more tickets than available. This differentiation is based on the assumption that tickets are mainly sold and rarely returned. On the other hand, for the ATS application, it is also reasonable to accept possibly_violated constraints under the assumption that the technical operator exactly knows the repaired component, but is not allowed to change the alarmKind that must be changed by the administrative operator, which operates in a different network partition, for example.

However, this enhanced set of possible validation results for a single constraint requires a specification of how the validation results of a set of con-

straints are combined into a single validation result for the whole set. Obviously, the overall outcome should be a consistency threat if at least one constraint validation provides a consistency threat. Therefore, the overall outcome is:

- *Satisfied*: if all constraints in the set are satisfied.
- *Possibly satisfied*: if all constraints are either satisfied or possibly satisfied and at least one constraint is possibly satisfied.
- *Possibly violated*: if all constraints are either satisfied, possibly satisfied, or possibly violated and at least one constraint is possibly violated.
- *Uncheckable*: if at least one constraint is uncheckable and none is violated.
- *Violated*: if at least one constraint is violated.

Determining possibly stale objects. Typically, in order to provide replication transparency, respectively application independence from a particular replication protocol, a proxy object serves as interface between the application and the replication protocol. For the application, this proxy object provides a local view onto the logical object based on the reachable replicas. In our case, this object view becomes possibly stale if updates on the same logical object can occur in another network partition. Whether or not an object[1] is possibly stale depends on the presence of node/link-failures and the underlying replication protocol. For example, in the primary partition protocol [RSB93], each object accessed in a non-primary partition is possibly stale. In the case of the primary-per-partition protocol [BBG+06], objects are possibly stale in every network partition.

Intra- vs. inter-object constraints. Intra-object constraints are constraints that can be evaluated on a single object and require access to the (primitive/value) attributes of the object only. Inter-object constraints need access to more than a single object (Figure 3.2). If the reconciliation process merges conflicting replicas of a single object through selection of one copy (and not by creating a new replica by merging values of disjoint sets of attributes of the different replicas), a differentiation of integrity constraints into

[1]For simplification, we use the term "object" as synonym for the local object view onto the logical object.

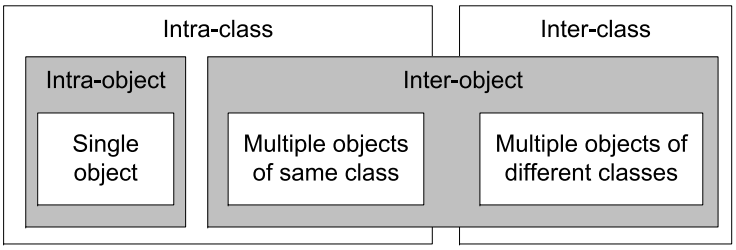


Figure 3.2: Intra- vs. inter-object constraints

intra- vs. inter-object constraints is reasonable as intra-object constraints will not be violated retrospectively by the replica reconciliation process. Hence, constraint validations performing an LCC can return "satisfied" instead of "possibly_satisfied" for intra-object constraints. This reduces the number of consistency threats and hence the amount of associated information gathered during degraded mode and required to be processed in the system reconciliation phase. Inter-object constraints could be further classified into intra-class (all objects of the same class, e.g., uniqueness of an attribute for all objects of a class) and inter-class (objects of different classes, e.g., Figure 1.6) constraints. Although this differentiation is useful for constraint implementation and tool support, it is not significant with respect to our balancing of dependability.

3.2 Balancing integrity and availability

Building upon explicit constraint management, constraint classifications and a validation result that takes system degradation into account enables us to explicitly balance integrity and availability during degraded system periods. For this balancing, we *decouple constraint validation* from the current business activity *in the time dimension* by postponing reliable constraint validation until we can perform an FCC for the threatened constraints of the current business activity. Obviously, to which extent integrity can be traded for availability depends on the particular application.

The application-specific trade-off is configured through the specification of tradeable and non-tradeable constraints. Consistency threats for non-tradeable constraints are automatically rejected with the usual effect that the current operation/transaction is aborted. Consistency threats for tradeable constraints are subject to a negotiation mechanism to decide whether to accept or reject the consistency threat. The negotiation mechanism will base

its decision on parameters such as the constraint satisfaction degree and/or the affected objects. However, if the consistency threat is accepted, the system stores this threat and allows to associate some information with this threat such as affected objects or application specific data.

If in the worst case all constraints are non-tradeable and all objects of the application are covered by at least one constraint, the application completely blocks during degraded system periods—a fallback to conventional system behaviour. Whether the system blocks for write-operations only or for read- and write-operations depends on the configuration of constraints, affected methods, and the applied replication protocol.

3.2.1 Negotiation of consistency threats

For negotiation of whether or not to accept consistency threats, we differentiate between two kinds:

- *Static (descriptive) negotiation*: is configured based on the satisfaction degree of a constraint and optionally some freshness criteria for possibly stale affected objects. For example, the consistency threat of a specific constraint might be acceptable if the satisfaction degree is "possibly_satisfied" and the last update of the affected objects is not older than n seconds. However, additional parameters could be considered as well.
- *Dynamic (algorithmic) negotiation*: is performed by using an application implemented callback handle—the NegotiationHandler. A NegotiationHandler can be registered with a transaction of the application to associate the negotiation mechanism with a specific use case. This kind of negotiation can be performed with or without user intervention.

The priority of the different negotiation mechanisms is set to prefer a dynamic negotiation handler over static negotiation decisions over a default application wide minimum constraint satisfaction degree. Alternatively, the descriptive declarations could be used as a boundary within which dynamic negotiation can be performed. A general overview of the negotiation process is provided in Figure 3.3.

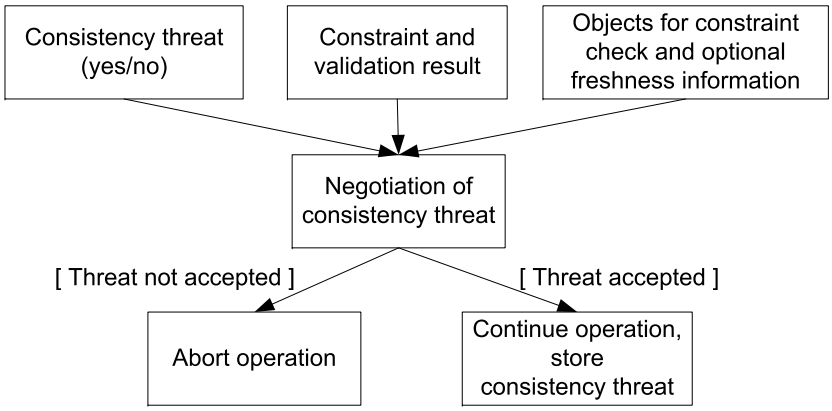


Figure 3.3: Overview of the negotiation process

3.2.2 Preparation for reconciliation

Whenever we accept a consistency threat, we have to store some information about the consistency threat to be able to evaluate during reconciliation time whether or not we have actually introduced an inconsistency into the system. For re-evaluation, we have to store at least the unique name identifying the constraint that produced the consistency threat. Moreover, depending on the "starting point" of constraint validation, we have to differentiate two cases:

1. Validation of the constraint starts from a context object. In this case we have to store at least an identifier for the context object which is later used as input to the constraint validation method.

2. Validation of the constraint starts from a set of objects, obtained by a query operation. In this case, the constraint needs no input to the validate method. Hence, no further information is required in addition to the unique name of a constraint.

The previous requirements only state the minimum information necessary to re-evaluate accepted consistency threats during the reconciliation process. This information can be further enriched by storing identifiers or even the serialized state of affected objects at the time the consistency threat occurred. Moreover, we allow the application to associate application specific data with a consistency threat. Finally, the application can also give reconciliation instructions such as to allow rollbacks to be performed during reconciliation.

On the other hand, stored information can be reduced, if rollback/undo operations to intermediate states are not required in which case *identical*

consistency threats need to be stored only once. Two consistency threats are identical if both of them refer to the same integrity constraint and—if applicable—to the same context object.

3.3 Reconciling constraint consistency

So far we considered operation in a healthy system and during degraded mode. After network links are repaired or nodes recovered, we have to re-evaluate accepted consistency threats. For this process, we perform a re-validation of associated constraints. Depending on the result of the constraint validation, we take different actions:

- *Constraint is satisfied.*
  - If there was no replica conflict (or no replication is used), remove the threat and all identical threats from the set of accepted consistency threats.
  - If there was a replica conflict for the constraint and a reconciliation instruction of at least one of the identical threats specifies that the application should be informed of this situation, notify the application.
- *Constraint is violated.*
  - If the accepted consistency threat has an associated reconciliation instruction specifying that rollback/undo is allowed, re-evaluation can be performed based on available (serialized) historical states. If a consistent state is found, the state of the affected objects is rolled back. Unfortunately, availability of the system is retrospectively reduced as some updates do not become effective. Even more so, as recovery may suffer from the "domino effect" [Ran75], the advantage of our approach may become completely diminished. If no consistent state is found at all, a callback handler provided by the application is invoked to solve the constraint violation.
  - If rollback/undo is not allowed, the system has to reconcile by using a compensation approach, e.g., similar to the WS-Business Activity standard. [ABH+05b]. For this process, the application provided callback handler is invoked to reconcile the constraint

violation. In this scenario, our approach provides the greatest benefit, because the overhead to store the threat information is minimal.

As an alternative to solving the violation, the system could deactivate violated constraints in order to reach the healthy state, thereby relaxing consistency. Similar to introducing new integrity constraints to the system, constraints that were disabled and are enabled again have to be checked for all context objects. This, however, is not within our focus.

- *Constraint is threatened.* If the constraint is still threatened in the reconciliation phase, at least one affected object is still not fully available. Hence, although some network partitions might have been re-unified, some partitions still exist and the system still operates in degraded mode. In this case, re-evaluation of the constraint has to be postponed until further partitions are re-unified.

Parallel reconciliation and business operations. During reconciliation, it is not feasible to block the system for business operations until the whole reconciliation process is finished. Business operations that partially involve still threatened objects can either block, if the reconciliation is already underway or be treated as if the partition were still in place, thereby introducing new threats. Additionally, business operations can also be used to remove existing consistency threats for constraints that are satisfied by the current operation. In parallel, business operations with only unthreatened objects can continue in healthy mode.

Chapter 4

Middleware Support for Adaptive Dependability

We integrated the concepts for adaptive dependability by trading integrity for availability into a platform independent system architecture [15], which has been implemented in different prototypes using several technologies (EJB [3], CORBA [BGM07], and .NET [17]). These prototype studies have been performed within industrial settings in strong cooperation with companies from communications and control engineering industry within the DeDiSys project (`http://www.dedisys.org/`).

Within this dissertation, we concentrate on how our general concepts and the general architecture were mapped to and integrated into the EJB middleware platform as provided by the JBoss Application Server (JBoss AS). First, we provide an overview of the system architecture as specifically mapped to EJB and the JBoss AS with a focus on the middleware (MW) layer. Second, we contribute with a detailed description of constraint consistency management as a new middleware service. Moreover, we will discuss the issue of replication support and give an insight into the implementation of the reconciliation phase. Finally, we contribute with a problem discussion of (negotiation) callbacks in Web-based applications along with a corresponding solution.

4.1 System architecture

Two components of our architecture are primarily responsible for the balancing of availability and integrity, the *replication service* (RS) and the *constraint consistency manager* (CCMgr) provided within the grey area of Figure 4.1.

Other important components are the *invocation service*, used for interception of point-to-point invocations, the *transaction manager*, managing distributed transactions, *persistence* to store application data, information about consistency threats, and historical replica versions to allow for rollback during reconciliation, the *group membership service* (GMS) to detect node and link failures as well as re-joins of nodes after recovery or network re-unification, and the *group communication* component that is used for update propagation (from primaries to backups) by the replication service. The *naming service*, allowing for name to object bindings, and the *activation service*, responsible for appropriate activation of objects, are not of immediate interest with respect to our solution.

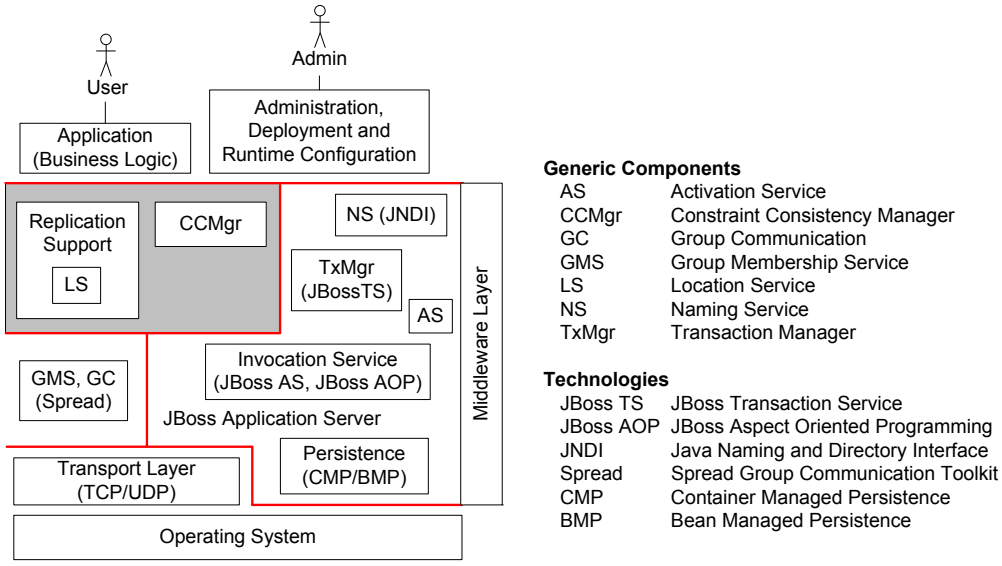


Figure 4.1: EJB/JBoss AS specific system architecture

Although quite common and concise, the layered representation in Figure 4.1 is not sufficient to illustrate how the components cooperate by using each other: First, strict layering is often not possible and second, layering does not imply actual usage. Therefore, Figure 4.2 provides an overview of usage-relations between the major components. This figure shows that transaction management and invocation service are the two central services where almost all of the other services depend upon.

Above the middleware, we differentiate between two types of applications and users. First, we have the administrator who is responsible for proper administration, deployment, and runtime configuration of the middleware as well as the application. These tasks may be supported by tools and appli-

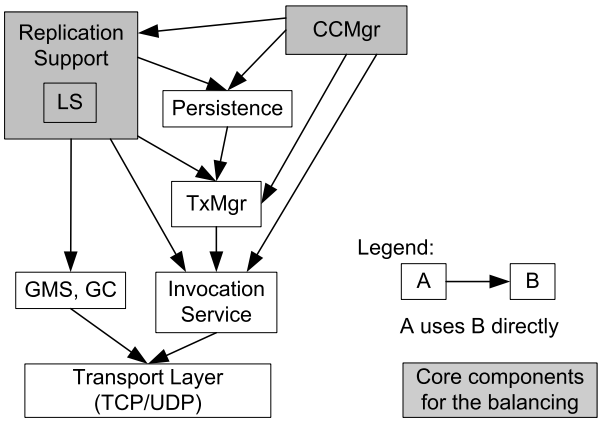


Figure 4.2: Usage relationships of system components

cations different from the application performing the business logic of the general users, which is the second category of users. Administrators are assumed to have special knowledge about the application and the middleware and maintenance and adaptation are their main tasks. The general users are mainly interested in performing their business and do not need in-depth knowledge about applications or middleware.

4.2 Constraint consistency management

Explicit runtime constraint consistency management is a new middleware service we introduced for balancing integrity and availability. In our approach, constraints are explicitly available during runtime and validated upon request of the middleware. The specification/implementation of constraints is up to the application developer as they result from the application requirements. On the other hand, triggering the validation of constraints as well as detection and management of consistency threats is performed by the middleware.

4.2.1 Explicit (runtime) constraint representation

Obviously, constraints are processed by the middleware (management, triggering validation, etc.) as well as the application (performing the actual validation). Hence, this concept needs a contract between the two parties. For this purpose, we encapsulate the integrity constraints within explicit con-

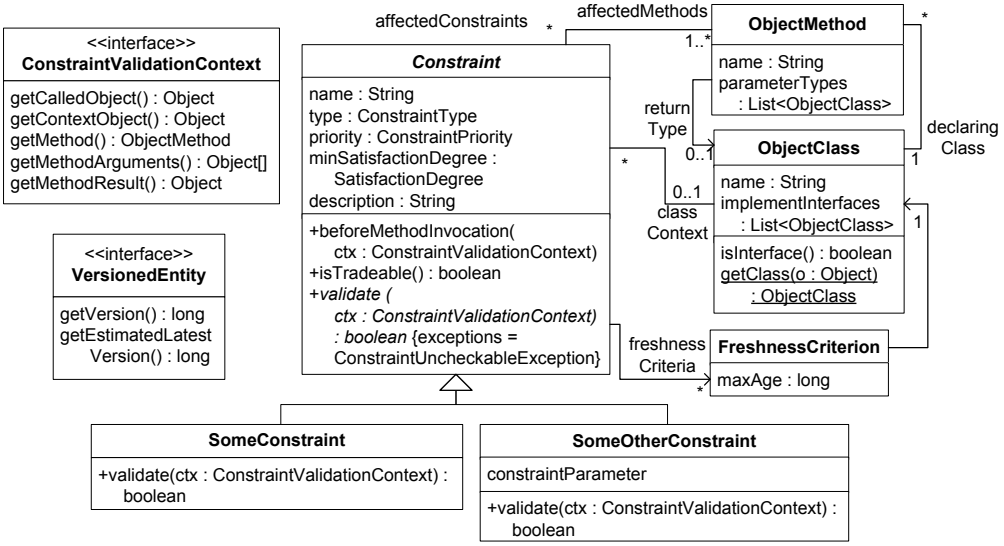


Figure 4.3: Constraint runtime model

straint classes similar to Verheecke et al. [VS02]. The primary contract between middleware and application is the Constraint.validate(ctx : ConstraintValidationContext) method (Figure 4.3) that has to be implemented by the application developer and provides true or false as return value or throws an exception to indicate that constraint checking is impossible, e.g., due to unreachable objects. The middleware's responsibility is to ensure that validate(...) is called appropriately. Moreover, the beforeMethodInvocation(...) call to a constraint supports postconditions that check whether state transitions caused by a method call are correct. Within this call, a postcondition might store some values (state before the method invocation) and check during the call to validate(...) whether the method invocation actually produced a correct result with respect to the state before the method invocation. This corresponds to the @pre operator of OCL.

The content of the ConstraintValidationContext provided to validate(...) depends on the type of a constraint and the circumstances under which the constraint is validated. It generally contains:

- The *context object* for invariant constraints, i.e., their "starting point" for constraint validation. Starting from this object, the constraint is able to reach all objects that are needed for validation of the constraint. For example, the context object for the OCL expression context Person inv: getAge() >= 18 would be an instance of the context class Person.

- The called object, method, and method arguments for preconditions.
- The called object, method, method arguments, and result for postconditions.

To allow the middleware to trigger constraint validation appropriately, the affected methods have to be specified in addition to the constraints. Moreover, the context class can be specified for invariant constraints. Some invariant constraints, however, may not need a context object as they, for example, use a query operation to get their affected objects.

Finally, constraints may have associated freshness criteria (maximum age), one per affected class of objects (ObjectClass). These classes have to implement the VersionedEntity interface that allows to retrieve the version of the object getVersion() and the estimated latest version getEstimatedLatestVersion(). The estimated latest version is the one that the object would expect to have. For example, if an object is usually updated every n seconds and the last update producing version v happened $3n$ seconds ago, getVersion() would return v while getEstimatedLatestVersion() would return $v+3$, indicating that the object most probably missed 3 updates. This mechanism can be used by the application developer to specify conditions for the negotiation of consistency threats.

4.2.2 Constraint configuration and registration

To allow appropriate validation, we need to know which constraints are affected by which method invocations. As motivated in Section 1.6, we require the application developer to declare constraints and affected methods as well as other details about a constraint, e.g., the constraint type or freshness criteria, in a configuration file. Similar to the EJB deployment descriptor, the constraint configuration file is read after deployment of an EJB application. The information contained in this file is then used to register the constraints within a constraint repository. This constraint repository allows to look up constraints, e.g., by class, method or constraint type. Listing 4.1 provides an example of a constraint specification within the configuration file.

The constraint ComponentKindReferenceConsistency implements the integrity constraint of the ATS application provided in Figure 1.5 on page 8. It is a hard constraint, specifies that the constraint implementation requires a context object, it can be relaxed during degraded mode, and the negotiation process will accept any consistency threats (minSatisfactionDegree="uncheckable")—if no negotiation callback handle is registered by the application to

Listing 4.1: Constraint configuration example

```
<constraint name="ComponentKindReferenceConsistency"
  type="HARD" priority="RELAXABLE" contextObject="Y"
  minSatisfactionDegree="UNCHECKABLE">
  <class>ComponentKindReferenceConstraint</class>
  <context-class>RepairReport</context-class>
  <affected-methods><affected-method>
    <context-preparation>
      <preparation-class>CalledObjectIsContextObject</preparation-class>
    </context-preparation>
    <objectMethod name="setAffectedComponent">
      <objectClass>RepairReport</objectClass>
       <arguments><argument>java.lang.String</argument></arguments>
    </objectMethod>
  </affected-method><affected-method>
    <context-preparation>
      <preparation-class>ReferenceIsContextObject</preparation-class>
      <params><param name="getter" value="getRepairReport"/></params>
    </context-preparation>
    <objectMethod name="setAlarmKind">
      <objectClass>Alarm</objectClass>
      <arguments><argument>java.lang.String</argument></arguments>
    </objectMethod>
  </affected-method></affected-methods>
</constraint>
```

be dynamically contacted for a threat-specific decision. A consistency threat occurs whenever the satisfaction degree of a constraint is possibly_satisfied or possibly_violated (constraint validation based on possibly stale objects) or uncheckable (e.g., due to unreachable objects). Considering constraint violations the least acceptable situation and satisfied constraints the desired case, we apply the following ordering of satisfaction degrees: violated < uncheckable < possibly_violated < possibly_satisfied < satisfied.

The <class> element specifies the Java implementation class of the constraint that will be instantiated while the configuration file is read during the deployment of an EJB application. The <context-class> is the class of the context object (RepairReport) required for constraint validation. Within the <affected-methods> element, affected methods of the constraint are provided. Each affected method is specified by stating the declaring class, the method name, and the method parameters. As the constraint is implemented for a specific context class, the ConstraintValidationContext (see Figure 4.3) must be initialized appropriately. Values such as called object, called method, and method parameters are already set by the middleware. However, the <preparation-class> is responsible to extract the context object based on these values. The context object for the method RepairReport.setAffectedComponent(...) is the called object itself while the context object for the method Alarm.setAlarmKind(...) is obtained by calling getRepairReport() upon the called object (an instance of Alarm).

4.2.3 Constraint consistency manager

The CCMgr is notified by the invocation service before and after method invocations. Upon such notifications, the CCMgr looks up preconditions, postconditions, hard and soft invariant constraints and triggers validation according to their constraint type. To allow such behavior of the CCMgr it is also registered with the transaction manager (TxMgr) as a transactional resource to take part in the two-phase commit. If any constraints are violated, the CCMgr sets the state of the current transaction to "rollback-only". Hence, any constraint violation (or unacceptable consistency threat) prevents an ongoing transaction from successful commit.

In degraded system mode, the CCMgr provides additional functionality to support the integrity/availability balancing by interacting with the replication manager in order to detect consistency threats caused by possibly stale objects. Before the CCMgr triggers the validation of a constraint, it starts to gather accessed objects, see Figure 4.4. After the constraint validation returns, the CCMgr asks the replication manager whether any of these objects are possibly stale. If this is the case, the validation result (satisfaction degree) of the constraint is changed from satisfied to possibly_satisfied, or from violated to possibly_violated, as the constraint validation is not fully reliable. If there were any unreachable objects, the validation result of the constraint is uncheckable. These situations indicate a consistency threat and trigger negotiation of the threat.

To perform algorithmic negotiation, the application must register a negotiation callback handler with the CCMgr. Such a negotiation handler is bound to the current transaction and responsible to decide whether to accept or not accept arising consistency threats. If no negotiation handler is registered at the CCMgr, declarative negotiation is performed based on the current satisfaction degree, the configured minimum satisfaction degree, and—if applicable—given freshness criteria. For this process, the current satisfaction degree of the constraint is compared with the minimum satisfaction degree. Moreover, the difference getEstimatedLatestVersion() - getVersion() is compared with the maximum age defined by available freshness criteria. Both, minimum satisfaction degree and optional freshness criteria are specified in the constraint configuration file.

Not accepting a consistency threat results in rollback of the current transaction. If a consistency threat is accepted, the consistency threat as well as application-specific information associated with the threat is persisted and used later during the constraint reconciliation phase. To reconcile con-

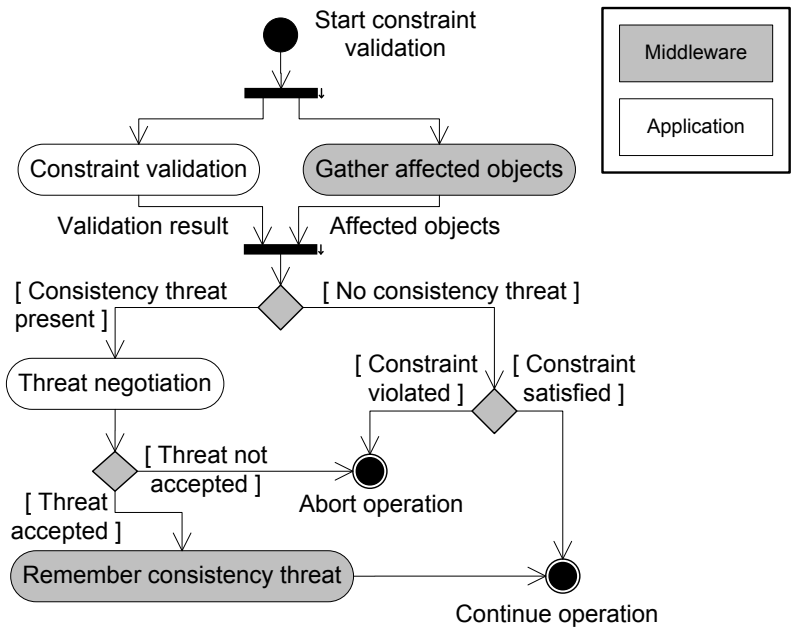


Figure 4.4: Detection and negotiation of consistency threats

straint consistency, the constraint consistency manager looks up accepted consistency threats, re-evaluates the corresponding constraints and takes appropriate actions according to Section 3.3.

4.2.4 Invocation interception

A key requirement for middleware integration of constraint consistency management is the possibility to intercept invocations. In EJB, each component and hence entity bean must provide a home and a business interface. These interfaces are implemented by the EJB container (a JBoss proxy in our case). After a call to the interface implementation, the EJB container can perform several middleware tasks, e.g., association of a security context or transaction with the call, before it finally forwards the call to the bean implementation.

In the case of JBoss, the JBoss proxy builds up an object representing the invocation and passes this object through the client-side interceptor chain, where each interceptor invokes the next interceptor until the final interceptor, the client-side JBoss invoker, transfers the invocation object over the network to the server-side JBoss invoker. Thereafter, the invocation object passes through the server side interceptor chain where again each interceptor invokes the next interceptor until the final interceptor, the JBoss EJB container,

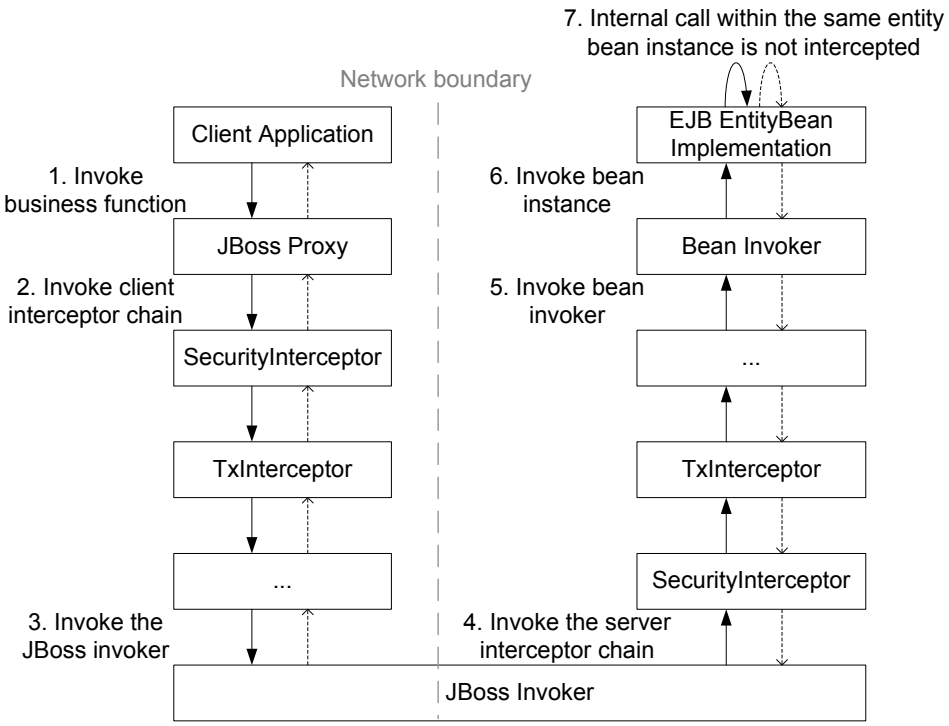


Figure 4.5: JBoss invocation interception

invokes the bean instance. The result of the invocation is passed back in the reverse order. This concept of client-side and server-side interceptor chain is illustrated in Figure 4.5. Of course, if the invocation is performed locally on the JBoss server instance, no marshalling over the network occurs.

The interceptors are responsible to provide middleware services for the invocation, e.g., transaction management or security checks like authentication or authorization of the call. Fortunately, the invocation interceptors of the chain can be specified in a configuration file of the JBoss AS (standardjboss.xml) and therefore enhancing JBoss with additional functionality is rather easy to achieve. Consequently, it was only necessary to implement a new interceptor and put it into the interceptor chain. This interceptor is then responsible for appropriately including the CCMgr for constraint consistency management within the process of an invocation. The implementation of the replication protocol is based on the ADAPT replication framework [BBM+04], which also hooks into JBoss through custom interceptors.

Unfortunately, the interceptor chain is only traversed if the invocation comes from a call to the interface implemented by the JBoss proxy, which then passes the invocation through the interceptor chain. If the bean instance

calls another method on itself, this (internal) invocation is not intercepted. This applies to call number 7 in Figure 4.5.

This behavior would prevent any affected constraints of internal invocations from being checked. This issue can be solved by using the JBoss aspect oriented programming (AOP) framework with which plain Java method invocations can be intercepted. Similarly to the approach above, the AOP framework transforms invocations into explicit invocation objects and calls interceptors registered with the AOP framework. Hence, we are able to use the same approach as above for triggering constraint validations for internal invocations as well.

4.3 Replication support

To maximize availability for systems capable of applying our concept of adaptive dependability, the middleware should provide replication support. Within our prototype, we implemented the primary-per-partition protocol (P4) [BBG+06] to replicate the state of entity beans. The P4 behaves like a traditional primary-backup replication protocol in a healthy system with the specific setup that each object might have its primary on a different node instead of using only a single designated primary server node. However, during degraded mode, a temporary primary is chosen per partition. This further increases availability because operations can be performed on objects in different partitions as long as only acceptable consistency threats occur. In order to prepare for the reconciliation phase and allow for generic rollback, the replication protocol stores intermediate states applied during the degraded mode. During repair, detected conflicts are solved either by rollback to previous states or by an application-specific compensation callback. For an in-depth specification of the protocol, please refer to [BBG+06].

For the implementation of this replication protocol, we build upon the ADAPT replication framework [BBM+04]. This framework is a convenient toolkit for prototyping EJB replication algorithms within the JBoss Application Server. ADAPT plugs into the application server via invocation interceptors and provides the necessary callbacks to intercept certain events, e.g., creation of, calls to, or deletion of EJB entity beans, and allows appropriate replication actions.

These callbacks are implemented by the client-side and server-side component monitor that have to be provided by the replication algorithm in order to achieve the replication tasks. The client-side component monitor can redirect

calls to different servers, for example. The server-side component monitor is notified about the different events mentioned above and can perform different actions before the control is passed to the bean implementation or after control returns from the bean implementation. For example, actions before the bean invocation include server-side request redirection, actions after the bean invocation include update propagation. Invocation interception also works for nested invocations as long as the request passes through the JBoss interceptor chain as described in Section 4.2.4.

The replication implementation performs update propagation after write requests to an entity bean. Detection of write requests is performed according to the EJB specification in the way that all methods starting with `set` plus an upper-case letter are considered write requests. After such a request, the state of the replica and the transaction propagation context are extracted by using the mechanisms provided by ADAPT, the data are packed into a message and multicast via the Spread group communication toolkit (`http://www.spread.org`) to the backup replicas. The backup replicas extract the information, associate the call with the transaction of the primary, apply the update within this transaction to the backup replica, and send a confirmation back to the primary replica. Due to this synchronous update propagation, read requests can always be performed locally. Further details about the replication implementation are found in [Ke07].

Further reading. Another replication protocol that allows for the same kind of balancing between availability and integrity is the Adaptive Voting Protocol [7]. This is a quorum-based protocol that adapts the quorum sizes in order to allow operation in degraded mode. Of course, only operations producing acceptable consistency threats are allowed. Based on the P4 and the Adaptive Voting Protocol, a generalized model for replication protocols allowing for a balancing between availability and integrity is presented in [11]. A discussion of a several replication middleware architectures including a generalized architecture is provided in [10]. Furthermore, a summarized and integrated version of all the replication issues can be found in the dissertation of Johannes Osrael [Osr07].

4.4 Reconciliation phase

In the reconciliation phase, inconsistencies within the system have to be solved in order to get a consistent, healthy system. This phase is performed

in two steps as illustrated in Figure 4.6. After the group membership service (GMS) notifies the replication service (RS) that one or more previously unreachable nodes joined the current partition, the replication service starts to propagate the updates of the current partition to the joined nodes. Of course, the updates performed on the joined nodes are propagated to the current partition.

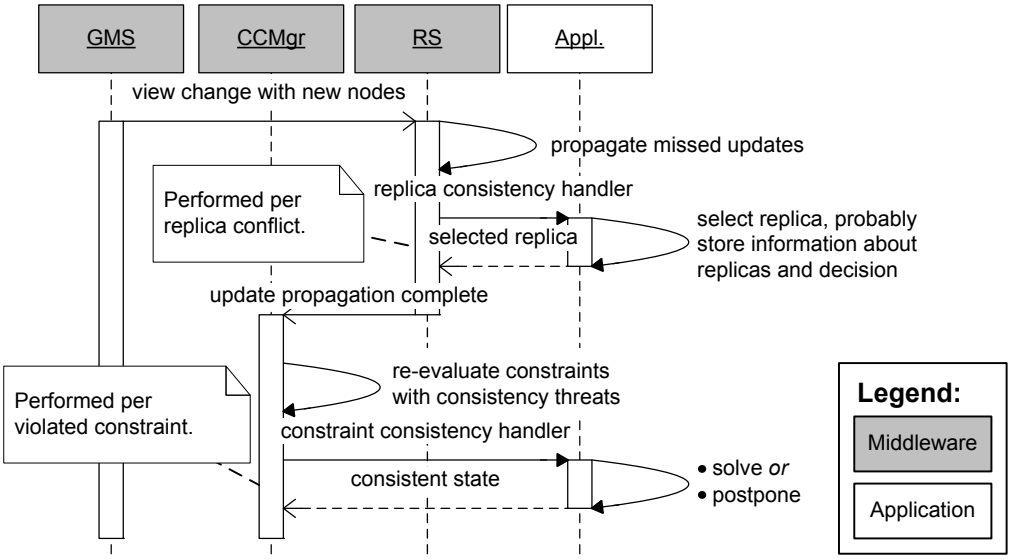


Figure 4.6: Reconciliation phase and callbacks

During this propagation of missed updates, write-write replica conflicts might be detected, i.e., two different replicas of the same object were changed in two different partitions during degraded mode. Consequently, the two inconsistent replicas have to be reconciled in order to produce a replica consistent state for the current partition. As the reconciliation strategy depends on the specific type of object, the middleware performs a callback to the application-provided *replica consistency handler* in order to produce a replica consistent state that is afterwards applied to all nodes of the current partition. Of course, replica consistency could also be established generically, e.g., by applying the state of the primary copy or the replica with the most updates during degraded mode.

However, after the replication service re-established a replica-consistent system state, it notifies the constraint consistency manager (CCMgr) that it should start re-establishment of constraint consistency. For this task, the CCMgr re-evaluates accepted consistency threats and if the corresponding constraint is violated, it notifies the application through the application-

provided *constraint reconciliation handler*. This handler might clean up the inconsistencies immediately, e.g., automatically or by blocking as long as the human operator is working on the clean-up, or return before the inconsistency is solved by providing this fact to the middleware. Differentiation between these two types (immediate vs. deferred reconciliation) works via the return parameter of the callback. If the application returns true, it specifies that the constraint violation is resolved. In this case, the CCMgr will revalidate the constraint and remove the threat if the inconsistency has actually be solved. Otherwise, it will contact the reconciliation handler again. If the application returns false, the reconciliation of the constraint violation will be performed at some time later under the application's responsibility. In this second case, the reconciliation handler might apply asynchronous message passing within the system or send an e-mail notification to the system operator. However, the cleanup of the threat by the application is detected by the CCMgr through the fact that the corresponding constraint is satisfied by a business operation. Subsequently, it will remove the consistency threat from persistent storage at that point in time.

Of course, generic rollback-based solutions to conflict resolution of replica conflicts as well as constraint violations are possible. Consequently, not every constraint violation unconditionally leads to an invocation of the constraint reconciliation handler. However, rollbacks to previous states of the system retrospectively reduces availability as some operations do not become effective at the end. Moreover, it will often not produce the solutions desired by the end user or application. Hence, generic rollback approaches were not within our focus.

4.5 Callbacks in Web applications

Before we investigate the issue of callbacks in Web applications, we shortly summarize the negotiation callback scenario within our middleware for distributed object systems as illustrated in Figure 4.7. At some time before a client makes a call to an entity bean, it registers a consistency threat negotiation handler with the constraint consistency manager (CCMgr). Calls to components in EJB and hence entity beans are made through the invocation service. Several interceptors responsible for different tasks can be registered with this invocation service. One interceptor responsible for constraint consistency management notifies the CCMgr before and after method invocations in order to allow the validation of constraints affected by the current operation. If the system operates in degraded mode, consistency threats

may arise from these constraint validations. These consistency threats have to be negotiated by contacting the registered negotiation handler, potentially crossing a network boundary between the server and the client.

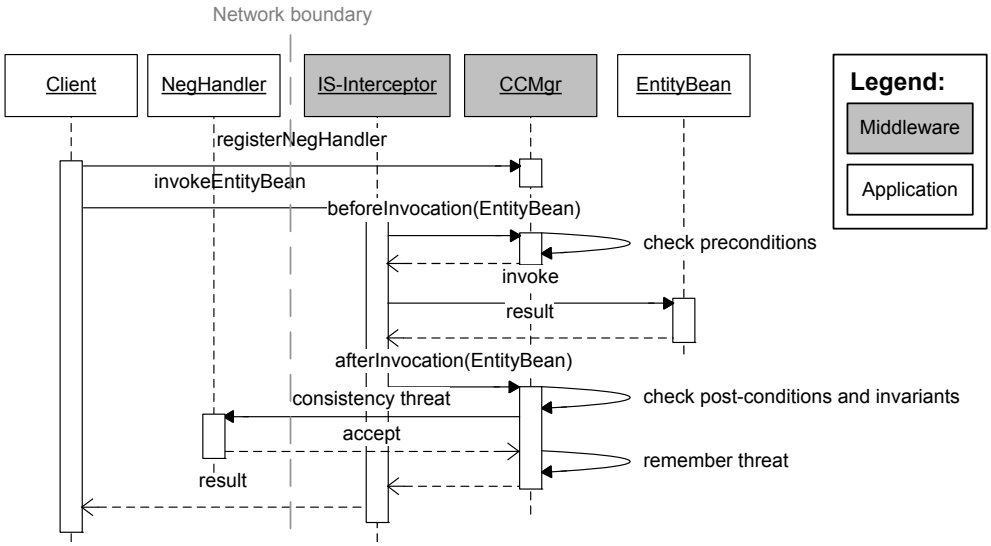


Figure 4.7: Consistency threat negotiation in distributed object systems

While such behaviour is possible and straight-forward to implement in distributed object systems using remote method invocation (RMI), it is more challenging in Web-based systems because a callback to the Web browser is simply not possible, see also first part of Figure 4.8. Moreover, the hypertext transfer protocol (HTTP) is based on a strict request/response behaviour. The business request is sent to the Web server and the browser is waiting for the response. Before the response is actually available, negotiation of potential consistency threats has to be performed, i.e., requests from the middleware to the application are necessary. Consequently, some logic has to be placed into the Web application to match this request/response discrepancy.

When a negotiation request arrives from the middleware, the negotiation logic in the Web application has to extract the information provided to the negotiation handler, examine the situation, provide an appropriate response to the Web browser and block the thread by which the negotiation request was received. Effectively, this forwards the negotiation callback request via the HTTP response for the business request to the Web browser. While the user examines the situation in front of the Web browser, the negotiation thread is blocked in the Web server. The negotiation decision is returned via a new HTTP request. This request is actually the response to the negotiation

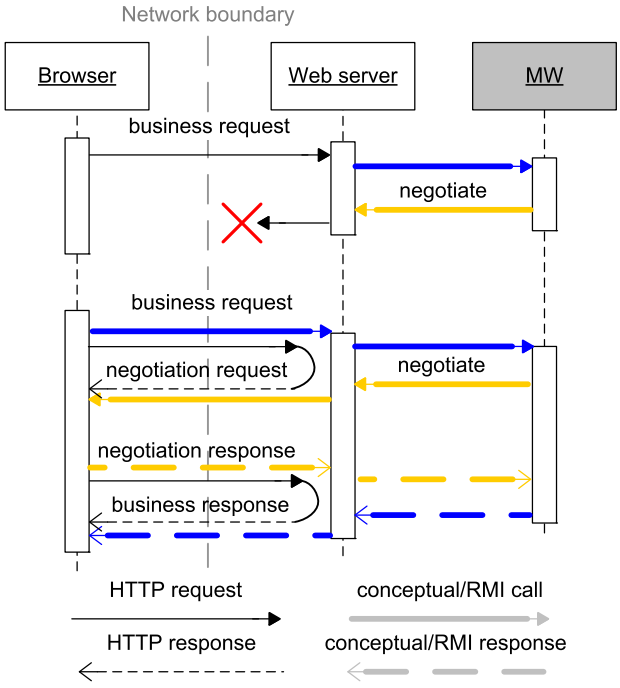


Figure 4.8: Different request/response behaviour of distributed object-based and Web-based systems

callback request. Hence, the Web negotiation logic has to map the request to the waiting negotiation thread. After certain parameters are set, e.g., a boolean flag of whether the consistency threat is accepted, the sleeping negotiation thread is interrupted and resumed. In order not to block the negotiation thread indefinitely, the Web negotiation logic can resume it by not accepting the consistency threat after a timeout. However, in case the threat was accepted, the business operation continues in its usual way, but the results have to be transmitted back to the browser via the HTTP response for the HTTP request providing the negotiation result. This further requires to suspend the new HTTP request with the negotiation result until the result of the business operation (or a new negotiation request) is received.

This request/response discrepancy and its solution are shown in Figure 4.8, where the conceptual requests and responses as they could be performed with RMI are drawn with thick grey lines[1], dark grey for the business requests/responses and light grey for the negotiation requests/responses. The

[1]In coloured printouts, blue illustrates the business requests/responses from the application and orange the negotiation requests/responses from the middleware.

HTTP requests/responses are drawn with thin black lines. Figure 4.8 first illustrates that a callback to the browser from the Web server is not possible in the way of straightforward calls. Thereafter, it shows the solution where the negotiation request is transferred over the HTTP response for the business request and the negotiation response is transferred back to the Web server via a new HTTP request. Finally, the business response is transferred back to the browser via the HTTP response for the HTTP request with the negotiation response.

While the previous concepts allowed the same behaviour and user experience for Web-based applications as for distributed object systems with respect to negotiation, it is partially impossible to achieve this for the reconciliation callback. The reconciliation handler of the application is called by the middleware whenever a violated constraint is detected in the reconciliation phase. Upon such a callback, distributed object systems have the possibility to immediately clean up the inconsistency by potentially involving a human operator and return true to the middleware, stating that the inconsistency was solved and the middleware should re-evaluate the corresponding constraint. Web-based applications can only usefully apply the second option of deferred reconciliation if interaction with a human operator is required. In this case, the application has to take note of the inconsistency, e.g., through database entries or sending an e-mail to an operator, and return false to the middleware to specify that the inconsistency will be solved later and the middleware should ignore the inconsistency at first. This decision is persistently stored by the middleware along with the consistency threat. For the ticket-constraint, the reconciliation handler would mark overbooked flights and send an e-mail to an operator. The operator afterwards rebooks passengers to other flights as appropriate, thereby cleaning up the inconsistencies. In this case it does not matter whether reconciliation is performed through an RMI client or a Web-based client.

We ensured the practical feasibility of our Web approach through an implementation of these concepts in a flight booking prototype following the examples provided in this dissertation. This application builds upon the Struts Web framework (`http://struts.apache.org/`) and an EJB application that is deployed within a JBoss Application Server (`http://www.jboss.org/`) enhanced with support for replication and constraint consistency management.

Chapter 5

Evaluation

The previous chapters introduced to the concept of middleware support for adaptive dependability and provided details about the technical implementation. This chapter subsequently performs an extensive evaluation of the implemented solution as well as the concepts and mechanisms applied. We start with an investigation of the performance impacts of our middleware enhancement for the healthy and the degraded mode before we analyze the reconciliation phase. We continue with lessons learned from implementation and tool usage before we summarize the applied middleware/application interaction mechanisms along with a discussion of specific design alternatives. Finally, we conclude this chapter with a summary of specific performance improvements based on our (intermediate) evaluations.

5.1 Healthy and degraded mode performance

For our performance measurements, we used a mixture of different computers, each between 2–3 GHz and 1 GB of RAM, connected via 100 MBit network links. The configuration denoted as "No DeDiSys" is a standard JBoss AS with JBoss TS as transaction service for distributed transactions and MySQL for persistent storage. The "DeDiSys" configuration additionally applies the principles provided within this dissertation as well as the P4 replication protocol and is measured in healthy mode as well as degraded mode. In order to ensure repeatability of the tests, we used the script-based DedisysTest application described in [Ke07].

The test case performed for measurement started with the creation of 1000 entity beans. Afterwards, a setter for String attributes of these entity beans

was called 1000 times followed by 1000 calls getter methods of String attributes and 1000 calls to an empty method without associated constraints. The next steps only applicable to the DeDiSys configurations were 1000 calls to an empty method with a satisfied constraint and 1000 calls to an empty method with violated constraints. Constraint satisfaction or violation was achieved by simply returning true or false within the Constraint.validate(...) method in order to eliminate the validation overhead for reasonable overhead comparison. This refers to the runtime slice R5 in Section 2.2. To measure the behaviour in degraded mode when consistency threats occur, we called an empty method with an associated constraint 1000 times. The occurring consistency threats were negotiated with a dynamic negotiation handler and persisted afterwards. Finally, the 1000 entity beans created in the first step were deleted. Obviously, the create and delete case operate on 1000 different objects. The "accepted threat" case is the primary issue to investigate for the degraded mode and therefore split into a good case and bad case scenario. The values for the other operations were obtained by taking the average of 1000 operations on the same object and 1000 operations on different objects, i.e., one operation per object.

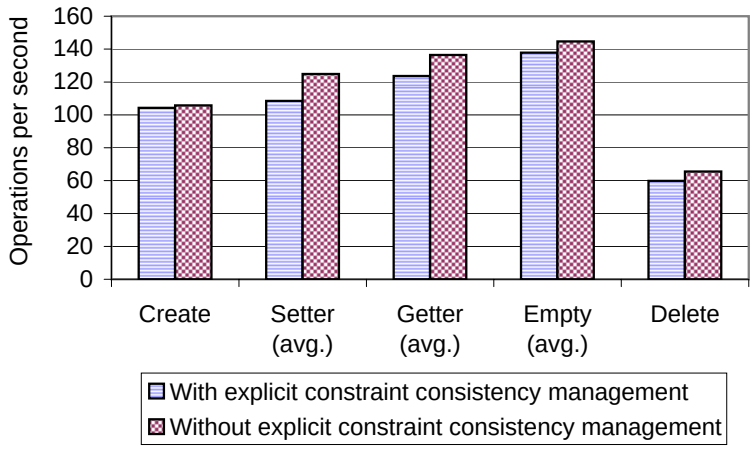


Figure 5.1: Overhead of explicit constraint consistency management

The prototype implementation indicates that explicit runtime management of constraints is a feasible approach, causing an almost negligible performance drop to about 87–99% as shown in Figure 5.1. Consequently, only in case of extremely demanding performance requirements, explicit runtime management of constraints might become too costly. Compared to our constraint validation performance evaluations in Section 2.3, these performance figures are surprising. There are two primary reasons for this significant dif-

ference in performance impact: First, the invocations in Section 2.3 are plain Java invocations while the invocations in our EJB evaluations are remote method invocations to EJB entity beans. Second, several other services such as authentication and authorization, transaction management, or entity bean locking are already performed for these invocations. Consequently, the relative overhead of constraint consistency management as an additional service does not have a major influence on invocation performance.

On the other hand, adding the implementation of the P4 replication protocol reduces performance (depending on the number of nodes and the performed operation) of the system to about 10% (create with four nodes) to 28% (delete with two nodes) of the values without constraint consistency management in Figure 5.1 for update operations and about 78% for local reads. As reads are always performed locally, the replication protocol of course increases the total number of reads that can be processed throughout the system—the benefit gained for the reduced update performance. This issue is investigated in more detail in the following paragraphs.

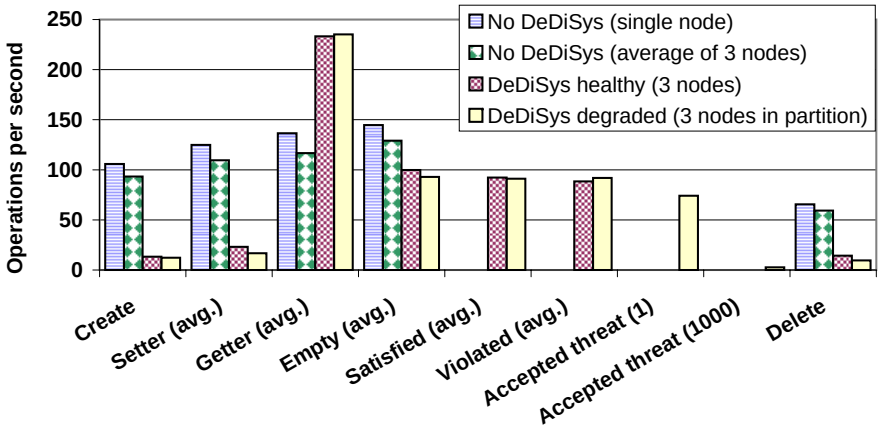


Figure 5.2: No DeDiSys vs. DeDiSys with same number of nodes in healthy and degraded mode

Figure 5.2 provides an overview of the performance of three different system configurations. "No DeDiSys" is performed on a single node (the fastest one), "No DeDiSys (average of 3 nodes)" is the average of the single-node performance of the three nodes taking part in the replicated setting, and the two DeDiSys configurations (healthy and degraded mode) use a setting with three replicated nodes. One drawback of the DeDiSys configurations is that creation, change, and deletion of entity beans is slower than the "No DeDiSys" setting. There are two main reasons for this performance loss. First, the replicated setting has to store data about the replicas of entity

beans, e.g., JNDI name and primary key to identify the corresponding entity bean and the (serialized) request used to create the entity bean (required to create backup replicas). Second, propagating the update messages from the primary copies to the backup copies requires network access in contrast to the single-node "No DeDiSys" setting. Although an efficient implementation of the P4 protocol was not in our primary focus, the provided figures give a rough estimation of the expected performance loss due to fault- and partition-tolerant replication.

Moreover, we observe that operation in degraded mode is slightly slower for write operations than operation in healthy mode. This is primarily caused by keeping a history of states per replica (requires database access). However, this comparison serves only to show the overhead of degraded mode compared to healthy mode if the number of nodes is equal. In practice, such a situation can not occur as at least a single node will not be reachable and therefore the number of nodes in degraded mode is at least one less than in healthy mode. Consequently, the degraded mode might be even faster than the healthy mode for operations triggering the replication protocol as Figure 5.3 shows. Whether this is true for a certain application further depends on the configuration of constraints and hence the number of consistency threats produced during degraded mode as the data about consistency threats have to be replicated, too. On the other hand, read performance decreases with a reduced number of nodes in a partition.

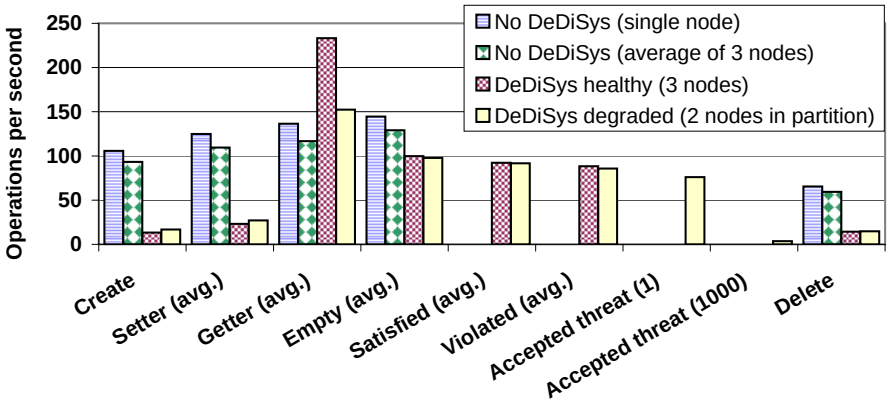


Figure 5.3: No DeDiSys vs. DeDiSys with three nodes (healthy) and two nodes (degraded)

The case where methods without associated constraints were called shows the interception overhead introduced by our middleware enhancement as well as the ADAPT replication framework [BBM+04]. This is on the one hand

the time required by the constraint consistency manager, e.g., accessing the constraint repository to search for affected constraints, and on the other hand running through the replication component that does not replicate if the called method is not a setter changing the state of an entity bean. In this case, the performance drops to about 73% of the "No DeDiSys" configuration, which we consider quite a good achievement as 22% of the 27% loss are caused by the ADAPT replication framework [BBM+04]. Consequently, the *overhead introduced by our middleware enhancement for empty operations is about 5%.*

Handling of satisfied and violated constraints only occurs in the DeDiSys configurations as this is a new middleware service added by our prototype. Although there are some minor differences between satisfied and violated constraints in certain scenarios, they show the same performance in average for the healthy as well as the degraded mode.

The "accepted threats" case for operation in degraded mode primarily shows the overhead introduced by consistency threat negotiation as well as persistence and replication of consistency threats in addition to the time required to handle satisfied constraints. In order to investigate a good case and a bad case scenario, we performed 1000 operations on a single object producing 1000 identical consistency threats on the one hand and 1000 operations producing 1000 different consistency threats on the other hand. Of course, depending on the system configuration, even more than 1000 threats would be possible. The good case scenario shows the advantage of storing identical threats only once. Consequently, only a single threat has to be stored in this case and we could serve 74 business operations per second. On the other hand, the bad case scenario requires replication and persistence of 1000 different consistency threats, which is a rather costly operation. In this case, we could only serve three business operations per second. Obviously, this case heavily depends on the specific application. However, the operation in *degraded mode shows the greatest benefit of our approach* compared to traditional systems that either block, i.e., are unavailable, or operate in an uncontrolled inconsistent way—thereby impairing dependability in one or the other way.

Although the contribution of this dissertation is not focused on an efficient implementation of the P4 replication protocol, the effects of replication on the different operations are of course an interesting aspect to investigate. Our implementation of the P4 protocol uses synchronous update propagation from the primary to all currently reachable nodes. While this slows down updates (create, setter, delete), the performance of read operations is enhanced as

reads can be performed on any node. Figure 5.4 shows that the performance of one node using DeDiSys (and hence the P4 replication protocol) drops to 71% for entity bean deletion, 43% for entity bean creation, and 57% for local writes. This shows the overhead of the replication protocol through database accesses to persist details about entity bean replicas. Adding a second DeDiSys-node further reduces update performance to 28% (delete), 15% (create), and 22% (writes) compared to the "No DeDiSys" case. This shows a little bit less than 50% performance of the single DeDiSys node case, caused by the fact that the primary first executes the update and afterwards propagates the updates synchronously to the backups. Even though the backup nodes process the update messages from the primary in parallel, adding additional nodes decreases update performance slightly further.

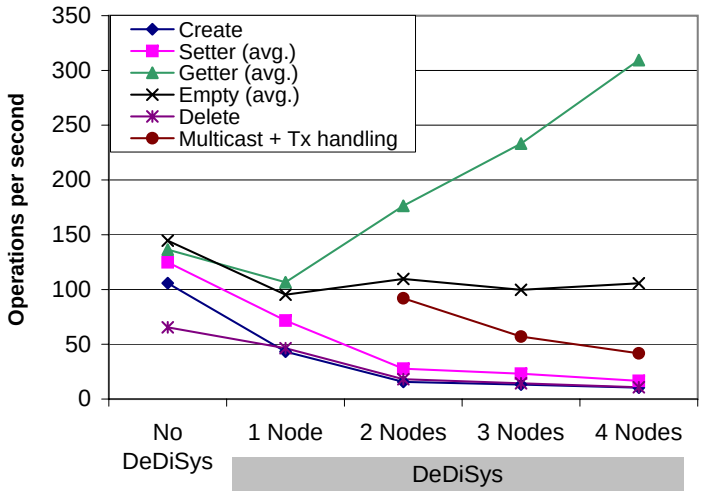


Figure 5.4: Replication effects on different operations

On the other hand, read performance is increased by roughly 50% of the single node per additional node, starting from 78% of the "No DeDiSys" case for the single node scenario and reaching 227% in the four node replicated setting. Empty operations only performed locally operate at a rather constant ratio independent of the number of nodes in the system. This is also true for the test cases with satisfied and violated constraints. However, the backup nodes show no CPU load for non-update operations and hence can serve further client requests.

In order to investigate the theoretical maximum of (update) operations per second possible due to restrictions of group communication and transaction handling, we started a transaction, sent 1000 ping messages from the primary to the backups, associated the transaction context at the backups, responded

with a pong message to the primary and finally committed the transaction. This is the "Multicast + Tx handling" case in Figure 5.4. Obviously, the round-trip time of multicasts and transaction handling become more and more influential with an increased number of nodes, limiting the possibilities for performance improvements. Finally, the empty method case shows the same performance independently of the number of nodes. The reason for this is that it does not trigger update propagation on the one hand. On the other hand, as this method does not adhere to any naming convention, we consider it a write operation—to be on the safe side—and therefore execute it only on the primary node.

5.2 Reconciliation phase

Out of the three major system modes, the healthy system and even the degraded mode are rather straight-forward to implement. The complexity of the reconciliation phase is incomparably higher. While it might be tempting to perform a rollback-approach until the system is consistent again, such behavior retrospectively reduces the availability of the system as (some to all) updates performed during degraded mode do not become effective after all. Moreover, trying to perform such a reconciliation through a generic, middleware-driven approach tends to be an complex optimization problem, mainly determined by the number of objects, number of constraints, number of partitions, and the size of the history (states/operations) of objects (Figure 5.5). Solving this issue in practice needs good heuristics that exclude a large number of combinations a priori. Such heuristics, however, can only be found with knowledge about the application requirements.

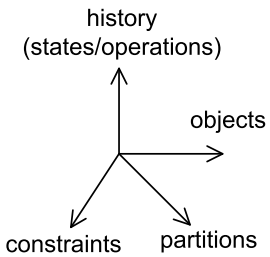


Figure 5.5: Major reconciliation dimensions

Therefore, in order to allow for more efficient compensation actions ("roll-forward" approach to reconciliation), replica and constraint consistency should not be re-established in a generic way by the middleware but rather in an

integrated, interactive way by including the application. This requires that middleware services are more open to applications, can be configured and extended with application-specific behavior, e.g., through the use of *plugin-based middleware architectures*. Consequently, middleware should provide re-usable infrastructure services and allow for flexible interaction and coupling between middleware and application to achieve the application-specific parts efficiently.

One of the major issues to consider is the amount of data gathered during degraded mode and to be processed during system reconciliation. For example, keeping a history of states/operations of the degraded period only makes sense, if it is required for reconciliation, e.g., for rollback to previous states. Similar considerations apply to whether identical consistency threats should be stored once or more than once. The more data are gathered during degraded mode, the more data needs to be processed in degraded mode—e.g., processing of already existing consistency threats and linking them to identical additional threats—as well as during reconciliation, where the reconciliation process might try a rollback to previous states. Obviously, the time taken for such automatic rollback-based reconciliation grows with the history of previous states/operations. Therefore, reconciliation should focus on reaching a consistent state through a roll-forward approach by performing compensating actions to remove inconsistencies. As the reconciliation phase is highly application-specific, this further implies that applications that do not require access to the history of the degraded mode in order to reconcile inconsistencies have a greater benefit from the balancing of integrity and availability. For example, if the flight booking application discovers during system reconciliation that 82 out of 80 available seats of an air plane are booked and two passengers have to be rebooked, the history of the degraded mode is not that important, i.e., how this situation was produced. The only requirement in this case is to rebook two passengers to another flight.

To evaluate the reconciliation phase, we performed several operations in degraded mode, resulting in 200 identical consistency threats or 1000 consistency threats if identical threats are stored more than once. A single threat initially requires at least three objects to be persistently stored in the database and two further objects per additional identical threat. After the network partition is reunified, the replication protocol starts to propagate missed updates—including consistency threats. Replica conflicts are provided to the constraint consistency manager to support constraint reconciliation. After the replica reconciliation phase finished, the CCMgr starts to re-evaluate the accepted consistency threats, which are all actually satisfied in our case to evaluate the best case. The worst case cannot be reasonably

evaluated as it might involve user interaction to clean up inconsistencies—possibly being performed only days after the network partition.

Figure 5.6 shows the time required for system reconciliation. As expected, the reconciliation phase becomes slower with an increased number of updates performed and threats occurred during degraded mode. While the number of updates was the same in both cases, the threats were stored according to the "identical threats only once" policy one time and another time with the "store all occurred threats" policy. Obviously, replica reconciliation scales worse with an increased number of identical threats than constraint reconciliation as it cannot benefit from identifying identical threats. On the other hand, re-evaluation of identical threats has to be performed only once (the validation result for identical threats is the same) and if the constraint is satisfied, all threats can be deleted. Constraint reconciliation can only benefit from multiple threats if a constraint violation is detected, which has to be subsequently resolved.

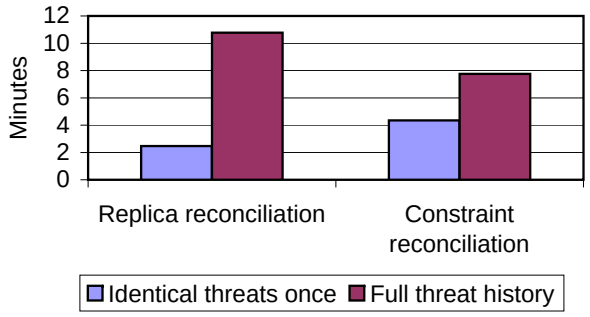


Figure 5.6: Time required for propagation of missed updates and re-evaluation of consistency threats

The primary conclusion of this evaluation is that it is not feasible to block the system during the reconciliation phase. Consequently, we allow for parallel reconciliation and business activities by following the approach described in Section 3.3.

Our approach to re-establish replica consistency before constraint consistency is primarily based on the motivation to re-establish replica consistency and hence denote a single (temporary) primary per object without too many delays for the reunified partition. Alternatively, it could be tried to re-establish replica and constraint consistency in a single step. However, this increases the complexity of the reconciliation phase and prolongs the time required to reach a replica consistent state. Subsequently, this may delay parallel business requests. Moreover, replica consistency can be established immediately

in several cases while re-establishment of constraint consistency might have to be postponed. Consequently, the separation of replica and constraint reconciliation is a reasonable approach as it does not unnecessarily delay re-establishment of replica consistency. On the other hand, details about replica (write-write) conflicts and the history of replicas in different partitions can support the automatic and manual re-establishment of constraint consistency. Therefore, details about replica reconciliation, e.g., conflicting replicas, should be available to the constraint reconciliation mechanism as well.

Simulation studies [Se05] have shown that our approach combined with the primary-per-partition protocol (P4) [BBG+06] can be used to increase availability in the presence of network partitions. However, the effort required for reconciliation due to continuing operation in different partitions during degraded mode is most probably only worth its costs in the case of longer lasting partitions for systems where the read-to-write ratio is high. Based on our experience with the prototype implementation, using a generic history-based rollback approach for consistency reconciliation tends to become a complex and processing intensive task. For example, trying several to all possible combinations of historical object states from different partitions is costly—even more so if the system recovers/reconciles from a longer lasting partition. Therefore, the reconciliation phase should focus on re-establishment of consistency through application-specific compensating actions instead of generic rollback.

5.3 Lessons from implementation and tools

Command pattern for invocations. One of the most enabling factors for middleware integration of our balancing of integrity and availability was the usage of the command pattern [GHJV95] for invocations by the JBoss AS and the fact that new interceptors can be added easily via specification in a configuration file. JBoss itself relies on this concept to provide middleware services, e.g., to associate security contexts or transactions with an invocation. Generally, any desired additional payload can be added to such an invocation that is explicitly represented by an object.

Aspect-oriented programming. An additional enabling factor for thorough support of constraint consistency management through a constraint consistency manager and a constraint repository was the aspect-oriented

programming paradigm and JBoss AOP as available toolkit. One limitation we discovered with AOP is that it performs out-of-context invocation interception making it necessary to explicitly maintain context information in a way accessible to AOP interceptors. In our case, for example, the constraint repository is application specific (constraint names have to be unique within an application and not within the whole application server) and uses information provided in the context information of an enterprise bean to differentiate between applications. This context information is available for JBoss AS invocation interceptors as it is associated with the invocation object. To make this context information available to AOP interceptors as well, we had to maintain an explicit mapping between bean instances and the context information. This, however, can also be achieved by using AOP.

Infinite loops between application and middleware. Providing middleware services requires to take special care of potentially infinite loops caused by a control flow between middleware components and application specific artefacts. For example, the CCMgr is integrated as a middleware service and notified of invocations through invocation interceptors. One of CCMgr's tasks is to trigger validation of constraints defined upon EJB entity beans. The validation code is again provided by the application, accessing EJB entity beans. Hence, we had to ensure that we do not trigger constraint validation again while we are already in the process of constraint validation. We consider this a reasonable approach as constraints are predicates that must not change the application's state.

Soft constraint limitations. The concept of soft constraints can only be applied with certain limitations. The thread committing the transaction holds a lock on the transaction. The transactional resources—one of which is the CCMgr—execute their activities including prepare() within another thread not associated with the transaction. Consequently, it is not possible to associate the thread executing CCMgr.prepare() with the committing transaction. Trying this leads to a deadlock as the second thread cannot acquire the lock on the transaction. This has the effect that the validation of soft constraints takes place within no or another transaction. Consequently, the second transaction for constraint validation must be allowed to access objects on which the first transaction of the business request holds a lock in order to prevent a deadlock.

Standards and interfaces. EJB uses entity beans to encapsulate the application data which fits our data-centric, object-oriented approach very well. The data integrity constraints are defined upon these entity beans and are implemented as explicit constraint classes. The constraints of an application are specified in a configuration file, which is read when the application is deployed into the application server. The information provided is used to register the constraints appropriately with the constraint repository. Consequently, our middleware support integrates quite well with the general concepts and processes of EJB, i.e., application data represented in specific objects, usage of metadata information, application deployment procedures, etc. Moreover and specifically due to the smooth integration of our middleware enhancement, the EJB specification was quite helpful during development as it provided the contract between the application and the middleware and enabled to start the development of middleware enhancements as well as the applications implemented on top of it in parallel.

5.4 Middleware/application interactions

This section discusses and summarizes the middleware/application interactions within our prototype implementations as well as specific design alternatives we investigated during the prototype studies.

Besides the explicit (callback) interaction through predefined interfaces we use metadata and invocation interception as programming abstractions for coordination of and implicit interaction between application and middleware. The metadata about constraints is provided by the application developer and includes information such as when to check a specific constraint or whether the constraint can be relaxed (potentially be violated) during degraded mode or reconciliation in order to enhance availability.

Invocation interception is a well-known mechanism to provide middleware services to an application and available in several middleware technologies, such as EJB, CORBA, or .NET. However, middleware traditionally is a layer between the presentation and the resource layers and hosts the application. Consequently, the middleware or the middleware services are primarily triggered in case of remote invocations. While EJB already triggers middleware services for local invocations, this is only the case for invocations made upon an interface but not the actual implementation of a bean itself. Therefore, method calls of a single instance to itself are plain Java invocations. Unfortunately, constraints can be triggered by an arbitrary method, not only

by methods invoked via remote method invocation (RMI) or via the bean interface. Hence, integration of constraint consistency management as a middleware service requires the possibility to intercept each and every method of an application. We use Aspect-Oriented Programming (AOP) to satisfy this requirement—thereby complementing the traditional mechanisms for invocation interception and introducing another kind of application interception by the middleware.

One more kind of interaction between middleware and application we use is the concept of exceptions. For example, the CCMgr throws a ConstraintViolation exception if it detects that constraints of the application are violated by a business operation in healthy mode. In degraded mode, it throws ConsistencyThreat exceptions for not accepted consistency threats. While the detection of inappropriate situations is performed by the middleware, the treatment of the consequences has to be performed by the application.

Furthermore, our middleware enhancement uses persistence to reliably manage consistency threats, i.e., it stores them when they occur to be able to re-evaluate them later during system reconciliation. Thereby, we decouple constraint validation from the business transactions. Application data associated with a consistency threat is stored along with the threat, effectively relieving the application from any management of threats and data associated with them.

While there are no reasonable alternatives for invocation interception (including AOP) or persistence, we evaluated alternatives for the explicit callbacks between middleware and application. The negotiation of consistency threats is a synchronous/blocking task, i.e., an operation/transaction cannot continue and especially not commit successfully as long as there is no decision on whether to accept or not accept a specific threat. However, negotiation can be performed immediately when a threat occurs or be deferred until the end of a transaction. In any case, we should be able to continue and commit the transaction if all threats are accepted. Due to this behaviour, a callback is most appropriate to achieve the negotiation task.

An alternative to the callback for negotiation of consistency threats seems to be to throw an exception to indicate the consistency threat. The application would have to investigate the exception details and retry the operation by signalling the middleware to accept the threat(s). This has the drawback that the threat that occurs the second time, e.g., 85 out of 80 tickets booked, might be different than the first time, e.g., 75 out of 80 tickets booked, but might be accepted due to the decision of the application. Unfortunately, continuing the operation at the point the exception occurred is generally not

possible, as the invocation stack is already lost. This is especially true for nested invocations and illustrated in Figure 5.7 where A would have to call B, B call C, and C call D again after the exception indicating a consistency threat occurred at some point in the method of D. Other mechanisms such as asynchronous calls or message passing are not useful for threat negotiation either due to the blocking behaviour of negotiation. The only advantage of asynchronous behaviour would be for longer-lasting transactions where deferred negotiation is possible. In this case, negotiation of threats could take place in parallel while the transaction continues with the assumption that all threats will be accepted. Of course, the transaction has to block before commit until the decisions for all occurred threats are available.

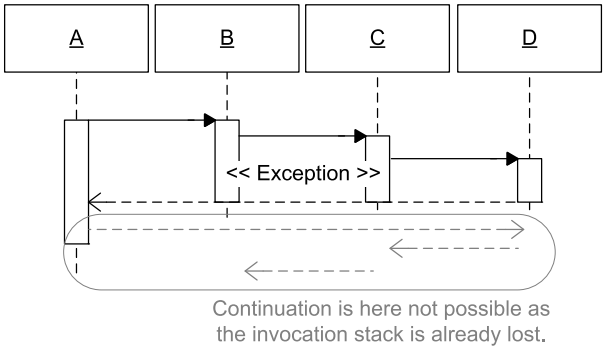


Figure 5.7: Exceptions break the flow of control

Due to the use of a callback for constraint reconciliation, we allow for immediate cleanup of constraint violations caused by accepted consistency threats as soon as the violation is detected. However, we allow constraint reconciliation via asynchronous behaviour as well. Based on our experience with the prototype implementations, asynchronous reconciliation is the usual case as constraint reconciliation might often require user intervention. Moreover, it is especially useful for Web applications where a callback to a Web browser is simply not possible. While it is possible to circumvent this limitation by letting the browser poll for constraint violations and perform the same technique as for the negotiation handler, this is a rather cumbersome and resource-wasting behaviour. Another alternative would be to run a Java Applet within the Web browser. Consequently, intermediate callbacks could be made to the applet supporting this behaviour. However, this—as any kind of "real callback" from the server to the client—has the drawback that intermediate firewalls might block the call.

Table 5.1 summarizes the middleware/application interaction mechanisms within our system. Callbacks, exceptions, asynchronous behaviour, and in-

teraction via persistence are interactions between middleware and application that an application developer has to explicitly address or use. Invocation interception is used only implicitly, i.e., transparently to the application developer, to achieve middleware tasks. Finally, the usage of metadata allows an application-specific configuration of the middleware.

Table 5.1: Middleware/application interactions

Mechanism	**Purpose**	**Remarks**
Invocation interception	Enables the middleware to provide middleware	AOP allows to intercept calls that otherwise would not trigger middleware services.
Callback	Immediate response required.	Support for asynchronous behaviour through callbacks that do not immediately have to succeed.
Exception	Indication that "something" failed, e.g., a constraint is violated.	Exceptions break the flow of control, thereby requiring an abort/retry behaviour.
Metadata	Application-specific configuration of the middleware.	of constraints (classes, affected methods, etc.) and callbacks.
Persistence	The middleware manages consistency threats while the application may access them.	This provides interaction based on shared memory semantics.
Asynchronous behaviour, e.g. message passing	For operations/tasks lasting for longer time periods such as constraint reconciliation.	Only indirectly supported via reconciliation callback that can opt for this behaviour.

5.5 Improvements based on evaluations

Our evaluations showed that i) write operations are rather slow due to synchronous replication and ii) the reconciliation phase takes quite some time already for the best case scenario without any conflicts or constraint violations. Consequently, it is beneficial to i) reduce the number of required write operations and ii) reduce the amount of data to be processed during the reconciliation phase along with a reduction of the probability of constraint violations caused by accepted consistency threats. Some improvements made based on our evaluations are provided within the following sections.

5.5.1 Reduced history

While, of course, the middleware has no influence on the number of write operations performed by the application, it can perform optimizations with respect to the data gathered during degraded mode. Our middleware enhancement allows the association of specific application data along with a consistency threat that will be persisted in order to be available during the reconciliation phase. However, some applications do not require the history of consistency threats or the threat-specific application data in order to clean up inconsistencies. Consequently, it is sufficient to store identical consistency threats only once.

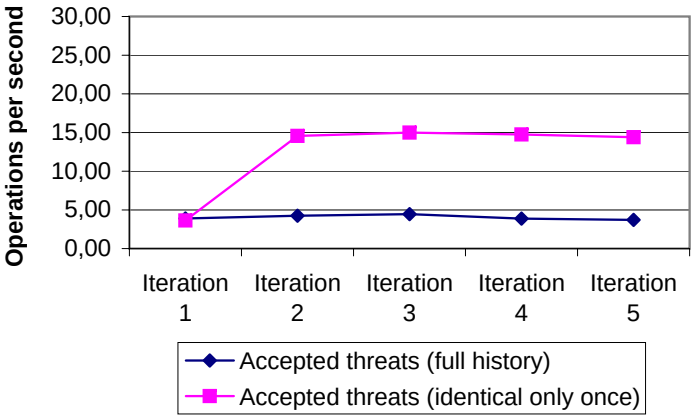


Figure 5.8: Improvements through reduced consistency threat history

Figure 5.8 shows the benefit gained from storing identical threats only once—an increase from about 4 operations per second to about 15 operations per second. For this evaluation, 200 operations were performed on 200 different

operations producing 200 different consistency threats within a single transaction. This scenario was repeated four times leading to five iterations in total. If the middleware was configured to store the full consistency threat history, identical threats were stored in all five iterations. If the middleware was configured to store identical threats only once, the consistency threats were stored in the first iteration. The following iterations only preform read operations on the database in order to detect that an identical consistency threat is already persisted.

5.5.2 Partition-sensitive constraints

The motivating flight booking example in Section 1.3 shows that although a constraint is satisfied in degraded mode while node or link failures are present, it might be violated afterwards when the system recovers from a previous failure. While rebooking five passengers to another flight solves the inconsistency, it would be desirable not to introduce such an inconsistency at all. For some applications, where the data can be partitioned like the tickets in the flight booking example, a significant improvement is possible. Our middleware makes use of group membership (GMS) and group communication (GC) primitives in order to implement the replication service. Similar to Gifford's solution of weighted replica copies [Gif79], we allow the association of weights with server nodes. The GMS component can thereafter calculate the weight of the current partition in relation to the whole system. This value is provided by the middleware to the application in order to take the current partition weight into account for constraint validation, effectively leading to *partition-sensitive integrity constraints*.

Based on these prerequisites, data can be partitioned during runtime. The ticket-constraint, for example, saves the number of tickets sold in healthy mode. In degraded mode, it partitions the number of still available tickets t (number of seats minus number of tickets sold in healthy mode) according to the partition weight, effectively leading to t_x available tickets for a partition x ($t = \sum_{x=1}^{n} t_x$). The constraint will only be satisfied within partition x, if the number of tickets sold during degradation is below or equal t_x.

This improves behaviour in degraded mode and introduces almost no inconsistencies based on the fact that tickets are mainly sold and rarely cancelled. In the best case, no inconsistencies are introduced at all, although write access in different partitions is possible. Inconsistencies could be introduced, if the same ticket is cancelled in two partitions and a different ticket is sold per partition afterwards. For example, assume that we already sold 80 tickets

for a flight with 80 seats. Then, we cancel the same ticket in two partitions, effectively having 79 sold tickets in both partitions. If we now sell one ticket per partition, we will finally end up with 81 sold tickets in the whole system—again having an overbooking situation.

Obviously, partition-sensitive constraints allow for more precise calculations of constraint validation results and consequently improve consistency during degraded mode. Furthermore, this improves the reconciliation phase as less inconsistencies have to be reconciled. On the other hand, the data partitioning might reduce availability during degraded mode. For example, some partitions might run out of tickets according to their partition weight while other partitions might still have unsold tickets.

5.5.3 Asynchronous constraints

The validation results of some constraints might not even be important to an application, if the result is unreliable because validation was performed on possibly stale objects. Consequently, this provides room for optimizations:

- The overhead of constraint validation can be removed by not validating the constraint in degraded mode at all. Consistency threat negotiation, however, could still take place in order to decide on whether to continue the current transaction.
- The overhead of consistency threat negotiation can be removed by simply accepting all consistency threats of a certain constraint.

In order to evaluate these optimizations, we introduced the notion of asynchronous constraints. Asynchronous constraints behave like soft constraints in the healthy system. In degraded mode, asynchronous constraints are no longer validated, but stored to be validated in the reconciliation phase, i.e., we no longer check these constraints or perform negotiation of consistency threats. Our evaluations showed that this increases degraded mode performance up to two times the number of operations per second possible for soft constraints where identical threats are stored only once.

Chapter 6

Related Work

This chapter groups related work into three major sections with respect to the concepts and one minor section with respect to the implementation. First, we start with discussion of related work on (in)consistency management in Section 6.1 before we continue with related work on explicit integrity constraints in Section 6.2. Related work on adaptive middleware and systems forms our third major block in Section 6.3. Finally, we conclude this chapter with Section 6.4 on related work for callbacks in Web applications.

6.1 (In)consistency management

The balancing of integrity/consistency and availability has already been investigated with respect to isolation [BBG+95, HW90] and replica consistency [DGMS85, FN02, PL91, YV02] and different strategies to optimistic replication are already well-known [SS05]. The trade-off between constraint consistency and availability did not yet receive much attention. However, several works address inconsistency management in different settings. One of the primary—or probably even the primary—and often cited work on "tolerating inconsistency" was performed by Balzer [Bal91]. He allows constraint violations temporarily to be able to perform certain business operations in separate steps, where an initial step might introduce inconsistencies and a final step removes them. He uses pollution markers corresponding to integrity constraints. If an integrity constraint is not satisfied, the corresponding *pollution marker* is set. The pollution marker is removed at the time the integrity constraint is satisfied again. The system tolerates inconsistent data in the way that report generators use the pollution markers to subsequently

mark reports that are affected by inconsistent data. Although the storage of consistency threats roughly corresponds to the pollution markers, Balzer accepts constraint violations in a healthy system and is not concerned about degraded mode due to node or link failures. On the other hand, we are aiming at fully consistent data during healthy system periods and only trade consistency threats (not violations) during degraded periods to increase availability. However, combining these two approaches would most likely provide further benefits.

Inspired by the work of Balzer, Cugola et al. investigate the issue of inconsistencies within the execution of (business) processes in [CNGM95]. Within their work, they distinguish between preconditions for state transitions and invariants defined upon the process model. If certain preconditions do not hold, e.g., due to user interaction, this is allowed as long as the invariants are still satisfied. Such a situation is called a *tolerable deviation* from the process definition. However, the data produced by this state transition might be polluted. Consequently, their system provides appropriate support for storing historical steps in the process execution in order to enable *pollution analysis*. Pollution analysis has to be performed if the process reaches an unacceptable state, identified by violated process invariants. The recovery from unacceptable process states is is left to the user.

Guinea continues the work of Cugola et al. to a certain extent in his PhD thesis [Gui07] in the area of service oriented architectures. In this work, he focuses on business processes defined in BPEL4WS, the Business Process Execution Language for Web Services—also BPEL for short [IBM+03]. The monitoring and recovery of business processes is carried out by the so called supervision framework. This framework includes data collectors to obtain the monitoring data, data analyzers to verify whether specific properties (conditions/constraints) hold, and the recovery manager responsible to activate appropriate/defined recovery strategies if process anomalies are detected. Recovery strategies include simple actions such as to ignore anomalies or halt the process execution, but also allow to rebind a service to another one with the same interface or perform a call to an external web service.

(In)consistency management model. From the previous works by Balzer, Cugola, and Guinea as well as our work, we can observe that all of them use explicit rules/constraints in order to monitor system state and/or behaviour. Moreover, violations or potential violations have to be treated explicitly in order to allow to make progress in the face of potential or even certain inconsistencies. Finally, inconsistencies are resolved through incon-

sistency analysis and repair actions. The major difference of these works lies in the application domain, in which inconsistency management is applied. Consequently, it is possible to abstract a general model for (in)consistency management. Tarr and Clarke present a such a model for *consistency management in complex applications* in [TC98]. Although this model is primarily motivated from consistency between source-code, abstract syntax trees, and control flow graphs in software engineering environments, it is certainly more generally applicable and maps to our research quite well. In fact, this work is quite a good starting point for anyone designing an (in)consistency management system. We reflect on their model in Appendix A by shortly addressing their requirements to consistency management systems along with the corresponding mechanisms we used in our constraint consistency management framework.

Inconsistency in requirements engineering. Finkelstein, Nuseibeh, Easterbrook, Kramer and others address the issue of inconsistencies in software requirements engineering based on a technique called viewpoints [FGH+94, EN96, NER01]. Their primary motivation comes from the fact that the development of large and complex software systems involves many people, viewing the system from different perspectives—*viewpoints*. While the viewpoints are locally consistent, several inconsistencies between the viewpoints might occur. In order to manage these inconsistencies, viewpoints are bound together via inter-viewpoint relations. These relations specify dependencies and mappings between system components. The enforcement of strict consistency between the different viewpoints in such environments is counter-productive: "Enforcement of consistency means the change has to be delayed until the problem is sorted out, during which the desired change cannot be represented. It is often desirable to tolerate and even encourage inconsistency [GH91], to maximise design freedom, to prevent premature commitment to design decisions and to ensure all views are taken into account" [EN96]. Their works are performed for software requirements engineering and hence consider the software development phase while we in our work address inconsistencies in the data of a running software system. The mechanisms used (explicit rules/constraints), however, are again similar.

Consistency checking of distributed documents. Ellmer, Nentwich, Emmerich, Finkelstein and others investigate the issue of inconsistency in distributed XML (eXtensible Markup Language) documents [EEF99, NCEF02, NEF03, NEFE03], e.g., UML design documents or EJB deployment descrip-

tors. They use XPath [CD99] expressions to select values of the XML documents in order to use them within their rule language to define consistency conditions. The support for consistency checking is implemented within a tool called xlinkit. Consistency checks produce a report with details on the inconsistencies within the documents. These repair reports can be used by the developer to fix the inconsistencies. However, some automatic repair actions can also be performed, e.g., by defining that one artefact has precedence over the others. Differently to us and some of the other works, there seems to be no support for explicit or preventive inconsistency management, i.e., for the decision of whether inconsistencies are allowed to be introduced by certain changes to the documents.

Several of the previous works are also addressed by Spanoudakis and Zisman in their survey of inconsistency management in software engineering along with an identification of open research issues in [SZ01].

6.2 Explicit integrity constraints

Several related work with respect to explicit integrity constraints focuses on constraint validation in the sense of design by contract [Mey92] as introduced by Meyer for the Eiffel programming language. One focus within this principle is to decide whether the producer or user of a certain piece of code violated the contract. This process is also called the problem of "assigning the blame" for incorrect code.

Lackner et al. [LKP02] discuss (pre-)compiler-based approaches supporting design by contract in Java. Starting with an overview of Jass [BFMW01], iContract [Kra98], jContractor [KHB99], and Handshake [DH98], they finally describe their own support for design by contract in Java through extension of the Java language with new keywords. Furthermore, they provide how this approach was integrated into the Kopi Java compiler and provide performance studies of their approach and a comparison with some of the other approaches. In their studies, the authors experience performance impacts for contract checking code between 2.22 and 1389.11 times the runtime of non-contract checking code. Obviously, these results also have such a wide range of performance impacts as shown by our studies of the different constraint validation approaches.

Plösch [Plö02] provides further details and an evaluation with respect to the degree of assertion support of some tools listed in Table 2.1 on page 21. However, these works focus on integrating constraint/contract checks into

Java byte code while our evaluation considers wrapper-based source-code instrumentation, byte-code instrumentation as well as interceptor-based approaches to make constraints first class runtime entities.

While design by contract might be interesting with respect to distributed software development and contracts between producers and users of code, other works focus on ensuring system integrity in the sense of considering a constraint violation an error that should be treated by corrective measures rather than to view the violation as system failure [ALRL04]. Verheecke and Van Der Straeten [VS02] perform a transformation from UML class diagrams enhanced with OCL constraints into Java objects (business objects) and Java constraint checking classes. The generated code of the business objects already contains the hard-wired trigger points for constraint validations and hence does not make use of a constraint repository. Therefore, although the constraints are explicitly encoded in Java classes, they are not explicitly managed during runtime. We on the other hand require that constraints are explicitly available and processable during runtime for adaptivity with respect to node and link failures. However, our representation of data integrity constraints as explicit constraint classes was inspired by their work.

Wang and Mathur [WM05] apply an interceptor-based approach for constraint violation detection in the area of autonomic systems. Based on declarative specifications of the monitoring constraints, invocation interceptors implementing the constraint checking code are generated. Although these constraints are slightly different from the data-centric constraints in our work (their constraints restrict method parameters and the order in which method invocations may take place), the performance evaluation of using the JBoss AOP framework can still be applied to our work as well. The authors identify a performance overhead of 18–20 microseconds for using AOP intercepted method invocations compared to simple method invocations. Consequently, the authors argue that interceptor based constraint validation is feasible because of this little AOP overhead, especially in component-based software, such as EJB, with medium size entities.

Oakasha et al. [OCS01] use an explicit constraints approach within the area of object-oriented databases that also regards constraints as first-class citizens registered within the so called constraint-catalog—corresponding to our constraint repository. However, the authors address the problem of consistency management in object-oriented databases and only make use of the flexibility of explicit constraints in the way that they may be turned on and off. This, however, is an interesting aspect when importing large amounts of data into a database, for example.

6.3 Adaptive middleware and systems

The EU funded project ADAPT (Middleware Technologies for Adaptive and Composable Distributed Components) provides a replication framework to allow rapid prototyping of replication protocols in J2EE (Java 2 Enterprise Edition) environments [BBM+04]. This framework is based upon the JBoss AS. The primary mechanism used is invocation interception at the client side as well as at the server side. The replication protocol building upon this framework is notified about different events, such as creation of, calls to, and deletion of enterprise beans. Consequently, this framework proved quite useful for our prototype implementation of the P4 replication protocol.

Geihs et al. [GKR+06] address adaptation of component-based distributed applications based on the model-driven paradigm. The adaptability of an application is specified within the model with the goal to provide the best possible service to the user according to context and user preferences. Based on this model, application and middleware artefacts can be generated. While our current systems do not apply model-driven elements, we are investigating this approach to support the application developer with code generation for integrity constraints similar to Verheecke et al. [VS02] and generation of the corresponding metadata.

The usage of metadata and reflection is also proposed by Capra et al. [CEM01] as a concept to address mobility. In their approach, metadata is specified by the application, but is managed by the middleware. Consequently, the middleware adapts to changes in the context, e.g., bandwidth, battery power, network connection/connectivity, etc. according to the metadata—also called an application profile. We support this argument to use metadata and also see metadata as a promising approach to master the increasing complexity in todays systems.

McKinley et al. address the issue of "composing adaptive software" in [MSKC04]. First of all, they differentiate between parameter adaptation and compositional adaptation. *Parameter adaptation* is performed through changes in program variables in order to adapt program behaviour. *Compositional adaptation* allows to change algorithms or structural components and is therefore more powerful than parameter adaptation. They argue that "middleware provides a natural place to locate adaptive behaviour" and see—and thereby they support our argumentation—the possibility of interception and redirection of interactions between program entities at the core of compositional adaptation. Moreover, McKinley et al. identify three key technologies

for reconfigurable software design: (i) separation of concerns and AOP as an enabling factor, e.g., for crosscutting concerns, (ii) computational reflection, which also includes metadata, and (iii) component-based design, e.g., to allow dynamic composition during runtime. Moreover, they provide a summary of software recomposition techniques and address questions such as how, when, and where to perform composition. Being in line with the arguments of McKinley et al., we introduced adaptive behaviour into the middleware, allow the application to perform parameter adaptation for basic configuration tasks, and use compositional adaptation for the complex tasks such as consistency threat negotiation or reconciliation. The resulting system adaptivity with respect to node an link failures, however, does not fall into the previous adaptation categories, especially as it focuses on adaptive system behaviour and not on composition.

Schmerl and Garlan investigate how existing design tools can be reused in order to support architectural adaptation [SG02]. Based on the component and connector paradigm, they use integrity constraints in order to specify correctness criteria for architectures developed in their design tool. These constraints are monitored at runtime. Moreover, runtime monitoring results can be observed within the design tool. If architectural adaptation at runtime leads to violated constraints, appropriate repair tactics are carried out in order to get back to a consistent architecture. While this work is similar to our work and others with respect to inconsistency management and explicit integrity constraints, the major emphasis is put on architectural adaptation during runtime. This, however, is different from our approach to adapt the system with respect to node and link failures in that we do not perform an adaptation of the system architecture.

Motivated by the fact that runtime monitoring is costly, Dwyer et al. use an adaptive approach to runtime monitoring [DKE07]. Within their system, they check program correctness with respect to finite state automata. To eliminate the costs of runtime monitoring, they only instrument the calls that lead to state transitions. If a state transition occurred, no longer needed monitoring code is removed to speed up performance while other calls might be instrumented according to the current state. Obviously, the primary idea is that the costs for continuous reinstrumentation are less than the gain in performance by running parts of the system as uninstrumented code. Their evaluations show a performance impact through adaptive analysis between 23–33% of additional runtime required. Based on our experience and compared to our results, this is quite a good achievement, especially as this includes interception overhead and monitoring code. On the other hand, these results heavily depend on the application and the size and complexity

of the monitored properties. In our case adaptive instrumentation would only make sense by replacing the generic interceptors with direct calls to the constraints, effectively excluding the search overhead within the repository. This has to be paired with a support for adding and removing constraints to/from the constraint repository, where such an operation would trigger re-instrumentation of affected methods in addition to the constraint validation of added constraints. While the effects of selective instrumentation of constrained objects, i.e., of the affected methods, would be an interesting aspect to investigate, the potential for performance optimizations is rather minimal as the overhead of explicit constraint consistency management is currently in the range of 1–13%. However, for plain Java applications an investigation of this approach could be worth the effort as in this case the potential for optimization is much higher as the evaluations in Section 2.3 show.

6.4 Callbacks in Web applications

XMLBlaster (`http://www.xmlblaster.org/`) is a message-oriented middleware tool that allows callbacks from a Web server to the browser via persistent HTTP connections. This connection is opened by the browser (specified via the Connection: keep-alive parameter) or a Java applet if the applet variant is used. The server sends messages—which might actually be callbacks—over this connection to the browser in form of data chunks. Client-side processing of these messages is performed either via JavaScript or within the Java applet depending on the technology used. This enables callbacks to the browser through a dedicated HTTP connection while we on the other hand use the HTTP connection of the original request and return a usual HTML (HyperText Markup Language) page. Moreover, we explicitly have to close the connection through Connection: close in order not to produce a deadlock caused by a single thread serving a single HTTP connection. However, these two approaches can be combined as well.

Chapter 7

Conclusion and Future Work

This dissertation presented a middleware approach to support adaptive dependability by balancing integrity and availability. We showed how explicit runtime management of constraints as a middleware service can support the application to provide the desired balancing with respect to an application's requirements and environment conditions. This concept allows detection and negotiation of consistency threats as a means to bound the potentially introduced inconsistency during degraded mode. Our approach allows decoupling constraint validation of threatened constraints from the business activities by postponing reliable validation to the reconciliation phase. This, however, comes at the price of increased complexity: Consistency threats that result in constraint violations during reconciliation have to be cleaned up at a point in time where the causing business activity is usually already finished.

Generic rollback-based reconciliation solutions require a lot of data to be gathered during degraded mode, e.g., the history of applied operations/states, which requires even more processing during reconciliation phase. Moreover, generic rollback (after the corresponding business activity is already finished) often does not lead to satisfactory solutions from the perspectives of the application developer and the end user. Therefore, our approach should primarily be applied to systems that are able to reconcile the system state without requiring a full history of the degraded mode and allow for flexible application and potentially user interaction to clean up the system.

In order to reduce consistency threats during degraded mode and reduce complexity and processing effort in the reconciliation phase, system partitioning should already be explicitly addressed in the system design phase by taking the specific application requirements into account. Within this task, data distribution and partitioning with the least amount of possible inter-node

dependencies is of major importance. Partition-sensitive constraints are one way towards this direction as they support a combination of optimistic replication and data partitioning in order to reduce the amount of inconsistency introduced into a distributed system while network partitions are present. This reduces the amount of inconsistencies to be cleaned up while the system reconciles the updates performed in different network partitions during degraded mode. Consequently, partition-sensitive constraints can increase the performance of the reconciliation phase. The price to be paid on the other hand is potentially reduced availability during the degraded mode.

While the balancing of the two dependability attributes availability and integrity is a rather complex task, it can still be achieved by building upon callbacks as the primary explicit middleware/application interaction mechanism. However, to fully achieve the desired behaviour, the callbacks are supported with exception handling, asynchronous behaviour, metadata, AOP, and persistence. Although callbacks are a well-known principle, they are sometimes hard if not even impossible to achieve in systems where they are not foreseen to be used, e.g., Web-based systems when a callback to the browser is required. This dissertation contributes with a solution to this callback issue as well as a discussion of several middleware/interaction mechanisms used to achieve the desired balancing of the two dependability attributes integrity and availability.

According to our prototype implementation, performance impairment due to explicit constraint consistency management is not an issue while the performance loss through synchronous replication is acceptable if (i) the read-to-write ratio is high, (ii) the number of replicated nodes within the system is small, and/or (iii) write performance is not a limiting factor. However, our approach increases availability at the expense of increased aggregate complexity during system reconciliation, which is handled best, if no access to the history of the degraded mode is required.

7.1 Future work

Within this section we summarize future work that is one step further from the current status of our work while we provide interesting research challenges requiring a lot more effort to achieve results in the next section.

The specification of affected methods of a constraint, is a rather tedious task. To relieve the application developer from this work as well as from the implementation of the constraints themselves, the MDA approach used

by Verheecke and Van Der Straeten [VS02] could be integrated with our constraint checking framework. This would support the overall software engineering process by generation of entity beans, constraints and metadata based on UML models annotated with OCL constraints. Alternatively, our approach could be combined with the "design by contract" principle by generation of explicit constraints and specification of affected methods based on the contracts instead of compile-time merging of business functionality with contract checking functionality. For both of the previous approaches we expect some necessary design extension (or workarounds) to UML and OCL or contract specification as constraints are not first class citizens as in our work. Especially, the specification of constraint metadata, e.g., tradeable vs. non-tradeable constraints might be problematic. However, a third option for improvement is to allow declarative specification of constraints (probably even during runtime), e.g., by using OCL, perform object-graph path analysis of the declarative constraints, and use the Java reflection mechanisms to access the appropriate properties of entity beans. This, however, would most likely result in significantly decreased constraint validation performance.

If constraints are stated in a declarative language, e.g., by using OCL for UML class diagrams, tool support for constraint generation or interpretation could be enriched with an automatic analysis of the consistency of the constraints themselves. For example, inconsistencies could be the result of defects in the specification, e.g., one constraint requires $A.x < 10$ and another constraint requires $A.x > 20$. However, the analysis does not only have to take into account the constraints defined for a single class, but also has to consider behavioural subtyping [Ame91, LW94, DL96, FF01] in order to address object inheritance and polymorphism.

Having data integrity constraints explicit at hand provides new possibilities to the management of data integrity in general. With respect to the coupling/decoupling of constraint validation, an "integrity robot" could continuously validate invariant constraints—completely decoupled from business activities. This, of course, is only applicable to systems that can live with a certain degree of inconsistency. While this is probably not of interest in small-scale tightly-coupled systems, it might well be an option for large-scale systems, e.g., "systems of systems". Another possibility gained through explicit runtime integrity constraints is runtime reconfigurability of constraints, e.g., to allow for mission changes (day/night). Furthermore, each constraint enforced in a healthy system could be paired with an alternate constraint to be validated during degraded system periods. For mobile databases, constraint validation could be performed during check-out and check-in to ensure that data checked out and checked in is consistent. Moreover, extensions to

the framework would be possible in order to integrate the notion of time, i.e., to validate constraints at certain points in time. Finally, future work could also consider the issue of checking the correctness of sequences of operations, similar to [WM05].

7.2 Future research challenges

Our approach is currently only applicable to tightly-coupled object oriented systems. Whether and how this approach can be applied to other system scenarios could be investigated in future work. Two major challenges we see with this respect are an increase in scale for the nodes maintaining application data and a transition of the balancing concept for integrity and availability into dynamic and mobile environments. Based on our experience, this would require different mechanisms as some of the assumptions for our current systems, e.g., server nodes are statically configured and maintained, will no longer be true.

Having integrity constraints explicitly at hand throughout the software lifecycle would allow for new approaches to software development and requirements engineering by combining the work on inconsistency management in requirements engineering with inconsistency management during system runtime. Moreover, integrity constraint maintenance effort could be reduced through the gained flexibility in constraint management. However, such effects should be subject to detailed studies within industrial settings in order to evaluate the benefits and drawbacks.

Today, we are often thinking in terms of strict consistency and that any threats to integrity—and hence, dependability—have to be avoided and undesirable effects have to be removed or repaired immediately. While this is acceptable for small-scale and tightly-coupled systems, we currently observe a trend towards large-scale integration (systems of systems) and pervasive computing, leading to ultra-large-scale systems [Pol06] in the future. Dependability will be an important aspect of these systems—and integrity management will be part of it. However, strict consistency is not affordable in large- to ultra-large-scale systems. Hence, interesting future research challenges will arise from a transition of thinking in terms of consistency management to thinking in terms of inconsistency management—not only for logical systems [GH91] or requirements engineering [NER01], but as a more holistic concept in software engineering considering the whole software lifecycle including runtime and maintenance. While this dissertation focused on “constraints-in-the-small” defined upon objects, we will have to think of

"constraints-in-the-large" defined upon and between systems to address the challenges of the future. Such constraints will most probably be fuzzy, imprecise, and potentially require negotiation to decide whether constraints are fulfilled. An analogy that can guide our way in this direction is to view "constraints-in-the-large" as kind of laws, often being precise enough, but sometimes requiring court decisions (negotiations) to decide whether something was actually lawful—or not.

Appendix A

Consistency Management Model

Tarr and Clarke present a model for *consistency management in complex applications* in [TC98]. This model is primarily motivated from consistency between source-code, abstract syntax trees, and control flow graphs in software engineering environments, but certainly more generally applicable and maps to our research quite well. The following sections shortly reflect on their work and address how their requirements and model for consistency management map to our research. This on the one hand strengthens the definition and generality of the presented consistency management model and on the other hand shows that our constraint consistency management framework satisfies the requirements for consistency management in complex applications.

A.1 Functional requirements

With respect to the requirements for consistency management, Tarr and Clarke differentiate between functional and cross-cutting requirements. In order to satisfy the functional requirements, they specify that a system must be able to perform the following tasks:

- *Define consistency conditions*: First of all, it has to be clarified what it means for objects to be consistent. While the differentiation between "consistent" and "inconsistent" is a quite common approach, more fine grained degrees of consistency or inconsistency might be required for certain applications.

→ Within our work, we basically follow the "consistent" and "inconsistent" classification for the healthy system. However, for the degraded mode, we differentiate between "possibly satisfied", "possibly violated", and "uncheckable" as additional satisfaction degrees.

- *Determine when to detect violations*: This primarily refers to the trigger points of a constraint, also discussed under the *scope of a constraint* by Verheecke and Van der Straeten in [VS02]. While Tarr and Clarke provide examples for *operation-driven* detection (constraint validation triggered by certain operations), they argue that it might also be desirable to allow checks to be performed at a client's request or at specific process steps.

  → The primary mechanism to trigger constraint validation are operations or more specifically method invocations in our work as well. However, the point in time when constraints are actually validated depends on several circumstances, e.g., soft constraints are only validated at the end of a transaction or validation of asynchronous constraints is postponed to the reconciliation phase during degraded mode.

- *Specify enforcement semantics*: It must be possible to define appropriate responses to (potential) consistency violations. This includes to reject actions leading to violations, perform roll-back or roll-forward strategies in order to reach a consistent state, or allow objects to be in an inconsistent state and manage them accordingly. However, it is quite important to *consider that an initial repair action may be unsuccessful and subsequent actions may be required.*

  → Basically, we support all of the provided examples. The specific actions taken, however, depend on the system mode. While we reject inconsistencies in the healthy system, e.g. throw a ConstraintViolation exception and possibly abort the ongoing transaction, we perform inconsistency management in the degraded mode. In the reconciliation phase, we support roll-back and roll-forward strategies in order to clean up the inconsistencies in the system.

- *Manage inconsistency*: "Managing inconsistency means being able to detect inconsistencies, ensure the meaningful manipulation of inconsistent objects, and ultimately, reachieve consistency."

  → This task is actually at the core of our middleware enhancements. In our case, it is even more complex as we have to perform inconsistency management with respect to potential inconsistencies where we cannot even be sure, whether an inconsistency is actually introduced into

the system. However, reachievement of consistency is performed during the reconciliation phase based on the accepted consistency threats that are the result of the inconsistency management performed during degraded mode. This task strongly correlates to the enforcement semantics requirement mentioned above.

- *Dynamically change consistency specifications*: Based on the observation that the consistency requirements of a system might change over time, a consistency management system should support to change consistency specifications as well. Similarly, the repair actions may change.

  → Due to the fact that our constraints are collected within the constraint repository, we are able to support this requirement in the way that constraints can be added or removed from this repository. During our studies, however, we did not make use of this functionality as system evolution was not in the primary focus of our work. Adding or removing constraints would also require additional handling of consistency threats, for example. If a constraint is removed from the repository and hence no longer presents an integrity criterion of the system, consistency threats for this constraint should also be removed from persistent storage. On the other hand, if a new constraint is added to the repository, it would have to be validated based upon all affected objects in order to detect whether the system is consistent with respect to this new constraint. However, the consistency requirements in our systems primarily changed with respect to the system mode. For example, constraints that have to be satisfied within the healthy system might potentially be violated during degraded mode. We support this kind of changing consistency requirements based on metadata and runtime negotiation of consistency threats.

A.2 Cross-cutting requirements

Besides the functional requirements addressed in the previous section, Tarr and Clarke further specify a set of cross-cutting requirements in order to produce a flexible, broad-spectrum consistency management system:

- *Completeness*: "*Computational completeness* supports the definition of arbitrarily complex algorithms"—for consistency conditions as well as for enforcement semantics. "*Type completeness* provides the ability to associate consistency conditions and enforcement mechanisms with any type of object."

→ While our work focused on application data and hence EJB entity beans, the constraint consistency management part also supports other types of objects as well, e.g., EJB session beans. Conceptually, it can even be applied to simple Java applications as provided in Chapter 2. The only requirement to be satisfied is that of invocation interception, which can be achieved through integration into the middleware invocation mechanism or aspect-oriented programming, for example. Besides the support for different types, we also support computational completeness in the way that arbitrary code can be executed within the validate() method of a constraint. However, the code should be limited to read-only operations as constraint validation is not expected to change an application's business data, for example.

- *Metadata*: Information about system artefacts and data, i.e., metadata, support dynamic decisions at system runtime. Examples with respect to consistency management are the consistency conditions that are currently enforced on an object and details about an object's consistency status.

  → The consistency conditions in our case are the constraints in the constraint repository that can be queried in order to get the consistency conditions of different objects. The consistency status of objects, e.g., persisted consistency threats, is data and not metadata within the constraint consistency management framework. However, it is of course metadata with respect to the application data. Besides these two examples, we use additional metadata information, primarily with respect to integrity constraints. Examples include the trigger points of constraints or the specification of whether a constraint can be relaxed during degraded mode.

- *Generality/heterogeneity*: "*Generality* means that a consistency management system must provide a set of primitive capabilities that facilitate the implementation of alternative consistency management paradigms. *Heterogeneity* means that a consistency management system must allow alternative consistency management models and implementations to coexist peacefully.

  → While generality is often a desirable goal, it often also comes at the price of decreased performance or applicability and increased complexity and configuration issues. Consequently, we limited our investigations to distributed object systems. The co-existence of different consistency management models might be an interesting thing to investigate and probably support. In practice, however, the well-established and commonly used consistency model, the one described in [TC98],

and the one used in our work defines pre- and post-conditions as well as invariant constraints. Moreover, we support sub-classifications of invariant constraints into hard, soft, and asynchronous and an extension of the model would be possible in order to support constraints on sequences of operations as well. To sum up, generality and heterogeneity are often desirable goals, but supporting all kinds of systems, e.g., different programming paradigms or languages, within a single consistency management system is rather unreasonable. Consequently, the scope of target systems has to be reasonably defined in order to achieve an optimal support for the developer applying the consistency management system.

- *First class status and identity*: "First class status provides the ability to treat all objects uniformly. The ability to pass a consistency condition or action as a parameter to an operation is an example of this requirement. Identity means that a given entity has a unique identifier that is separate from its state."

  → First class status of constraints is achieved in our framework through encapsulation of consistency conditions within explicit integrity classes. Consequently, constraints can be processed like any other object in an object oriented programming language. The identity requirement is addressed in the way that each constraint in the system has a unique name by which it can be referred and looked up from the constraint repository.

A.3 Model of consistency management

Based on their requirements, Tarr and Clarke provide a model of consistency management. This model primarily includes four different issues:

1. Definition of consistency (conditions)
2. Trigger points of a condition—when to perform violation detection
3. Definition of enforcement mechanisms
4. Inconsistency management

Definition of consistency. Within their model, they define that the consistency status of a condition might be *consistent*, *inconsistent*, *partially consistent* if some but not all parts of a condition are consistent and *unknown*

if not enough information is available to evaluate the condition. Partially consistent and unknown address situations of uncertainty. Differently to our consistency threats, partially consistent means that parts of a condition are certainly inconsistent while "possibly satisfied" and "possibly violated" in our case only provide an indication of the validation result that is unreliable in the face of a network partition. The "unknown" state corresponds to our "uncheckable" satisfaction degree.

Trigger points. With respect to the trigger points of constraints, Tarr and Clarke differentiate between:

- *Preinvoke*: condition is checked before the invocation of the specified operation. This type should be applied where a "failure to satisfy the condition results in the invocation of an operation other than the one specified."
- *Precondition*: checked during the execution of the specified operation, but before the operation takes any other actions. Preconditions as well as postconditions require access to the runtime context of the operation.
- *Postcondition*: checked after the operation has performed its task, but before it terminates.
- *Postinvoke*: condition is checked after the operation executed and committed.

This differentiation seems to be mainly related to the fact that their consistency management system PLEIADES [TC93] is implemented as a preprocessor for Ada. Consequently, the discussion refers to the difference between in-line code instrumentation and wrapper-based approaches presented in Chapter 2. As they state in [TC98] that only pre- and postconditions are supported, they chose the in-line code instrumentation approach.

Several considerations apply to this decision. First, their system does not apply the concept of transactions and therefore the only option to prevent an inconsistent operation from commit is within the operation itself. Second, there is no notion of invocation interception which could be used in order perform the commit within one of the interceptors but after the operations returned—similar to our situation in the EJB environment. Moreover, the considerations with respect to in-line code instrumentation vs. wrapper-based approaches apply.

With respect to their definition, all of our pre- and postconditions are actually preinvoke and postinvoke constraints as they are checked within the interceptor chain before and after the actual method of an object was performed. As our system is integrated into EJB and thereby supports transactions, the commit of an operation is not an issue as the effects of the whole transaction can be undone through a transaction rollback. Moreover, we also integrate consistency management with transactions through the concept of soft constraints.

Enforcement mechanisms. The definition of enforcement mechanisms allows to specify one or more (repair) actions within action chains to be taken when a specific consistency condition is violated. The default action is to throw an exception, but this can be even dynamically modified. Actions are written in the programming language and therefore can execute basically anything in order to solve the inconsistency or send a notification the developer.

Inconsistency management. Finally, Tarr and Clarke point out inconsistency management as an important task within a consistency management system. The primary issue is that the consistency management system must be able to handle situations in which some objects are inconsistent and consider the fact that an initial action to repair the inconsistency may fail. This further implies the support for longer-lasting repair activities, i.e., operations performed on inconsistent objects in order to reach a consistent state.

Bibliography

[ABH+05a] Arjuna, BEA, Hitachi, IBM, IONA, and Microsoft. Web services atomic transaction, 2005. `http://www-128.ibm.com/developerworks/library/specification/ws-tx/`.

[ABH+05b] Arjuna, BEA, Hitachi, IBM, IONA, and Microsoft. Web services business activity framework, 2005. `http://specs.xmlsoap.org/ws/2004/10/wsba/`.

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.

[Bal91] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering*, pages 158–165. IEEE Computer Society Press, 1991.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[BBG+06] Stefan Beyer, Mari-Carmen Bañuls, Pablo Galdámez, Johannes Osrael, and Francesc Daniels Muñoz-Escoi. Increasing availability in a replicated partitionable distributed object system. In *The 2006 International Symposium on Parallel and Distributed Processing and Applications (ISPA 2006)*. Springer, December 2006.

[BBM+04] Özalp Babaoglu, Alberto Bartoli, Vance Maverick, Simon Patarin, Jaksa Vuckovic, and Huaigu Wu. A framework for prototyping J2EE replication algorithms. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE (2)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1413–1426. Springer, 2004.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with assertions. In Klaus Havel and Grigore Rosu, editors, *Proceedings of the First Workshop on Runtime Verification*, July 2001.

[BGM07] Stefan Beyer, Pablo Galdámez, and Francesc Daniels Muñoz-Escoi. Implementing network partition-aware fault-tolerant CORBA systems. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security (ARES 2007)*, pages 69–76, Washington, DC, USA, 2007. IEEE CS.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BMST93] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In S.J. Mullender, editor, *Distributed systems*, chapter 8, pages 199–216. ACM Press, Addison-Wesley, Wokingham, United Kingdom, 2nd edition, 1993. ISBN 0-201-62427-3.

[CD99] James Clark and Steve DeRose. XML path langugage (XPath), 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design.* Addison-Wesley, fourth edition, 2005.

[CEM01] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection*, volume 2192 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2001.

[CNGM95] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione. How to deal with deviations during process model enactment. In *ICSE '95: Proceedings of the 17th international conference on Software*

engineering, pages 265–273, New York, NY, USA, 1995. ACM Press.

[ÇÖSF01] Ugur Çetintemel, Banu Özden, Abraham Silberschatz, and Michael J. Franklin. Design and evaluation of redistribution strategies for wide-area commodity distribution. In *ICDCS*, pages 154–161, 2001.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370, 1985.

[DH98] Andrew Duncan and Urs Hölzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, Dec. 1998.

[DKE07] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[EEF99] Ernst Ellmer, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management of distributed documents using xml and related technologies. Technical Report 99-94, Dept. of Computer Science, University College, London, UK, 1999.

[EN96] Steve Easterbrook and Bashar Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, 1996.

[FB99] Armando Fox and Eric A. Brewer. Harvest, yield and scalable tolerant systems. In *Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

[Fe07] Lorenz Froihofer (ed.). FTNS technology comparison. Technical Report D4.1.1, DeDiSys Consortium, 2007. `http://www.dedisys.org/`.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 1–15, New York, NY, USA, 2001. ACM Press.

[FGH+94] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FN02] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *Proc. of Confederated Int'l Conf. DOA, CoopIS and ODBASE 2002*, pages 737–754, London, UK, 2002. Springer.

[GCG02] J. S. Goonetillake, T. W. Carnduff, and W. A. Gray. An integrity constraint management framework in engineering design. *Comput. Ind.*, 48(1):29–44, 2002.

[GH91] Dov M. Gabbay and Anthony Hunter. Making inconsistency respectable: a logical framework for inconsistency in reasoning. In Philippe Jorrand and Jozef Kelemen, editors, *FAIR*, volume 535 of *Lecture Notes in Computer Science*, pages 19–32. Springer, 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[GKR+06] Kurt Geihs, Mohammad Ullah Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, and Simon Merral. Modeling of component-based adaptive distributed applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 718–722, New York, NY, USA, 2006. ACM Press.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[Glo07] Gerhard Glos. Performance and usability evaluation of constraint checking approaches in java. Master's thesis, Vienna University of Technology, 2007.

[Gui07] Sam Guinea. *Dynamo: a Framework for the Supervision of Web Service Compositions*. PhD thesis, Politecnico di Milano, 2007.

[HHB96] Abdelsalam A. Helal, Abdelsalam A. Heddaya, and Bharat B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[IBM+03] IBM, BEA Systems, Microsoft, SAP, and Siebel Systems. Business process excecution language for web services BPEL4WS, 2003. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`.

[JQ92] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 469–480. Morgan Kaufmann Publishers Inc., 1992.

[Ke07] Hubert Künig (ed.). FTNS/EJB system design & first prototype & test report. Technical Report D3.2.2, DeDiSys Consortium, 2007. `http://www.dedisys.org/`.

[KHB99] Murat Karaorman, Urs Hölzle, and John L. Bruno. jContractor: A reflective Java library to support design by contract. In

Pierre Cointe, editor, *Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 1999.

[KIL+96] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[Kra98] Reto Kramer. iContract - The Java design by contract tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.

[LKP02] Martin Lackner, Andreas Krall, and Franz Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76, 2002. Special issue: TOOLS.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[NCEF02] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Inter. Tech.*, 2(2):151–185, 2002.

[NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.

[NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 12(1):28–63, 2003.

[NER01] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, September 2001.

[OCS01] Hussien Oakasha, Stefan Conrad, and Gunter Saake. Consistency management in object-oriented databases. *Concurrency and Computation: Practice and Experience*, 13(11):955–985, 2001.

[Osr07] Johannes Osrael. *Replication Techniques for Balancing Data Integrity with Availability.* PhD thesis, Vienna University of Technology, 2007.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: an asynchronous approach. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 377–386, New York, NY, USA, 1991. ACM Press.

[Plö02] Reinhold Plösch. Evaluation of assertion support for the Java programming language. *Journal of Object Technology*, 1(3):5–17, 2002.

[Pol06] Bill Pollak, editor. *Ultra-Large-Scale Systems.* Software Engineering Institute, Carnegie Mellon University, July 2006.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Trans. on Softw. Eng.*, SE-1(2):220–232, June 1975.

[RG00] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2000.

[RSB93] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "non partition" assumption. In *IEEE Proc. of Fourth Workshop on Future Trends of Distributed Systems*, 1993.

[Se05] Diana Szentiványi (ed.). Metrics. Technical Report D1.3.1, DeDiSys Consortium, 2005. `http://www.dedisys.org/`.

[SG02] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248, New York, NY, USA, 2002. ACM Press.

[SG03] Robert Smeikal and Karl M. Goeschka. Fault-tolerance in a distributed management system: a case study. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 478–483, Washington, DC, USA, 2003. IEEE Computer Society.

[SG04] Robert Smeikal and Karl Michael Goeschka. Trading constraint consistency for availability of replicated objects. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.

[Sme04] Robert Smeikal. *Trading Consistency for Availability in a Replicated System.* PhD thesis, Vienna University of Technology, 2004.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of Software Engineering and Knowledge Engineering*, 1, 2001.

[TC93] Peri Tarr and Lori A. Clarke. Pleiades: an object management system for software engineering environments. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 56–70, New York, NY, USA, 1993. ACM Press.

[TC98] Peri Tarr and Lori A. Clarke. Consistency management for complex applications. In *Proceedings of the 20th International Conference on Software Engineering*, pages 230–239. IEEE Computer Society, 1998.

[Ver01] Bart Verheecke. From declarative constraints in conceptual models to explicit constraint classes in implementation models. Master's thesis, Vrije Universiteit Brussel, 2001.

[VS02] Bart Verheecke and Ragnhild Van Der Straeten. Specifying and implementing the operational use of constraints in object-oriented applications. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 23–32. Australian Computer Society, Inc., 2002.

[Wie00] Ralf Wiebicke. Utility support for checking OCL business rules in Java programs. Master's thesis, Dresden University of Technology, Dec. 2000.

[WM05] Qianxiang Wang and Aditya Mathur. Interceptor based constraint violation detection. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 457–464, Washington, DC, USA, 2005. IEEE Computer Society.

[YV02] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

Glossary

ACID	Atomicity, Consistency, Isolation, Durability, page 4
AOP	Aspect-Oriented Programming, page 20
BPEL	Business Process Execution Language, page 79
BPEL4WS	Business Process Execution Language for Web Services, page 79
CAP	Consistency, Availability, Partition-tolerance, page 1
CCMgr	Constraint Consistency Manager, page 44
CORBA	Common Object Request Broker Architecture, page 20
DTMS	Distributed Telecommunication Management System, page 8
EJB	Enterprise JavaBeans, page 7
GMS	Group Membership Service, page 45
HTTP	Hypertext Transfer Protocol, page 57
IDE	Integrated Development Environment, page 26
J2EE	Java 2 Enterprise Edition, page 83
JBoss AS	JBoss Application Server, page 44
JIT	just-in-time, page 28
JML	Java Modeling Language, page 18
MW	Middleware, page 44
OCL	Object Constraint Language, page 9
RMI	Remote Method Invocation, page 57

RS	Replication Service, page 44
UML	Unified Modeling Language, page 9
VCS	Voice Communication System, page 8
XML	eXtensible Markup Language, page 80
XOR	Exclusive or, page 9

Index

Curriculum Vitae

Lorenz Froihofer

Year of birth: 1979

Email: lorenz@froihofer.net
WWW: `http://www.froihofer.net/`

Address: 8654 Fischbach 69a
Austria

Education

2004–2007	Ph.D. study at the Vienna University of Technology
2005–2006	Master's study in Computer Science Management at the Vienna University of Technology
1998–2004	Master's study in Computer Science at the Vienna University of Technology
1997–1998	Military Service
1993–1997	Grammar school with specialization in computer science

Work Experience

01/2005– **Research Assistant** at the Distributed Systems Group, Vienna University of Technology, within the EU research project Dependable Distributed Systems (DeDiSys).

Field of activity: Research on dependability in distributed systems, middleware, replication, (in)consistency management, system architecture. Work package leader of several work packages in the range of 1.5–3 person years.

10/2006–	**Lecturer** of "Distributed Computing" and "Messaging Design" at the University of Applied Sciences FH Joanneum in Kapfenberg.
02/2006–	**Lecturer** of "Distributed Computing and Middleware" at the University of Applied Sciences FH Campus Vienna.
07/2003– 09/2003	**Internship** at SIEMENS Austria within the scope of my master's thesis regarding WLAN Security.
07/2002– 09/2002	**Research Assistant** at the Distributed Systems Group of the Mediateam Oulu / University Oulu (Finland) *Field of activity:* Analyzed, designed, and implemented improvements for a distributed mobile application.
07/2002– 09/2002	**Study Assistant** at the Vienna University of Technology, Institute for Software Technology *Field of activity:* Hold lectures about UML. Managed two teams during a software engineering process.
08/2000– 09/2000, 02/2001	**Software Programmer** at CRP Schützinger, Graz *Field of activity:* Database application development with MS Access, VBA, and MS SQL server.

Languages

German (native)
English (excellent)
Italian (basic knowledge)

Awards

2005	Prize of the city of Vienna for excellent master's thesis

Publications

[1] Lorenz Froihofer, Markus Baumgartner, Johannes Osrael, and Karl M. Goeschka. Data partitioning through integrity constraints. In *Fast Abstract Proc. of the 37th Int. Conference on Dependable Systems and Networks*, 2007.

[2] Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 313–322, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael. Middleware support for adaptive dependability. In *Proc. of the 8th Int. Middleware Conference*. Springer, 2007.

[4] Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka. Trading integrity for availability by means of explicit runtime constraints. In *Proc. of the 30th Intl. Conf. on Computer Software and Applications (COMPSAC 2006)*, pages 14–17. IEEE Computer Society, 2006.

[5] Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka. Decoupling constraint validation from business activities to improve dependability in distributed object systems. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security (ARES 2007)*, pages 443–450. IEEE CS, 2007.

[6] Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka. Middleware/application interactions to support adaptive dependability. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with EuroSys 2007*, pages 31–36, New York, NY, USA, 2007. ACM Press.

[7] Johannes Osrael, Lorenz Froihofer, Norbert Chlaupek, and Karl M. Goeschka. Availability and performance of the adaptive voting repli-

cation protocol. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security (ARES 2007)*, pages 53–60. IEEE CS, 2007.

[8] Johannes Osrael, Lorenz Froihofer, Matthias Gladt, and Karl M. Goeschka. Adaptive voting for balancing data integrity with availability. In *On the Move to Meaningful Internet Systems 2006: Confederated Int. Workshops Proc.*, volume 4278 of *LNCS*, pages 1510–1519. Springer, 2006.

[9] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. A replication model for trading data integrity against availability. In *Proc. of the 12th Pacific Rim Int. Symposium on Dependable Computing (PRDC'06)*, pages 377–378, Washington, DC, USA, 2006. IEEE CS.

[10] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. What service replication middleware can learn from object replication middleware. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 18–23, New York, NY, USA, 2006. ACM Press.

[11] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. Availability/consistency balancing replication model. In *Workshop Proc. of the 21st Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–8. IEEE, 2007.

[12] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. Experiences from building service and object replication middleware. In *Workshop Proc. of the 6th Int. Symp. on Network Computing and Applications*. IEEE CS, 2007.

[13] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. On the need for dependability research on service oriented systems. In *Fast Abstract Proc. of the 37th Int. Conference on Dependable Systems and Networks*. IEEE CS, 2007.

[14] Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. *Software Engineering of Fault Tolerant Systems*, chapter Replication in Service-Oriented Systems. World Scientific Publishing, 2007.

[15] Johannes Osrael, Lorenz Froihofer, Karl Michael Goeschka, Stefan Beyer, Pablo Galdámez, and Francesc Daniels Muñoz-Escoi. A system architecture for enhanced availability of tightly coupled distributed systems. In *Proc. 1st Int. Conf. on Availability, Reliability and Security*

(ARES 2006), pages 400–407, Washington, DC, USA, April 2006. IEEE Computer Society.

[16] Johannes Osrael, Lorenz Froihofer, Hubert Kuenig, and Karl Michael Goeschka. Scenarios for increasing availability by relaxing data integrity. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy - Issues, Applications, Case Studies*, volume 2, pages 1396–1403. IOS Press, 2005.

[17] Johannes Osrael, Lorenz Froihofer, Georg Stoifl, Lucas Weigl, Klemen Zagar, Igor Habjan, and Karl M. Goeschka. Using replication to build highly available .NET applications. In *Workshop Proc. of the 17th Int. Conf. on Database and Expert Systems Applications*, pages 385–389, Washington, DC, USA, 2006. IEEE CS.

[18] Johannes Osrael, Lorenz Froihofer, Martin Weghofer, and Karl M. Goeschka. Axis2-based replication middleware for Web services. In *Proc. of the Int. Conf. on Web Services*. IEEE CS, 2007.