Middleware Support for Adaptive Dependability

Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael

Vienna University of Technology
Institute of Information Systems
Argentinierstrasse 8/184-1
1040 Vienna, Austria
{lorenz.froihofer,karl.goeschka,johannes.osrael}@tuwien.ac.at

Abstract. Generic middleware can often not provide satisfactory solutions, but neither is it acceptable to let the application developer reinvent the wheel each time. Therefore, middleware shall support reuse of infrastructural services while leaving the application in control. In particular, we contribute with a middleware approach to support adaptive dependability by balancing integrity and availability in distributed systems. To achieve this goal, we add a new middleware service for explicit runtime management of data integrity constraints. In order to provide the desired balancing with respect to an application's requirements and environment conditions, our approach supports the application developer with explicit interaction between middleware, application, and metadata. Based on our prototype implementation, we show how adaptive balancing of integrity and availability improves the overall dependability. The performance impairments of our approach are typically worth their costs in systems where the read-to-write ratio is high or write performance is not a limiting factor.

Keywords: Middleware, dependability, adaptivity, constraint consistency, inconsistency, replication.

1 Introduction

Today's software systems often face availability requirements of 24 hours per day, 7 days a week. While availability close to 100% is already hard to achieve in a healthy system, e.g., due to system maintenance operations, the situation becomes even worse if parts of a system suffer from failures and the system therefore operates in a degraded mode. However, availability (the readiness for correct service) is only one attribute of dependability [1]. Integrity, the absence of improper system alterations, is another. Within our work, we focus on consistency of data with respect to data integrity constraints, i.e., *constraint consistency* [2]. The constraints stem from an application's requirements and have typically to be satisfied in the course of business transactions. Consistency of the constraints themselves, e.g., whether they represent conflicting requirements, is not within the focus of our work.

Failures are threats to dependability and hence to availability and integrity. While failures affecting availability might lead to a non-responsive system, integrity violations may lead to inconsistent data. We focus on node and link

R. Cerqueira and R.H. Campbell (Eds.): Middleware 2007, LNCS 4834, pp. 308–327, 2007.
© IFIP International Federation for Information Processing 2007

failures, assuming the *crash failure model [3] 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 [4], we initially treat node failures as partitions with a single node. Whether a node or link failed can be detected after the node is reachable again.

Replication [5], the process of maintaining several copies (replicas) 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 replication of entities, however, introduces a new integrity criterion: *replica consistency*. Replica consistency requires that replicas of an entity are consistent according to the used replica consistency model, e.g., 1-copy-serializability [6] or looser consistency models like ϵ-serializability [7] and eventual consistency. As replica consistency may impair constraint consistency, it is in the focus of our work as well.

It is well known that **C**onsistency, **A**vailability, and **P**artition-tolerance (CAP) cannot be optimized independently of each other. This interdependency is stated more precisely in the (strong) *CAP principle* [8,9] providing that *only two of the three requirements can be achieved*, e.g., a system can be available and consistent but not be partition-tolerant. However, the weak CAP principle specifies that *the stronger guarantees are provided for two of these properties, the weaker guarantees can be provided for the third*. Obviously, these three properties have to be balanced according to an application's requirements. Moreover, it would be beneficial if the system could adapt to changing requirements during runtime, e.g., due to node or link failures.

While in the past adaptation mechanisms were incorporated into software on a per system basis, where they are hard to change, reuse, or analyze, the proliferation of such systems suggests to include adaptation support into the middleware. Thus, the adaptation mechanisms can be reused in numerous systems, analyzed separately from the system being adapted, and easily changed to incorporate new adaptations. Moreover, they provide a natural home for encoding the expertise of system designers and implementers about adaptation strategies and policies.

It is important to note that we are *not* aiming at *transparent* adaptivity: While the initial motivation to introduce middleware stems from the goal to re-use infrastructure code and encapsulate it behind coherent service interfaces for the application programmer, soon transparency was introduced to conceal the distribution of components from the user and the application programmer, so that the system is perceived as a single coherent system rather than as a collection of independent components. Different kinds of transparency have been standardized by ISO (International Standardization Organization, ISO 10746-1:1998, `http://www.iso.org/`) and ANSA (Advanced Network Systems Architecture) [10] and have been in the focus of middleware research.

Unfortunately, generic transparency (if not impossible at all) often comes at the cost of impaired performance and other quality properties. Additionally, users or application developers sometimes require knowledge about certain distribution aspects (e.g., in the presence of certain failure scenarios). Consequently, transparency in itself is not the ultimate design goal of a distributed system, but

neither is it advisable to unconditionally follow the so-called "end-to-end argument" that some properties can only reasonably be provided under consideration of the application semantics and therefore have to be actually implemented by the application itself. Rather should middleware support the integration of applications with configurable re-used infrastructure services.

If application requirements are available during run-time in a processable form, they can explicitly be manipulated, configured, and processed by the application as well as the middleware, which allows such a system to balance or trade certain requirements against each other during run-time. By applying these principles, the application can be left in control to avoid costly generic solutions. In particular, we contribute with a middleware approach to support adaptive dependability by balancing integrity and availability. In order to provide the desired balancing with respect to an application's requirements and environment conditions, our approach supports the application developer with explicit management of data integrity constraints.

Paper Overview. First, Sect. 2 introduces our concept of balancing availability and integrity before Sect. 3 describes a prototype implementation with explicit runtime constraint consistency management. Section 4 then provides evaluation results and corresponding conclusions. We give an insight into related work in Sect. 5 and conclude this paper in Sect. 6.

2 Balancing Concept

For investigating the trade-off between integrity and availability, we concentrate on *data-centric* systems [11], which 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. 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.

One example for such an application scenario is a distributed telecommunication management system (DTMS) [12]. The DTMS is a software application that 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.

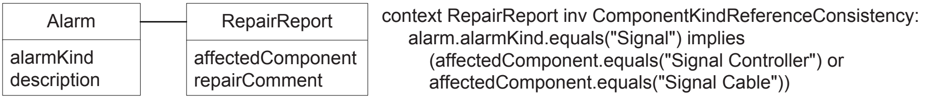

Fig. 1. Simplified ATS model with constraint

Another application scenario, where a prototype has been implemented by an industry partner based upon our middleware, is a distributed alarm tracking system (ATS) [13]. A simplified model of this system is given in Fig. 1 for our studies within this paper. The simplified ATS has two objects 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 Fig. 1 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.

2.1 Constraints and Consistency Threats

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) for Unified Modeling Language (UML) class diagrams. We follow the well-established approach to differentiate between preconditions (bound to and checked before a specific method invocation), postconditions (bound to and checked after a method invocation), and invariant constraints (bound to a certain class—the *context class*) [14]. 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) [15]. 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 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—an *affected method* of the constraint—was called.

Although an invariant constraint is defined for a certain context class, affected methods might belong to other classes as well. For our example in Fig. 1, 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. Obviously, the affected methods of a constraint cannot generically be determined without further knowledge of the constraint. Moreover, checking the ComponentKindReference-Consistency 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.

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. For example, if the technical operator of an ATS application sets componentKind in a RepairReport while the system operates in degraded mode. The administrative operator might have changed the corresponding Alarm in the meantime in another partition. Consequently, the constraint validation performed because of the changed RepairReport is not fully reliable. Hence, we call such a situation a *consistency threat* [2].

2.2 Availability Improvements and Reconciliation

Systems with a strict consistency model require to block or abort operations if a consistency threat occurs. However, some constraints might not be critical for "sufficiently" correct system operation and can temporarily be relaxed (traded). The application developer decides which constraints are tradeable and specifies the according metadata about the constraint. During runtime, the middleware is responsible to appropriately trigger constraint validation.

Whenever a consistency threat is detected and the corresponding constraints are tradeable, the middleware triggers the negotiation process to decide whether to accept or not accept the current consistency threat. Negotiation can either be performed descriptively, e.g., accept the threat if no affected objects are older than n seconds, or algorithmically via an application callback. In an algorithmic negotiation process, the application may associate application-specific information with accepted threats. Whether a threat is accepted can be decided by the callback handle provided by the application developer on its own, or it might even contact the end user for a threat-specific decision. However, if the threat is not accepted, the current transaction is aborted. If the consistency threat is accepted, the middleware stores this threat (including the application data) to be re-evaluated at a later time when node or link failures are repaired.

After two partitions are reunified (node or link failure repaired), our system starts the reconciliation phase. First of all, the replication protocol starts

to propagate updates performed in one partition to the replicas in the other partition(s). If replica (write-write) conflicts are detected, conflict resolution can either be performed generically, e.g., by performing a rollback to previous states, or application-specific, e.g., through an application callback by the replication protocol. However, replica conflicts are also provided to the constraint consistency component in order to support re-establishment of constraint consistency.

After replica consistency is re-established, constraint consistency has to be restored (constraints are defined upon objects and not between replicas of a single object). For this purpose, the stored consistency threats are re-evaluated. If re-evaluation is successful, i.e., the corresponding constraint is satisfied, no inconsistency was actually introduced by the consistency threat—if not, appropriate actions have to be taken. Such actions, again, can be generic (e.g., roll-back) or application-specific (e.g., call-back) including compensation. One further generic option is to validate constraints based on different selections of replicas for objects with replica conflicts detected during replica reconciliation. If any of these combinations satisfies the constraint, the solution can automatically be established or be presented to the application for confirmation.

2.3 Relationship to the Concept of Transactions

Traditional systems apply ACID (Atomicity, Consistency, Isolation, Durability) transactions [16], requiring that all four properties are met. Replication ("R") can synchronously be bound to transactions. 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. [17] do not include consistency in their list of transaction properties and rather specify that the "C" is under the responsibility of the application developer.

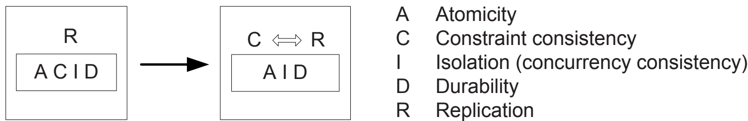

Fig. 2. Trading transactional properties for adaptive dependability

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 in degraded mode and one or more transactions to resolve conflicts during reconciliation). These considerations rather correspond to the concepts of atomic transactions [18] vs. business activities [19] 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 Fig. 2.

3 Middleware Support

These concepts for adaptive dependability have been integrated into a platform independent system architecture [20], which has been implemented in different prototypes using several technologies (EJB, CORBA, .NET). Within this section, we first provide 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 (AS). Second, we contribute with a detailed description of constraint consistency management as a new middleware service.

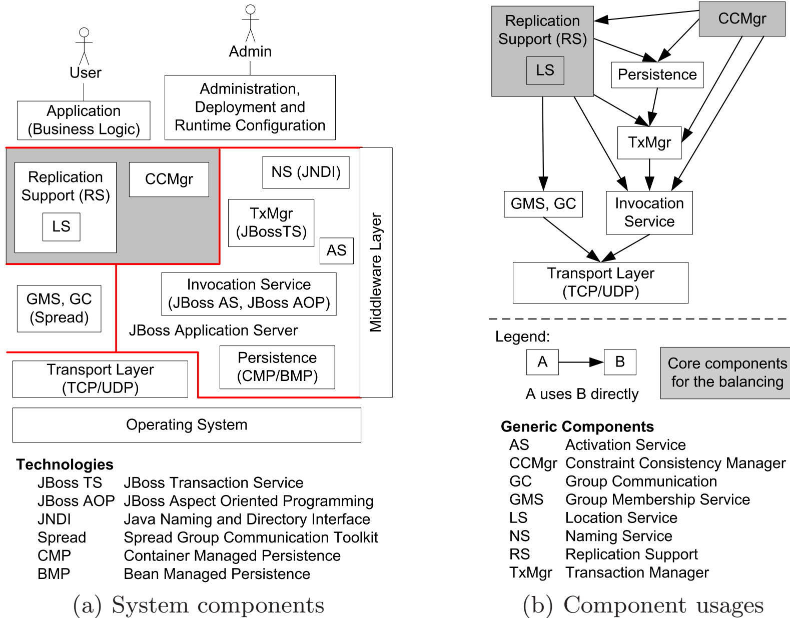

Fig. 3. EJB/JBoss AS specific system architecture

3.1 System Architecture

Two components of our architecture are primarily responsible for the balancing of availability and integrity, the *replication support* (RS) and the *constraint consistency manager* (CCMgr) provided within the grey areas of Fig. 3. 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* to detect node and link failures, and the *group communication* component that is used for update propagation (from primaries to backups) by the replication support. The *naming service*, allowing for name to object bindings, and the *activation service*, responsible for appropriate activation of objects, are not of immediate interest within our platform.

Although quite common and concise, the layered representation in Fig. 3(a) 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, Fig. 3(b) further provides an overview of usage-relations between the major components. This part of the figure shows that transaction management and invocation service are the two central services where almost all of the other services depend upon.

3.2 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 run-time 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.

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 constraint classes similar to Verheecke et al. [21]. The primary contract between middleware and application is the Constraint.validate(ctx : ConstraintValidationContext) method (Fig. 4) 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.

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, called 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,

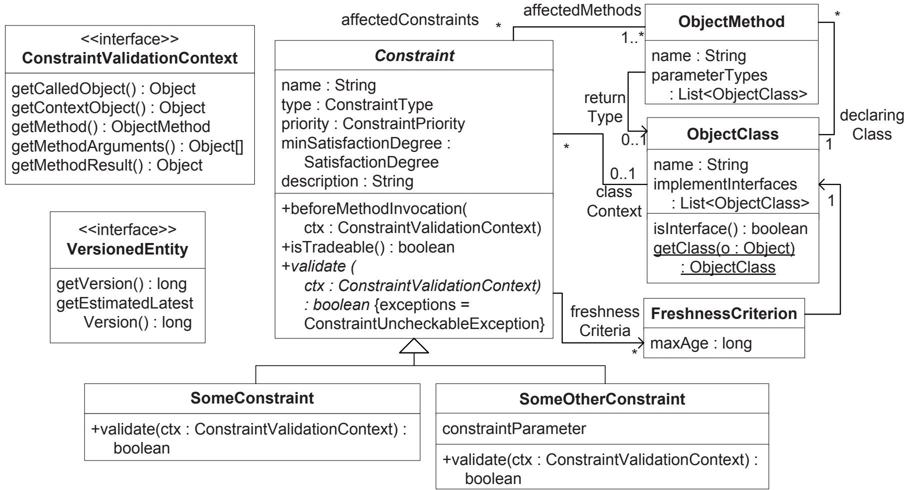

Fig. 4. Constraint runtime model

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.

Constraint Configuration and Registration. To allow appropriate validation, we need to know which constraints are affected by which method invocations. As motivated in Sect. 2.1, 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 1.1 provides an example of a constraint specification within the configuration file.

The constraint ComponentKindReferenceConsistency implements the integrity constraint of the ATS application provided in Fig. 1. It is a hard constraint, specifies that the constraint implementation requires a context object, it can be relaxed

Listing 1.1. Constraint configuration example

```
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>
```

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 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 Fig. 4) 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).

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 support in order to detect consistency threats caused by 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 or link failures and the underlying replication protocol. For example, in the primary partition protocol [22], each object accessed in a non-primary partition is possibly stale. In the case of the primary-per-partition protocol [23], objects are possibly stale in every network partition. However, before the CCMgr triggers the validation of a constraint, it starts to gather accessed objects, see Fig. 5. 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.

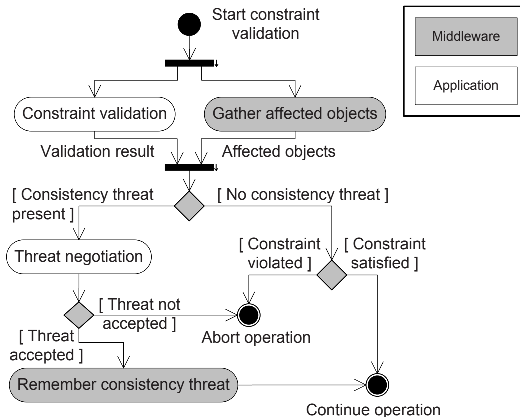

Fig. 5. Detection and negotiation of consistency threats

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

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. Reconciliation of constraint consistency is performed after reconciliation of replica consistency. Consequently, the CCMgr only starts its work after having received a notification from the replication manager that it has finished its reconciliation work.

To reconcile constraint consistency, the constraint consistency manager looks up accepted consistency threats and re-evaluates the corresponding constraints. If a constraint is satisfied, no inconsistency was introduced during degraded mode and the data about the consistency threat are removed. If a constraint is still threatened (node or link failures affecting the constraint are still present), re-evaluation of the corresponding threat is postponed until further repair. If a constraint is violated, an inconsistency was introduced during degraded mode and appropriate actions have to be taken to satisfy the constraint.

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 an interceptor chain where each interceptor invokes the next interceptor until the final interceptor invokes the bean instance. The result of the invocation is passed back in the reverse order. The interceptors are responsible to provide middleware services for the invocation—enhanced by constraint consistency management and replication in our case. Fortunately, the invocation interceptors of the chain can be specified in a configuration file of the JBoss AS 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 within the process of an invocation. The implementation of the replication protocol is based on the ADAPT replication framework [24], which also hooks into JBoss through custom interceptors.

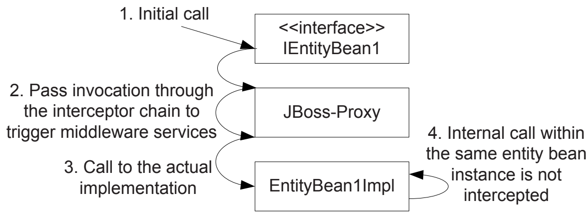

Fig. 6. JBoss invocation interception

Unfortunately, the interceptor chain is only traversed if the invocation comes from a call to the interface which is passed through the interceptor chain by the JBoss proxy (EJB container). If the bean instance calls another method on itself, this (internal) invocation is not intercepted, e.g., call number four in Fig. 6. 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.

3.3 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) [23] 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 non-critical constraints are affected. During repair, detected conflicts are solved either by rollback to previous states or by an application-specific compensation callback—potentially even involving a system operator.

However, the reconciliation process of the replication protocol has an influence on the handling of constraints and consistency threats. Consequently, constraints are further divided in intra- and inter-object constraints to address this fact. 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.

If the replica reconciliation process resolves replica conflicts (replicas of a single logical object were written in different partitions) through selection of one copy (and not by creating a new state for the object by merging values of disjoint sets of attributes of the different replicas), a differentiation of integrity constraints into intra- vs. inter-object constraints is useful. In this case, intra-object

constraints will not be violated retrospectively by the replica reconciliation process. Therefore, constraint validations based upon possibly stale objects can still return satisfied or violated instead of possibly_satisfied or possibly_violated 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 constraint 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., Fig. 1) constraints. Although this differentiation is useful for constraint implementation, it is not significant with respect to our balancing of dependability.

4 Evaluation

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 Ethernet network links. The configuration denoted as "No DeDiSys" is a standard JBoss AS 4.0.4 with JBoss TS 4.2.1b1 as transaction service for distributed transactions and MySQL 5.0.21 for persistent storage. The "DeDiSys" configuration additionally applies the principles provided within this paper 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 [13].

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. Details on this issue are available in [25]. 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 7 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,

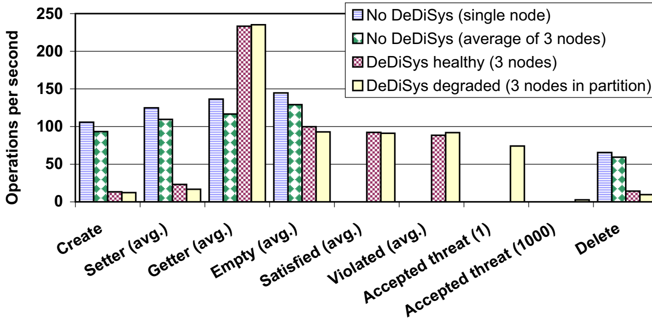

Fig. 7. No DeDiSys vs. DeDiSys in healthy and degraded mode

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. 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.

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 [24]. 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 [24]. 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[2] 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 paper is not focused on an efficient implementation of the P4 replication protocol, the effects of introducing replication 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 8 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 primarily shows the overhead of the ADAPT replication framework and 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.

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

[2] Two consistency threats are identical if they refer to the same constraint and—if applicable—to the same context object.

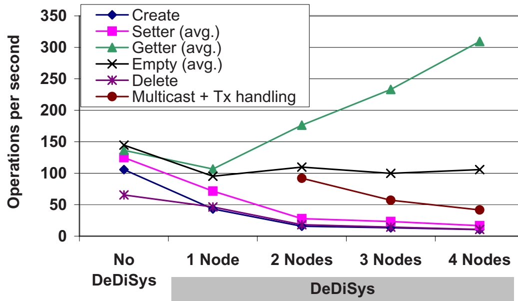

Fig. 8. Replication effects on different operations

for the single node scenario and reaching 227% in the four node replicated setting. Empty methods operate at a rather constant ratio independent of the number of nodes in the system. The reason for this is that they do not trigger update propagation on the one hand. On the other hand, as these methods do not adhere to any naming convention, we consider them as write operations—to be on the safe side—and therefore execute them only on the primary node. This behaviour is the same as 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 possible per second 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 Fig. 8. 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.

We conclude that only under extremely demanding performance requirements the performance impairment due to explicit constraint management may turn out to be a problem, see also [2,25] for further details. On the other hand, synchronous replication significantly reduces system performance for update operations while the performance of read operations is improved. Therefore, synchronous replication should be applied if the read to write ratio is high and/or write performance is not the limiting factor. In other cases, asynchronous replication protocols or only partial replication (updates are only propagated to some but not all nodes within the system) should be applied.

5 Related Work

The balancing of integrity and availability has already been investigated with respect to isolation [26,27] and replica consistency [28,7,29] and different strategies to optimistic replication are already well-known [30]. The focus of our work to trade constraint consistency for availability did not yet receive too much

attention. Balzer [31] uses pollution markers to temporarily allow and denote constraint violations in a healthy system to allow certain business cases. The application using this approach tolerates inconsistency in the way that reports are marked appropriately if they contain data affected by pollution markers. Although our stored consistency threats roughly correspond to pollution markers, Balzer accepts constraint violations in a healthy system and does not consider node or link failures while we aim at fully consistent data in the healthy system and accept consistency threats during degraded mode. However, integration of both approaches would most likely provide further benefits to an application developer.

Representing data integrity constraints as explicit constraint classes was inspired by Verheecke et al. [21] who perform a transformation from UML class diagrams enhanced with OCL constraints into Java objects and Java constraint checking classes. Their approach generates skeletons for the classes with hardwired triggers for constraint validation while we require that constraints are explicitly manageable at runtime for adaptivity with respect to node and link failures. However, 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 model driven approach used by Verheecke et al. could be integrated with our constraint checking framework. Consequently, entity beans, constraints and metadata could be generated based on UML models annotated with OCL constraints.

ADAPT [24] provides a replication framework to allow rapid prototyping of replication protocols in J2EE (Java 2 Enterprise Edition) environments. 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.

6 Summary and Conclusion

This paper presents a middleware approach to support adaptive dependability by balancing integrity and availability. We show how explicit runtime management of constraints as a middleware service can support the application to provide the envisaged 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. According to our prototype implementation and additional evaluation studies [2,25], 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. Therefore, this system mode is subject to improvements and future research questions.

Acknowledgments

We thank Hubert Künig for many in-depth discussions, Markus Horehled and Klaus Fuchshofer who contributed major parts of the proof-of-concept EJB prototype implementation integrated into the JBoss application server, and Dominik Ertl who was strongly involved in the performance tests. We further thank the anonymous reviewers for their useful comments. This work has been partially funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract 004152, http://www.dedisys.org/).

References

1. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
2. Froihofer, L., Osrael, J., Goeschka, K.M.: Decoupling constraint validation from business activities to improve dependability in distributed object systems. In: Proc. 2nd Int. Conf. on Availability, Reliability and Security, pp. 443–450. IEEE Computer Society, Los Alamitos, CA (2007)
3. Cristian, F.: Understanding fault-tolerant distributed systems. Communications of the ACM 34(2), 56–78 (1991)
4. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
5. Helal, A.A., Heddaya, A.A., Bhargava, B.B.: Replication Techniques in Distributed Systems. Kluwer Academic Publishers, Dordrecht (1996)
6. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
7. Pu, C., Leff, A.: Replica control in distributed systems: an asynchronous approach. In: SIGMOD 1991: Proceedings of the 1991 ACM SIGMOD international conference on Management of data, pp. 377–386. ACM Press, New York (1991)
8. Fox, A., Brewer, E.A.: Harvest, yield and scalable tolerant systems. In: Workshop on Hot Topics in Operating Systems, pp. 174–178 (1999)
9. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
10. Architecture Projects Management: The advanced network systems architecture (ANSA) reference manual (1989), http://www.ansa.co.uk/ANSATech/89/ANSAREF/
11. Osrael, J., Froihofer, L., Kuenig, H., Goeschka, K.M.: Scenarios for increasing availability by relaxing data integrity. In: Cunningham, P., Cunningham, M. (eds.) Innovation and the Knowledge Economy - Issues, Applications, Case Studies, vol. 2, pp. 1396–1403. IOS Press, Amsterdam (2005)
12. Smeikal, R., Goeschka, K.M.: Fault-tolerance in a distributed management system: a case study. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, pp. 478–483. IEEE Computer Society, Washington, DC (2003)
13. Künig, H. (ed.): FTNS/EJB system design & first prototype & test report. Technical Report D3.2.2, DeDiSys Consortium (2007), http://www.dedisys.org/
14. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
15. Jagadish, H.V., Qian, X.: Integrity maintenance in object-oriented databases. In: Proceedings of the 18th International Conference on Very Large Data Bases, pp. 469–480. Morgan Kaufmann Publishers Inc., San Francisco (1992)

16. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (1983)
17. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems - Concepts and Design, 4th edn. Addison-Wesley, Reading (2005)
18. Arjuna, BEA, Hitachi, IBM, IONA, Microsoft: Web services atomic transaction (2005), http://www-128.ibm.com/developerworks/library/specification/ws-tx/
19. Arjuna, BEA, Hitachi, IBM, IONA, Microsoft: Web services business activity framework (2005), http://specs.xmlsoap.org/ws/2004/10/wsba/
20. Osrael, J., Froihofer, L., Goeschka, K.M., Beyer, S., Galdámez, P., Muñoz Escoi, F.D.: A system architecture for enhanced availability of tightly coupled distributed systems. In: Proceedings of the 1st International Conference on Availability, Reliability and Security, IEEE Computer Society, Los Alamitos (2006)
21. Verheecke, B., Straeten, R.V.D.: Specifying and implementing the operational use of constraints in object-oriented applications. In: Proceedings of the Fortieth International Conference on Tools Pacific, Australian Computer Society, Inc,. pp. 23–32 (2002)
22. Ricciardi, A., Schiper, A., Birman, K.: Understanding partitions and the non partition assumption. In: IEEE Proc. of Fourth Workshop on Future Trends of Distributed Systems, IEEE Computer Society Press, Los Alamitos (1993)
23. Beyer, S., Bañuls, M.C., Galdámez, P., Osrael, J., Muñoz Escoi, F.: Increasing availability in a replicated partitionable distributed object system. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, Springer, Heidelberg (2006)
24. Babaoglu, Ö., Bartoli, A., Maverick, V., Patarin, S., Vuckovic, J., Wu, H.: A framework for prototyping J2EE replication algorithms. In: Meersman, R., Tari, Z. (eds.) On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. LNCS, vol. 3291, pp. 1413–1426. Springer, Heidelberg (2004)
25. Froihofer, L., Glos, G., Osrael, J., Goeschka, K.M.: Overview and evaluation of constraint validation approaches in Java. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 313–322 (2007)
26. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. SIGMOD Rec. 24(2), 1–10 (1995)
27. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
28. Davidson, S.B., Garcia-Molina, H., Skeen, D.: Consistency in a partitioned network: a survey. ACM Comput. Surv. 17(3), 341–370 (1985)
29. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. Comput. Syst. 20(3), 239–282 (2002)
30. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
31. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th international conference on Software engineering, pp. 158–165. IEEE Computer Society Press, Los Alamitos (1991)