Decoupling Constraint Validation from Business Activities to Improve Dependability in Distributed Object Systems

Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka
Vienna University of Technology
Institute of Information Systems
Argentinierstrasse 8/184-1
1040 Vienna, Austria
{*lorenz.froihofer|johannes.osrael|karl.goeschka*}*@tuwien.ac.at*

Abstract

Integrity constraints are an important means to discover and specify application requirements. Although they are explicitly available and discussed during the system analysis and design phases, the constraint validation functionality is generally still tangled with other implementation code, e.g., the business logic, in today's systems. We contribute with an approach to decouple the integrity constraints from the business logic as well as the setpoints of constraint validation from the business activities. This allows us to balance dependability with respect to node and link failures by temporarily relaxing constraint consistency. Our prototype implementation indicates that this approach is typically worth its effort in systems where availability is of higher priority than strict consistency and a roll-forward approach to system repair, e.g., through compensating actions, is preferred over generic rollback-based solutions.

1 Introduction

Application requirements are often analyzed in terms of use cases, classes/entities, sequence diagrams—and constraints. In this analysis phase, the requirements and constraints are explicitly available, e.g., by stating "The system must not sell more tickets than available seats for an event." The constraints of an application are still explicitly available in the design of a system. For example, nowadays it is common practise to design systems by using the Unified Modelling Language (UML). UML includes an explicit language for specifying constraints, the Object Constraint Language (OCL). OCL can be used to explicitly define constraints upon a UML class model—in addition to constraints, such as cardinality or XOR, that can already be specified in the graphical notation of UML. Figure 1 provides an example for the previous constraint.

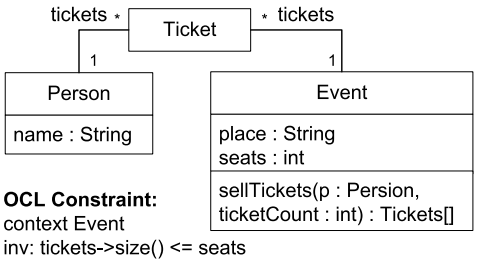

Figure 1. Design-phase constraint example

Unfortunately, these constraints are often intertwined with business code during implementation. Listing 1 provides such an example. However, depending on how the system is implemented, there could be several other places of where to perform the constraint validation. 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.

Meyer [15] 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 exam-

Listing 1. Constraint implementation

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

ple, if nodes are unreachable, some constraints might not be checkable due to unavailable objects. Systems requiring strict consistency will have to 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 constraints that are explicitly available and manageable during runtime. This approach allows for constraint validation at any time or adapting to changing consistency requirements. Within this paper we discuss how explicit runtime constraints as first class citizens within a distributed object system can be used to balance dependability [1] with respect to node and link failures.

Paper overview. Section 2 provides an introduction into target systems, applications, and failure model. Section 3 introduces our constraint model and the notion of a consistency threat. In Section 4 we discuss how the decoupling of constraint validation can be used to improve dependability. The consequences of our trading—the reconciliation of possible inconsistencies—is discussed in Section 5. Section 6 shortly introduces the prototype implementation and provides results and lessons learned from it. We give an overview of related work in Section 7 and provide our conclusions in Section 8.

2 System model and applications

Within this work, we focus on the class of tightly-coupled data-centric distributed object systems. Our system model has three major states (Figure 2). 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

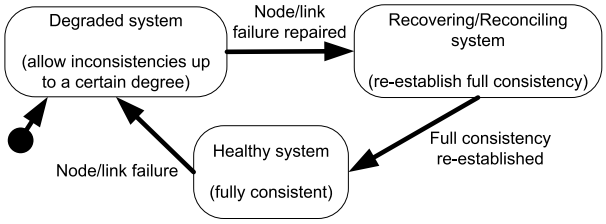

Figure 2. Major system states

introduced in degraded system periods, we require that software manages constraints explicitly during runtime. For this purpose, we also need metadata about constraints, e.g., whether a constraint must never be violated or might potentially be violated in degraded mode. Given this input, our middleware afterwards provides the support for runtime-management of constraint consistency in each of the three major system states.

These system states are primarily defined with respect to node and link failures. Failures are threats to dependability [1] and affect its attributes—availability, the readiness for correct service, and integrity, the absence of improper system alterations, in our case. While failures affecting availability might lead to a non-responsive system, integrity violations may lead to inconsistent data. Within our work, we consider node and link failures, assuming the *crash failure model [7] for nodes*—pause-crash for server nodes—and *links may fail by losing some messages but do not duplicate or corrupt messages*. However, as node and link failures cannot be differentiated at the time when they occur [9], we initially treat node failures as partitions with only a single node. Whether a node or link failed will be detected at the time the node is reachable again.

Replication [11], 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 [19], 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 distinguish three different kinds of consistency with respect to data integrity [21]:

- *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 "logi-

cal" entity with respect to a particular replica consistency model (e.g., 1-copy-serializability [4]). Replica inconsistency is caused by staleness, e.g., if the backup copies differ from the primary copy in a primary-backup replication protocol [6]. Isolation of concurrent access to different replicas of the same logical entity is also achieved by replica control (in cooperation with the isolation mentioned above).

- *Constraint consistency*: defines the correctness of data with respect to data integrity constraints that stem from the application requirements. This is the kind of consistency we focus on within this paper.

An application example for such a system scenario is a distributed telecommunication management system (DTMS) [20]. A DTMS manages a voice communication system (VCS), e.g., it establishes communication channels between partners. In this system, distributed objects are bound to physical hardware facilities but are replicated to other nodes to prepare for degraded system periods. The reason is that, for example, the parameterization of a communication channel requires data of objects located at different nodes. For correct parameterization, the objects have to be consistent with respect to certain integrity constraints.

3 Constraints and consistency threats

The "design by contract" principle as well as the Object Constraint Language (OCL) of UML, provide the basis for our first classification of constraints: preconditions, postconditions and invariant constraints. Preconditions have to be validated before the call to a method, postconditions have to be satisfied after the call to a method returns. 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.

We distinguish between transactional and non-transactional applications to decide when to trigger validation of invariant constraints. For non-transactional applications, an invariant constraint must be checked immediately after the call to a method which might change the state constrained by this specific invariant—an *affected constraint* of the method. Vice versa, the method is an *affected method* of the constraint. All objects restricted by a certain constraint are *affected objects* of the constraint. The validation of a constraint requires access to all affected objects. For transactional applications, we further differentiate between hard and soft constraints [13]. Hard constraints are validated like postconditions immediately after the call to an affected method. All affected soft constraints of a transaction are validated at the end of the transaction.

In order 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 of *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, invariant constraints can be used for re-establishing constraint consistency during system reconciliation. Pre- and postconditions can 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 Constraint runtime representation

A data integrity constraint is generally a predicate on the system state and can either be satisfied or violated. Similar to [23], we implement data integrity constraints by constraint validation classes where one class represents exactly one integrity constraint. Each class provides a `validate(...)` method which is called to validate the constraint. Constraints are defined within the context of a class for invariants (the *context class* of the constraint) or the context of a method for pre- and postconditions. To validate invariant constraints, an instance of the context class (the *context object*) is provided as parameter to the validate method of the constraint. Certain invariant constraints might not even need a context object because they obtain the objects needed for validation through a query operation. For pre- and postconditions, we provide the called method arguments in addition to the context object as parameter to the validate method of a constraint. For postconditions, we further provide the result of the method invocation.

3.2 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.

- *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 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).

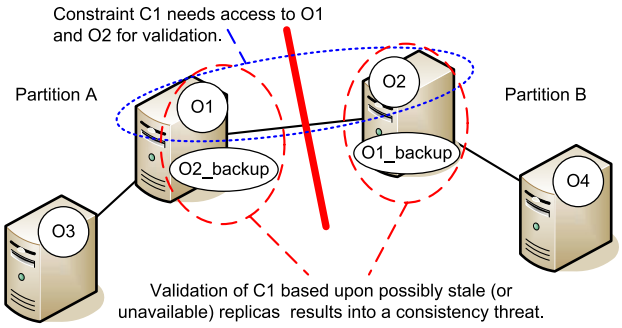

Figure 3. Consistency threats

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 example in Figure 1, 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.

However, this enhanced set of possible validation results for a single constraint requires a specification of how the validation results of a set of constraints 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 [19], each object accessed in a non-primary partition is possibly stale. In the case of the primary-per-partition protocol [5], objects are possibly stale in every network partition.

4 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 a particular application.

The application-specific trade-off is configured through the specification of tradeable and non-tradeable constraints.

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

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.

4.1 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. A general overview of the negotiation process is provided in Figure 4.

4.2 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:

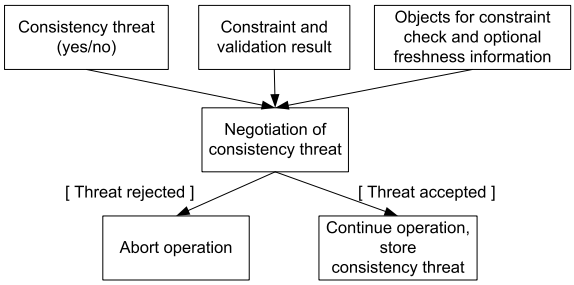

Figure 4. Overview of the negotiation process

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.

5 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 (i) 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 (ii) 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" [18], 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. 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 a different kind of decoupling constraints, we do not focus on.

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 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.

6 Results and lessons learned from prototype

We integrated these concepts into a system architecture for tightly-coupled data-centric systems where management of explicit constraints is one essential part of the overall architecture [16]. This management of explicit constraints is performed by the *Constraint Consistency Manager (CCMgr)*, which triggers the validation of constraints and the negotiation of consistency threats and drives the reconciliation of accepted consistency threats.

To allow such behavior, the CCMgr is notified before and after method invocations. Constraints affected by these method invocations are looked up from the constraint repository that stores the constraints of an application and allows to find constraints by different criteria, such as the class of an object, the called method, or the constraint type (pre-, postcondition, invariant hard, or invariant soft). Validation of affected constraints is triggered by the CCMgr according to the constraint type. Depending on the outcome, appropriate actions are taken by the CCMgr (e.g., abort the current transaction in case of constraint violations).

Furthermore, we mapped the system architecture onto the Enterprise JavaBeans (EJB) platform and integrated our prototype implementation into the JBoss application server [10, 14]. To investigate the overhead for different operations, we created, changed, called empty methods, and deleted EJB entity beans. The prototype implementation indicates that explicit runtime management of constraints is a feasible approach, causing almost negligible performance loss between 1–10% as shown in Figure 5. Consequently, only in case of extremely demanding performance requirements, explicit runtime management of constraints might become too costly.

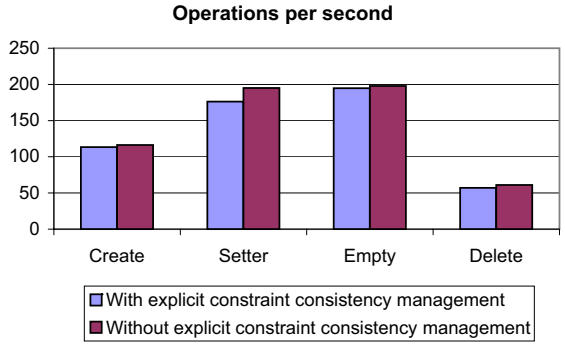

Figure 5. Performance overhead caused by explicit constraint consistency management

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 8–33% of the values without constraint consistency management in Figure 5 for update operations and about 72% 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.

However, it is important to consider 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 reconciliation through rollback to previous states is actually acceptable. 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.

To evaluate the costs of decoupled constraint validation for improved dependability, 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 situation cannot be reasonably be evaluated as it might involve user interaction to clean up inconsistencies—possibly being performed only days after the network partition.

Figure 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 another time. 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.

Simulation studies [22] have shown that our approach combined with the primary-per-partition protocol (P4) [5] 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.

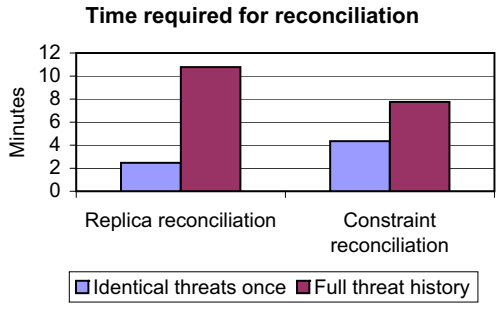

Figure 6. Propagation of missed updates and re-evaluation of consistency threats

7 Related work

Balancing integrity with availability has already been thoroughly investigated with respect to concurrency consistency [3, 12] and replica consistency [8, 17, 24]. The trade-off between constraint consistency and availability, however, is still rather poorly researched. Balzer [2] allows constraint violations temporarily to allow certain business operations to be split into separate steps. This approach 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.

8 Conclusion

In this paper, we showed how decoupled constraint validation can be used to balance the two dependability attributes integrity and availability. Such behavior allows to adapt to node and link failures and hence to improve the overall dependability of a system. Our approach allows decoupling of threatened constraints 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 with a high read-to-write ratio that are able to reconcile the system state without requiring a full history of the degraded mode and allow for flexible application and/or user interaction to clean up the system.

Acknowledgments

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] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

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

[3] 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.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] S. Beyer, M.-C. Bañuls, P. Galdámez, J. Osrael, and F. 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.

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

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

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370, 1985.

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

[10] L. Froihofer, J. Osrael, and K. M. Goeschka. Trading integrity for availability by means of explicit runtime constraints. In *Proc. of the 30th Intl. Conf. on Computer Software and Applications*, volume 2, pages 14–17, 2006.

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

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

[13] H. V. Jagadish and X. 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.

[14] H. Kuenig (ed.). FTNS/EJB system design & first prototype & test report. Technical Report D3.2.1, DeDiSys Consortium (www.dedisys.org), 2006.

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

[16] J. Osrael, L. Froihofer, K. M. Goeschka, S. Beyer, P. Galdámez, and F. D. Muñoz Escoi. 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, April 2006.

[17] C. Pu and A. 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.

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

[19] 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.

[20] R. Smeikal and K. 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.

[21] R. Smeikal and K. M. 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.

[22] D. Szentiványi (ed.). Metrics. Technical Report D1.3.1, DeDiSys Consortium (www.dedisys.org), 2005.

[23] B. Verheecke and R. V. D. 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.

[24] 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.