Trading Integrity for Availability by Means of Explicit Runtime Constraints

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

Data integrity is one of the dependability attributes in data-centric applications. However, applications exist, e.g., safety or mission critical systems, where availability is more important for dependability than strict data integrity. Consequently, in such systems availability can be increased by temporarily relaxing data integrity. Potential inconsistencies are accepted by constraint validation on replicated copies, which are potentially stale in the face of network partitions. Such consistency threats need to be bound and eventually resolved during reconciliation.

The contribution of this paper is a solution approach to this trade-off between availability and integrity by means of explicit runtime-management of data integrity constraints and consistency threats as well as reconciliation support.

1 Introduction

Dependability [1], with two of its attributes being availability (readiness for correct service) and integrity (absence of improper system alterations), is of major importance in today's software systems [14], especially for distributed systems, which are prone to several errors. As failure model for our work, we consider node and link failures, *assuming the crash failure model [5] 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 [7], we initially treat node failures as partitions with only a single node. Whether a node or link failed will be detected afterwards, when the node is reachable again.

Replication [8] is a well-known mechanism to provide fault tolerance for improved availability in case of node and link failures. For example, the primary partition replication protocol [13] allows a single partition (the primary partition) to continue operation while other partitions are blocked. This 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.

Besides *replica consistency* that defines the correct effect of operations on different replicas with respect to a particular replica consistency model, e.g., 1-copy-serializability [4], we distinguish between two further kinds of consistency with respect to data integrity: *Concurrency consistency (isolation)* defines the correctness of data with respect to concurrent, interleaving access to single data items; *constraint consistency* defines the correctness of data with respect to data integrity constraints that stem from the application requirements. Within this paper, we focus on constraint consistency, influenced by replica consistency, as both are affected by node and link failures.

We classify systems into three different categories based on the major design focus for building a system: service-centric, data-centric, and resource-centric systems. Service-centric systems build upon the notion of a service and one major design focus for these systems is to concentrate on composition and interaction. Data-centric systems focus on the data of an application—entity relationship (ER) diagrams or Unified Modeling Language (UML) class diagrams are often used during the design of data-centric applications. Resource-centric applications are mostly concerned about (possibly physical) resources such as CPU power or network bandwidth.

Moreover, we differentiate between tightly-coupled and loosely-coupled systems. Tightly-coupled systems are nowadays typically implemented by using object-oriented technologies, apply Remote Method Invocation (RMI) as the typical communication paradigm and are often deployed in enterprise networks. Loosely-coupled systems are more

heterogeneous in their nature, often apply message passing as the communication paradigm, increasingly use Web services technology, and are intended to be deployed in Internet-scale networks. Within this paper, we focus on the category of tightly-coupled, data-centric applications, implemented by means of object-oriented software engineering.

Our system model has three major states: Healthy system (no failures, no inconsistencies), degraded mode (failures, inconsistencies are possibly introduced), and reconciliation mode (no failures, inconsistencies have to be cleaned up). To limit the degree of inconsistency introduced to the system during the degraded period, we require that software manages constraints and consistency threats explicitly during runtime. This approach requires that integrity constraints are implemented within constraint checking classes, similar to [15], and provide metadata about these constraints. Our middleware afterwards provides the support for runtime-management of constraint consistency in each of the three major system states.

Paper overview. Section 2 provides a short overview of our constraint classifications, the constraint model and introduces the notion of a consistency threat. In Section 3 we describe how the trade-off between availability and integrity is performed. The consequences of our trading—the reconciliation of possible inconsistencies—is described in Section 4. Section 5 gives some insight into the implementation of our approach. We give an overview of related work in Section 6 and conclude our paper and provide an outlook to future work in Section 7.

2 Constraints and consistency threats

The Object Constraint Language (OCL), which is part of the UML specification, already provides the basis for our classification of constraints: preconditions, postconditions and invariant constraints. Preconditions have to be validated before the call to a method will be performed, 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. Here, we distinguish between hard and soft invariant constraints [10]. Hard invariants are validated immediately after a call to a method which might change the state constrained by this specific invariant. We call that invariant an *affected constraint* of the method. Affected soft invariants are checked at the end of a transaction. Similar to an affected constraint, a method that triggers constraint validation is called an *affected method* of the constraint. All objects restricted by a constraint are called *affected objects* of the constraint. Of course, validation of a constraint requires access to all affected objects.

Generally, a data integrity constraint is a predicate on the system state and can either be satisfied or violated. 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 and the arguments in addition to the context object as parameter to the validate method of a constraint.

In a distributed system, the validation of integrity constraints is more complex as constraint validation itself becomes subject to node and link failures. Hence, constraints might be uncheckable if affected objects cannot be reached. If the objects are replicated, we might be able to validate constraints based on backup copies but the result might not be reliable as the corresponding primary copies might already have changed in another partition—the copies used for validation are *possibly stale*. Hence, validation based on primary copies—although not possible in degraded mode—might have produced a different validation result. We call such situations a *consistency threat*.

3 Availability improvements

Following a strict consistency model would require to block or reject operations causing consistency threats in degraded mode. To increase availability, we temporarily accept potential inconsistencies (consistency threats) in the system and hence explicitly trade integrity for availability. The application-specific trade-off is configured by the application developer through the specification of tradeable constraints (may be relaxed during degraded periods) and non-tradeable constraints (critical for correct system operation)—according to an application's requirements. Consistency threats for non-tradeable constraints are automatically rejected with the usual effect that the current operation/transaction is aborted. Operations causing consistency threats for tradeable constraints are subject to a negotiation mechanism to decide whether to accept or reject the consistency threat. Negotiation can either be statically configured during application deployment or dynamically performed through an application-specific callback handle. 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.

Whenever we accept a consistency threat, we 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: (i) if validation of the constraint starts from a context object, we have to store at least an identifier for the context object that is later used as input to the constraint validation method and (ii) if validation of the constraint starts from a set of objects obtained by a query operation, we only have to store the constraint as no input to the validate method is required. These requirements state only the minimum information necessary to re-evaluate constraints. In practice, we allow to enrich the data of a consistency threat with application-specific information. As pre- and post-conditions cannot simply be re-evaluated, we do not store consistency threats for such constraints. Hence, the effects of trading pre- or post-conditions have to be covered by invariant constraints. Consequently, developers should prefer invariant constraints over pre- and post-conditions.

4 Reconciling constraint consistency

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, our middleware takes different actions. If the *constraint is satisfied* and there was no replica conflict (or no replication is used), the middleware removes the threat and all identical threats from the set of accepted consistency threats. If there was a replica conflict for the constraint and a reconciliation instruction[1] of at least one of the identical threats specifies that the application should be informed of this situation, the middleware notifies the application. If the *constraint is violated*, this constraint violation must be resolved—either by middleware-supported rollback to previous states or by asking the application to perform compensating actions. If the *constraint is still threatened*, the reconciliation of the consistency threat has to be postponed until further network partitions are re-unified.

5 Prototype implementation

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 [11]. This management of explicit constraints is performed by the *Constraint Consistency Manager (CCMgr)*, which also triggers the negotiation of new consistency threats and drives the reconciliation of accepted consistency threats.

Our implementation of this system architecture is based on the Enterprise JavaBeans (EJB) platform and integrated into the JBoss application server. EJB uses entity beans to encapsulate the application data. This architecture fits our data-centric, object-oriented approach very well. The data integrity constraints are defined upon these entity beans and are implemented as explicit constraint classes. The constraints of an application are specified in a configuration file which is read when the application is deployed into the application server. The information provided is used to register the constraints appropriately with the constraint repository.

One key requirement for the implementation of our approach with explicit runtime constraints is to be able to intercept the calls to methods and to know the method and the object on which the method was called. This can be achieved by registration of interceptors with the JBoss invocation service and the JBoss Aspect Oriented Programming (AOP) framework. These interceptors further register the CCMgr as transactional resource at the transaction manager to support the notion of soft constraints.

The invocation interceptors notify the CCMgr before and after method invocations to enable appropriate constraint validation. The CCMgr looks up affected constraints from the constraint repository by using different criteria, such as the class of an object/entity bean, 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. Finally, the CCMgr also takes appropriate actions depending on the validation outcome (e.g., abort the current transaction in case of constraint violations).

Detection of consistency threats can also be performed through invocation interception. Directly before the CCMgr triggers the validation of a constraint, it starts to collect entity beans on which invocations are performed and stops this behavior after the validation of the constraint returns. This behaviour is also supported by invocation interceptors. Consequently, the CCMgr knows which entity beans were used for constraint validation and can ask the replication manager (responsible for managing the different replicas of a single "logical" object) for each entity bean, whether the currently used replica was possibly stale. If one of the affected entity beans was possibly stale, a consistency threat occurred and negotiation of this threat is triggered.

After the Group Membership Service (GMS) notifies the replication manager of reunified network partitions, the system starts re-establishment of integrity. For this process, the replication manager propagates missed updates from primary copies to backup copies that were located in other partitions. Replica conflicts are solved based on rollback

[1] Such a reconciliation instruction might be introduced by the application during negotiation of a consistency threat.

or with application callback. Constraint consistency is re-established according to the description provided in Section 4.

6 Related work

The trade-off between consistency and availability has already been investigated with respect to concurrency consistency [3, 9] and replica consistency [6, 12, 16]. The trade-off between constraint consistency and availability, however, is still rather poorly researched. Balzer [2] allows constraint violations temporarily by using pollution markers. Each pollution marker corresponds to an integrity constraint. If the integrity constraint is not satisfied, the corresponding pollution marker is set. At the time the integrity constraint is satisfied again, the pollution marker is removed. The system tolerates inconsistent data in the way that the 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, we do not accept constraint violations and rather aim at fully consistent data in a healthy system. Consequently, the pollution markers are a means to trade integrity within a healthy system while consistency threats are a means to cope with degraded system situations.

7 Conclusion

In this paper, we provided a solution approach to the trade-off between data integrity and availability. The application-specific trade-off configuration is supported through explicit data integrity constraints. The runtime-management of these constraints (validation, detection and proper treatment of consistency threats) is afterwards performed by our middleware enhancement. Hence, our middleware supports this trade-off while the tasks requiring application specific knowledge (specification of constraints, negotiation and reconciliation of consistency threats) have to be performed by the application.

8 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/).

We thank Hubert Künig for many in-depth discussions of our solution approach. We further thank Markus Horehled and Klaus Fuchshofer who contributed the proof-of-concept prototype implementation integrated into the JBoss application server.

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

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.