This is the author's version. The original publication is available at: http://dx.doi.org/10.1109/ARES.2006.19

A System Architecture for Enhanced Availability of Tightly Coupled Distributed Systems

J. Osrael, L. Froihofer, K. M. Goeschka
Vienna University of Technology
Institute of Information Systems
Argentinierstrasse 8/184-1,
1040 Vienna, Austria
{*osrael|froihofer|goeschka*}*@tuwien.ac.at*

S. Beyer, P. Galdámez, F. Muñoz
Universidad Politécnica de Valencia
Instituto Tecnológico de Informática
Camino de Vera,
46022 Valencia, Spain
{*stefan|fmunyoz|pgaldamez*}*@iti.upv.es*

Abstract

We present a system architecture which facilitates enhanced availability of tightly coupled distributed systems by temporarily relaxing constraint consistency.

Three different types of consistency are distinguished in tightly coupled distributed systems - replica consistency, concurrency consistency, and constraint consistency. Constraint consistency defines the correctness of the system with respect to a set of data integrity rules (application defined predicates). Traditional systems either guarantee strong constraint consistency or no constraint consistency at all. However, a class of systems exists, where data integrity can be temporarily relaxed in order to enhance availability, i.e. constraint consistency can be traded against availability. This allows for a context- and situation-specific optimum of availability.

This paper presents the basic concepts of the trading process and the proposed system architecture to enable a fine-grained tuning of the trade-off in tightly coupled distributed systems.

1. Introduction

Distributed systems are of unprecedented interest and importance today, ranging from applications present in daily life such as banking or health care applications to highly specialized distributed systems used in control engineering and air traffic control. Dependability [1] is necessary not only for safety-critical applications but also to cope with the complexity of distribution in general. Replication [2], the process of maintaining several copies of the same entity (e.g. data item, object, process) at different nodes, is used to enhance performance and fault tolerance of distributed systems.

Three different kinds of consistency [3] are distinguished in distributed systems: Replica consistency defines the correctness of replicas; i.e. it is a measure how replicas of the same logical entity differ from each other. Concurrency consistency is a correctness criterion for concurrent access to a particular data item (local isolation), usually employed in the context of transactions. Thirdly, constraint consistency defines the correctness of the system with respect to a set of data integrity rules (application defined predicates). If consistency has to be ensured all the time (e.g. banking applications) - even in the presence of failures - the system becomes (at least partially[1]) unavailable in degraded scenarios (e.g. node or link failures). On the other hand, some applications exist where consistency can be temporarily relaxed in order to achieve higher availability. For instance, in some safety-critical systems (e.g. [5]) or in some control engineering applications (e.g. [6]) availability is more important than consistency.

We focus on tightly coupled distributed systems, i.e. systems where components depend on each other to a high extent. In contrast, interdependencies between components in loosely coupled systems (e.g. Web Services) are generally lower than in tightly coupled systems [7]. Another typical characteristic of tightly coupled systems is that components interact via remote procedure calls (RPC) whereas message or document

[1] Even if a majority partition or more generally - a quorum - exists [4], significant parts of the system become unavailable.

exchange are primary communication means in loosely coupled systems.

In section 3, we present a simple example that shows how constraint consistency (data integrity) can be temporarily relaxed in order to gain improved availability in tightly coupled distributed systems. In section 5 we present our main contribution, a system architecture (called **DeDiSys**) that is targeted to the system model shown in section 4 and allows to configure this trade-off in real-world distributed systems. Finally, in section 6 we show how the core components of our system architecture interact.

2. Related Work

Trading concurrency consistency or replica consistency for increased availability has been investigated in various research projects. For example, replica consistency is addressed in [8] [9] [10] [11] and concurrency consistency in [12] and [13]. The latter systems basically enhance availability by using weaker models than serializability.

Traditional replicated systems either guarantee strong replica consistency or no replica consistency at all. TACT (Tunable Availability and Consistency Trade-offs) [14] fills in the space between by providing a continuous consistency model based on logical consistency units (*conits*). The consistency level of each conit is defined using three application-independent metrics – numerical error, order error, and staleness. For instance, in a replicated bulletin board service, where users can post messages to any replica or retrieve messages from any replica, a conit covers all news messages. Numerical error limits the total number of messages posted system-wide but not seen locally, order error bounds the number of out-of-order messages on a given replica, and staleness limits the delay of messages. The TACT replication system enforces that the specified limits are not exceeded. TACT provides a fine-grained trade-off between replica consistency and availability but does not deal with constraint consistency.

In [8], the application developer can define replica consistency on *Data Objects* using a large set of parameters. Data objects are passive entities that encapsulate data and provide operations on the data but do not - in contrast to objects in our target applications - invoke other objects.

All of the above mentioned projects have one commonality - they either do not deal with constraint consistency or presume strong constraint consistency. Therefore, our specific approach of temporarily relaxing constraint consistency (data integrity) to enhance availability in tightly coupled distributed systems is not researched.

The architecture of the above mentioned systems is either not described at all or rather implicitly (as in the case of TACT). General considerations regarding distributed systems architectures are for example described in [15] but do not address our specific objective of trading consistency against availability. The strength of our approach is to identify coherence of concerns and translate them into system components that compose a novel system architecture for enhanced availability. We believe both researchers and practitioners building dependable distributed systems will benefit from this paper.

3. Relaxing Data Integrity

The correctness criterion for tightly coupled distributed systems are data integrity constraints, such as value constraints, relationship constraints (cardinality, XOR), uniqueness constraints and other predicates [16]. We propose a middleware system architecture (called DeDiSys) that allows to temporarily relax data integrity constraints when node or link failures occur (degraded operation) in order to enhance availability. Full consistency is re-established (reconciliation) after the failures are eventually repaired.

3.1. Constraint Priorities

Not all constraints of an application are of equal importance. Some have to be satisfied at any point in time while others might be relaxed temporarily when failures occur. To allow such flexibility, we provide different constraint classes with respect to the trading of constraint consistency for availability:

- *Critical constraints*: must not be traded.
- *Regular constraints*: are tradeable during degraded operation and full reconciliation support is provided by the DeDiSys middleware.
- *Relaxable constraints*: are tradeable during degradation but full reconciliation is not done by the DeDiSys middleware. We view them as "user enforced constraints" in the way that the application/user has to re-establish consistency when the system becomes healthy again.

3.2. Cardinality Constraint Example

Cardinality constraints restrict the number of associations between classes. We have presented a simple ticket booking system in [17], which consists of

three replicated (database) servers that store information about persons that buy tickets for an event. A customer who wants to buy a ticket interacts with one of the server nodes.
The system is characterized by the following cardinality constraints:

1. A person is allowed to buy no more than three tickets for an event. (C1)

2. A ticket is owned by exactly one person. (C2)

The first constraint is a relaxable one, i.e. it can be temporarily relaxed but the second constraint can never be relaxed - it is a critical constraint.

During normal operations both constraints have to be fulfilled and the servers are always synchronized, i.e. both replica consistency and constraint consistency are maintained. However, when network partitions occur, the servers cannot synchronize their state any more. Traditionally, the system or parts of it would block till the link failure is repaired. However, in order to enhance availability, we temporarily relax C1. For example, assume a person has already bought two tickets for an event before the network split occurs. Our system allows the person to buy a third ticket at each one of the three server nodes, since the constraint is relaxable. The servers cannot communicate due to the network splits; hence, it is possible that the person buys five tickets in total. This means, the constraint is potentially (temporarily) violated during degraded operation.

Full constraint consistency has to be re-established when the network failure is repaired (reconciliation mode) - therefore, the person can only tentatively buy tickets during degraded operations. In the example system this can be achieved, if the tickets are only sent to the customer, when the degradation is repaired. This means, if five tickets have been bought, two of the ticket purchases are rejected during reconciliation and only three are eventually sent to the customer.

4. System Model

Some of the system model parameters presented in this section have been taken and generalized from two real-world distributed object systems that are typical target applications for our proposed system architecture - the Distributed Telecommunication Management System (DTMS) [5] and the Advanced Control System (ACS) [6]. The DTMS is a control and monitoring system for distributed voice communication systems and is used in air traffic control. The ACS is a CORBA-based framework for building complex control systems and is for example currently used in high energy physics facilities.

Based on the two scenarios, we focus on tightly coupled, object-based distributed systems with up to about 30 server nodes and an arbitrary number of client nodes. Server nodes host objects which are replicated to other server nodes in order to achieve fault-tolerance. The system is preconfigured, i.e. the number and name of nodes is known in advance. The average size of objects is about 2 - 5kB, the maximum size is around 100kB. The number of objects ranges from 10 000 to 1 million but not many concurrent operations on objects are performed, i.e. up to 20 objects are accessed per second throughout the system. The read/write ratio of object invocations is about 10:1 and we assume nested object invocations. About 10 percent of object invocations access objects on other nodes.

We assume a partially synchronous distributed system where node clocks are not synchronized, but where the message transmission time can be bound. In this system a group membership service [18] is taken as the basis to monitor the state of each server node, reporting thus about node failures and node recoveries. Since objects have a persistent state, the failure model being adopted is the "pause-crash" model as described in [19].

The underlying network is a local or wide area network, where partitions might occur. However, we assume a network with guaranteed bandwidth, e.g. leased lines with around 2Mbit. Communication services follow the "link" failure model suggested in the "crash+link" failure semantics of [20]. In order to provide link failure semantics for the communication services we use communication protocols that also use the membership services outlined above. The link failure semantics assumes that a communication link only loses messages when it fails, but otherwise it is able to deliver all its messages reliably in FIFO order. This is not difficult to achieve for point-to-point messages (TCP can be a valid solution), but the replication protocols require broadcasts instead of point-to-point messages. Therefore, we need a communication toolkit that provides at least reliable FIFO broadcasts [21].

A relaxed passive replication model is employed. In passive replication [22, 23], which is also known as primary-backup or primary-copy, the so-called primary replica initially processes the request and then sends the updates, or backups, to the other replicas. We relax this for read-only operations, i.e. read-only operations can be served by any secondary replica.

5. System Architecture

5.1. Overview

Figure 1 shows our (platform-independent) middleware system architecture that supports the fine-grained trading between constraint consistency and availability. Interactions between architectural components are denoted using arrows. Some interactions have been omitted to enhance the readability of the figure, e.g. the Transaction Manager communicates with several components. Our middleware extension called **DeDiSys** is built upon a standard operating system, existing middleware like CORBA, EJB, or COM/COM+/.NET, and uses standard point-to-point transport protocols (TCP/UDP). Furthermore DeDiSys uses off-the-shelf stable storage mechanisms (typically a database) and existing persistence solutions (e.g. persistence frameworks). Some other components might also be already provided by the specific middleware (e.g. the Transaction Manager); however, since this is platform-dependent we consider these components explicitly. Applications (business logic) are built on top of DeDiSys.

The core components of DeDiSys are the *Constraint Consistency Manager* and the replication system (consisting of the *Replication Manager* and the *Replication Protocol*). The replication system utilizes the *Group Membership Service*, the *Group Communication* tool and interacts with the *Invocation Service* and the *Naming Service*. Further components are the *Transaction Manager* and the *Activation Service*.

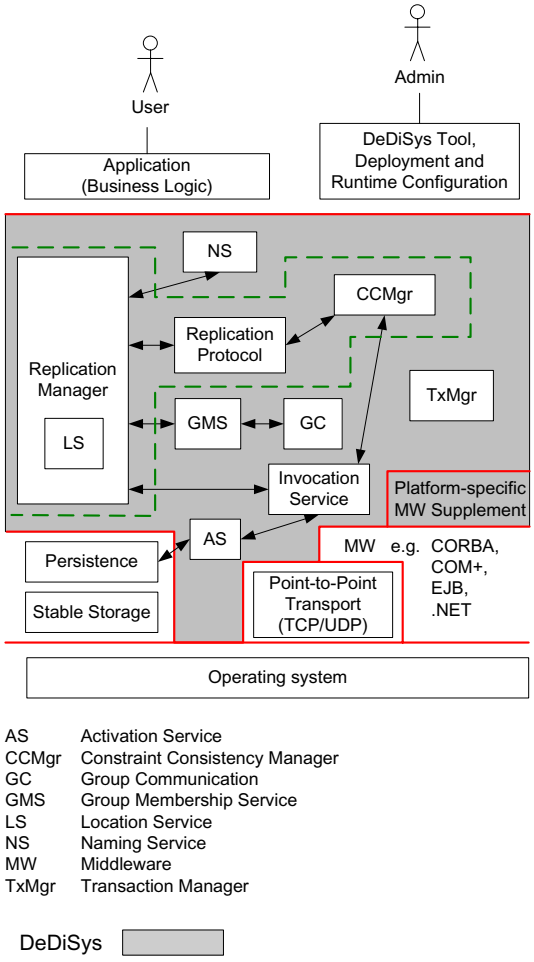

Figure 1. DeDiSys system architecture

5.2. Architectural Components

5.2.1 Replication System

Replication is the primary means to achieve fault-tolerance in our middleware infrastructure. The replication system is divided into two sub-parts: the Replication Manager and the Replication Protocol.

Replication Manager: The Replication Manager keeps track of object replicas in the system. Thus, it maintains a mapping between global object IDs and replica IDs with their location and role (primary or backup replica). It supports different passive replication protocols and performs the following tasks:

- Add or remove replicas for a logical entity
- Change the role of a replica
- Select a given replica according to some criteria: e.g. primary, backup, readable, writable, etc.
- Provide the locations of the replicas of a logical entity

The location service for replicas is closely integrated with the replication manager.

Replication Protocol: We have developed a passive replication protocol that supports the trading of consistency against availability. The key idea is that some operations can be performed in all partitions when network failures occur. In contrast to the traditional primary partition approach [24], which allows only one partition to continue, a new primary is chosen if the original primary copy is not available in a partition. Hence we call it the "primary-per-partition protocol" or simply "P4 protocol" [25].
The protocol consists of three modes: *normal* mode, *degraded* mode, and *reconciliation* mode.

During the normal mode, it behaves similar as a traditional (relaxed) primary-backup protocol. Write operations are forwarded to the primary replica[2] and read operations can be served by any replica. However, the difference is that constraints have to be checked before an operation can proceed. The updates are only propagated to the backup replicas, if the constraints are satisfied; otherwise, the operation is aborted. The constraint checking process is described in detail in [16] and [25].

In degraded mode, i.e. when node and/or link failures are present, write operations cannot be directed to the original primary copy if the primary is not in the partition. The operation is aborted immediately if a critical constraint is affected. If only non-critical constraints (regular, relaxable) are involved, a secondary copy is promoted to a new primary copy (according to some pre-defined strategy) and performs the operation.

Since updates that are not covered by critical constraints are allowed in different partitions (i.e. several primaries can be elected in different partitions), conflicts might occur which have to be resolved during the reconciliation mode. This mode consists of three steps: First, consistency among primary replicas is re-established (e.g. by application interaction). Afterwards, all affected constraints have to be re-evaluated. If some of the relaxable constraints are violated, the application is asked again to restore constraint consistency. Constraint violations concerning regular constraints can be solved automatically without application interaction by means of a roll-back. However, this is more costly since previous versions of objects have to be kept. Finally, all secondary replicas are updated.

We are currently about to refine the algorithm (e.g. by supporting an automatic roll-forward mechanism as alternative to roll-back) and to provide safety and liveness properties [26]. The modes of the primary-per-partition protocol and an analytical comparison with the primary partition model are described in detail in [25]. The analytical study shows that the P4 performs better than the primary partition approach if the number of critical constraints is small, i.e. if constraint consistency can be temporarily relaxed.

5.2.2 Group Membership Service

DeDiSys requires a membership monitor in order to keep track of reachable server nodes in the system. Handler routines are registered with the view-based membership monitor and are called when a view change occurs. A view change can occur when a node fails, partitioning occurs, a node recovers or partitions are re-joined. Further details on the proposed group membership service can be found in [18].

[2]Primary replicas of different objects can be hosted on different nodes.

5.2.3 Group Communication

The group communication component provides multicast support for groups of replicas. FIFO reliable multicast is sufficient for our primary-backup replication protocol.

5.2.4 Constraint Consistency Manager

This component manages constraint consistency of application data based on the definition of data integrity constraints. Currently, static constraints are supported since they are the common case; however, in principle it is possible to consider dynamic constraints as well. (Static constraints restrict the state of an object itself whereas dynamic constraints restrict state changes.) Basically, the functionality of the constraint consistency manager can be divided into three major tasks:

- Ensuring data integrity in a healthy system by triggering constraint validation.
- Triggering negotiation of whether data integrity should be actually traded for individual non-critical (regular, relaxable) constraints in a degraded system.
- Supporting re-establishment of constraint consistency after a node/link-failure was repaired through re-evaluation of constraints that were affected by an availability-consistency trade-off.

5.2.5 Transaction Manager

The Transaction Manager provides support for distributed (possibly nested) transactions.

5.2.6 Naming Service

In literature, naming service and location service are used in various ways, dealing with human readable names, identities and addresses - location-dependent or location-independent. In our case, the naming service resolves names to object identities, while the replication manager - among others - provides the location service for business objects and maps object identities to (location-dependent) addresses. However, in the case of infrastructural objects the naming service resolves their names to local addresses directly, if a naming service is used at all.

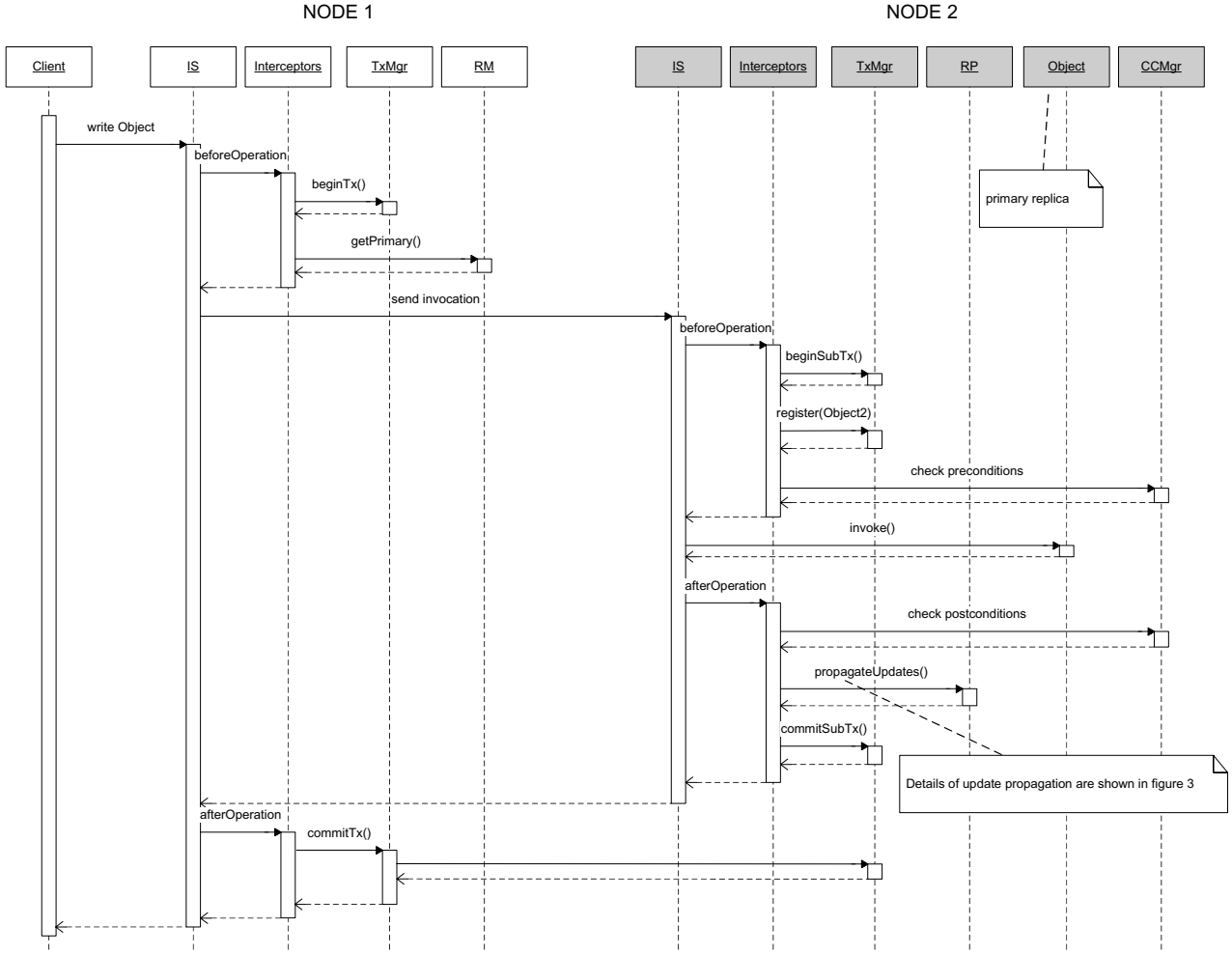

Figure 2. Interaction between DeDiSys core components

5.2.7 Activation Service

The Activation Service, a component local to its node, activates objects based on the serialized/persisted object state.

5.2.8 Invocation Service

The invocation service, another local service, provides the invocation logic used for invocation of methods within the system and provides specific guarantees with respect to node or link failures. It further provides the possibility to intercept object invocations and transmits additional data with an object invocation, e.g. the identifier of a transaction to associate a specific call with a transaction. This, however, is not visible to upper layers.

6. Interaction between DeDiSys Core Components

The replication protocol is triggered by interceptors that are registered with the invocation service (IS). Figure 2 shows a sequence diagram of a non-nested invocation in a healthy system with our replication protocol. For simplicity it is assumed in the diagram that all constraint evaluations are positive. Furthermore, details of the transaction manager (TxMgr), e.g. commit protocol and other interactions between individual transaction manager components, are omitted. When a client invokes a method of an object, an interceptor is executed. This interceptor starts a transaction. Next, the primary copy of the invoked object is obtained from the replication manager (RM). The primary copy's id is installed as the destination for the invocation (this is omitted from the diagram). The invocation is then

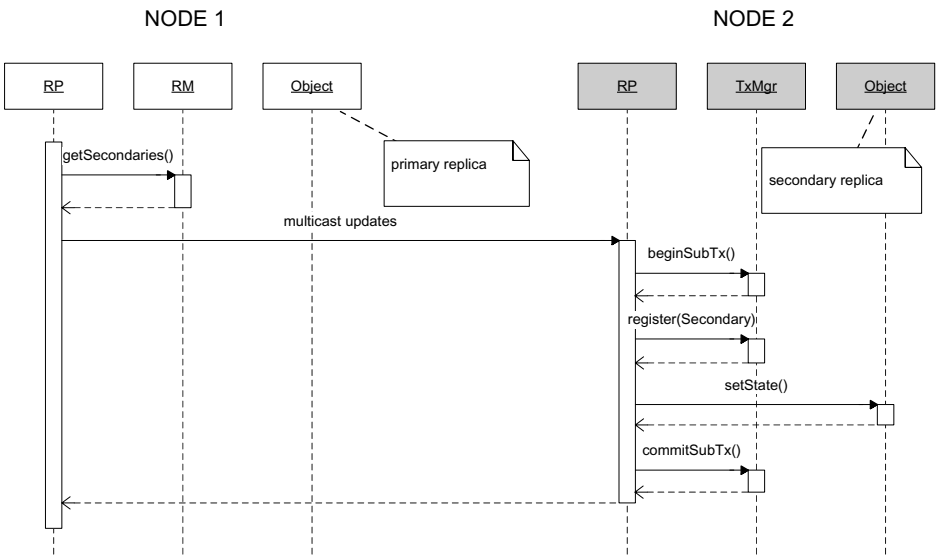

Figure 3. Update propagation

sent to the node hosting the primary copy of the object, where the actual invocation is intercepted again. The interceptor starts a sub-transaction. Before the invocation can occur, precondition constraints have to be evaluated by the constraint consistency manager (CCMgr). If the constraint evaluation is positive, the object operation is invoked. When the call returns, another interceptor is executed, which initiates evaluation of postcondition constraints. The update propagator component of the replication protocol (RP) is called afterwards, if the constraint evaluation is positive. The update propagator gets a list of all secondary copy owner nodes from the replication manager and multicasts the local updates to that list of nodes. On all nodes that receive an update the updates are installed within a sub-transaction (illustrated in Figure 3). Before the interceptor terminates, the sub-transaction is committed. The invocation service returns the result to the client object, where another interceptor starts the commit protocol. Details of the commit protocol are omitted from the diagram.

7. Conclusions and Future Work

Our key idea is to temporarily relax non-critical data integrity constraints when node or link failures arise in order to enhance availability of the system. We have designed a system architecture that supports the trade-off between availability and constraint consistency in tightly coupled distributed systems. Consistency is re-established after the system continues to operate normally. The core components of our proposed architecture are the constraint consistency manager that facilitates the trading and a replication system consisting of a replication manager and specific replication protocols that support this trade-off.

So far we have evaluated our replication protocols and the core components using our Java-based DeDiSys Lite Environment [25]. Future work includes deployment of our system architecture to two industrial systems - the Distributed Telecommunication Management System [5] and the Advanced Control System [6] using EJB, CORBA, and .NET technologies.

8. Acknowledgments

This work has been funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract number 004152).

References

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

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

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

[4] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2), 1979.

[5] R. Smeikal and K.M. Goeschka. Fault-tolerance in a distributed management system: a case study. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 478–483. IEEE CS, 2003.

[6] K. Zagar. Fault tolerance scenarios in control engineering. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy - Issues, Applications, Case Studies*, volume 2, pages 1389–1395. IOS Press, 2005.

[7] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services.* Springer, 2004.

[8] C. Ferdean and M. Makpangou. A generic and flexible model for replica consistency management. In *ICDCIT*, pages 204–209, 2004.

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

[10] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. In *OOPSLA '98: Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 244–258, 1998.

[11] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, and R.E. Schantz. Aqua: An adaptive architecture that provides dependable distributed objects. In *SRDS '98: Proc. of The 17th IEEE Symp. on Reliable Distributed Systems*, page 245, 1998.

[12] 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.

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

[14] 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.

[15] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, 2001.

[16] L. Froihofer (ed.). Trade-off: Availability - consistency. Technical Report D1.1.1, DeDiSys Consortium (www.dedisys.org), 2005.

[17] J. Osrael, L. Froihofer, H. Kuenig, and K.M. Goeschka. Scenarios for increasing availability by relaxing data integrity. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy - Issues, Applications, Case Studies*, volume 2, pages 1396–1403. IOS Press, 2005.

[18] M.C. Bañuls and P. Galdámez. Extended membership problem for open groups: Specification and solution. In M. Daydé et al., editor, *VECPAR 2004: High Performance Computing for Computational Science*, number 3402 in LNCS, pages 288–301. Springer, 2004.

[19] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2), 1991.

[20] F.B. Schneider. Replication management using the state-machine approach. In S.J. Mullender, editor, *Distributed Systems*, chapter 2. ACM Press, Addison-Wesley, 2nd edition.

[21] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S.J. Mullender, editor, *Distributed systems*, chapter 5. ACM Press, Addison-Wesley, 2nd edition.

[22] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In S.J. Mullender, editor, *Distributed systems*, chapter 8. ACM Press, Addison-Wesley, 2nd edition.

[23] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[24] 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.

[25] L. Froihofer (ed.). Ftns system model. Technical Report D2.2.1, DeDiSys Consortium (www.dedisys.org), 2005.

[26] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.