Middleware/Application Interactions to Support Adaptive Dependability

Lorenz Froihofer, Johannes Osrael, 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

Today's software systems often face complex, challenging, and even contradicting requirements that cannot be jointly optimized. In order to achieve satisfying results, the systems have to adapt to changes of context and user needs during runtime. While such adaptivity can be supported by middleware, it typically requires interaction with the specific application to achieve this task. Well-known principles such as callback functions are straightforward to implement in, e.g., distributed object systems, but they are harder to achieve if the end user in front of a Web browser should be involved. Our contribution in this paper is twofold. First, we introduce to our concept of adaptive dependability and provide a solution for the issue of callbacks in Web applications. Second, we contribute with a discussion and summary of several middleware/application interactions (MAI) we applied within our prototype implementations in several industrial settings. Thereby we prove our approach and show how several kinds of MAI mechanisms have to be combined in order to achieve the desired adaptivity. The experienced increase in complexity will have to be addressed by better integration and coordination of different mechanisms.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault Tolerance, Reliability, Availability, and Serviceability; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

General Terms

DESIGN, RELIABILITY

Keywords

Dependability, middleware/application interaction, Web, integrity, availability, constraint consistency, replication

MAI '07, March 20, 2007 Lisbon, Portugal

1. INTRODUCTION

Software systems become more and more complex, integrated, and face challenging if not even contradictory requirements. For example, it is well known that **C**onsistency, **A**vailability, and **P**artition-tolerance (CAP) cannot be optimized independently of each other. The (strong) *CAP principle* [6, 11] provides that *only two of the three requirements can be achieved*, e.g., a system can be available and consistent but not be partition-tolerant. However, the weak CAP principle provides room for system optimizations by viewing the interdependency in a non-binary way: *the stronger guarantees are provided for two of these properties, the weaker guarantees can be provided for the third.*

Obviously, these three properties have to be balanced according to an application's requirements. Moreover, it would be beneficial if the system could adapt to changes of context, requirements, or user needs. In our work we specifically focus on context changes and the associated changes of user needs in the case of node or link failures. We assume the *crash failure model [4] for nodes*—pause-crash for server nodes—and *links may fail by losing some messages but do not duplicate or corrupt messages.* Link failures may subsequently lead to network partitions, effectively splitting a system into parts that are not able to communicate. However, as node and link failures cannot be differentiated at the time when they occur [5], we initially treat node failures as partitions with a single node. Whether a node or link failed is detected later when the node is reachable again.

Failures are threats to dependability [1] and hence to availability and integrity. While failures affecting availability might lead to a non-responsive system, integrity violations may lead to inconsistent data. Within this paper, we focus on *constraint consistency* [9], i.e., consistency of data with respect to data integrity constraints. These constraints stem from an application's requirements, which are typically stated in informal language, such as "There must not be more passengers on an airplane than available seats." for example. We will subsequently call this the *ticket-constraint.*

While the requirement is clear and easy to state, its implementation and maintenance at runtime might be challenging due to other conflicting requirements such as high availability. Replication [12], the process of maintaining several copies (replicas) of the same entity (data item, object), is well-known to provide fault tolerance for improved availability in case of node and link failures. To improve availability even further, write access to replicas in different partitions is desirable in case of network partitions. This, however,

introduces inconsistency into the system, potentially violating the application requirements. To manage such situations, we integrated *constraint consistency management* as a new middleware service [8]. This service primarily uses callback functions to provide control to the application at any time application-specific processing is required. While our target systems initially were distributed object systems, allowing for remote method invocation (RMI) to perform callbacks to clients, we faced challenges for Web-based applications where a callback from the Web server to the Web browser in order to involve the end-user is simply not possible. Consequently, we contribute with a solution to this callback issue. Moreover, we discuss and summarize several middleware/application interaction mechanisms applied to support adaptive dependability within several prototype implementations [2, 8, 13] in diverse industrial settings such as control and information systems.

Paper overview.

We provide an overview of our approach to support adaptive dependability in Section 2 before we contribute with our solution to the callback problem for Web-based systems in Section 3. In Section 4, we discuss our implementation as well as the middleware/application interactions applied to support adaptive dependability. Finally, we provide related work in Section 5 and conclude this paper in Section 6.

2. MIDDLEWARE SUPPORT FOR ADAPTIVE DEPENDABILITY

Our concept of middleware support for adaptive dependability is based on an explicit runtime balancing of the two dependability attributes [1] integrity (consistency) and availability with respect to node and link failures. In order to achieve this goal, we make parts of an application's requirements (the data integrity constraints) explicitly available and processable during runtime. We further decouple constraint validation from the business transactions (typically an atomic set of operations) by partially deferring it to a later point in time. Hence, explicitness and decoupling play a key role to gain the required flexibility for adaptive behaviour.

Generally, a data integrity constraint is a predicate on the system state or state transitions. Within our work, we focus on static constraints, i.e., constraints defined solely on the system state that is, for example, represented by Enterprise JavaBeans (EJB) entity beans. Dynamic constraints defined upon state transitions, sequences of state transitions, or temporal predicates are not within the focus of our work. To make constraints explicitly available at runtime, each constraint is implemented in a separate class according to a pre-defined interface. This interface specifies a validate(...) method that has to return true in case the constraint is satisfied or false if it is violated. While these constraints are straightforward to implement, the management of the constraints and their consistency becomes rather complex in replicated distributed object systems that are subject to node and link failures. In this case, constraint validation is affected by failures as well.

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

The previous example shows that although a constraint is satisfied in degraded mode while node or link failures are present, it might be the case that it is violated afterwards when the system recovers from a previous failure. Consequently, constraint validation is not reliable during degraded mode and we are only able to determine that a constraint is possibly satisfied, possibly violated or even uncheckable if the constraint requires unreachable objects for validation. We call such situations a *consistency threat* [9].

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

Accepted consistency threats are persistently stored by the middleware and processed again during the reconciliation phase in order to evaluate whether a consistency threat actually introduced a constraint violation. If a constraint is violated, e.g., 85 tickets sold for a plane with 80 seats, we use another callback to the application-provided *reconciliation handler* in order to trigger the clean-up of the inconsistencies. The reconciliation handler might clean up the inconsistencies immediately, e.g., automatically or by blocking as long as the human operator is working on the clean-up, or return before the inconsistency is solved by providing this fact to the middleware. Differentiation between these two types (immediate vs. deferred reconciliation) works via the return parameter of the callback. If the application returns true, it specifies that the constraint violation is resolved. In this case, the middleware will revalidate the constraint and remove the threat if the inconsistency has actually be solved. Otherwise, it will contact the reconciliation handler again. If the application returns false, the reconciliation of the constraint violation will be performed at some time later under the application's responsibility. In this second case, the reconciliation handler might apply asynchronous message passing within the system or send an e-mail notification to the system operator. However, the cleanup of the threat by the application is detected by the middleware through the fact that the corresponding constraint is satis-

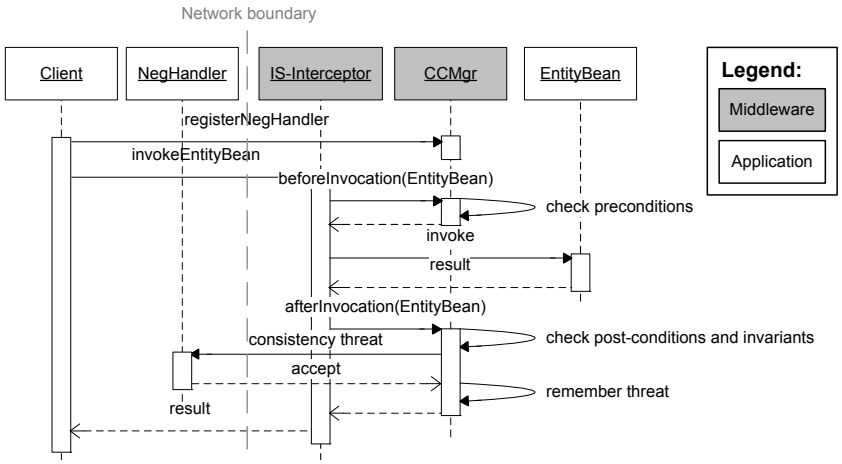

Figure 1: Consistency threat negotiation in distributed object systems

fied by a business operation. Subsequently, it will remove the consistency threat from persistent storage.

3. CALLBACKS IN WEB APPLICATIONS

Before we discuss the issue of callbacks in Web applications, we will revisit the negotiation callback scenario within our middleware for distributed object systems as illustrated in Figure 1. At some time before a client makes a call to an entity bean, it registers a consistency threat negotiation handler with the constraint consistency manager (CCMgr). Calls to components in EJB and hence entity beans are made through the invocation service. Several interceptors responsible for different tasks can be registered with this invocation service. One interceptor responsible for constraint consistency management notifies the CCMgr before and after method invocations in order to allow the validation of constraints affected by the current operation. If the system operates in degraded mode, consistency threats may arise from these constraint validations. These consistency threats have to be negotiated by contacting the registered negotiation handler, potentially crossing a network boundary between the server and the client.

While such behaviour is possible and straight-forward to implement in distributed object systems communicating via remote method invocation (RMI), it is more challenging in Web-based systems because a callback to the Web browser is simply not possible, see also first part of Figure 2. Moreover, the hypertext transfer protocol (HTTP) is based on a strict request/response behaviour. The business request is sent to the Web server and the browser is waiting for the response. Before the response is actually available, negotiation of potential consistency threats has to be performed, i.e., requests from the middleware to the application are necessary. Consequently, some logic has to be placed into the Web application to match this request/response discrepancy.

When a negotiation request arrives from the middleware, the negotiation logic in the Web application has to extract the information provided to the negotiation handler, examine the situation, provide an appropriate response to the Web browser and block the thread by which the negotiation request was received. Effectively, this forwards the negotiation callback request via the HTTP response for the business

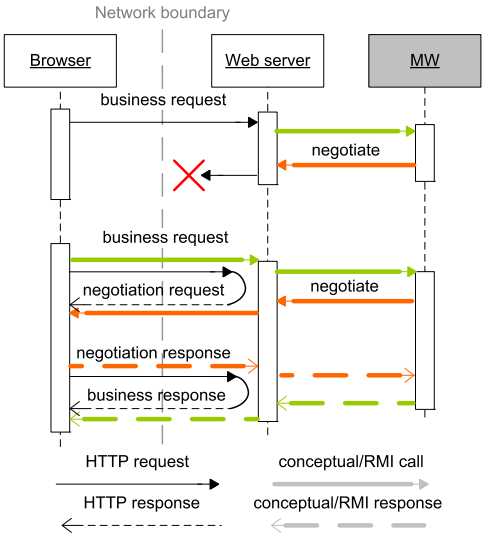

Figure 2: Different request/response behaviour of distributed object-based and Web-based systems

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

This request/response discrepancy and its solution are shown in Figure 2, where the conceptual requests and responses as they could be performed with RMI are drawn with thick grey lines[1] and the HTTP requests/responses are drawn with thin black lines. Figure 2 first illustrates that a callback to the browser from the Web server is not possible in the way of straightforward calls. Thereafter, it shows the solution where the negotiation request is transferred over the HTTP response for the business request and the negotiation response is transferred back to the Web server via a new HTTP request. Finally, the business response is transferred back to the browser via the HTTP response for the HTTP request with the negotiation response.

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

We ensured the practical feasibility of our approach with an implementation of these concepts in a flight booking prototype. This application builds upon the Struts Web framework (`http://struts.apache.org/`) and an EJB application that is deployed within a JBoss application server (`http://www.jboss.org/`) enhanced with support for replication and constraint consistency management [9].

4. DISCUSSION

The previous sections introduced to the problem and the usage of callbacks to achieve a balancing of the two dependability attributes integrity and availability. As a proof of concept, prototype studies have been performed within industrial settings in strong cooperation with companies from communications and control engineering industry within the DeDiSys project (`http://www.dedisys.org/`). This section provides further details on middleware/application interaction within these prototypes as well as a discussion of specific design alternatives we investigated during the prototype studies [2, 8, 13].

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

Besides the explicit interaction through predefined interfaces we use metadata and invocation interception as programming abstractions for coordination of and implicit interaction between application and middleware. The metadata about constraints is provided by the application developer and includes information such as when to check a specific constraint or whether the constraint can be relaxed (potentially be violated) during degraded mode or reconciliation in order to enhance availability. The constraints themselves are implemented by the application developer. However, triggering constraint validation and management of consistency threats and associated data is performed by the middleware according to the metadata specifications. Therefore, the middleware provides support for the balancing between availability and integrity but the actual balancing is achieved through a combination of services and artifacts provided by the middleware (CCMgr, replication, etc.) and the application (constraints, metadata, etc.).

Invocation interception is a well-known mechanism to provide middleware services to an application and available in several middleware technologies, such as EJB, CORBA, or .NET. However, middleware traditionally is a layer between the presentation and the resource layers and hosts the application. Consequently, the middleware or the middleware services are primarily triggered in case of remote invocations. While EJB already triggers middleware services for local invocations, this is only the case for invocations made upon an interface but not the actual implementation of a bean itself. Therefore, method calls of a single instance to itself are plain Java invocations. Unfortunately, constraints can be triggered by an arbitrary method, not only by methods invoked via remote method invocation (RMI) or via the bean interface. Hence, integration of constraint consistency management as a middleware service requires the possibility to intercept each and every method of an application. We use Aspect-Oriented Programming (AOP) to satisfy this requirement—thereby complementing the traditional mechanisms for invocation interception and introducing another kind of application interception by the middleware.

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

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

While there are no reasonable alternatives for invocation interception (including AOP) or persistence, we evaluated alternatives for the explicit callbacks between middleware and application. The negotiation of consistency threats is a synchronous/blocking task, i.e., an operation/transaction cannot continue and especially not commit successfully as

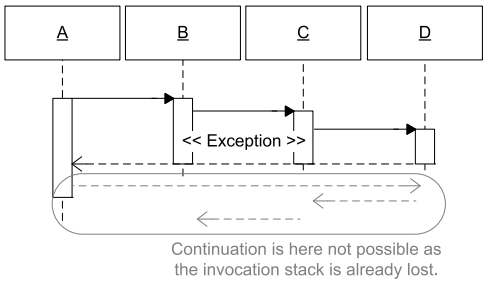

Figure 3: Exceptions break the flow of control

long as there is no decision of whether to accept or reject a specific threat. However, negotiation can be performed immediately when a threat occurs or be deferred until the end of a transaction. In any case, we should be able to continue and commit the transaction if all threats are accepted. Due to this behaviour, a callback is most appropriate to achieve the negotiation task.

However, an alternative to the callback for negotiation of consistency threats would be to throw an exception to indicate the consistency threat. The application would have to investigate the exception details and retry the operation by signalling the middleware to accept the threat(s). This has the drawback that the threat that occurs the second time, e.g., 85 out of 80 tickets booked, might be different than the first time, e.g., 75 out of 80 tickets booked, but might be accepted due to the decision of the application. Unfortunately, continuing the operation at the point the exception occurred is generally not possible, as the invocation stack is already lost. This is especially true for nested invocations and illustrated in Figure 3 where A would have to call B, B call C, and C call D again after the exception indicating a consistency threat occurred at some point in the method of D. Other mechanisms such as asynchronous calls or message passing are not useful for threat negotiation either due to the blocking behaviour of negotiation. The only advantage of asynchronous behaviour would be for long-lasting transactions where deferred negotiation is possible. In this case, negotiation of threats could take place in parallel while the transaction continues with the assumption that all threats will be accepted. Of course, the transaction has to block before commit until the decisions for all occurred threats are available.

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

Table 1: Middleware/Application interactions

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

Table 1 summarizes the middleware/application interaction mechanisms within our system. Callbacks, exceptions, asynchronous behaviour, and interaction via persistence are interactions between middleware and application that an application developer has to explicitly address or use. Invocation interception is used only implicitly, i.e., transparently to the application developer, to achieve middleware tasks. Finally, the usage of metadata allows an application-specific configuration of the middleware.

5. RELATED WORK

Geihs et al. [10] address adaption of component-based distributed applications based on the model-driven paradigm. The adaptability of an application is specified within the model with the goal to provide the best possible service to the user according to context and user preferences. Based on this model, application and middleware artifacts can be generated. While our current systems do not apply model-driven elements, we are investigating this approach to support the application developer with code generation for integrity constraints similar to Verheecke et al. [14] and generation of the corresponding metadata. Related work with respect to the trade-off between availability and integrity/-consistency is further discussed in [8].

The usage of metadata and reflection is also proposed by Capra et al. [3] as a concept to address mobility. In their approach, metadata is specified by the application, but is managed by the middleware. Consequently, the middleware adapts to changes in the context, e.g., bandwidth, battery power, network connection/connectivity, etc. according to the metadata—also called an application profile. We support this argument to use metadata. However, we also see metadata as a promising approach to master the increasing complexity in todays systems.

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

6. CONCLUSION

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

Based on the experience of prototype implementations in industrial settings we require middleware/application interaction to be performed in a configurable and flexible way. This is strongly supported by middleware architectures that allow application-specific configuration and behaviour based on metadata and application-provided middleware "plug-ins". Therefore, future research should not only concentrate on new mechanisms of middleware/application interaction but also on how to build/architect middleware and how to concert the different interaction mechanisms in order to reduce complexity and support middleware/application interaction in a flexible and application-specific way. A key for efficient interaction between middleware and application is coherence of concern, achieved through decoupling, localization, and explicitness of particular policies and properties. The resulting performance impairments are typically offset by the gain in flexibility [7]—at least after a while when the technologies mature.

Acknowledgments

We thank Hubert Künig for many in-depth discussions and Bernhard Rieder who strongly contributed to the prototype for Web callbacks. This work has been partially funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, `http://www.dedisys.org/`, contract 004152).

7. 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] S. Beyer, P. Galdámez, and F. D. Muñoz Escoi. Implementing network partition-aware fault-tolerant CORBA systems. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security*. IEEE CS, 2007.

[3] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, volume 2192 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2001.

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

[5] 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.

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

[7] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In *Proc. 29th Int. Conf. on Software Engineering*. IEEE CS, 2007.

[8] 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*, 2006.

[9] L. Froihofer, J. Osrael, and K. M. Goeschka. Decoupling constraint validation from business activities to improve dependability in distributed object systems. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security*. IEEE CS, 2007.

[10] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 718–722. ACM Press, 2006.

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

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

[13] J. Osrael, L. Froihofer, G. Stoifl, L. Weigl, K. Zagar, I. Habjan, and K. M. Goeschka. Using replication to build highly available .NET applications. In *Workshop Proc. of the 17th Int. Conf. on Database and Expert Systems Applications*, pages 385–389. IEEE CS, 2006.

[14] 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.