# DeDiSys Lite: An Environment for Evaluating Replication Protocols in Partitionable Distributed Object Systems[*]

Stefan Beyer, Alexander Sánchez, Francesc D. Muñoz-Escoí and Pablo Galdámez
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
Spain
{stefan, asanchez, fmunyoz, pgaldamez}@iti.upv.es

## Abstract

*Distributed object systems for partitionable systems present a challenge, in that there is a trade-off between availability and consistency. Changes in one partition are not visible in another partition. Therefore, if strong consistency is required, certain operations cannot be permitted. This reduces availability. In the DeDiSys project we aim at allowing this trade-off between consistency and availability to be configurable.*

*The DeDiSys distributed object system relies heavily on replication protocols that allow high-availability, whilst ensuring a level of consistency that is required by a particular application. We have developed DeDiSys Lite, a prototype of the DeDiSys system, which provides a platform to implement and evaluate these replication protocols.*

*Infrastructure components are provided in a minimal implementation. Configuration files allow system parameters, such as the degree of replication or the nesting of object invocations, to be modified, without having to adapt application code.*

*We use DeDiSys Lite as both a simulation environment for the development of new replication protocols and as a basis for the continuous development of the DeDiSys system. Some results obtained using the platform to optimise a new replication protocol are presented in this paper.*

## 1 Introduction

Distributed systems, especially those deployed on a wide area network, are vulnerable to network failure. Network link failures can cause such systems to partition. Replication can be used as a means to limit the reduction of availability that occurs in a partitioned distributed system. However, replicas can diverge, as replica synchronisation is not always possible in a partitioned system. For applications which require strong consistency, this divergence of replica states might be unacceptable. In such applications certain operations must not be permitted during partitioning, which in turn reduces availability. Therefore, there is a trade-off between consistency and availability in partitionable distributed systems.

In the DeDiSys distributed object system we allow this trade-off to be configured. That is, we allow data consistency to be temporarily relaxed, according to rules specified by the application. We base these rules on data integrity constraints [13].

In this paper we introduce DeDiSys Lite, a simplified prototype implementation of the DeDiSys architecture. DeDiSys Lite provides a platform for implementing and evaluating different replication protocols. Lightweight architecture components allow objects with a simple read/write interface to invoke each other. The architecture components are configured through configuration files. These files allow system parameters, such as the location and number of replicas of a specific object and the nesting of object invocations to be specified easily.

The main contribution of this work is a simplified partitionable environment that serves as a test bed for implementing and evaluating replication protocols. The design of the architecture facilitates the implementation of new protocols. Furthermore, we demonstrate the simplicity with which the environment can be configured, in order to allow flexible protocol evaluation. Finally, we show how DeDiSys Lite has been used to optimise a new replication protocol [2] that has been designed for the DeDiSys project.

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

The remainder of this paper is organised as follows: Section 2 introduces the DeDiSys project. Section 3 compares DeDiSys to related work. The proposed architecture of DeDiSys Lite, the prototype implementation of the DeDiSys system, is introduced in section 4, whereas section 5 explains how the architecture can be used to implement and evaluate replication protocols. In section 6 we demonstrate, how we have started to evaluate a new replication protocol and provide some first results. The concept of using the architecture for the implementation of replication protocols is widened to other system components in section 7 and in section 8 we outline our plans for future uses of DeDiSys Lite, before concluding in section 9.

## 2 The DeDiSys Approach

DeDiSys is aimed at partitionable environments. Distributed objects are located on a known set of server nodes, which are connected by a wide area network. In contrast to other systems, we support nested invocations. That is, methods of one object might invoke methods of other objects in the same transactional context.

A partially synchronous system model is used. In this model clocks are not synchronised, but message time can be bound. As a failure model, the "pause-crash model" [4] is used for node failures, and the "link failure model" [12] for communication services. As we cannot distinguish between a failed node and an isolated node until recovery time, every failure is treated as partitioning.

Objects are replicated, in order to improve availability. Replication is partial. That is, an object does not have to be replicated on every node. This has the advantage of having to maintain consistency between fewer replicas, which in turn should reduce communication costs. On the other hand, there is a risk of not having access to any replica of an object during network partitioning in partial replication. This risk can be reduced by placing replicas carefully, taking into account network links that are more likely to fail.

We employ a relaxed *passive replication* model. In passive replication [3] [7] requests are only processed by one *primary copy*. Updates are then propagated to the *secondary copies*. However, we relax this for read-only operations. Read-only operations can be served by any secondary copy. The passive model lends itself to a system in which consistency is to be configured, as it allows variations in the way updates are propagated. If *synchronous* update propagation is used, a primary copy must propagate any updates immediately; that is, before the result of the operation that has caused the update is returned to the client. In *asynchronous* update propagation the result is returned and the propagation of state changes performed some time later.

In DeDiSys consistency is based on integrity constraints. Each constraint is associated with an operation and defined over parts of the state of one or more objects. The system is said to be fully constraint consistent, if only changes to objects caused by operations whose constraints have been met are accepted. During partitioning the primary copy of an object might not be available for constraint evaluation and available secondary copies might be stale.

We allow consistency to be temporarily relaxed for certain constraints. This is achieved by a classification of integrity constraints into three categories:

- **Critical constraints** always have to be evaluated on up-to-date replicas. Write operations which are covered by a critical constraint are only permitted, if all replicas used for constraint evaluation are known to be up-to-date.
- **Regular constraints** can be evaluated on possibly stale copies. They are re-evaluated at reconciliation time. If a constraint is not met at reconciliation time a consistency conflict has occurred. Consistency conflicts caused by a violated regular constraint are removed automatically.
- **Relaxed constraints** can also be evaluated on stale copies. They also have to be re-evaluated at reconciliation time. However, dealing with consistency conflicts is left to the application.

The DeDiSys approach allows a variety of replication and reconciliation protocols to be used. DeDiSys Lite serves as a convenient test bed for developing and evaluating these protocols.

## 3 Related Work

The trade-off between consistency and availability has been investigated in the context of other distributed systems. These systems generally require the application programmer to specify the required consistency or the required availability.

The authors of [14] use consistency units (conits) to specify the bounds on allowed inconsistency. A conit is a set of three values representing "numerical error", "order error" and "staleness". Numerical error defines a weight of writes on a conit that can be applied to all replicas, before update propagation has to occur. Order error defines the number of outstanding write operations that are subject to re-ordering on a single conit. Finally, staleness defines the time update propagation can be delayed. Conits are associated to elements of a replicated data store, which can range from a simple file-system to a persistent object system. Operations on replicas are restricted to primitive read and write operations. The system does not support dynamic objects with nested invocations. Furthermore, the system does not deal with partitioning.

In CoRe [6] the principle of specifying consistency is extended to allow the programmer to define consistency using a larger set of parameters. A set of concerns is identified and arranged in a hierarchy. To each of these concerns a variety of possible parameters are attached. CoRe forces the application programmer to be aware of these concerns and adjust a large set of parameters. Furthermore, the system only focuses on data objects; that is, objects that do not cause invocations to other objects.

AQua [5] approaches the trade-off from the other direction by allowing availability requirements to be specified. In AQua "quality objects" are used to specify quality of service requirements. In this case, the quality objects are used to describe availability constraints. A contract defines the required availability, whereas "system condition objects" are used to monitor the current availability in the system. The system uses a general object model, in which any object may act as a client and invoke operations in other objects. AQua considers crash failures, value faults and time faults, but does not consider partitioning.

None of the distributed systems described above considers constraint consistency.

## 4 The DeDiSys Lite Architecture

### 4.1 Overview

DeDiSys Lite is a Java prototype implementation of the DeDiSys system. The prototype models a distributed object system in a simplified way. Objects do not implement real code, but provide an interface that allows to simulate read and write invocations. Nested operations are supported by specifying the nesting in a configuration file.

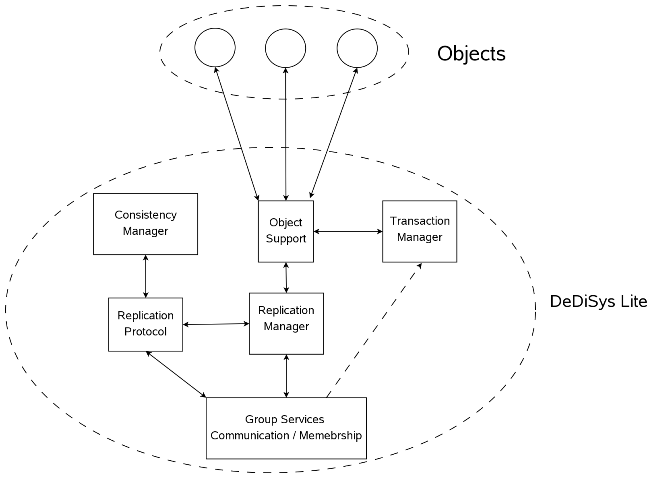

**Figure 1. System Overview**

Figure 1 shows an overview of the system components that are present on each node.

As can be seen, the application consists of a number of objects, which use the following system components to invoke each other:

- **Object Support (OS)** This component is an invocation service. It exchanges invocation request and result messages between nodes.
- **Consistency Manager (CM)** The CM simulates a component that is invoked to evaluate data integrity constraints.
- **Transaction Manager (TM)** Transactions are simulated by the TM component. The nested transaction model [10] is used, although no real commit protocol is implemented.
- **Replication Manager (RM)** The RM provides a mapping between object identifiers and the implementations of the objects they refer to, in the form of primary and secondary replicas.
- **Replication Protocol (RP)** This component allows the implementation of different replication and reconciliation protocols.
- **Group Services** The group services consist of a membership service (MS) and a group communication service (GC). The MS keeps track of connected nodes. It provides a view of the nodes in the current partition. The GC provides reliable group multicast. It is mainly used to propagate updates to secondary replicas. It can be configured to provide different delivery guarantees, depending on the requirements of the replication protocol. The replication model employed in the DeDiSys approach requires FIFO reliable multicast.

### 4.2 Objects

Objects in DeDiSys Lite provide a read operation and a write operation. The state of an object is an integer field. A write operation increments this state field. Furthermore, a `getState()` method and a `setState()` method allow the replication protocol to capture an object's state, send it over a network and install it in another replica of the same object.

Having such simple objects has the advantage that during experiments object access times should remain short and constant. This allows measurements to focus on the performance of the distributed algorithms of the system, rather than being "diluted" by the execution times of code in individual object implementations.

Nested invocations are simulated in the object support component described below. It is possible to define which other objects should be invoked during an object's invocation.

### 4.3 Object Support

Objects use the OS component to invoke each other. The OS is basically a simple invocation service. It provides routines to invoke read and write operations of other objects and to direct incoming read and write messages to the corresponding local object implementations. The OS is a multithreaded component. For each invocation a new thread is started which waits for the invocation result to be received. As objects do not implement any real code, a configuration file in the OS is used to specify the required nesting of object invocations. The format of the configuration file allows to vary the ratio of read and write operations easily. The following listing is an example of such a configuration file:

```
nodes = 2

o2.w = 2
o2.w.oper1 = o3.w
o2.w.oper2 = o5.r
```

The `nodes` directive specifies the number of nodes in the system. The next line signifies that a write operation on object `o2` causes two nested write operations. The next two lines specify these nested write operations as a write operation on object `o3` and a read operation on object `o5`.

The final DeDiSys system will be implemented in three different middleware systems. Hence, the invocation functionality of the OS will be implemented in CORBA, Enterprise Java Beans or .NET, depending on the particular implementation.

### 4.4 Consistency Manager

The CM is used to simulate constraint evaluation. No real constraints are supported, but it is possible to specify objects covered by a constraint. Such a simulated constraint is associated with a constraint type (critical, regular or relaxed) and a set of rules that define how the constraint should evaluate in consecutive evaluation requests.

The following example of a CM configuration file demonstrates how constraint evaluation can be defined:

```
c1.priority = critical
c1.type = pre-condition
c1.objects = o1, o2 ,o3
c1.true = 3
c1.false = 4
c1.number_ratio = 3
c1.percent = 0.4
```

The constraint `c1` is a critical constraint. Furthermore, it is a pre-condition. That is, the constraint has to be met before the object can be written to. `c1` covers the objects `o1`, `o2` and `o3`. It has to be evaluated before the invocation of a write operation on any of these three objects. In the DeDiSys approach constraints are associated with individual object methods, rather than whole sets of objects, but for the implementation and evaluation of replication protocols, associating constraints with sets of objects simplifies the configuration files and therefore allows experiments to be carried out more efficiently.

The final four lines of the above configuration file define the way the constraint evaluates in consecutive evaluations. First, a constraint evaluates as many times successfully as specified in the `true` field. Next, the constraint evaluation fails as many times as specified in the `false` field. Finally, the `number_ratio` field defines the number of times the constraint should evaluate according to a success ratio specified in the `percent` field. The succession of evaluation results is then re-started. That is, the last four lines of the configuration file provide an iteration of constraint evaluation results. This way of specifying the results of simulated constraint evaluation provides a large range of possible configurations in a very simple format. By setting, for instance, the `true` and `false` fields to 0 and the `percent` field to 0.5, a purely random constraint evaluation can be achieved.

### 4.5 Transaction Manager

The TM simulates nested transactions. It has an interface that provides `create`, `abort` and `commit` primitives. Rollback is fully implemented. When a transaction is aborted, modified object states are reset to the state at the beginning of the transaction. However, no commit protocol is implemented. When a transaction commits, the datastructures related to that transaction are removed, but no voting or locking protocol is executed. Thus we provide the atomicity and durability properties of a transaction, but not isolation and consistency. However, due to the nature of DeDiSys Lite operations and the concurrency control embedded in the object support, these two missing properties can be assumed to be provided by the system.

In order to simulate different abortion rates the TM can be configured through a configuration file similar to that of the consistency manager. The following is an example of such a TM configuration file:

```
commit = 3
abort = 4
number_ratio = 3
percent = 0.4
```

As in constraint evaluation the file defines the sequence in which a simulated commit protocol should lead to transaction being committed or aborted. In the above example the first three commit attempts succeed. This is followed by four commit failures. Finally, the next three commit attempts succeed with a success ratio of 40%. This sequence is then repeated.

### 4.6 Replication Manager

The RM provides a mapping between object ids and replicas. It maintains lists of the locations of the primary and secondary copies of each object. An interface is provided that allows the OS to discover the primary copy of an object. An additional interface visible to the replication protocol allows all secondary copies to be discovered for update propagation. Furthermore, the role of a replica can be changed. Optimistic replication protocols can use this feature of the RM to promote secondary copies to temporary primary copies, when the real primary copy is not available.

The RM is configured through a configuration file of the following format:

```
nodes = 3
objs = 2

o1.backups = 0
o1.primary = #proc1#host1

o2.backups = 2
o2.primary = #proc2#host2
o2.b1 = #proc3#host3
o2.b2 = #proc1#host1
```

The configuration file is the same on all nodes. The first two lines indicate the total number of nodes and objects in the system. The statement `o1.backups = 0` indicates that object `o1` is not replicated. The next line provides the node name on which the single copy is located. Note, that the node name is made up of a process name and a host name. One of the simplifications of DeDiSys Lite is that a process is equivalent to a node. That is, each process hosts all the system components. This simplification allows various nodes to be simulated on a single machine. This enables the testing of new replication protocols without having to deploy them on a real network. Object `o2` has two secondary copies in addition to the primary copy. The location of these replicas is specified in the last two lines. At system startup all the replicas are automatically deployed. This allows the degree of replication and location of replication to be modified easily between experiments, without having to change any application code.

In addition to replica creation through the configuration file the RM interface allows the replication protocol to add or remove replicas. This allows protocols to dynamically modify the location and number of replica. A protocol, might for instance monitor access patterns and create a local replica on nodes from which an object is frequently accessed.

The RM uses the group services described below to keep track of which replicas on which nodes are reachable.

### 4.7 Replication Protocol

The RP component is the place where different replication protocols can be plugged in easily. Different update propagation and reconciliation policies can be implemented. The configuration files of the above components can be modified, in order to simulate different system behaviour. This allows to measure the performance and availability of replication protocols in different scenarios. Default behaviour of some common interface functions is implemented in a `ReplicationProtocol` class, which can be extended. Section 5 describes in detail how new replication protocols can be implemented easily.

### 4.8 Group Services

The membership service provides a common view of nodes that are believed to be reachable within the local partition. It notifies the RP and the RM of membership view changes. It also interacts closely with the group communication service described below. The group communication service provides reliable group multicast primitives with configurable delivery guarantees. Replication protocols can use these primitives to synchronise replicas.

We currently use Spread [1] for group services, as it provides the extended virtual synchrony model [9]. This model simplifies the reconciliation process of potential replication protocols, as nodes are aware which views have been installed in re-joining partitions.

## 5 Implementing and Evaluating Replication Protocols

### 5.1 Overview

The main purpose of DeDiSys Lite is to easily implement and evaluate new replication protocols. In order to implement a new replication protocol for DeDiSys Lite, protocol writers should extend the `ReplicationProtocol` class. Default implementations of common protocol methods are provided. For example, a simple update propagator is already implemented. If a protocol wishes to change the way object updates are propagated to secondary copies, it can simply overwrite a method. The timing of update propagation can be determined by the point in the execution of the protocol at which the method is called. Possible alternatives are for example after each invocation, at the end of a transaction or at some later point.

Every protocol needs to implement a certain interface. There are two entry points for the object support. The method `preWrite()` is called by the OS before each invocation and the method `postWrite()` after each invocation. Furthermore, a constructor that takes a reference to the local object support, replication manager, consistency manager and group communication components is required. A `handleMessage()` method has to be implemented. The OS, which is responsible for receiving incoming messages passes all messages of type `REPMSG` to this method. A default implementation in the `ReplicationProtocol` super-class handles messages that contain replica state updates. Finally, a `reconcile()` method is required so that the OS can inform the RP of membership messages that indicate the re-joining of unreachable nodes.

Replication protocols will typically distinguish three system modes. In **normal mode** all nodes are reachable and the replication protocol just has to propagate updates to secondary copies. When a node fails or the system is partitioned, **degraded mode** is entered. In this mode, the replication protocol will have to keep track of updates that might be missed by replicas on unreachable nodes. When partitions are re-merged or a crashed node re-joins the system, missed updates must be propagated and possible conflicts removed in a **reconciliation** phase. In each of these three modes a variety of policies can be implemented. The choice of policies affects the overall time and space efficiency and the availability of the system. Replication protocols can distinguish between the three system modes by registering a view change handler with the membership service.

### 5.2 Example Protocols

We are currently in the process of evaluating two protocols with different policies. We are planning to gain detailed measurements on the availability these protocols provide in the DeDiSys system and application model. The results will be compared with a theoretical study we have performed [2].

#### 5.2.1 The Primary Partition Protocol

In the primary partition protocol [11] only a majority partition is allowed to proceed in the case of failure. A majority partition is a partition that contains at least one more than half the nodes in the system. Updates to objects in other partitions are not allowed. In the case that no partition has a majority of nodes, the whole system is prevented from accepting update operations.

#### 5.2.2 The Primary per Partition Protocol

The primary per partition protocol (P4) [2] is a protocol we have developed for the DeDiSys system and application model. The P4 allows objects in all partitions to be updated. If a primary copy is not available a secondary copy is chosen and promoted to a temporary primary copy. Conflicts can be dealt with automatically or can be left to the application.

The protocol should provide higher availability than the primary partition protocol in the type of applications at which DeDiSys is aimed. The higher availability has been demonstrated mathematically in our theoretical study mentioned above. Probability based availability models have been used to compare the two models within the parameters of an example application. However, there are many replication policy variations, which are difficult to model, that can have a big impact on the performance or on the availability. For instance, propagating updates at the end of a transaction, instead of at the end of every invocation, could theoretically have a performance impact. DeDiSys Lite allows us to compare these two policies easily.

The P4 has been implemented for DeDiSys Lite and we have obtained some preliminary performance results, that will allow us to fine tune the protocol. These results are presented in the following section.

## 6 Preliminary Experiments

### 6.1 Experimental Setup

In order to optimise the P4 protocol we have run two simple performance experiments. These experiments do not represent a complete evaluation of the P4 protocol. They merely serve as a demonstration of the power of DeDiSys Lite as an evaluation tool. All tests were run on a cluster of four nodes. Each node contained a Pentium 4 2.8 GHz CPU and 1GByte of RAM. The nodes were interconnected through a 1 GBit/s Ethernet network. DeDiSys Lite was executed in Sun's Java Virtual Machine version 1.5 on Linux with kernel version 2.4.22. One of the nodes acted as a client, whereas the other three nodes acted as servers, hosting 10 objects which were replicated on each host.

### 6.2 Overhead of Degraded Mode

The first experiment was designed to measure the extra cost of the *degraded mode* of the P4 protocol. In degraded mode, the constraint evaluation process is more complicated. It has to be established whether accessible replicas of objects participating in a constraint might be stale. If there are stale copies and the constraint is critical the operation is

| | |
|---|---|
| normal mode write (ms) | 827 |
| degraded mode write (ms) | 998 |
| normal mode read (ms) | 844 |
| degraded mode read (ms) | 917 |

**Table 1. Overhead of degraded mode**

rejected. If the constraint is not critical it is marked for re-evaluation at reconciliation time. The operation is allowed to proceed and the object is marked as revocable. If the primary copy of the object is not available, a secondary is chosen as a temporary primary. This choice is broadcasted to all reachable nodes. Furthermore, the object version of the object being written to is incremented. If an operation tries to write to an object that is covered by a critical constraint that also contains a revocable object, the operation is rejected. This is done, in order to avoid violating critical constraints retrospectively at reconciliation time, when an operation might be undone.

Three hundred object write invocations and three hundred read operations were performed in normal mode. All objects were covered by constraints, but these constraints were configured to be always met. The average time for a single write and read invocation was calculated. The experiment was then repeated, in degraded mode. To enter degraded mode, one of the nodes was forced to crash.

Table 1 shows the results of this experiment. As can be seen write invocations were on average 171ms slower in degraded mode. Thus, degraded mode adds an overhead of about 20% for write operations. For read operations the difference was only 73ms, thus the overhead is only about 8%.

### 6.3 Update Propagation

The next experiment was designed to investigate whether delaying the propagation of replica updates until the end of the transaction could be a worthwhile performance optimisation. Updates are currently propagated after each individual invocations, even if various objects on the same node are updated in a nested invocation. In this special case updates could be propagated in less messages, if the propagation was delayed until the end of the transaction. However, these fewer messages might have to be sent to more nodes, as object replicas might not be distributed uniformly.

We measured the overhead of multiple update operations in the ideal case for this optimisation, where all updated objects reside on the same node. To this end, we have repeated the first experiment for write invocations only. The level of nesting was increased in consecutive runs of the experiment. All nested invocations were local invocations on the server node that received the initial invocation.

Table 2 lists the average write times of the whole nested invocation at the various levels of nesting[1]. The results are surprising, in that the invocation times only show a slight tendency to increase with increasing nesting. This indicates that the main cost in our system is the initial remote invocation and that the repeated update propagation message in this simple case is not very expensive.

| level of nesting | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| write time (ms) | 827 | 976 | 981 | 976 | 1029 | 1031 |

**Table 2. Nesting of write operations**

Therefore, according to this experiment, it seems doubtful that delaying update propagation until the end of the transaction significantly improves the performance. However, we plan to investigate this issue further, as more elaborate experiments are needed to confirm the cost of update propagation when more nodes and objects are involved and a larger number of concurrent invocations from various clients occur.

## 7 Evaluation of Other System Components

Although DeDiSys Lite is aimed at evaluating replication protocols, it also serves as an expandable prototype for the implementation of the DeDiSys middleware. The system can be used to evaluate other system components. Each of the simulated components described in section 4 can be replaced by a real implementation.

The system can be used to run experiments with different algorithms for each component. For example a full transaction manager could be added to the system. By replacing the commit protocol different policies could be evaluated.

## 8 Future Work

We are currently using DeDiSys Lite to further fine tune and improve our P4 replication protocol and design and implement other replication protocols.

Furthermore, we are planing to evaluate the P4 by comparing it with existing protocols. As mentioned above, we have already performed an analytical study comparing the protocol to the primary partition approach. DeDiSys Lite will be used to verify this study. To this end we will simulate more elaborate application scenarios taken from DeDiSys target applications, such as the Distributed Telecommunication Management System (DTMS). The DTMS monitors and controls a distributed voice communication system used in air traffic control. It is is provided as an application scenario by one of the industrial partners of the DeDiSys

[1]Note, that the nested local invocations are included in these measurements.

project. The DTMS has already been used to extract typical system parameters. In a typycal DTMS scenario objects are hosted on no more than 100 hosts and each object has no more than ten replicas. Each integrity constraint involves on average three objects and a maximum of 10% of the constraints are critical. We plan to use the same parameters in our DeDiSys Lite simulation, in order to verify the results of the analytical study. In addition we hope to use these parameters to compare the DeDiSys approach to Fault Tolerant Corba with the primary partition model.

In order to simplify protocol evaluation an additional component, a **Performance Monitor**, is to be introduced. The purpose of this configurable component is to measure various performance and availability metrics automatically.

DeDiSys Lite also serves as a basis for the implementation of the DeDiSys middleware. By replacing simulated components with real implementations we are hoping to be able to thoroughly test and evaluate the majority of the DeDiSys components.

Finally, we hope to use results and insights obtained from this work in partitionable environments in the MADIS project [8]. MADIS is a middleware system primarily aimed at database replication.

## 9 Conclusion

We have presented DeDiSys Lite, a simplified prototype implementation of the DeDiSys distributed object system. DeDiSys Lite serves as a simulation environment for evaluating replication protocols in partitionable distributed object systems. The design of the environment and the flexibility of the configuration files, allows efficient modification of system parameters which might affect the replication protocols. An introduction to two of the replication protocols we are evaluating with the use of DeDiSys Lite has been given.

We have also provided a short introduction to the DeDiSys approach of trading consistency for availability, and compared it to related work. The implementation of the full DeDiSys system is work in progress and DeDiSys Lite is used as a basis for its implementation.

## References

[1] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society.

[2] S. Beyer, M. Bañuls, P. Galdámez, and F. D. Muñoz-Escoí. Increasing availability in a replicated partionable distributed object system. Technical Report ITI-ITE-05/10, Instituto Tecnológico de Informática, 2005.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The primary-backup approach*, pages 199–216. ACM Press, Addison-Wesley, 1993.

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

[5] 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: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 245, 1998.

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

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

[8] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J. E. Armendáriz, and F. D. Muñoz-Escoí. MADIS: A Slim Middleware for Database Replication. In *11th International Euro-Par Conference*, pages 349–359, 2005.

[9] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[10] E. B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[11] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "non partition" assumption. In *IEEE Proc Fourth Workshop on Future Trends of Distributed Systems*, 1993.

[12] F. B. Schneider. What good are models and what models are good? In *Distributed Systems*, chapter 2, pages 17–26. ACM Press, Addison-Wesley, 2nd edition, 1993.

[13] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, May 1999.

[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.