International Workshop on Self-Managing Database Systems (SMDB2005),
April 8-9, 2005, Tokyo, Japan.

A Self-Managing High-Availability Database: Industrial Case Study

Antoni Wolski
Solid Information Technology
wolski@solidtech.com

Béla Hofhauser
Fujitsu Siemens Computers
bela.hofhauser@fujitsu-siemens.com

Abstract

While it is obvious that a highly available database requires some degree of self-management, it is not clear to what extend and how the responsibilities should be divided among the DBMS, the surrounding HA framework or even applications. We start with analyzing expectations of an HA database in a telecom setup. We check how the expectations are met in a commercial implementation of a telecom platform. We analyze the way of allocating various HA responsibilities to different parts of the system. We end up with a reference division of self-managing responsibilities in an HA DBMS and the surrounding HA framework.

1. Introduction

The way high-availability (HA) systems deal with failures is that failures are allowed, but the concept of *service continuity* is embraced. It is based on the notion that very short breaks in operation of service-providing applications are tolerable to a certain extent. For example, on these premises, the Service Availability Forum (SA Forum)[1] is producing API specifications for the use by manufacturers of high-availability frameworks that are being delivered to the telecom field as components of telecom platforms. To say to what extent failures are tolerable, one may use the measure of availability *A*. It is expressed as the percentage of the time a service is operational, as related to the total time the service is supposed to be operational. Other important measures are: a maximum duration of an outage (equal to mean time to repair, MTTR) and a frequency of outages (represented with mean time between failures, MTBF). The three quantities are bound together with the formula:

$$A = \frac{MTBF}{MTBF + MTTR} \bullet 100\%$$

Typically, the required level of availability is expressed as the number of "nines". In a "five nines" system, the required availability is not less then 99.999% (a typical telecom requirement). This represents a maximum of about 5 minutes of downtime during one year. Additionally, the maximum outage time is required to be between 50 ms and 5 s, depending on the application. It is obvious that a great deal of self-management is to be utilized in such systems in order to meet the availability requirements, especially in unattended operations. Thus, HA technologies go hand in hand with autonomous approaches as they complement each other. Autonomous computing [2] [8] (synonymous to self-management) is "best considered a strategic refocus for the engineering of effective systems" [13]. The attributes of an autonomous system are *self-protecting*, *self-configuring*, s*elf-healing* and *self-optimizing*. System availability falls, naturally, under the category of self-healing.

Databases have become an important ingredient of telecom platforms. Being at a focal point of the service applications, they bear the highest availability expectations. However, most of the advances in autonomic database research have been addressing the attributes of self-optimizing and self-configuring, rather then self-healing in the sense of availability preserving [5].

When you are building a HA database management system (HADBMS), you can ask the following question from the dependability point of view [10]: what are the threats to availability and how are you going to deal with them in a self-managed way? The answer seems to be, traditionally, that the threats are errors and the resulting faults [10]. Errors and faults may appear both in hardware and software. However, a new significant source of errors and faults is the process of dynamic software upgrades [11] [14]. The graveness of the phenomenon has caught both researchers and practitioners by surprise: a casual scan of the trade press reveals that more then half of serious mobile network outages (1 hour or more) was caused by failed software upgrades.

In an HA system model like that of the SA Forum's Availability Management Framework (AMF) [1] [7], all threats translate to failures. The unit of failure is a *service unit* that may comprise of one or more components. To deal with failures, the system

[1] www.saforum.org

embodies redundancy both in hardware and software, managed by the framework like AMF. Each software component is connected to AMF by way of a related API. The most common redundancy model supported is that of active-standby whereby the framework commands standby components (in a standby unit) to action should any component in the active unit fail. The AMF model is a synthesis based on many existing commercial HA framework implementations.

In this paper, we set a question whether the DBMS-internal self-management capabilities are sufficient to deal with typical threats in a highly-available system. Another question is whether an HA framework may be useful in improving the database availability. We analyze the problems and solutions encountered in the development of an industrial platform: Resilient Telco Platform for Continuous Services (RTP4CS™) by Fujitsu Siemens Computers. Having done the study, we conclude that the answer to the former question is "no", and to the latter one "yes". As a result, we propose a reference division of responsibilities in a system supporting highly available data services.

2. Highly Available Databases and Threats to Availability

2.1. Solid CarrierGrade Database

In highly available databases, redundancy may be applied both to processes (running program instances) and data, thus resulting in numerous redundancy models [4]. In this paper, we deal with the most basic one, that is a fully replicated active-standby configuration, called here Hot-Standby Database (HSBDB).

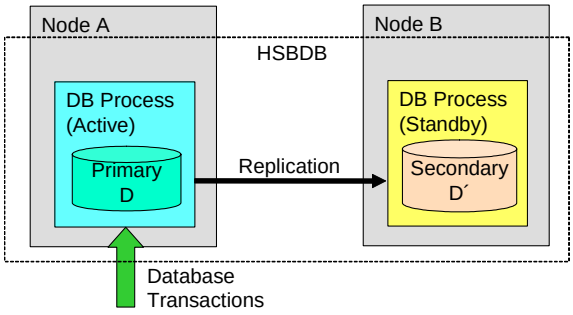

Fig. 1. A fully replicated hot-standby database.

Fig. 1 depicts the architecture of HSBDB. In the case study introduced in the next Section, a corresponding HSBDB implementation by Solid[2] was utilized.

[2] www.solidtech.com

The process components are called *Active* and *Standby* and the corresponding data components are called, *Primary* and *Secondary*, respectively (for simplicity, we will call the processes Primary and Secondary, too). The state of Secondary is maintained by way of a continuous transactional replication. We deal with one-way replication only, as it is commonly preferred over more complex replication schemes, for performance reasons. The effect of one-way replication is that the Secondary process can barely offer any transaction service other than read-only transactions.

The replication protocols may be roughly divided into 1-safe and 2-safe protocols [6] and, especially for 2-safe protocols, further subclasses may be defined [4]. The word "hot" in "hot standby" means that the standby process may take over the load immediately, regardless of the protocol used. "Immediately" is understood as allowing for delays between a few tens of milliseconds and a few seconds. This precludes any possibility to perform a full database recovery or restore. The failover time is spent entirely on internal state processing and flushing the transaction queues. The time required by the latter task depends on the replication protocol used, as different types of protocols represent different trade-off between run-time performance and failover time [4]. The self-healing capabilities of an HADBMS (in the sense of preserving availability) are reflected in the externally perceived state behavior. A corresponding simplified state diagram of the Solid CarrierGrade Database Engine [12] is shown in Fig. 2.

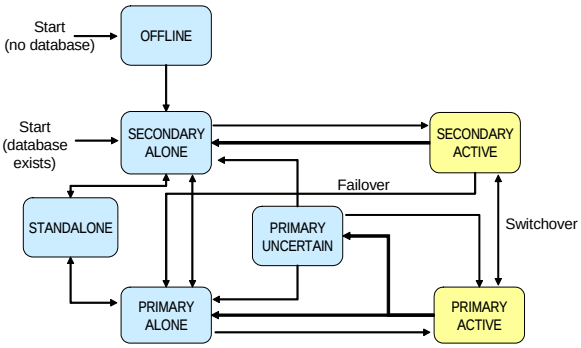

Fig. 2. HA State diagram of the Solid Carrier Grade Database Engine.

The operational hot-standby states are shown on the right-hand side of the diagram: PRIMARY ACTIVE (i.e. active) and SECONDARY ACTIVE (i.e. standby). Other states come into the picture when taking care of various failure, startup and reconfiguration scenarios.

The state behavior is externalized in the form of commands for invoking state transition and querying the state. The commands are available to applications (or an HA manager) as extensions of the SQL lan-

guage. For a full description of the states and transitions, see [12]. The transitions shown in bold, in Fig. 2, are executed autonomously by the database server process. They have to do with falling back to a “disconnected” state (i.e. PRIMARY ALONE or. SECONDARY ALONE), both on the Primary and Secondary side, should a communication failure occur between Primary and Secondary. This behavior is possible thanks to a built-in heartbeat functionality. All other transitions are invoked with administration commands.

Thus, the crucial failover transition is also invoked by an external entity, like an HA manager or an HA framework. It is performed with a single command, 'SET PRIMARY ALONE' that may be issued in both the SECONDARY ACTIVE and SECONDARY ALONE state (because the Secondary server might have fallen back to the ALONE state already). The resulting state is PRIMARY ALONE that is HA-aware in the sense that it involves collecting committed transactions to be delivered later to the Secondary, upon reconnect and the resulting “catchup” (i.e. re-synchronizing the database state).

If a failure case we are dealing with is such that no reconnect is likely to happen in the near future, there is a possibility to move to a “pure” STANDALONE state that has no HA-awareness. In that case, future actions may include restarting a Secondary database server without a database, and sending a database over the network (so-called “netcopy”). For this purpose there exists a startup state OFFLINE whose only purpose is to receive the database with a netcopy. After the successful netcopy, the state SECONDARY ALONE is reached, and the command 'CONNECT' brings both servers back into the operational hot-standby state. On the other hand, if the candidate Secondary involves a database that can be resynchronized (caught-up), the startup state is SECONDARY ALONE.

The auxiliary state PRIMARY UNCERTAIN is meant for reliable dealing with Secondary failures. If the internal heartbeat alerts the Primary server that the communication with the Secondary has failed, and there are transaction commits that have been sent but not acknowledged by the Secondary, the resulting state is PRIMARY UNCERTAIN. In this state the outstanding commits are blocked until resolved. The resolution may happen automatically when the Secondary becomes reconnected. Or, if the Secondary is assumed to become defunct for a longer period of time, command-invoked transitions are possible, e.g. to PRIMARY ALONE whereby the outstanding commits are accepted. (NOTE: the PRIMARY UNCERTAIN state is not mandatory—it may be by-passed with a configuration parameter setting.)

In addition to the above transitions dealing with failures, a role switch (switchover) may be performed for maintenance purposes. It is invoked with dedicated commands 'SWITCH [TO] PRIMARY' and 'SWITCH [TO] SECONDARY'.

2.2. Why So Little Autonomy?

A question immediately arises: “why the failover cannot be performed fully autonomously, by the database server?”. The answer is: because of network partitions. When the Secondary looses the Primary connection, there is just not enough information about the total state of the system to perform a failover. The inactive connection may be a result of a network partition whereby the Primary, in fact, functions properly and serves the applications but cannot reach the Secondary. If the Secondary unilaterally decided upon failover, the system would end up with two Primaries, and that would be an invitation to a database consistency disaster (with no copy update reconciliation method available). Therefore, the responsibilities for self-management have to be distributed among various levels of the system, as will be shown in the sequel.

2.3. More Self-Healing

Failover capability is not the only possibility to address self-healing at runtime. Two other examples are *adaptive durability* and *log stop*.

Adaptive durability. The goal of adaptive durability is to guarantee durability of transactions in a most efficient way. Advantage is taken of the fact that, in normal hot-standby operation, the log is effectively written to Secondary in a synchronous way, if only a 2-safe protocol is used. We will call this effect *hot-standby durability*. Additionally, writing over the network, to another computer's memory turns out to be at least an order of magnitude faster than writing synchronously to a local disk. Therefore, in the operational hot-standby state, the local log can be written asynchronously. Should a failure of Secondary or Primary happen, leading to the PRIMARY ALONE state, the local log operation is automatically changed to synchronous, thus guaranteeing full durability in a single-node operation. By *full durability* we mean that all committed transactions are recoverable after any level of failure other than media failure. Once the two-node hot-standby operation is resumed, the local log writing falls back to the asynchronous mode. Note: the approach does not guarantee full durability in the presence of a total system failure, with some replication protocols like *2-safe received* [4].

Log stop. Transaction log files occasionally grow in an unexpected way because of reasons like failed backups, delayed checkpoints, or simply high data load. As a result, the system may run out of disk space. In a non-adaptive DBMS, such situation would most likely result in a system crash, or, at best, in service denial. On the other hand, in HSBDB, we may again take advantage of the Secondary that is logging the data anyway. We may apply an adaptive mechanism whereby the local log writing is stopped and periodically re-enacted, for example in connection with checkpoints or backups (when hopefully some disk space will be freed). Note: neither this approach guarantees full durability in the presence of a total system failure, with some replication protocols.

2.4. Threats to Continuous Data Management

Below is a list of threats that an HADBMS may face in real-life environments. The environment we assume is a multi-node cluster computer. The database is run on selected nodes of the cluster. The purpose of the list is to serve as a basis for the case study analysis in the next Section.

Hardware threats

- CPU hardware fault (has to be replaced)
- Disk hardware fault (has to be replaced)
- Disk is full (has to be reorganized)
- Network adapter fault
- Network partition
- Power supply failure: isolated
- Power supply failure: total
- HW upgrade or maintenance

HADBMS software threats

- Server process exception or assertion
- Process hang-up
- Database corruption
- Unexpected build-up of resources, e.g. memory allocation
- Unexpected degradation of performance
- DBMS upgrade

External software threats

- OS failure
- HA framework failure
- HA framework hang-up
- OS upgrade
- HA framework upgrade
- Application upgrade

3. The Case of Solid and RTP4CS

3.1. About RTP4CS

Fujitsu Siemens Computers[3] offers SAFE4CS™ (SA Forum Environment for Continuous Services) as a SA Forum-compliant carrier-grade middleware suite. Carrier-grade environments require extremely high availability (99.999%+) and manageability. To meet this requirement, Fujitsu Siemens Computers have developed SAFE4CS consisting of PRIME-CLUSTER™, a standard clustering software solution, RTP4CS carrier-grade high-availability middleware, and ServerView, the company's systems management tool. The configuration of a server node is shown in Fig. 3.

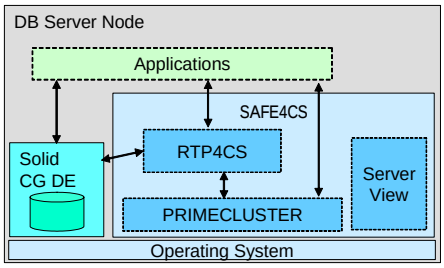

Fig. 3. Components of a server node.

RTP4CS (Resilient Telco Platform for Continuous Services) [9] is targeted for utilization in contemporary network elements like that of 3G mobile networks, Intelligent Networks and Voice-over-IP services. It runs on multi-node clusters built on several hardware platforms, currently under Linux and Solaris operating systems. The overall architecture of the platform is shown in Fig. 4.

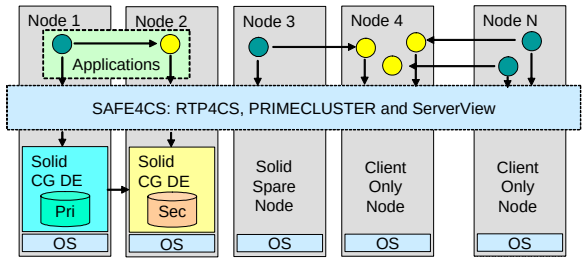

Fig. 4. Cluster architecture.

RTP4CS is a distributed system managing the redundancy embedded in the system. By "managing redundancy" we mean providing a single system image and isolating the applications from the effects of redundancy, failovers, or even replacement of failed components with dedicated spare components.

The proven versions of RTP4CS offer the required functionality via proprietary interfaces tai-

[3] www.fujitsu-siemens.com

lored to the needs of telco applications. Forthcoming versions of RTP4CS will include the implementation of the evolving standardized Service Availability Forum Application Interface Specifications (SA FORUM AIS).

RTP4CS embeds Solid's CarrierGrade Database Engine (Solid CG DE) for the purposes of configuration management and application data management.

3.2. Failure Handling Scenarios

As concerns threats to the database availability, we try to cover all possible failure handling scenarios. One of the main goals of our project is to provide service availability by coordinating redundant resources within a cluster to deliver a system with no single point of failure. Therefore we don't deal here with double failures.

Beyond the SA Forum AIS functionality, RTP4CS provides the Audit, Recovery and Alarm Manager Instances to handle different failure scenarios. The Audit Manager may watch arbitrary resources, e.g. disk space. The Recovery Manager uses active objects called *reactors* which will be activated, if the resource consumption reaches predefined levels. In such a case, a backup of the database or a cleanup of a file system may be started. With these methods, a lot of error situations can be avoided, and preventive system maintenance may be applied. If the situation can't be handled automatically in any way, the problem will be escalated by the Alarm Manager to the maintenance team. As a last remedy, a cluster restart may be activated manually.

In a telco environment we assume that a redundant hardware and network configuration is used.

Advanced cluster management products, like PRIMECLUSTER of Fujitsu Siemens Computers, supervise not only the nodes, but the network resources too. In the very rare case of a serious network error (inability of a node to communicate), PRIMECLUSTER, using special hardware features, eliminates (resets, reboots) the erroneous node. This solution reduces a network error to the node failure. Consequently, the node failure of the primary node inflicts a failover controlled by the redundancy manager of RTP4CS.

The most important scenarios are listed below, together with the threats they address. The responsibility for a given step is marked in the following way: 'MF' designates the availability management framework (represeneted by RTP4CS) and 'DB' means the DBMS (represented by Solid CG DE).

(A) Failover and quick DB service repair

Threats

- DBMS Server process exception or assertion in the Primary.
- Primary DBMS hang-up.
- DBMS performance degradation in the Primary.
- Unexpected build-up of resources in the Primary DBMS.

Solution

1) DB: Monitor the Primary with:
   - heart beat
   - reading, in real-time, the RDBMS event log.
2) DB (possibly): Secondary: fall back to SECONDARY ALONE.
3) MF: Conclude that the performance of the Primary is not sufficient, or it has failed.
4) MF: terminate the old primary (if not terminated)
5) MF: Execute a failover.
6) DB: On MF's request, command the old Secondary to become a new Primary. Adjust log writing to the required level of durability.
7) MF: Restart the new Secondary DBMS server.
8) MF: Provided the DB recovery is successful, reconnect (if recovery failed, switch to scenario C).
9) DB: resynchronize the databases.
10) DB: Adjust log writing to the required level of durability.
11) MF: (Optionally) execute switchover to return to the preferable Primary/Secondary configuration.

(B) Failover and cold restart of the new Secondary

Threats

- Total software failure of the Primary node.
- Operating system malfunction of the Primary node.

Solution

Same as in scenario A, except that the step 7) should be: reboot the failed node.

(C) Failover and reconstruction of a database

Threats

- Corruption of the Primary database.
- Disk is full.

Solution

1) DB: Filter out the events pertaining to corrupted or inoperable database.
2) MF: Conclude that the database is corrupted.

3) MF: terminate the old primary (if not terminated).
4) MF: Execute failover.
5) DB: On MF's request, command the old Secondary to become a new Primary. Adjust log writing to the required level of durability
6) MF: Repair the failed node and restart it as a new Secondary in the OFFLINE state.
7) DB: Netcopy the database back to the new Secondary.
8) MF: Reconnect.
9) DB: Resynchronize the databases.
10) DB: Adjust log writing to the required level of durability.

(D) Failover and activation of a Spare

Threats

- Serious hardware failure (damage of the CPU or disk).
- Network partition (isolated node).
- Network adapter fault.
- Local power supply failure.

Solution

1) DB/MF: Detect inoperable or isolated primary node.
2) DB: Secondary: fall back to SECONDARY ALONE.
3) MF: Execute failover.
4) DB: On MF's request, command the old Secondary to become a new Primary. Adjust log writing to the required level of durability.
5) MF: Designate a Spare node to be a new Secondary.
6) MF: Start the new Secondary in the OFFLINE state.
7) DB: Netcopy the database to the Secondary.
8) MF: Reconnect.
9) DB: Resynchronize the databases.
10) DB: Adjust log writing to the required level of durability.

(E) Secondary failure and migration to a Spare

Threats (in Secondary)

- Serious hardware failure (damage of the CPU or disk).
- Network partition (isolated node).
- Local power supply failure.

Solution

1) DB/MF: Detect inoperable or isolated Secondary node.
2) DB: Primary: fall back to PRIMARY ALONE.
3) DB: Adjust log writing to the required level of durability.
4) MF: Designate a Spare node to be a new Secondary.
5) MF: Start the new Secondary in the OFFLINE state.
6) DB: Netcopy the database to the Secondary.
7) MF: Reconnect.
8) DB: Resynchronize the databases.
9) DB: Adjust log writing to the required level of durability.

(F) Short-term Secondary failure

Threats (in Secondary)

- Process exception or assertion.
- Process hang-up.
- Database corruption.
- Unexpected build-up of resources, e.g. memory allocation.
- Unexpected degradation of performance.

Solution

1) DB/MF: Detect inoperable or isolated Secondary node.
2) DB: Primary: fall back to PRIMARY ALONE.
3) DB: Adjust log writing to the required level of durability.
4) MF: Restart the Secondary node
5) MF: Reconnect.
6) DB: Resynchronize the databases.
7) DB: Adjust log writing to the required level of durability

(G) Maintenance Secondary migration to a Spare

Threats

- Hardware upgrade.
- Hardware maintenance.
- Any software upgrade to be performed and tested in isolation,by way of a spare node.

Solution

1) MF: Set Primary to PRIMARY ALONE.
2) MF: Terminate Secondary.

Continue with step 3) of scenario E.

(H) Short-term Secondary maintenance

Threats

- quick OS or HA framework upgrade.
- quick HW upgrade or maintenance.

Solution

1) MF: (If needed) Execute switchover.
2) MF: Set Primary to PRIMARY ALONE.
3) DB: Adjust log writing to the required level of durability.
4) MF: Terminate Secondary.
5) Perform the Secondary upgrade or maintenance.

Continue with step 7) of scenario A.

(I) Switchover to a new DBMS version

Threats

- DBMS version upgrade.

Solution

1) MF: Terminate the old Secondary and upgrade.
2) MF: Start the old Secondary and reconnect.
3) MF: Execute switchover.
4) DB: On MF's request, command the old Secondary to become a new Primary and vice versa.
5) MF: Shut down the new Secondary and upgrade.
6) MF: Start the new Secondary.
7) DB: Reconnect.
8) DB: Resynchronize the databases.
9) DB: Adjust log writing to the required level of durability.

(J) Miscellaneous failures

Threats

- Total power supply failure.
- HA framework failure.
- HA framework hang-up.

Solutions

The above failures cannot be dealt with by the HA framework because it is not operational. A cold restart is required whereby the state of all components is rebuilt from the persistent storage. A normal database recovery procedure will assure that the databases (both Primary and Secondary) are recovered to a consistent state.

4. Failure Scenarios: What Have We Learned?

4.1. Discussion

From the implemented scenarios shown above, you can see that the responsibilities for management of redundant DBMS components were divided between the DBMS and the AMF.

Was it needed? Could not the HADBMS manage totally its own redundancy? To answer this question, let us assume that there existed some layer, or component, of the HADBMS to do that. Then, for example, in scenario D (Failover and activation of a Spare), the DBMS might choose a certain node to become a new Secondary. In the same time the cluster's own availability management framework may "have a different idea" about the choice. For example, the selected node might be engaged in serving other purposes. Allocation of resources might be also done dynamically and thus the two redundancy managers would compete for resources. So far, no work has been done on developing methods for coordination of competing redundancy managers. Additionally, the DBMS may not be able to collect all the information about the cluster status—the information a well-instrumented AMF would have at its disposal. The conclusion is that the redundancy management has to be centralized, in a cluster, in order for the resources to be used efficiently and in a consistent way. This goal is achieved using means available to AMF: configuration definitions assigning available resources to *service units* and *service groups* (sets of units), rules for cluster membership whereby nodes and other components may join and leave the cluster, redundancy models supported, and redundancy management policies that embody scenarios as shown above.

What is left to DBMS? An HADBMS should be built to cooperate seamlessly with the AMF. For this to happen in general case, standard interfaces have to be upheld. The work by SA Forum has already produced such interfaces [1].

Based on the experience of this case study, the division of responsibilities between a DBMS and AMF proposed below seems to be plausible.

4.2. HADBMS: division of responsibilities

Responsibilities of the Availability Management Framework

- Maintain cluster membership.
- Maintain redundancy configuration (components, service units, service groups, etc.) as specified.
- Execute the redundancy management policies and failover scenarios as specified, for the HADBMS components. Specifically, start and terminate DBMS processes, control the DBMS HA states.
- Monitor the performance of the HADBMS components.

Responsibilities of the HADBMS

- Perform self-healing actions that are restricted to a single process, database or node (no global system state knowledge necessary—like adaptive durability).
- Perform safe state transitions based on the events in the immediate (local node) environment (like falling back to the ALONE state)
- Execute HA state transitions as requested by AMF.
- Comply with the AMF interface.

5. Conclusions

By analyzing a concrete example of implementing a highly available DBMS in the environment of an HA cluster, we have come to the conclusion that the responsibility to deal with various threats should be distributed at least between two layers of the system: the DBMS itself and the availability management framework. In order to build total HA systems out of heterogeneous and pre-existing components, the division of responsibilities and the suitable interface have to be defined. In this respect, the work by the Service Availability Forum is a welcome activity. The interaction between the HA framework and a DBMS may be defined in the context of SA Forum interfaces. However, database developers have to agree on what is a set of functionality supported in an HA DBMS to be mapped to the SA Forum interfaces.

References

[1] "Application Interface Specification, SAI-AIS-B.01.01", Service Availability Forum, November 2004, available at www.saforum.org.

[2] "Autonomic Computing Manifesto", IBM White Paper, IBM, 2001, available at www.research.ibm.com/autonomic/manifesto/autonomic_-computing.pdf.

[3] S. Brossier, F. Herrmann, E. Shokri, "On the Use of the SA Forum Checkpoint and AMF Services" *Proc. International Service Availability Symposium (ISAS 2004*), May 13-14, 2004, Munich, Germany.

[4] S. Drake, W. Hu, D. M. McInnis, M. Sköld, A. Srivastava, L. Thalmann, M. Tikkanen, Ø. Torbjørnsen and A. Wolski "Architecture of Highly Available Databases", *Proc. International Service Availability Symposium (ISAS 2004*), May 13-14, 2004, Munich, Germany.

[5] S. Elnaffar, W. Powley, D. Benoit, and P. Martin, "Today's DBMSs: How Autonomic Are They?", Proc. First International Autonomic Systems Workshop, DEXA 2003, Prague.

[6] J. Gray and A. Reuter, "Transaction Processing Systems, Concepts and Techniques", Morgan Kaufmann Publishers, 1992-

[7] T. Jokiaho, F. Herrmann, D. Penkler, L. Moser, "The Service Availability Forum Application Interface Specification", *The RTC Magazine*, June 2003, pp. 52-58.

[8] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", *IEEE Computer* 36(1): 41-50 (January 2003).

[9] J. Neises, "Benefit Evaluation of High Availability Middleware", *Proc. International Service Availability Symposium (ISAS 2004*), May 13-14, 2004, Munich, Germany.

[10] B. Randell, "Turing Memorial Lecture – Facing Up to Faults", Comp. J. 43(2), pp 95-106, 2000.

[11] M. Segal, O. Frieder, "On-the-fly Program Modification: Systems for Dynamic Upgrading", *IEEE Software*, March 1993, pp. 53-65.

[12] "Solid High Availability User Guide, Version 4.2", Solid Information Technology, June 2004, available at http://www.solidtech.com.

[13] R. Sterritt, D. W. Bustard, "Autonomic Computing - A Means of Achieving Dependability?", *Proc. 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2003)*, pp. 247-251.

[14] A. Wolski and K. Laiho, "Rolling Upgrades for Continuous Services", *Proc. International Service Availability Symposium (ISAS 2004)*, May 13-14, 2004, Munich, Germany.