Fault-tolerance in a Distributed Management System: a Case Study

Robert Smeikal
Vienna University of Technology
Gusshausstrasse 27-29/384
A-1040 Vienna, Austria
smeikal@ict.tuwien.ac.at

Karl M. Goeschka
Frequentis Nachrichtentechnik GmbH
Spittelbreitengasse 34
A-1120 Vienna, Austria
goeschka@acm.org

Abstract

Our case study provides the most important conceptual lessons learned from the implementation of a Distributed Telecommunication Management System (DTMS), which controls a networked voice communication system. Major requirements for the DTMS are fault-tolerance against site or network failures, transactional safety, and reliable persistence. In order to provide distribution and persistence both transparently and fault-tolerant we introduce a two-layer architecture facilitating an asynchronous replication algorithm. Among the lessons learned are: component based software engineering poses a significant initial overhead but is worth it in the long term; a fault-tolerant naming service is a key requirement for fail-safe distribution; the reasonable granularity for persistence and concurrency control is one whole object; asynchronous replication on the database layer is superior to synchronous replication on the instance level in terms of robustness and consistency; semi-structured persistence with XML has drawbacks regarding consistency, performance and convenience; in contrast to an arbitrarily meshed object model, a accentuated hierarchical structure is more robust and feasible; a query engine has to provide a means for navigation through the object model; finally the propagation of deletion operation becomes more complex in an object-oriented model. By incorporating these lessons learned we are well underway to provide a highly available, distributed platform for persistent object systems.

1. System Context

High availability is not only demanded for safety-critical networked voice communication systems (VCS) but also for the management systems controlling them. In our case, these VCS networks are managed by a **D**istributed **T**elecommunication **M**anagement **S**ystem (DTMS) as illustrated in figure 1. Multiple DTMS servers are connected via a WAN and every DTMS server configures, controls and monitors its associated VCS systems. Every DTMS stores VCS parameter data using an object-oriented model.

Following the general design principles of Component-based Software Engineering (CBSE) [6] like strong component coherence and well-defined interaction standards, we consider the following requirements, which are relevant for availability considerations:

Fault-tolerance: The system has to be tolerant against network and site failures. This is achieved through replicating object-states to other sites.

- All objects need to be available for read access at any available site at all times even in presence of arbitrary system degradation scenarios.
- For write access it is sufficient
  - if objects are available at all sites during periods of healthy system condition.
  - if objects are available at a particular site (which we call their home site) during periods of degraded system condition.

Moreover, the framework itself has to be fully distributed. A single unique framework component must not exist in order to avoid *any* single point of failure throughout the whole system.

Transaction: The objects shall be able to participate in transactions. Typically, small write and large read transactions need to be performed frequently.

Persistence: The persistent objects' data shall be written to a stable storage.

Transparency: Clients shall not be aware of distribution, replication and persistence issues, which are therefore transparent to clients.

In our case, both, distribution *and* persistence, have to be transparent *and* fault-tolerant, where ***fault-tolerance*** is the

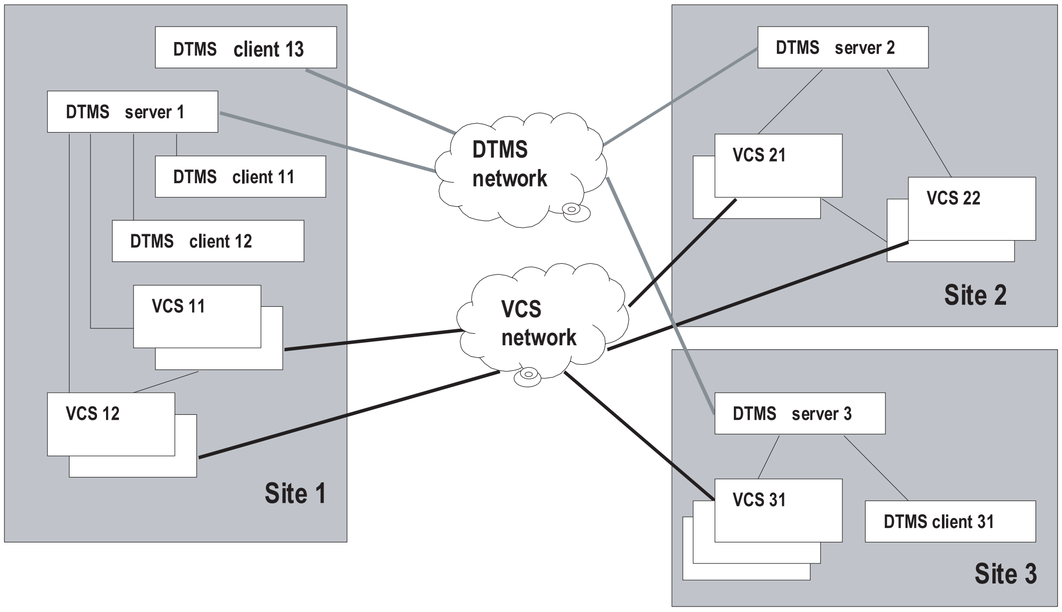

Figure 1. DMTS overview.

ability of a system to continue functioning while a failure is still unrepaired or even undetected [7]. Since fault-tolerance needs to be incorporated into the system architecture itself, we replicate the persisted object to other sites, in order to provide enough redundant information to replace any failed object dynamically. In the best case, the application is available if *any* site is available, because now objects can be restored at every site.

Replication as a means to provide fault-tolerance is well suited and a plethora of replication protocols exist. However, the deployment of replication requires the management of the replicated object data and introduces the new requirement of transparency of the replication. Moreover, if deployed in an object-oriented, persistent and distributed software system, managing multiple physical copies that constitute the state of a single logical copy poses problems on conventional concepts of such systems, which are usually designed to operate on a single copy:

- the preservation of consistency among different logical objects
- the concurrent access to logical objects
- the transactional access to logical objects
- handling of object identities, which are unique among logical objects

The contribution of this case study is to describe the major rationale for the key decisions and to derive general lessons, which we believe can be learned from our experience.

The major findings are:

- Distribution and replication need to be installed as two-layer architecture. This way, conventional distribution mechanisms can be deployed unaltered.
- Fail-safe naming services are a key requirement to fail-safe distribution.
- Performance requirements are more difficult to ensure due to the complexity of the systems.

Both authors have been heavily involved in the DTMS development project:

Karl M. Goeschka is Chief Scientist at Frequentis Nachrichtentechnik GmbH with headquarters in Austria and several subsidiaries in other countries (USA, Canada, Germany). Frequentis is world market leader for fast and highly available voice communications in air traffic ontrol. The products are also used for public safety and safe communications. Due to the innovative character of the DTMS development he is also the responsible manager of this project.

Robert Smeikal is a research assistant at the Vienna University of Technology in Austria. He is currently working on his Ph.D. and engaged in consulting the development team at Frequentis concerning practical and conceptual software engineering issues.

2. DTMS Architecture and Components

Figure 2 depicts the system architecture using a component diagram (Unified Modeling Language UML syntax).

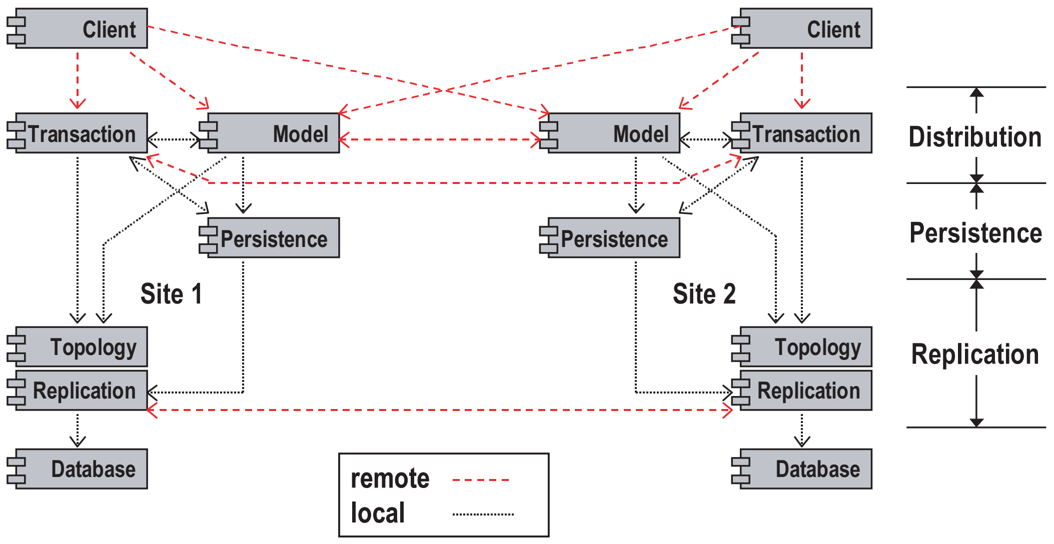

Figure 2. Proposed architecture.

The picture shows how components use each other either locally or remotely. A single site consists of the following components: **`Transaction`**, **`Model`**, **`Persistence`**, **`Topology`**, **`Replication`** and **`Database`**. **`Client`** components do not belong to a site. A client can be connected to an arbitrary site. Please note, that remote communication is established between components of the same type only (eg. local **`Replication`** and remote **`Replication`**). As denoted on the righthand side, the **`Model`** and **`Transaction`** components belong to the distribution aspect, the **`Persistence`** component to the persistence aspect and the **`Topology`** and **`Replication`** component to the replication aspect of the architecture.

Model. The **`Model`** component contains the model of the VCS. This model

- is complex, i.e. it consists of many classes, small object-states and few object-states per class.
- requires complex validation, i.e. many algorithmic constraints involving many object-states.
- requires complex read operations concerning many objects, e.g. assemble information for configuration of the VCS.

Each model object has a system wide unique ID. Each class has an associated factory, which can be located using the class name (kind of naming service). The factory creates and deletes objects. The factory validates object-constraints.

Objects and factories register at component **`Transaction`**. Objects store their serialized object-state at **`Persistence`**. Factories query **`Topology`** for the currently responsible site and initialize accordingly using either remote **`Model`** or local **`Persistence`**. Factories provide Interoperable Object References (IORs, refer to CORBA [1]) pointing at their associated local or remote objects given an ID or another query. Factories obtain IORs pointing at remote objects by communicating with remote factories. Clients can obtain IORs for every object at every associated factory.

Transactional behavior and persistence are mandatory for all model objects. Therefore, all model objects implement the the `validate()`, `commit()`, `rollback()`, and `store()` method.

Client. The **`Client`** component performs invocations on objects in local or remote **`Model`**, where the physical location of the model object is transparent to **`Client`**. An arbitrary number of **`Client`** components can operate on the model concurrently. The **`Client`** component starts and finishes transactions at **`Transaction`**.

Topology. The **`Topology`** component maintains a list of all sites and their availability and provides information about the currently responsible site for a particular object. It is queried by the **`Model`** and **`Transaction`** components. Therefore, **`Topology`** tells other components at which site they have to look for an object, also if the home site of that object cannot be contacted.

Transaction. The **`Transaction`** component is responsible for serializing concurrent transactions. It coordinates the invocation of `commit()`, `rollback()`, and `validate()` on the objects in the **`Model`**, it controls the transaction context at **`Persistence`** and it coordinates distributed transactions in co-operation with remote **`Transaction`** components (2-phase commit [3]).

Currently, the smallest unit of lock granularity is a whole site. Since write transactions are not expected to last long and validation involves many object-states from different classes, this is sufficient. We differentiate between write and read transactions.

Persistence. The **`Persistence`** component stores and retrieves serialized object-states using the **`Replication`** component. Additionally, it provides methods to access object-states in a transactional way, which are used by **`Transaction`**. **`Persistence`** does not distinguish between object-states that are home at the local site and object-states that are home at a remote site, but always persists to and retrieves from the local **`Replication`** component. Factories always ask their local **`Persistence`** component for retrieving persisted object-states. An object persists itself providing its ID, its type and its object-state to **`Persistence`**, which in turn provides transactional access to the stable object-store.

Database. The Database component essentially encapsulates the actual stable storage. It provides function primitives to persist and retrieve data. It may also encapsulate local database redundancy for even higher availability.

Replication. The **`Replication`** component is at the heart of the architecture, as it implements the replication algorithm. It performs a replication protocol in accordance with **`Replication`** at other sites to propagate object-states, to build constructed object-states, and to calculate their object readiness accordingly. It uses the **`Database`** component to persist data locally and it provides methods to access the stored object-states.

3. Lessons learned

We provide the most important lessons learned with a focus on conceptual aspects rather than implementation details or findings regarding particular COTS (commercial off the shelf) products and tools.

Semantics of methods versus read/write: There is a considerable amount of theory on the subject of exploiting method semantics in order to increase availability of certain methods (e.g. [8], where "intersection relations" capture the information flow among method invocations). In practice, we have not been able to deploy any further classification of methods than read-methods and write-methods, especially in regard to common use cases. Apart from a significant increase of implementation complexity (consider a replication mechanism, which knows about method semantics of every object type and calculates the objects' availability accordingly) it renders the system incomprehensible from the users' point of view.

Mapping of object identity to object reference: Obtaining an object reference (i.e. locating an object) given an object identifier (i.e. an object name) is always of concern if objects are instantiated out of process or even remotely (compare the CORBA Naming Service [1] or the Java Naming and Directory Interface [2]). Though such a naming service is already needed for basic persistence, the matter gets complicated if objects might live in different places due to a particular degradation scenario. On the one hand, the naming service maintains information about the references to objects (where a reference also determines the objects' location), on the other hand, the replication component decides where objects are instantiated if degradation occurs. Thus, the mapping of identity to reference is very dependent on the underlying mechanisms instilled to provide failure tolerance. Furthermore, the required fail-safe naming service has to be available at every site at every time, because the naming service itself has to be fault-tolerant as well.

In our case we resolve an object identifier using two steps. At first, the **`Topology`** component is queried for the presently responsible naming service (the factories in the DTMS), which can be either remote or local depending on the degradation scenario. Secondly, the object identifier is finally resolved.

Additionally, the mandatory use of object identifiers to store references (we call them "soft references") introduces the problem of either having to resolve an identity every time the respective reference is used (which usually yields bad performance) or to maintain a kind of identity/reference cache within every object.

Granularity: Defining the unit of granularity for system operations is a fundamental design decision. It can vary for different parts of the system entailing different implementation complexity and system behavior. The boundaries of granularity in an object-oriented model range from single attributes to the whole model.

We experienced that single objects as unit of granularity are a reasonable approach, mainly because objects are designed to be the units of coherence especially from the use-cases' point of view. However, we use different granularity in different components:

Persistence: Persisting object data using whole objects as unit of granularity was comparatively easy to achieve, because we generate the necessary code from the modelling information using the APIs (application programming interfaces) of our CASE (computer aided software engineering) tools (Rational in our case).

Transactions and concurrency: Currently, a site is the unit of lock granularity. Since this is clearly not sufficient, we strive to use object granularity here as well. This entails the ability of each object to associate method invocations with a particular transaction context and to calculate whether a commit is possible or not. Additionally, more complexity is introduced because of "indirect" write operations due to object relationships and because of the necessary locking of "uninvolved" objects that are used during validation of constraints. We plan to generate the necessary code, which obviously is far more complex.

Two-layer architecture: The design of our two-layer architecture focuses mainly on one requirement: Read access to every object must be possible at all times, while write access must be possible *at a particular site* (its home site) at all times. To free the object implementation from the additional burden of handling synchronous replication (propagate the data and coordinated commit using an atomicity control protocol like in group communication [4]), we decided to use asynchronous replication (commit first, propagate afterwards using queues). Asynchronous replication jeopardizes consistency, if multiple instances of a single object live in the system (e.g. an objects is accessed before changes from a remote site arrive). Therefore, during periods of a healthy system, we use a conventional setup of a distributed system: Every object is instantiated only once, objects communicate with other objects in an synchronous way and concepts of distributed computing can be smoothly applied. This makes up the upper layer, called distribution layer. Furthermore, persisted object-data is propagated asynchronously to prepare the system for a degraded scenario. This makes up the lower layer, called replication layer. All of the above considerations apply to model objects only, combined in the the "model" component. In contrast, "system" components exist at every site and manage the distributed model objects.

Derived from our experience, the lesson here is to use asynchronous replication of object data rather than synchronous replication at the instance level.

CBSE: Persistence and Transaction are usually considered as services of the component model implementation (CMI), but since orthogonality is desired (i.e. replace the replication algorithm) CMI itself should be component-based. The DTMS is not implemented using a particular component model implementation but its design is based on a basic principle of CBSE: strong coherence of concern and accompanying interface standards. From our experience, this approach increases the effort at the beginning of the project, but later on the coordination is reduced significantly, both in terms of technical and organizational issues. So far, our software is used in three projects for different customers on different platforms (Microsoft Windows and Sun Solaris) and only two components were altered to accommodate customer requirements: The rest of the framework remains unchanged and is stable.

However, also a disadvantage of CBSE was encountered. Due to the strict encapsulation of components, multiple mappings from component-internal interfaces to component-external ("provided") interfaces are necessary instead of one single invocation (object to object). That yields a loss in runtime performance, which can pile up considerably as the size of the system grows. Therefore, there is a hidden trade-off between component coherence and runtime performance.

WAN-Bandwidth: During the design phase of the project we did not distinguish between in-process, out-of-process and CORBA calls, because we expected the middleware to hide any related issues from the application programmers point of view. Though this is true, we afterwards experienced that such a unregulated use of CORBA calls congested our 2MBit WAN links dramatically. Apart from CORBA optimization, we had to reconsider carefully where CORBA calls are really necessary.

Object-states to database mapping: We designed our database scheme to accommodate object-states in a generic way using a table with three columns: object id, object type and serialized XML-formatted object-data. This setup provides the flexibility to handle changes in the model without altering either the database scheme or the persistence and replication component. However, we experienced some drawbacks:

- Since no static validation can be applied, the database allows inconsistency of object-states. Though obvious, this proved to be quite unpleasing, because now even small failures in the object model may render the database in an inconsistent state.
- Queries that operate directly on the database are very laborious to implement, since neither pure SQL nor pure XPath/XQuery can be used.
- XML is readable for a human consumer, but this comes at the price of a large amount of redundant information. The size of the XML data is increasing quickly as the size of the system grows and therefore becomes increasingly inefficient to handle. In turn the human

readability of these large and complex structures vanishes. Thus, we currently develop a binary format to substitute XML at the database level.

Hierarchical structure: The hierarchy has proven to be one of the most robust, feasible and practical structures [9]. On the other hand, the structure of the model cannot be limited, because many-to-many relations are needed as well as additional arbitrary associations. Therefore, we found it useful to build a hierarchical structure by means of aggregation references. We introduced a model rule, demanding each object to be reachable by exactly one aggregation hierarchy.

Query engine: Two simple types of query are supported: Get object by ID and get all objects of a certain type. For efficient UI implementation purposes, additional queries for navigation through the object hierarchy are essential, e.g. get the ancestor or get the descendants of an object where additional conditions on attributes are fulfilled. Though a descriptive query language like OQL (object query language [1]) is desirable, the mentioned queries are sufficient.

Propagation of delete operation: When an object instance is to be deleted, there may exist other objects referencing this object. Similar to relational databases, it has to be determined whether or not the reference or the referencing object has to be deleted. This may result in cascading delete operations or in delete restrictions when constraints would be violated. Sometimes this results in quite complex delete implementations on the client side. We do not have an appropriate general solution yet, but we consider this to be an important issue for further investigation.

4. Future work

The following improvement possibilities have been identified:

- Support for a query engine, that allows descriptive queries using OQL.
- Support for finer granularity of locking to yield better concurrency of model access for clients.
- Invocation of write methods at model objects at all sites even during system degradation is a major future requirement.
- A better exploitation of transaction types (currently read/write-available) regarding the trade-off between availability of object-states and consistency requirements is desired [5, 10].

By improving these current system weaknesses we are well underway to provide a highly available, distributed platform for persistent object systems.

References

[1] *http://www.omg.org/*. The Object Management Group.
[2] *http://www.sun.com/*. Sun Microsystems.
[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[4] K. Birman. The process group approach to reliable distributed computing. *Communication of ACM*, 36(12):37–53, December 1993.
[5] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.
[6] G. Heineman and W. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.
[7] A. Helal, A. Heddaya, and B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1995.
[8] M. Herlihy. A quorum consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.
[9] P. Kahn. Information architecture: a new discipline for organizing hypertext. In *Proceedings of the twelfth ACM conference on Hypertext and Hypermedia*, pages 1–2. ACM, September 2001.
[10] D. Skeen. Achieving high availability in partitioned database systems. In *Proceedings of the International Conference on Data Engineering*, pages 159–166. IEEE, 1985.